Язык программирования Java SE 8

Подробное описание, 5-е издание

Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли

ORACLE

Язык программирования Java SE 8

Подробное описание, 5-е издание

The Java® Language Specification

Java SE 8 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Язык программирования Java SE 8

Подробное описание, 5-е издание

Джеймс Гослинг
Билл Джой
Гай Стил
Гилад Брача
Алекс Бакли

ВИЛЬЯМС

Москва • Санкт-Петербург • Киев
2015

ББК 32.973.26-018.2.75
Г72
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*
Перевод с английского *И. Карася*
Рецензент канд. физ.-мат. наук *Д.Е. Намиот*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Гослинг, Джеймс, **Джой**, Билл, **Стил**, Гай, **Брача**, Гилад, **Бакли**, Алекс.

Г72 Язык программирования Java SE 8. Подробное описание, 5-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 672 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1875-8 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли

Язык программирования Java SE 8. Подробное описание

5-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 14.04.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 54,18. Уч.-изд. л. 35,6.
Тираж 400 экз. Заказ № 1907.

Отпечатано способом ролевой струйной печати
в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1875-8 (рус.)
ISBN 978-0-13-390069-9 (англ.)

Оглавление

Содержание

Маурицио, с глубочайшей признательностью.

Предисловие к Java SE 8 Edition

В 1996 году Джеймс Гослинг, Билл Джой и Гай Стил написали в первом издании книги *The Java® Language Specification*:

"*Мы считаем, что язык программирования Java является зрелым языком, готовым для широкого использования. Тем не менее мы ожидаем определенной эволюции этого языка в предстоящие годы. Мы намерены управлять этой эволюцией таким образом, чтобы сохранялась полная совместимость с существующими приложениями*".

Java SE 8 представляет собой единое наибольшее развитие языка Java в его истории. Относительно небольшое количество возможностей — лямбда-выражения, ссылки на методы и функциональные интерфейсы — объединены для получения модели программирования, которая сочетает объектно-ориентированный и функциональный стили. Под руководством Брайана Гетца (Brian Goetz) это слияние достигнуто таким образом, что поощряет лучшие практики — неизменность, отсутствие состояния, композиционность — при сохранении "чувства Java" — удобочитаемости, простоты, универсальности.

Самое главное то, что библиотеки платформы Java SE эволюционировали вместе с языком Java. Это означает, что использование и лямбда-выражений, и ссылок на методы для представления поведения — например, операции, которая должна применяться к каждому элементу списка — дает возможность написания продуктивных и мощных приложений на основе готовых компонентов. Аналогично совместно с языком Java развивалась и виртуальная машина Java, гарантируя, что методы по умолчанию поддерживают эволюцию библиотеки как можно более последовательно как во время компиляции, так и во время выполнения, с учетом ограничений отдельной компиляции.

Инициативы по добавлению функций первого класса[1] в язык Java имели место начиная примерно с 1990-х годов. Предложения *BGGA* и *CICE* около 2007 года добавили новую энергию, а создание *Project Lambda* в OpenJDK около 2009 года привлекло беспрецедентный интерес. Добавление дескрипторов методов JVM в Java SE 7 открыло дверь новым методам реализации при сохранении принципа "написано один раз, работает везде". Со временем изменения языка были пересмотрены в JSR 335 (Java Specification Request), *Lambda Expressions for the Java Programming Language*, группа экспертов которого состояла из Джошуа Блоха (Joshua Bloch), Кевина Бурриллиона (Kevin Bourrillion), Андрея Бреслава (Andrey Breslav), Реми Форакс (Rémi Forax), Дэна Хейдинги (Dan Heidinga), Дуга Ли (Doug Lea), Боба Ли (Bob Lee), Дэвида Ллойда (David Lloyd), Сэма Пулара (Sam Pullara), Сриканта Санкарана (Srikanth Sankaran) и Владимира Захарова (Vladimir Zakharov).

Разработка языка программирования обычно включает борьбу со сложностями языка, обычно совершенно скрытыми от пользователей. (По этой причине ее часто сравнивают с айсбергом: 90% ее остается невидимой.) В JSR 335 наибольшая сложность таилась во

---

[1] First-class functions — это функции, которые можно передавать в качестве аргументов, возвращать как результат и т.д. — *Примеч. ред.*

взаимодействии неявно типизированных лямбда-выражений с разрешением перегрузки. В этой и многих других областях выдающуюся работу проделал Дэн Смит (Dan Smith) из Oracle, тщательно определив желаемое поведение. Сказанное им можно найти везде в данной спецификации, в том числе в новой главе о выводе типа.

Еще одной инициативой в Java SE 8 стало расширение возможностей аннотаций, одной из наиболее популярных возможностей языка программирования Java. Во-первых, была расширена грамматика Java, которая теперь допускает аннотации типов во многих конструкциях языка, формируя основу для новейших инструментов статического анализа, таких как *Checker Framework*. Эта возможность была определена комитетом JSR 308, *Annotations on Java Types*, с возглавляемой Майклом Эрнстом (Michael Ernst) группой экспертов, в которую входили я, Дуг Ли (Doug Lea) и Срикант Санкаран (Srikanth Sankaran). Внесенные в спецификацию изменения были достаточно велики, и неустанные многолетние усилия Майкла Эрнста (Michael Ernst) и Вернера Дитля (Werner Dietl) были благодарно приняты сообществом программистов. Во-вторых, аннотации могут "повторяться" в конструкции языка, что немаловажно для тех API, которые моделируют предметную конфигурацию типами аннотаций. Майкл Кейт (Michael Keith) и Билл Шеннон (Bill Shannon) инициировали эту возможность в Java EE и руководили ею.

Значительный вклад в данную спецификацию внесли многие коллеги из Java Platform Group в Oracle: Леонид Арбузов (Leonid Arbouzov), Мэнди Чанг (Mandy Chung), Джо Дарси (Joe Darcy), Роберт Филд (Robert Field), Джоль Франк (Joel Franck), Сонали Гёл (Sonali Goel), Йон Гиббонс (Jon Gibbons), Джаннетт Ханг (Jeannette Hung), Стюарт Маркс (Stuart Marks), Эрик Мак-Коркл (Eric McCorkle), Метери Нунец (Matherey Nunez), Марк Райнхольд (Mark Reinhold), Винсент Ромеро (Vicente Romero), Джон Роуз (John Rose), Джорджес Сааб (Georges Saab), Стив Сайдс (Steve Sides), Бернард Траверсат (Bernard Traversat) и Майкл Трюдо (Michel Trudeau).

Возможно, наибольшую благодарность следует выразить программистам, которые превратили спецификации в реальное программное обеспечение. Маурицио Чимадаморе (Maurizio Cimadamore) из Oracle героически работал с первых дней проекта лямбда-выражений и их реализации в `javac`. Над поддержкой возможностей Java SE 8 работали Джаяпракаш Артанарисваран (Jayaprakash Arthanareeswaran), Шанха Банержи (Shankha Banerjee), Анирбан Чакраборти (Anirban Chakraborty), Эндрю Клемент (Andrew Clement), Стефан Херрманн (Stephan Herrmann), Маркус Келлер (Markus Keller), Джеспер Мёллер (Jesper Møller), Маной Палат (Manoj Palat), Срикант Санкаран (Srikanth Sankaran) и Оливер Томанн (Olivier Thomann) из Eclipse; Анна Козлова (Anna Kozlova), Алексей Кудрявцев (Alexey Kudravtsev) и Роман Шевченко (Roman Shevchenko) из IntelliJ. Они заслуживают благодарность от всего сообщества Java.

Java SE 8 является возрождением языка программирования Java. Хотя некоторые ищут "очередной большой язык", мы считаем, что программирование на Java является более интересным и продуктивным, чем когда-либо. Мы надеемся, что этот язык программирования и далее будет соответствовать вашим потребностям и запросам.

Алекс Бакли (Alex Buckley)
Санта-Клара, Калифорния
Март 2014

ГЛАВА 1

Введение

JAVA является языком программирования общего назначения, ориентированным на параллельное выполнение и основанным на классах объектно-ориентированным языком. Он специально разрабатывался так, чтобы быть достаточно простым, так что многие программисты могут легко достичь высокой скорости работы. Хотя язык программирования Java связан с C и C++, он существенно отличается от них; ряд аспектов C и C++ в Java отсутствует, а из других языков включено несколько идей, не имеющих аналогов в C и C++. Java — это, в первую очередь, производственный, а не исследовательский язык программирования, а потому, как указывал в своей классической работе по дизайну языка программирования Java Ч.Э.Р. Хоар (C.A.R. Hoare), при его разработке создатели языка старательно избегали включения новых и неопробованных возможностей.

Язык программирования Java является строго и статически типизированным. В данной спецификации четко различаются ошибки *времени компиляции*, которые могут и должны быть обнаружены во время компиляции, и ошибки, которые происходят во время выполнения. Процесс компиляции обычно состоит из перевода программы в независимый от машины байт-код. Процесс выполнения программы включает загрузку и компоновку классов, необходимых для выполнения программы, необязательные генерацию машинного кода и динамическую оптимизацию программы, а также выполнение подготовленной программы.

Язык программирования Java — язык относительно высокого уровня, что проявляется, в частности, в том, что детали представления машинного кода в языке недоступны. Сюда входят автоматическое управление памятью, обычно с использованием сборщика мусора (чтобы избежать проблем, связанных с явным освобождением памяти, как при вызовах `free` в C или `delete` в C++). Высокопроизводительные реализации сборки мусора могут ограничиваться работой в паузах выполнения программы, чтобы обеспечить возможность создания системных программ и приложений реального времени. Язык не включает ни одной небезопасной конструкции, такой как доступ к массиву без проверки диапазонов индексирования, поскольку такие небезопасные конструкции могут приводить к неопределенному поведению программы.

Программа на языке Java обычно компилируется в набор команд байт-кода и бинарный формат, определенный в спецификации виртуальной машины Java *The Java Virtual Machine Specification, Java SE 8 Edition.*

§1.1. Организация книги

В главе 2, “Грамматика”, описываются грамматика и обозначения, используемые для представления лексических и синтаксических элементов языка.

В главе 3, “Лексическая структура”, описывается лексическая структура языка программирования Java, основанного на языках программирования C и C++. Язык использует кодировку Unicode и поддерживает соответствующие символы в системах, которые работают только с ASCII.

В главе 4, “Типы, значения и переменные”, описываются типы, значения и переменные. Типы подразделяются на примитивные и ссылочные.

Примитивные типы определены одинаково для всех компьютеров и во всех реализациях и представляют собой целые числа различных размеров в формате дополнения к 2, числа с плавающей запятой одинарной и двойной точности в стандарте IEEE 754, тип `boolean` и тип `char` для хранения символов Unicode.

Ссылочными типами являются классы, интерфейсы и массивы. Ссылочные типы реализуются динамически создаваемыми объектами, которые являются экземплярами классов или массивов. На каждый объект может существовать несколько ссылок. Все объекты (включая массивы) поддерживают методы класса `Object`, который представляет собой (единственный) корень иерархии классов. Предопределенный класс `String` поддерживает строки символов Unicode. Классы существуют для инкапсуляции примитивных значений в объектах. Во многих случаях инкапсуляция и извлечение выполняются компилятором автоматически (в этом случае данные процессы называются упаковкой (boxing) и распаковкой (unboxing)). Объявления класса и интерфейса могут быть обобщенными, т.е. могут быть параметризированы другими ссылочными типами. Такие объявления могут затем использоваться с конкретными типами.

Переменные являются типизированными ячейками памяти. Переменная примитивного типа содержит значение именно данного примитивного типа. Переменная типа класса может содержать пустую ссылку или ссылку на объект, тип которого является данным классом или любым подклассом этого класса. Переменная интерфейсного типа может содержать пустую ссылку или ссылку на экземпляр любого класса, реализующего данный интерфейс. Переменная типа массива может содержать пустую ссылку или ссылку на массив. Переменная типа класса `Object` может содержать пустую ссылку или ссылку на любой объект, как на экземпляр класса, так и на массив.

В главе 5, “Преобразования и контексты”, описаны преобразования и приведения к общему типу (numeric promotion — числовое повышение). Преобразования изменяют тип времени компиляции, а иногда и значение выражения. Сюда включаются преобразования упаковки и распаковки между примитивными и ссылочными типами. Приведение к общему типу используется для преобразования операндов числового оператора в общий тип, над которым может быть выполнена данная операция. Лазеек в языке нет; приведения к ссылочным типам проверяются во время выполнения, чтобы обеспечить полную безопасность типов.

В главе 6, “Имена”, описываются объявления, имена и способы определения того, что обозначает то или иное имя. Язык не требует объявления типов или их членов до исполь-

зования. Порядок объявлений важен только для локальных переменных, локальных классов и порядка инициализаторов полей в классе или интерфейсе.

Язык программирования Java обеспечивает контроль над областью видимости имен и поддерживает ограничения на внешний доступ к членам пакетов, классов и интерфейсов. Это помогает при написании больших программ, позволяя отделять реализации типов от их пользователей и от расширяющих эти типы программистов. Здесь также описаны рекомендуемые соглашения по именованию, которые делают программы более удобочитаемыми.

В главе 7, "Пакеты", описывается структура программы, которая организована в пакеты, подобные модулям языка программирования Modula. Членами пакета являются классы, интерфейсы и подпакеты. Пакеты делятся на единицы компиляции. Единицы компиляции содержат объявления типов и могут импортировать типы из других пакетов, давая им при этом короткие имена. Имена пакетов находятся в иерархическом пространстве имен, а для образования уникальных имен пакетов может использоваться система доменных имен Интернета.

В главе 8, "Классы", описываются классы. Членами классов являются классы, интерфейсы, поля (переменные) и методы. Переменные класса имеются по одной для всего класса. Методы класса работают без ссылки на конкретный объект. Переменные экземпляров создаются динамически в объектах, являющихся экземплярами классов. Методы экземпляра вызываются для экземпляров классов; такие экземпляры становятся текущим объектом `this` на время работы метода, поддерживая объектно-ориентированный стиль программирования.

Классы поддерживают единичное наследование, при котором реализация каждого класса является производной от единственного суперкласса, а в конечном счете — от класса `Object`. Переменные типа класса могут ссылаться на экземпляр данного класса или любого подкласса этого класса, позволяя полиморфно использовать новые типы с существующими методами.

Классы поддерживают параллельное программирование с помощью методов, объявленных как `synchronized`. Методы объявляют, какие исключения могут возникнуть в процессе их выполнения, что позволяет во время компиляции убедиться в том, что все исключительные ситуации будут обработаны. Объекты могут объявлять метод `finalize`, который будет вызываться перед тем, как объект будет удален сборщиком мусора, что обеспечивает возможность объекту выполнить все необходимые действия по "уборке за собой".

Для простоты язык не имеет отдельных от реализации "заголовков", а также не разделяет иерархии типов и классов.

Особая форма классов — перечисления — поддерживает определение небольших наборов значений и манипуляций ими безопасным с точки зрения типов образом. В отличие от перечислений в других языках программирования, перечисления Java являются объектами и могут иметь собственные методы.

В главе 9, "Интерфейсы", описаны интерфейсные типы, которые объявляют набор абстрактных методов, типов членов и констант. Классы, не связанные друг с другом ничем иным, могут реализовывать один и тот же интерфейс. Переменная, имеющая тип интер-

фейса, может содержать ссылку на любой объект, реализующий этот интерфейс. В языке поддерживается множественное наследование интерфейсов.

Типы аннотаций представляют собой специализированные интерфейсы, используемые для аннотированных объявлений. Такие аннотации никоим образом не влияют на семантику программы на языке программирования Java, но предоставляют различные полезные возможности для стороннего инструментария.

В главе 10, "Массивы", описаны массивы. Доступ к массиву включает в себя проверку диапазона. Массивы представляют собой динамически создаваемые объекты и могут быть присвоены переменным типа `Object`. Вместо многомерных массивов язык поддерживает массивы массивов.

В главе 11, "Исключения", описываются исключения, которые являются неотделимой, полностью интегрированной с семантикой и механизмами параллелизма возможностью языка. Существует три вида исключений: проверяемые исключения, исключения времени выполнения (run-time) и ошибки (error). Компилятор гарантирует, что проверяемые исключения будут обработаны надлежащим образом, требуя, чтобы метод или конструктор мог привести к генерации проверяемого исключения, только если объявили его. Таким образом, во время компиляции обеспечивается проверка существования обработчиков исключений, что является большим подспорьем для программиста. Большинство определяемых пользователем исключений должны быть проверяемыми. Некорректные операции в программе, обнаруживаемые виртуальной машиной Java, приводят к исключениям времени выполнения, например `NullPointerException`. Ошибки являются результатом обнаруженных виртуальной машиной Java сбоев, например исключение `OutOfMemoryError`. Большинство простых программ не пытаются каким-то образом обрабатывать ошибки.

В главе 12, "Выполнение", описывается, как происходит выполнение программы. Программа обычно хранится в виде бинарных файлов, которые представляют собой скомпилированные классы и интерфейсы. Эти бинарные файлы могут быть загружены в виртуальную машину Java, скомпонованы с другими классами и интерфейсами и инициализированы.

После инициализации могут использоваться методы и переменные классов. Для создания новых объектов классов могут инстанцироваться некоторые иные классы (т.е. создаваться экземпляры этих классов). Объекты, являющиеся экземплярами класса, содержат также экземпляры каждого суперкласса этого класса, так что создание объекта включает рекурсивное создание экземпляров суперклассов.

Когда на объект больше не имеется ссылок, он может быть удален сборщиком мусора. Если у объекта объявлен метод-финализатор, он выполняется непосредственно перед утилизацией объекта, чтобы дать объекту последний шанс освободить захваченные им ресурсы. Когда класс больше не нужен, он может быть выгружен из памяти.

В главе 13, "Бинарная совместимость", описана бинарная совместимость, с указанием влияния изменений некоторых типов на другие типы, которые используют измененные, не будучи перекомпилированными. Этот материал представляет интерес для разработчиков типов, которые будут широко распространяться в виде целой серии версий, зачастую через Интернет. Хорошая среда разработки программ автоматически перекомпилирует

код, зависящий от измененного типа, так что большинству программистов не приходится сталкиваться с этими деталями.

В главе 14, "Блоки и инструкции", описываются блоки и конструкции, которые основаны на языках программирования C и C++. Язык Java не имеет оператора `goto`, но включает операторы `break` и `continue` с метками. В отличие от языка C, язык программирования Java требует в операторах управления потоком выполнения выражения типа `boolean` (или `Boolean`) и не выполняет неявного приведения типов к `boolean` (за исключением распаковки) в надежде отловить большее количество ошибок во время компиляции. Оператор `synchronized` обеспечивает базовую блокировку монитора на уровне объектов. Инструкция `try` может включать инструкции `catch` и `finally` для защиты от нелокальной передачи управления.

Глава 15, "Выражения", посвящена выражениям. В документации полностью определен порядок вычисления выражений, что увеличивает детерминизм и переносимость Java. Компилятор выполняет разрешение перегруженных методов и конструкторов, выбирая среди всех применимых в рассматриваемой ситуации метод или конструктор, определенный наиболее точно.

В главе 16, "Определенное присваивание", описан точный метод, с помощью которого язык программирования Java гарантирует корректность установки значений локальных переменных до их использования. В то время как все прочие переменные автоматически инициализируются значением по умолчанию, локальные переменные в Java автоматически не инициализируются во избежание маскировки ошибок программирования.

Глава 17, "Потоки и блокировки", посвящена семантике потоков и блокировок, основанной на параллельности на основе мониторов (впервые такая параллельность появилась в языке программирования Mesa). Язык программирования Java определяет высокопроизводительную модель многопроцессорной системы с общей памятью.

Глава 18, "Вывод типов", описывает различные алгоритмы вывода типов, используемые для проверки применимости обобщенных методов и для выводов типов в вызовах обобщенных методов.

Глава 19, "Синтаксис", содержит формальное описание грамматики языка программирования Java.

§1.2. Примеры программ

Большинство приведенных в книге примеров программ готовы к выполнению и имеют вид, подобный приведенной ниже программе.

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
        System.out.println();
    }
}
```

На машине с установленным Oracle JDK этот класс, сохраненный в файле `Test.java`, может быть скомпилирован и выполнен следующими командами.

```
javac Test.java
java Test Hello, world.
```

Это приведет к выводу на экран строки

```
Hello, world.
```

§1.3. Обозначения

Везде в книге мы ссылаемся на классы и интерфейсы, взятые из Java SE API. Всякий раз, когда мы будем ссылаться на класс или интерфейс (отличный от объявленных в примере) с использованием только лишь идентификатора *N*, подразумевается ссылка на класс или интерфейс с именем *N* из пакета `java.lang`. Для классов или интерфейсов из пакетов, отличных от `java.lang`, мы используем каноническое имя (§6.7). Ненормативная информация, приводимая для пояснений спецификации, приводится уменьшенным шрифтом с небольшим отступом.

> Это — ненормативная информация. К вашим услугам интуиция, обоснования, советы, примеры и т.п.

Система типов языка программирования Java иногда опирается на понятие подстановки (*substitution*). Запись $[F_1 := T_1, \ldots, F_n := T_n]$ означает подстановку T_i вместо F_i для $1 \le i \le n$.

§1.4. Связь с предопределенными классами и интерфейсами

Как отмечалось выше, в книге мы часто ссылаемся на классы Java SE API. В частности, некоторые классы имеют особое положение в языке программирования Java. Примеры подобных классов включают такие классы, как `Object`, `Class`, `ClassLoader`, `String`, `Thread`, а также, среди прочих, классы и интерфейсы из пакета `java.lang.reflect`. В книге нет полной спецификации таких классов и интерфейсов, так что читателю рекомендуется обратиться к документации по Java SE API.

Поэтому читатель не найдет здесь сколь-нибудь детального описания рефлексии. Многие языковые конструкции имеют аналоги в соответствующих API — Core Reflection API (`java.lang.reflect`) и Language Model API (`javax.lang.model`), но они, как правило, в данной книге не обсуждаются. Например, перечисляя способы создания объектов, мы обычно не включаем способы выполнения задания с помощью API рефлексии. Читатель должен ознакомиться с этими дополнительными механизмами самостоятельно ввиду отсутствия их упоминания в тексте книги.

§1.5. Литература

Apple Computer. *Dylan Reference Manual*. Apple Computer Inc., Cupertino, California. September 29, 1995.

Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990, ISBN 1-55558-041-6, 770–864.

Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990, reprinted with corrections October 1992, ISBN 0-201-51459-1.

Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989, ISBN 0-201-13688-0.

Harbison, Samuel. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-596396.

Hoare, C. A. R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, December 1973. Reprinted in SIGACT/SIGPLAN Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, October 1973.

IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8. (Брайан У. Керниган, Деннис М. Ритчи, Язык программирования С, 2-е изд., пер. с англ., ISBN 978-5-8459-1975-5, ИД "Вильямс", 2015 г.)

Madsen, Ole Lehrmann, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993, ISBN 0-201-62430-3.

Mitchell, James G., William Maybury, and Richard Sweet. *The Mesa Programming Language, Version 5.0*. Xerox PARC, Palo Alto, California, CSL 79-3, April 1979.

Stroustrup, Bjarne. *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

Unicode Consortium, The. *The Unicode Standard, Version 6.0.0*. Mountain View, CA, 2011, ISBN 978-1-936213-01-6.

ГЛАВА 2

Грамматика

В этой главе описана контекстно-свободная грамматика, использованная для определения лексической и синтаксической структуры программы.

§2.1. Контекстно-свободные грамматики

Контекстно-свободная грамматика (бесконтекстная грамматика) состоит из ряда *продукций*. В левой части каждой продукции содержится абстрактный символ, именуемый *нетерминалом*, а в правой — последовательность из одного или более нетерминальных и *терминальных* символов. В каждой грамматике терминальные символы выбираются из определенного *алфавита*.

Заданная контекстно-свободная грамматика, начиная с выражения, состоящего из единственного выделенного нетерминала, именуемого *целевым символом* (goal symbol), определяет язык, т.е. множество возможных последовательностей терминальных символов, которые могут получиться в результате многократных замен встречающихся нетерминалов последовательностями из правой части продукций, в которых этот нетерминал находится в левой части.

§2.2. Лексика

Лексика языка программирования Java рассматривается в главе 3, “Лексическая структура”. В качестве терминальных символов *лексической грамматики* Java выступают символы из набора символов Unicode. Она определяет множество продукций, начиная с целевого символа *Input* (§3.5), которые описывают, как последовательности символов Unicode (§3.1) преобразуются в последовательности входных элементов (§3.5).

Эти входные элементы, с отброшенными пробельными символами (§3.6) и комментариями (§3.7), образуют терминальные символы для синтаксической грамматики языка программирования Java и называются *токенами* (§3.5). Токены представляют собой идентификаторы (§3.8), ключевые слова (§3.9), литералы (§3.10), разделители (§3.11) и операторы (§3.12) языка программирования Java.

§2.3. Синтаксис

Синтаксическая грамматика языка программирования Java описана в главах 4, 6–10, 14 и 15. В качестве терминальных символов данная грамматика использует токены, определенные лексической грамматикой. Она определяет множество продукций, начиная с целевого символа *CompilationUnit* (§7.3), которые описывают, как последовательности токенов могут образовывать синтаксически правильные программы.

В главе 19 также приведена синтаксическая грамматика языка программирования Java, однако она в большей степени подходит для реализации языка, чем для чтения человеком и понимания. Обе синтаксические грамматики определяют один и тот же язык программирования.

§2.4. Обозначения грамматики

В продукциях лексической и синтаксической грамматик терминальные символы показаны `моноширинным шрифтом`, и он используется на протяжении этой книги всякий раз, когда текст непосредственно ссылается на терминальный символ. В программе терминальные символы присутствуют в точности в том виде, в котором они показаны в книге.

Нетерминалы выделяются *курсивом*. Определение нетерминала начинается с его имени с двоеточием в левой части определения. За двоеточием располагается одна или несколько альтернативных правых частей на отдельных строках.

> Например, синтаксическое определение
>
> *IfThenStatement*:
>
> `if (` *Expression* `)` *Statement*
>
> гласит, что нетерминал *IfThenStatement* представляет собой токен `if`, за которым следует токен “левая круглая скобка”, за которым следует *Expression*, затем — токен “правая круглая скобка”, после чего идет *Statement*.

Синтаксис *{x}* в правой части продукции обозначает наличие нуля или нескольких *x*.

> Например, продукция
>
> *ArgumentList*:
>
> *Argument* `{,` *Argument*`}`
>
> гласит, что *ArgumentList* может представлять собой *Argument*, за которым следует нуль или несколько запятых с последующим *Argument*. В результате получается, что *ArgumentList* может содержать любое положительное количество *Argument*.

Синтаксис *[x]* в правой части продукции обозначает наличие нуля или одного *x*, т.е. *x* представляет собой *необязательный символ*. Альтернатива, которая содержит необязательный символ, фактически определяет две альтернативы: одну, которая опускает необязательный символ, и вторую, которая включает его.

Это означает, что

BreakStatement:
    `break` *[Identifier]* `;`

является удобным сокращением для

BreakStatement:
    `break ;`
    `break` *Identifier* `;`

В качестве еще одного примера это означает, что

BasicForStatement:
    `for (` *[ForInit]* `;` *[Expression]* `;` *[ForUpdate]* `)` *Statement*

является удобным сокращением для

BasicForStatement:
    `for (` `;` *[Expression]* `;` *[ForUpdate]* `)` *Statement*
    `for (` *ForInit* `;` *[Expression]* `;` *[ForUpdate]* `)` *Statement*

что, в свою очередь, является сокращением для

BasicForStatement:
    `for (` `;` `;` *[ForUpdate]* `)` *Statement*
    `for (` `;` *Expression* `;` *[ForUpdate]* `)` *Statement*
    `for (` *ForInit* `;` `;` *[ForUpdate]* `)` *Statement*
    `for (` *ForInit* `;` *Expression* `;` *[ForUpdate]* `)` *Statement*

что, опять же, является сокращением для

BasicForStatement:
    `for (` `;` `;` `)` *Statement*
    `for (` `;` `;` *ForUpdate* `)` *Statement*
    `for (` `;` *Expression* `;` `)` *Statement*
    `for (` `;` *Expression* `;` *ForUpdate* `)` *Statement*
    `for (` *ForInit* `;` `;` `)` *Statement*
    `for (` *ForInit* `;` `;` *ForUpdate* `)` *Statement*
    `for (` *ForInit* `;` *Expression* `;` `)` *Statement*
    `for (` *ForInit* `;` *Expression* `;` *ForUpdate* `)` *Statement*

Так что, как видите, нетерминал *BasicForStatement* в правой части продукции на самом деле имеет восемь альтернатив.

Очень длинная правая часть может быть продолжена на второй строке, при этом на этой второй строке имеется существенный отступ.

Например, синтаксическая грамматика содержит следующую продукцию, определяющую одну правую часть для нетерминала *NormalClassDeclaration*.

NormalClassDeclaration:
    {ClassModifier} `class` *Identifier [TypeParameters]*
        [Superclass] [Superinterfaces] ClassBody

Когда в определении грамматики за двоеточием следуют слова “одно из”, они означают, что каждый из терминальных символов на следующей строке или строках является альтернативным определением.

> Например, лексическая грамматика содержит продукцию
>
> *ZeroToThree*: одно из
>
> ```
> 0 1 2 3
> ```
>
> которая просто представляет собой удобное сокращение для
>
> *ZeroToThree*:
>
> ```
> 0
> 1
> 2
> 3
> ```

Когда альтернатива в лексической продукции является токеном, последний представлен последовательностью символов, составляющих этот токен.

> Таким образом, определение
>
> *BooleanLiteral*: одно из
>
> ```
> true false
> ```
>
> в продукции лексической грамматики представляет собой сокращение для
>
> *BooleanLiteral*:
>
> ```
> t r u e
> f a l s e
> ```

В правой части лексической продукции может быть указано, что определенные раскрытия сокращений не разрешены. Это указывается с помощью фразы “но не”, за которой идут запрещенные варианты раскрытия.

> Например, такую фразу можно встретить в продукциях для *InputCharacter* (§3.4) и *Identifier* (§3.8):
>
> *InputCharacter*:
>
> *UnicodeInputCharacter* но не CR или LF
>
> *Identifier*:
>
> *IdentifierName* но не *Keyword* или *BooleanLiteral* или *NullLiteral*

Наконец, несколько нетерминальных символов описаны обычной фразой — в тех случаях, когда было бы нецелесообразно перечислять все возможные альтернативы.

> Например:
>
> *RawInputCharacter*:
>
> любой символ Unicode

ГЛАВА 3

Лексическая структура

В этой главе рассматривается лексическая структура языка программирования Java.

Программы пишутся с применением набора символов Unicode (§3.1), но при этом предоставляется возможность лексической трансляции (§3.2), так что можно использовать только символы ASCII, а для включения символов Unicode (§3.3) воспользоваться специальными управляющими последовательностями. Определены также ограничители строк (§3.4) для поддержки различных соглашений существующих базисных систем для обеспечения последовательной нумерации строк.

Символы Unicode, получающиеся в результате лексической трансляции, приводятся к последовательности входных элементов (§3.5), которые представляют собой пробельные символы (§3.6), комментарии (§3.7) и токены. Токены представляют собой идентификаторы (§3.8), ключевые слова (§3.9), литералы (§3.10), разделители (§3.11) и операторы (§3.12) синтаксической грамматики.

§3.1. Unicode

Программы пишутся с помощью набора символов Unicode. Информация об этом наборе символов и связанных с ним кодировках имеется по адресу `http://www.unicode.org/`.

Платформа Java SE отслеживает изменения в спецификации Unicode. Точная версия Unicode, используемая текущим выпуском языка, указана в документации класса `Character`.

> Версии языка программирования Java до 1.1 используют Unicode версии 1.1.5. Обновление на новые версии стандарта Unicode выполнено в JDK 1.1 (для Unicode 2.0), JDK 1.1.7 (для Unicode 2.1), Java SE 1.4 (для Unicode 3.0) и Java SE 5.0 (для Unicode 4.0).

Стандарт Unicode изначально был разработан как кодировка символов фиксированной 16-битовой ширины. С тех пор он был изменен, чтобы разрешить использовать символы, представление которых требует более 16 бит. Диапазон корректных кодов в настоящее время простирается от U+0000 до U+10FFFF (с использованием шестнадцатеричной записи U+*n*). Символы, коды которых превышают U+FFFF, называются дополнительными (supplementary). Для представления полного диапазона символов с использованием только 16-битовых значений стандарт Unicode определяет кодировку, которая называется

UTF-16. В этой кодировке дополнительные символы представлены в виде пары 16-битовых кодовых слов, первое — из диапазона от U+D800 до U+DBFF, второе — из диапазона от U+DC00 до U+DFFF. Для символов из диапазона от U+0000 до U+FFFF значения кодов совпадают с кодовыми словами UTF-16.

Язык программирования Java представляет текст в виде последовательности 16-битовых кодовых слов кодировки UTF-16.

> Некоторые API-платформы Java SE, главным образом в классе `Character`, используют для представления кодов символов как отдельных объектов 32-битовые целые числа. Платформа Java SE предоставляет методы для преобразования между 16- и 32-битовыми представлениями символов.

В данной книге термины *код символа* и *кодовое слово UTF-16* используются только там, где существенную роль играет способ представления символов. Там, где способ представления не имеет значения для обсуждаемого вопроса, применяется обобщенный термин *символ*.

За исключением комментариев (§3.7), идентификаторов и содержимого символьных и строковых литералов (§3.10.4, §3.10.5), все входные элементы (§3.5) программы формируются только из символов ASCII (или управляющих последовательностей (§3.3) с применением символов ASCII).

> ASCII (ANSI X3.4) представляет собой аббревиатуру от American Standard Code for Information Interchange (американский стандартный код обмена информацией). Первые 128 символов кодировки Unicode UTF-16 представляют собой символы ASCII.

§3.2. Лексическая трансляция

Входной поток символов Unicode преобразуется в последовательность токенов с помощью следующих трех последовательно применяемых этапов лексической трансляции.

1. Трансляция управляющих последовательностей (§3.3) в потоке в соответствующий символ Unicode. Управляющие последовательности имеют вид `\uxxxx`, где `xxxx` — шестнадцатеричное значение, представляющее кодовое слово UTF-16. Этот этап трансляции позволяет записать любую программу с применением одних лишь символов ASCII.
2. Трансляция получившегося на первом этапе потока символов Unicode в поток входных символов и ограничителей строк (§3.4).
3. Трансляция потока входных символов и ограничителей строк, получающихся на втором этапе, в последовательность входных элементов (§3.5), которая, после удаления пробельных символов (§3.6) и комментариев (§3.7), состоит из токенов (§3.5), являющихся терминальными символами синтаксической грамматики (§2.3).

На каждом этапе используется наиболее длинная возможная трансляция, даже если она приводит к некорректной программе, в то время как другая трансляция могла бы дать корректную с точки зрения синтаксиса программу.

Таким образом, входные символы `a--b` токенизируются (§3.5) как `a`, `--`, `b`, которые не могут быть частью никакой грамматически правильной программы, хотя токенизация `a`, `-`, `-`, `b` вполне может быть частью грамматически корректной программы.

Без правила для символов > две последовательные угловые скобки > в типе наподобие `List<List<String>>` токенизируются как оператор правого знакового сдвига >>, в то время как три последовательные угловые скобки > в типе наподобие `List<List<List<String>>>` токенизируются как оператор правого беззнакового сдвига >>>. Еще хуже ситуация с четырьмя последовательными угловыми скобками > в типе наподобие `List<List<List<List<String>>>>`, где наблюдается неоднозначность в том, какая именно комбинация токенов >, >> и >>> представлена последовательностью символов >>>>.

§3.3. Управляющие последовательности Unicode

Компилятор языка программирования Java ("компилятор Java") сначала распознает во входном потоке управляющие последовательности Unicode, транслируя ASCII-символы `\u` с последующими четырьмя шестнадцатеричными цифрами в кодовое слово UTF-16 (§3.1) с указанным шестнадцатеричным значением, оставляя все остальные символы без изменений. Представление дополнительных символов требует двух последовательных управляющих последовательностей. Этот этап трансляции дает на выходе последовательность символов Unicode.

UnicodeInputCharacter:
    UnicodeEscape
    RawInputCharacter

UnicodeEscape:
    \ *UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

UnicodeMarker:
    u *{u}*

HexDigit: одно из
    `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

RawInputCharacter:
    любой символ Unicode

Здесь символы \, `u` и шестнадцатеричные цифры являются символами ASCII.

Кроме обработки, обусловленной грамматикой, для каждого необработанного входного символа обратной косой черты \ следует учитывать количество непосредственно предшествующих ему других символов \. Если это четное количество, то рассматриваемый символ \ может начинать управляющую последовательность; если же это нечетное количество, то такой символ \ не может быть началом управляющей последовательности.

> Например, необработанный вход `"\\u2126=\u2126"` приводит к одиннадцати символам `" \ \ u 2 1 2 6 = Ω "` (`\u2126` представляет собой кодировку Unicode для символа Ω).

Если за символом `\`, могущим быть началом управляющей последовательности, не следует символ `u`, то он рассматривается как *RawInputCharacter* и остается частью входного потока.

Если же за таким символом `\` следует `u` (или более одного `u`) и за последним из них не следуют четыре шестнадцатеричные цифры, то возникает ошибка времени компиляции.

Символ, сгенерированный управляющей последовательностью, в других управляющих последовательностях не участвует.

> Например, необработанный вход `\u005cu005a` приводит к шести символам `\ u 0 0 5 a`, потому что `005c` является значением Unicode для `\`. Мы не получим символ `Z`, значением Unicode которого является `005a`, потому что символ `\` получен как результат обработки управляющей последовательности `\u005c`, а потому не интерпретируется как начало еще одной управляющей последовательности.

Язык программирования Java определяет стандартный способ преобразования в ASCII написанной на Unicode программы. Этот способ приводит программу к виду, который может быть обработан инструментами, работающими с ASCII. Преобразование включает в себя замену всех управляющих последовательностей в исходном тексте программы путем добавления дополнительного символа `u`. Так, `\uxxxx` превращается в `\uuxxxx`, в то время как одновременно символы исходного текста, не являющиеся ASCII-символами, преобразуются в управляющие последовательности с одним символом `u`.

Такая преобразованная версия вполне приемлема для Java-компилятора и представляет собой точно ту же программу, что и исходная. Точный исходный текст в кодировке Unicode можно впоследствии восстановить из ASCII-версии путем преобразования каждой управляющей последовательности, в которой имеется несколько символов `u`, в управляющую последовательность с одним символом `u`, а управляющие последовательности с одним символом `u` в соответствующий символ Unicode.

> Компилятор Java должен использовать запись `\uxxxx` в качестве выходного формата для отображения символов Unicode, если подходящий шрифт недоступен.

§3.4. Ограничители строк

После трансляции управляющих последовательностей компилятор Java делит входную последовательность символов Unicode на строки, распознавая в потоке *ограничители строк* (line terminator).

LineTerminator:
ASCII-символ LF, известный как “новая строка”
ASCII-символ CR, известный как “перевод каретки”
ASCII-символ CR, за которым идет символ LF

InputCharacter:
UnicodeInputCharacter но не CR или LF

Строки завершаются ASCII-символами CR или LF или последовательностью CR LF. Два символа CR, идущие сразу же после символа LF, рассматриваются как признак конца одной строки, а не двух.

Конец строки указывает также завершение комментария вида // (§3.7).

> Строки, указываемые ограничителями строк, могут определять номера строк, генерируемые компилятором Java.

Результатом этого этапа лексической трансляции является последовательность ограничителей строк и входных символов, которые являются терминальными символами для третьего этапа процесса токенизации.

§3.5. Входные элементы и токены

Входные символы и ограничители строк, получающиеся в результате обработки управляющих последовательностей (§3.3) и распознавания строк (§3.4), преобразуются в последовательность *входных элементов*.

Input:
    {InputElement} [Sub]

InputElement:
    WhiteSpace
    Comment
    Token

Token:
    Identifier
    Keyword
    Literal
    Separator
    Operator

Sub:
    ASCII-символ SUB, известный также как “Ctrl-Z”

Входные элементы, не являющиеся пробельными символами (§3.6) или комментариями (§3.7), являются токенами. Токены представляют собой терминальные символы синтаксической грамматики (§2.3).

Пробельные символы (§3.6) и комментарии (§3.7) могут служить для разделения токенов, которые, находясь один непосредственно за другим, могут быть токенизированы другим способом. Например, во входном потоке символы ASCII - и = при отсутствии между ними пробельных символов или комментариев формируют токен оператора -= (§3.12).

В качестве уступки для совместимости с некоторыми операционными системами ASCII-символ SUB (`\u001a`, или Ctrl-Z) игнорируется, если это последний символ во входном потоке.

Рассмотрим два токена, `x` и `y`, в получающемся в результате входном потоке. Если `x` предшествует `y`, то мы говорим, что `x` находится *слева* от `y` и что `y` находится *справа* от `x`.

> Например, в простом фрагменте кода
>
> ```
> class Empty {
> }
> ```
>
> мы говорим, что токен `}` находится справа от токена `{`, несмотря на то что в двумерном представлении на странице он находится внизу слева от токена `{`. Соглашение о терминах "слева" и "справа" позволяет нам говорить, например, о правом операнде бинарного оператора или левой части присваивания.

§3.6. Пробельные символы

Пробельные символы определяются как ASCII-символы пробела, горизонтальной табуляции, подачи страницы и конца строки (§3.4).

WhiteSpace:
ASCII-символ SP, известный также как "пробел"
ASCII-символ HT, известный также как "горизонтальная табуляция"
ASCII-символ FF, известный также как "прогон страницы"
LineTerminator

§3.7. Комментарии

Имеются два вида комментариев.

- `/* текст */`
- *Традиционный комментарий*: все символы, представляющие собой текст, заключенный между ASCII-символами `/*` и `*/`, компилятором игнорируются (как в C и C++).
- `// текст`
- *Комментарий в конце строки*: все символы, представляющие собой текст, заключенный между ASCII-символами `//` и концом строки, компилятором игнорируются (как в C++).

Comment:
TraditionalComment
EndOfLineComment

TraditionalComment:
/ * *CommentTail*

CommentTail:
* *CommentTailStar*
NotStar CommentTail

CommentTailStar:
    /
    * *CommentTailStar*
    NotStarNotSlash CommentTail

NotStar:
    InputCharacter но не *
    LineTerminator

    NotStarNotSlash:
        InputCharacter но не * *или* /
        LineTerminator

    EndOfLineComment:
        / / *{InputCharacter}*

Из приведенных продукций вытекают следующие свойства.

- Комментарии не бывают вложенными.
- /* и */ не несут специального смысла в комментариях, начинающихся с //.
- // не несет специального смысла в комментариях, начинающихся с /* или /**.

> Таким образом, текст
> ```
> /* Этот комментарий /* // /** заканчивается здесь: */
> ```
> представляет собой один завершенный комментарий.

Из лексической грамматики вытекает, что комментарии не могут встречаться в пределах символьных (§3.10.4) или строковых (§3.10.5) литералов.

§3.8. Идентификаторы

Идентификатор представляет собой последовательность *букв и цифр Java* неограниченной длины, первой из которых должна быть *буква Java*.

Identifier:
    IdentifierChars но не *Keyword* или *BooleanLiteral* или *NullLiteral*

    IdentifierChars:
        JavaLetter {JavaLetterOrDigit}

    JavaLetter:
        любой символ Unicode, являющийся буквой Java (см. ниже)

    JavaLetterOrDigit:
        любой символ Unicode, являющийся буквой или цифрой Java (см. ниже)

"Буква Java" — это символ, для которого метод `Character.isJavaIdentifierStart(int)` возвращает значение `true`.

"Буква или цифра Java" — это символ, для которого метод `Character.isJavaIdentifierPart(int)` возвращает значение `true`.

> "Буквы Java" включают прописные и строчные латинские буквы ASCII `A–Z` (`\u0041–\u005a`), `a–z` (`\u0061–\u007a`) и, по историческим причинам, ASCII-символ подчеркивания (`_` или `\u005f`) и знак доллара (`$` или `\u0024`). Символ доллара должен использоваться только в сгенерированном автоматически исходном тексте или, реже, для доступа к ранее существовавшим именам в устаревших системах.
>
> "Цифры Java" включают ASCII-цифры `0–9` (`\u0030–\u0039`).

Буквы и цифры могут выбираться из всего набора символов Unicode, который поддерживает большинство языков в мире, включая большие наборы иероглифов китайского, японского и корейского языков. Это позволяет программистам использовать в своих программах идентификаторы, написанные на своих родных языках.

Идентификатор не может совпадать (представлять собой одинаковую последовательность символов Unicode) с ключевым словом (§3.9), логическим литералом (§3.10.3) или литералом `null` (§3.10.7), иначе генерируется ошибка времени компиляции.

Два идентификатора являются одинаковыми, только если они идентичны, т.е. если во всех позициях их символы Unicode совпадают. Идентификаторы, имеющие одинаковый внешний вид, на самом деле могут быть разными.

> Например, идентификаторы, состоящие из одной прописной латинской буквы (`A`, `\u0041`), строчной латинской буквы `a` (`a`, `\u0061`), греческой прописной буквы альфа (`A`, `\u0391`), кириллической строчной буквы `a` (`a`, `\u0430`) и математической полужирной курсивной строчной `a` (*a*, `\ud835\udc82`), все разные.
>
> Составные символы Unicode отличаются от канонических эквивалентных знаков. Например, в идентификаторах латинская прописная буква `A` с ударением (`Á`, `\u00c1`) отличается от латинской прописной буквы `A` (`A`, `\u0041`), за которой следует символ ударения (`´`, `\u0301`). (См. стандарт Unicode, раздел 3.11, "Нормализованные формы".)
>
> Примеры идентификаторов приведены ниже.
>
> - `String`
> - `i3`
> - `αρετη`
> - `MAX_VALUE`
> - `isLetterOrDigit`

§3.9. Ключевые слова

50 последовательностей символов, образованных из букв ASCII, зарезервированы для использования в качестве *ключевых слов* и не могут использоваться как идентификаторы (§3.8).

Keyword: одно из

```
abstract   continue   for          new          switch
assert     default    if           package      synchronized
boolean    do         goto         private      this
break      double     implements   protected    throw
byte       else       import       public       throws
case       enum       instanceof   return       transient
catch      extends    int          short        try
char       final      interface    static       void
class      finally    long         strictfp     volatile
const      float      native       super        while
```

Ключевые слова `const` и `goto` зарезервированы, хотя в настоящее время и не используются. Появление этих ключевых слов C++ в программах Java приведет к появлению сообщения об ошибке компиляции.

Хотя слова `true` и `false` могут показаться вам ключевыми, технически они являются логическими литералами (§3.10.3). Аналогично `null` также является не ключевым словом, а литералом `null` (§3.10.7).

§3.10. Литералы

Литералом является представление в исходном тексте значения примитивного типа (§4.2), типа `String` (§4.3.3) или типа `null` (§4.1).

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

§3.10.1. Целочисленные литералы

Целочисленный литерал может быть выражением в десятичной, шестнадцатеричной, восьмеричной или двоичной системе счисления.

IntegerLiteral:
DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
BinaryIntegerLiteral

DecimalIntegerLiteral:
DecimalNumeral [IntegerTypeSuffix]

HexIntegerLiteral:
HexNumeral [IntegerTypeSuffix]

OctalIntegerLiteral:
OctalNumeral [IntegerTypeSuffix]

BinaryIntegerLiteral:
BinaryNumeral [IntegerTypeSuffix]

IntegerTypeSuffix: одно из
`l L`

Целочисленный литерал имеет тип `long`, если он завершается суффиксом в виде ASCII-буквы `L` или `l`; в противном случае он имеет тип `int` (§4.2.1).

> Суффикс `L` предпочтительнее, потому что строчную латинскую букву `l` часто трудно отличить от цифры `1`.

В качестве разделителей между цифрами, образующими целое число, допускаются подчеркивания.

В шестнадцатеричном или двоичном литерале целое число состоит из цифр, идущих после префикса `0x` или `0b` соответственно и до суффикса любого типа. Таким образом, подчеркивания не могут находиться сразу после `0x`, `0b` или последней цифры числа.

В десятичном или восьмеричном литерале целое число состоит из *всех* цифр литерала до суффикса любого типа. Таким образом, подчеркивания не могут находиться перед первой или после последней цифры числа. Подчеркивание может присутствовать после начальной цифры `0` восьмеричного числа (поскольку `0` представляет собой цифру, являющуюся частью целого числа) и после начальной ненулевой цифры в ненулевом десятичном литерале.

Десятичная запись числа представляет собой либо единственную ASCII-цифру `0`, представляющую целое число, равное нулю, либо ASCII-цифру от `1` до `9`, за которой (необязательно) следует одна или более ASCII-цифр от `0` до `9`, перемежающихся подчеркиваниями, и в этом случае запись представляет целое положительное число.

DecimalNumeral:
`0`
NonZeroDigit [Digits]
NonZeroDigit Underscores Digits

NonZeroDigit: одно из
`1 2 3 4 5 6 7 8 9`

Digits:
Digit
Digit [DigitsAndUnderscores] Digit

Digit:
`0`
NonZeroDigit

DigitsAndUnderscores:

DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:
Digit
_

Underscores:
_ {_}

Шестнадцатеричная запись числа состоит из ведущих ASCII-символов `0x` или `0X`, за которыми следуют одна или более шестнадцатеричных ASCII-цифр, перемежающихся подчеркиваниями, и которая может представлять положительное, нулевое или отрицательное целое число.

Шестнадцатеричные цифры со значениями от 10 до 15 представлены ASCII-буквами от `a` до `f` или от `A` до `F` соответственно. Каждая буква, используемая как шестнадцатеричная цифра, может быть как строчной, так и прописной.

HexNumeral:
`0x` *HexDigits*
`0X` *HexDigits*

HexDigits:
HexDigit
HexDigit [HexDigitsAndUnderscores] HexDigit

HexDigit: одно из
`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

HexDigitsAndUnderscores:
HexDigitOrUnderscore {HexDigitOrUnderscore}

HexDigitOrUnderscore:
HexDigit
_

Приведенная продукция для *HexDigit* взята из §3.3.

Восьмеричная запись числа состоит из ASCII-цифры `0`, за которыми следуют одна или более ASCII-цифр от `0` до `7`, перемежающихся подчеркиваниями, и которая может представлять положительное, нулевое или отрицательное целое число.

OctalNumeral:
`0` *OctalDigits*
`0` *Underscores OctalDigits*

OctalDigits:
OctalDigit
OctalDigit [OctalDigitsAndUnderscores] OctalDigit

OctalDigit: одно из
```
0 1 2 3 4 5 6 7
```

OctalDigitsAndUnderscores:
OctalDigitOrUnderscore {OctalDigitOrUnderscore}

OctalDigitOrUnderscore:
OctalDigit
_

> Обратите внимание, что запись восьмеричного числа всегда состоит из двух или более цифр; единственная цифра `0` всегда считается десятичной (что не так уж важно на практике, поскольку числа `0`, `00` и `0x0` представляют в точности одно и то же целочисленное значение).

Двоичная запись числа состоит из ведущих ASCII-символов `0b` или `0B`, за которыми следует одна или более ASCII-цифр `0` и `1`, перемежающихся подчеркиваниями, и которая может представлять положительное, нулевое или отрицательное целое число.

BinaryNumeral:
`0b` *BinaryDigits*
`0B` *BinaryDigits*

BinaryDigits:
BinaryDigit
BinaryDigit [BinaryDigitsAndUnderscores] BinaryDigit

BinaryDigit: одно из
```
0 1
```

BinaryDigitsAndUnderscores:
BinaryDigitOrUnderscore {BinaryDigitOrUnderscore}

BinaryDigitOrUnderscore:
BinaryDigit
_

Наибольший десятичный литерал типа `int` — `2147483648` (2^{31}).

Все десятичные литералы от `0` до `2147483647` могут находиться в любом месте, где может находиться литерал типа `int`.

Ошибка времени компиляции генерируется, если десятичный литерал типа `int` оказывается больше, чем `2147483648` (2^{31}), или если десятичный литерал `2147483648` появляется в месте, отличном от операнда оператора “унарный минус” (§15.15.4).

Наибольшими положительными шестнадцатеричным, восьмеричным и двоичным литералами типа `int`, каждый из которых представляет десятичное значение `2147483647` ($2^{31} - 1$), являются соответственно

- `0x7fff_ffff`,
- `0177_7777_7777` и
- `0b0111_1111_1111_1111_1111_1111_1111_1111`.

Наибольшими отрицательными шестнадцатеричным, восьмеричным и двоичным литералами типа `int`, каждый из которых представляет десятичное значение `-2147483648` (-2^{31}), являются соответственно

- `0x8000_0000`,
- `0200_0000_0000` и
- `0b1000_0000_0000_0000_0000_0000_0000_0000`.

Приведенные далее шестнадцатеричный, восьмеричный и двоичный литералы представляют десятичное значение `-1`:

- `0xffff_ffff`,
- `0377_7777_7777` и
- `0b1111_1111_1111_1111_1111_1111_1111_1111`.

В случае, если шестнадцатеричный, восьмеричный или двоичный литерал типа `int` не может быть размещен в 32 бит, генерируется ошибка времени компиляции.

Наибольший десятичный литерал типа `long` — `9223372036854775808L` (2^{63}).

Все десятичные литералы от `0L` до `9223372036854775807L` могут находиться в любом месте, где может находиться литерал типа `long`.

Ошибка времени компиляции генерируется, если десятичный литерал типа `long` оказывается больше, чем `9223372036854775808L` (2^{63}), или если десятичный литерал `9223372036854775808L` появляется в месте, отличном от операнда оператора "унарный минус" (§15.15.4).

Наибольшими положительными шестнадцатеричным, восьмеричным и двоичным литералами типа `long`, каждый из которых представляет десятичное значение `9223372036854775807L` ($2^{63}-1$), являются соответственно

- `0x7fff_ffff_ffff_ffffL`,
- `07_7777_7777_7777_7777_7777L` и
- `0b0111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L`.

Наибольшими отрицательными шестнадцатеричным, восьмеричным и двоичным литералами типа `long`, каждый из которых представляет десятичное значение `-9223372036854775808L` (-2^{63}), являются соответственно

- `0x8000_0000_0000_0000L`,
- `010_0000_0000_0000_0000_0000L` и
- `0b1000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000L`.

Приведенные далее шестнадцатеричный, восьмеричный и двоичный литералы представляют десятичное значение `-1L`:

- `0xffff_ffff_ffff_ffffL`,
- `017_7777_7777_7777_7777_7777L` и
- `0b1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111L`.

В случае, если шестнадцатеричный, восьмеричный или двоичный литерал типа `long` не может быть размещен в 64 бит, генерируется ошибка времени компиляции.

Примеры литералов типа `int`:

```
0  2  0372  0xDada_Cafe  1996  0x00_FF__00_FF
```

Примеры литералов типа `long`:

```
01  0777L  0x100000000L  2_147_483_648L  0xC0B0L
```

§3.10.2. Литералы с плавающей точкой

Литерал с плавающей точкой состоит из следующих частей: целой части, десятичной или шестнадцатеричной точки (представлена ASCII-символом точки), дробной части, показателя степени и суффикса типа.

Литерал с плавающей точкой может быть выражен в десятичной или шестнадцатеричной системе счисления.

В случае десятичного литерала с плавающей точкой требуется наличие по крайней мере одной цифры (в целой или дробной части) и либо десятичной точки, либо показателя степени, либо суффикса типа. Все остальные части являются необязательными. Показатель степени, если таковой присутствует, обозначается ASCII-буквой `e` или `E`, за которой следует дополнительное целое число.

В случае шестнадцатеричного литерала с плавающей точкой требуется наличие по крайней мере одной цифры (в целой или дробной части), как и показателя степени. Наличие суффикса типа является необязательным. Показатель степени обозначается ASCII-буквой `p` или `P`, за которой следует дополнительное целое число.

Подчеркивания разрешаются в качестве разделителей цифр, которые определяют целую часть числа, между цифрами, которые определяют дробную часть, и между цифрами, которые определяют показатель степени.

FloatingPointLiteral:
    DecimalFloatingPointLiteral
    HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
    Digits . *[Digits] [ExponentPart] [FloatTypeSuffix]*
    . *Digits [ExponentPart] [FloatTypeSuffix]*
    Digits ExponentPart [FloatTypeSuffix]
    Digits [ExponentPart] FloatTypeSuffix

ExponentPart:
    ExponentIndicator SignedInteger

ExponentIndicator: одно из
`e E`

SignedInteger:
[Sign] Digits

Sign: одно из
`+ -`

FloatTypeSuffix: одно из
`f F d D`

HexadecimalFloatingPointLiteral:
HexSignificand BinaryExponent [FloatTypeSuffix]

HexSignificand:
HexNumeral [.]
`0 x` *[HexDigits] . HexDigits*
`0 X` *[HexDigits] . HexDigits*

BinaryExponent:
BinaryExponentIndicator SignedInteger

BinaryExponentIndicator: одно из
`p P`

Литерал с плавающей запятой имеет тип `float`, если он имеет суффикс в виде ASCII-буквы `F` или `f`; в противном случае его тип — `double` и он может (необязательно) иметь в качестве суффикса ASCII-букву `D` или `d` (§4.2.3).

Элементами типа `float` и `double` являются те значения, которые могут быть представлены соответственно с помощью 32-битовых одинарной точности и 64-битовых двойной точности бинарных форматов чисел с плавающей точкой стандарта IEEE 754.

> Детали корректного преобразования входных данных, представленных в виде строки Unicode, в бинарное представление числа с плавающей точкой в стандарте IEEE 754 описаны для методов `valueOf` классов `Float` и `Double` пакета `java.lang`.

Наибольший положительный конечный литерал типа `float` — `3.4028235e38f`.
Наименьший положительный конечный ненулевой литерал типа `float` — `1.40e-45f`.
Наибольший положительный конечный литерал типа `double` — `1.79769313486231570e308`.
Наименьший положительный конечный ненулевой литерал типа `double` — `4.9e-324`.

Если ненулевой литерал с плавающей точкой слишком велик, так что при преобразовании во внутреннее представление он становится равным бесконечности IEEE 754, генерируется ошибка времени компиляции.

Программа может представить бесконечность без генерации ошибки времени компиляции с помощью константных выражений, таких как `1f/0f` или `-1d/0d`, или исполь-

зуя предопределенные константы `POSITIVE_INFINITY` и `NEGATIVE_INFINITY` из классов `Float` и `Double`.

Если ненулевой литерал с плавающей точкой слишком мал, так что при преобразовании во внутреннее представление он становится равным нулю, генерируется ошибка времени компиляции.

Если ненулевой литерал с плавающей точкой имеет столь малое значение, что при преобразовании с округлением во внутреннее представление оно становится ненулевым денормализованным числом, генерируется ошибка времени компиляции.

Предопределенные константы, представляющие значения NaN (Not-a-Number — не число), определены в классах `Float` и `Double` как `Float.NaN` и `Double.NaN`.

Примеры литералов типа `float`:

```
1e1f  2.f  .3f  0f  3.14f  6.022137e+23f
```

Примеры литералов типа `double`:

```
1e1  2.  .3  0.0  3.14  1e-9d  1e137
```

§3.10.3. Логические литералы

Тип `boolean` имеет два значения, представленные *логическими (булевыми) литералами* `true` и `false`, образованными из ASCII-букв.

BooleanLiteral: одно из

```
true false
```

Логический литерал всегда имеет тип `boolean` (§4.2.5).

§3.10.4. Символьные литералы

Символьный литерал выражается в виде символа или управляющей последовательности (§3.10.6), заключенной в одинарные кавычки ASCII. (Символом одинарных кавычек, или апострофа, является `\u0027`.)

CharacterLiteral:
' *SingleCharacter* '
' *EscapeSequence* '

SingleCharacter:
InputCharacter но не ' или \

Определение *EscapeSequence* см. в §3.10.6.

Символьные литералы могут представлять только кодовые слова UTF-16 (§3.1), т.е. они ограничены значениями от `\u0000` до `\uffff`. Дополнительные символы должны быть представлены либо суррогатными парами в последовательности `char`, либо как целое число, в зависимости от API, с которым они используются.

Символьный литерал всегда имеет тип `char` (§4.2.1).

Если после *SingleCharacter* или *EscapeSequence* идет символ, отличный от ', то генерируется ошибка времени компиляции.

Если между открывающей и закрывающей одинарными кавычками имеется ограничитель строки (§3.4), то генерируется ошибка времени компиляции.

Как указано в §3.4, символы CR и LF никогда не являются *InputCharacter*; они распознаются как составляющие *LineTerminator*.

Вот примеры символьных литералов.

- `'a'`
- `'%'`
- `'\t'`
- `'\\'`
- `'\''`
- `'\u03a9'`
- `'\uFFFF'`
- `'\177'`
- `'Ω'`

Поскольку управляющие последовательности Unicode обрабатываются очень рано, некорректно написать "`\u000a`" для символьного литерала, значение которого представляет собой перевод строки (LF); управляющая последовательность `\u000a` преобразуется в фактический символ перевода строки на первом этапе трансляции (§3.3), а затем этот символ становится *LineTerminator* на втором этапе (§3.4), так что на третьем этапе этот символьный литерал становится недопустимым. Вместо него следует использовать управляющую последовательность "`\n`" (§3.10.6). Аналогично неверно писать "`\u000d`" для символьного литерала, значение которого — символ возврата каретки (CR). Вместо этого используйте управляющую последовательность "`\r`".

В C и C++ символьный литерал может содержать представление более чем одного символа, но значение такого литерала зависит от конкретной реализации компилятора. В языке программирования Java символьный литерал всегда представляет ровно один символ.

§3.10.5. Строковые литералы

Строковый литерал состоит из нуля или большего количества символов, заключенных в двойные кавычки. Символы могут быть представлены в виде управляющих последовательностей (§3.10.6): одна последовательность — для символов из диапазона U+0000 до U+FFFF, две последовательности — для суррогатных кодовых слов UTF-16 суррогатных символов из диапазона U+010000 до U+10FFFF.

StringLiteral:
" *{StringCharacter}* "

StringCharacter:
InputCharacter но не " *или* \
EscapeSequence

Определение *EscapeSequence* см. в §3.10.6.

Строковый литерал всегда имеет тип `String` (§4.3.3).

Если между открывающей и закрывающей кавычками имеется ограничитель строки, генерируется ошибка времени компиляции.

Как указано в §3.4, символы CR и LF никогда не являются *InputCharacter*; они распознаются как составляющие *LineTerminator*.

Длинный строковый литерал всегда можно разбить на короткие фрагменты и записать как выражение (возможно, в круглых скобках) с помощью оператора конкатенации строк + (§15.18.1).

Вот примеры строковых литералов.

```
""                  // Пустая строка
"\""                // Строка из одного символа "
"Это строка"        // Строка из 10 символов
"Конкатенация " +   // В действительности это строковая кон-
    "двух строк"    // станта, образованная двумя литералами
```

Поскольку управляющие последовательности Unicode обрабатываются очень рано, некорректно написать `"\u000a"` для строкового литерала, содержащего один символ перевода строки (LF); управляющая последовательность `\u000a` преобразуется в фактический символ перевода строки на первом этапе трансляции (§3.3), а затем этот символ становится *LineTerminator* на втором этапе (§3.4), так что на третьем этапе этот символьный литерал становится недопустимым. Вместо него следует использовать управляющую последовательность `"\n"` (§3.10.6). Аналогично неверно писать `"\u000d"` для символьного литерала, значение которого — символ возврата каретки (CR). Вместо этого используйте управляющую последовательность `"\r"`. Наконец невозможно записать `"\u0022"` для строкового литерала, содержащего символ двойных кавычек (").

Строковый литерал представляет собой ссылку на экземпляр класса `String` (§4.3.1, §4.3.3).

Кроме того, строковый литерал всегда ссылается на *один и тот же* экземпляр класса `String`. Это связано с тем, что строка литералов (или, в общем случае, строки, которые являются значениями константных выражений (§15.28)) “интернируется" так, чтобы совместно работать с единственными экземплярами, используя метод `String.intern`.

Пример 3.10.5-1. Строковые литералы

Программа, состоящая из модуля компиляции (compilation unit) (§7.3)

```
package testPackage;
class Test {
    public static void main(String[] args) {
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((other.Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
class Other { static String hello = "Hello"; }
```

и модуля компиляции

```
package other;
public class Other { public static String hello = "Hello"; }
```

генерирует следующий вывод:

```
true true true true false true
```

Этот пример иллюстрирует шесть пунктов.

- Литеральные строки в одном и том же классе (§8) в одном и том же пакете (§7) представляют собой ссылки на один и тот же объект `String` (§4.3.1).
- Литеральные строки в различных классах в одном и том же пакете представляют собой ссылки на один и тот же объект `String`.
- Литеральные строки в разных классах в разных пакетах также представляют собой ссылки на один и тот же объект `String`.
- Строки, вычисленные с помощью константных выражений (§15.28), вычисляются во время компиляции, а затем обрабатываются так, как если бы они были литералами.
- Строки, вычисленные путем конкатенации во время выполнения, являются вновь создаваемыми, и потому отличными от других.
- В результате явного интернирования вычисленная строка является той же строкой, что и любая ранее существовавшая литеральная строка с тем же содержимым.

§3.10.6. Управляющие последовательности для символьных и строковых литералов

Символьные и строковые *управляющие последовательности* позволяют представлять некоторые неграфические символы, такие как одинарные кавычки, двойные кавычки и обратная косая черта, в символьных (§3.10.4) и строковых (§3.10.5) литералах.

EscapeSequence:
```
\ b   /* \u0008: забой BS */
\ t   /* \u0009: горизонтальная табуляция HT */
\ n   /* \u000a: новая строка LF */
\ f   /* \u000c: перевод страницы FF */
\ r   /* \u000d: возврат каретки CR */
\ "   /* \u0022: двойные кавычки " */
\ '   /* \u0027: одинарные кавычки ' */
\ \   /* \u005c: обратная косая черта \ */
```
OctalEscape /* восьмеричное значение от `\u0000` до `\u00ff` */

OctalEscape:
```
\ OctalDigit
\ OctalDigit OctalDigit
\ ZeroToThree OctalDigit OctalDigit
```

OctalDigit: одно из
```
0 1 2 3 4 5 6 7
```

ZeroToThree: одно из
```
0 1 2 3
```

Если после обратной косой черты в управляющей последовательности идет символ, отличный от ASCII-символа `b`, `t`, `n`, `f`, `r`, `"`, `'`, `\`, `0`, `1`, `2`, `3`, `4`, `5`, `6` или `7`, генерируется ошибка времени компиляции. Управляющие последовательности Unicode `\u` обрабатываются раньше (§3.3).

> Восьмеричные управляющие последовательности предназначены для совместимости с языком программирования C, но поскольку так можно выразить только символы Unicode от `\u0000` до `\u00FF`, обычно предпочтительнее использовать управляющие последовательности Unicode.

§3.10.7. Литерал null

Тип null имеет единственное значение — пустую ссылку, представленную литералом `null`, сформированным из символов ASCII.

NullLiteral:
```
null
```

Литерал `null` всегда имеет тип null (§4.1).

§3.11. Разделители

Двенадцать токенов, сформированных из символов ASCII, представляют собой *разделители*.

Separator: одно из
```
( ) { } [ ] ; , . ... @ ::
```

§3.12. Операторы

38 токенов, сформированных из символов ASCII, представляют собой *операторы*.

Operator: одно из
```
=    >    <    !    ~    ?    :    ->
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
```

ГЛАВА 4

Типы, значения и переменные

Язык программирования Java является *статически типизированным* языком. Это означает, что каждая переменная и каждое выражение имеют тип, который известен во время компиляции.

Язык программирования Java является также *строго типизированным* языком, потому что типы ограничивают значения, которые могут храниться в переменных (§4.12) и могут быть результатом выражения. Типы также ограничивают операции, которые могут выполняться с этими значениями, и определяют смысл выполняемых операций. Строгая статическая типизация помогает обнаруживать ошибки во время компиляции.

Типы языка программирования Java разделены на две категории: примитивные и ссылочные типы. К примитивным типам (§4.2) относятся тип `boolean` и числовые. Числовые типы подразделяются на целочисленные типы `byte`, `short`, `int`, `long` и `char` и типы с плавающей точкой `float` и `double`. Ссылочные типы (§4.3) включают типы классов, интерфейсов и массивов. Имеется также специальный тип `null`. Объект (§4.3.1) представляет собой динамически созданный экземпляр типа класса или динамически созданный массив. Значения ссылочного типа являются ссылками на объекты. Все объекты, включая массивы, поддерживают методы класса `Object` (§4.3.2). Строковые литералы представлены объектами типа `String` (§4.3.3).

§4.1. Виды типов и значений

Имеется два вида типов в языке программирования Java: примитивные типы (§4.2) и ссылочные типы (§4.3). Соответственно, существует два вида значений данных, которые можно хранить в переменных, передавать как аргументы и возвращать методами и над которыми можно выполнять операции: примитивные значения (§4.2) и ссылочные значения (§4.3).

Type:
    PrimitiveType
    ReferenceType

Существует также специальный *тип null*, тип выражения `null` (§3.10.7, §15.8.1), который не имеет имени.

Поскольку этот тип не имеет имени, невозможно объявить переменную типа null или выполнить приведение к типу `null`.

Пустая ссылка `null` является единственным возможным значением выражения типа null.

Пустая ссылка всегда может быть присвоена или приведена к любому ссылочному типу (§5.2, §5.3, §5.5).

> На практике программист может игнорировать тип null и просто делать вид, что `null` — не более чем специальный литерал, который может быть любого ссылочного типа.

§4.2. Примитивные типы и значения

Примитивные типы предопределены в языке программирования Java и именованы с применением зарезервированных ключевых слов (§3.9).

PrimitiveType:
    {Annotation} NumericType
    {Annotation} `boolean`

NumericType:
    IntegralType
    FloatingPointType

IntegralType: одно из
    `byte short int long char`

FloatingPointType: одно из
    `float double`

Примитивные значения не разделяют свои состояния с другими примитивными значениями.

Числовыми типами являются целочисленные типы и типы с плавающей точкой.

Целочисленными типами являются `byte`, `short`, `int` и `long`, значения которых представляют собой целые числа размером 8, 16, 32 и 64 бит соответственно в формате дополнения до 2, а также тип `char`, значениями которого являются 16-битовые целые числа без знака, представляющие кодовые слова UTF-16 (§3.1).

Типами с плавающей точкой являются `float`, значения которого включают 32-битовые числа с плавающей точкой стандарта IEEE 754, и `double`, значения которого включают 64-битовые числа с плавающей точкой стандарта IEEE 754.

Тип `boolean` имеет ровно два значения: `true` и `false`.

§4.2.1. Целочисленные типы и значения

Значения целочисленных типов представляют собой целые числа в следующих диапазонах.

- `byte`: от –128 до 127 включительно
- `short`: от –32768 до 32767 включительно

- `int`: от –2147483648 до 2147483647 включительно
- `long`: от –9223372036854775808 до 9223372036854775807 включительно
- `char`: от `'\u0000'` до `'\uffff'` включительно, т.е. от 0 до 65535

§4.2.2. Целочисленные операции

Язык программирования Java предоставляет ряд операторов, работающих с целочисленными значениями.

- Операторы сравнения, которые дают результат типа `boolean`.
  - ✦ Операторы числового сравнения <, <=, > и >= (§15.20.1).
  - ✦ Операторы числового равенства == и != (§15.21.1).
- Числовые операторы, дающие в результате значения типа `int` или `long`.
  - ✦ Операторы унарных плюса и минуса + и – (§15.15.3, §15.15.4).
  - ✦ Мультипликативные операторы *, / и % (§15.17).
  - ✦ Аддитивные операторы + и – (§15.18).
  - ✦ Оператор инкремента ++, как префиксный (§15.15.1), так и постфиксный (§15.14.2).
  - ✦ Оператор декремента --, как префиксный (§15.15.2), так и постфиксный (§15.14.3).
  - ✦ Операторы знакового и беззнакового сдвига <<, >> и >>> (§15.19).
  - ✦ Оператор побитового дополнения ~ (§15.15.5).
  - ✦ Целые побитовые операторы &, ^ и | (§15.22.1).
- Условный оператор ? : (§15.25).
- Оператор приведения (§15.16), который может конвертировать целочисленное значение в значение любого указанного числового типа.
- Оператор конкатенации строк + (§15.18.1), который, получая в качестве операндов объект `String` и целочисленное значение, преобразует последнее в объект `String`, представляющий значение в десятичном виде, а затем создает новый объект `String`, являющийся конкатенацией указанных строк.

Другие полезные конструкторы, методы и константы предопределены в классах `Byte`, `Short`, `Integer`, `Long` и `Character`.

Если целочисленный оператор, отличный от оператора сдвига, имеет по крайней мере один операнд типа `long`, то операция осуществляется с 64-битовой точностью, и результатом является значение типа `long`. Если другой операнд не принадлежит типу `long`, он сначала расширяется (§5.1.5) до типа `long` с помощью числового повышения (numeric promotion) (§5.6).

В противном случае операция осуществляется с 32-битовой точностью, и результатом является значение типа `int`. Если один из операндов не принадлежит типу `int`, он сначала расширяется до типа `int` с помощью числового повышения.

Любое значение любого целочисленного типа может быть приведено к любому другому целочисленному типу. Приведение между целочисленными типами и типом `boolean` невозможно.

> Смотрите в §4.2.5 идиому для конвертации целочисленных выражений в тип `boolean`.

Целочисленные операторы никак не оповещают о переполнении или потере точности. Целочисленный оператор может вызвать исключение (§11) по следующим причинам.

- Любой целочисленный оператор может сгенерировать `NullPointerException`, если требуется распаковка (§5.1.8) ссылки `null`.
- Оператор целочисленного деления `/` (§15.17.2) и оператор целочисленного остатка `%` (§15.17.3) могут сгенерировать исключение `ArithmeticException`, если правый операнд равен нулю.
- Операторы инкремента и операторы декремента `++` (§15.14.2, §15.15.1) и `--` (§15.14.3, §15.15.2) могут сгенерировать исключение `OutOfMemoryError`, если требуется упаковка (§5.1.7), но для выполнения преобразования недостаточно памяти.

Пример 4.2.2-1. Целочисленные операции

```
class Test {
    public static void main(String[] args) {
        int i = 1000000;
        System.out.println(i * i);
        long l = i;
        System.out.println(l * l);
        System.out.println(20296 / (l - i));
    }
}
```

Вывод программы:

```
-727379968
1000000000000
```

После этого генерируется исключение `ArithmeticException` при делении на `l-i`, поскольку это значение равно нулю. Первое умножение выполняется с 32-битовой точностью, в то время как второе умножение представляет собой "длинное" умножение. Значение `-727379968` представляет собой младшие 32 бит математического результата умножения, `1000000000000`, слишком большого для размещения в типе `int`.

§4.2.3. Типы, форматы и значения с плавающей точкой

Типы с плавающей точкой в языке программирования Java — `float` и `double`, которые концептуально связаны со значениями и операциями в формате IEEE 754 с одинарной (32-битовой) и двойной (64-битовой) точностью, как указано в стандарте *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York).*

Стандарт IEEE 754 включает не только положительные и отрицательные числа, состоящие из знака и величины, но и положительные и отрицательные нули, положительные и отрицательные *бесконечности* и специальные значения *Not-a-Number* (*не число*; далее сокращенно — NaN). Значение NaN используется для представления результата определенных недопустимых операций, например деления нуля на нуль. NaN-константы типа `float` и `double` предопределены как `Float.NaN` и `Double.NaN`.

Каждая реализация языка программирования Java должна поддерживать два стандартных набора значений с плавающей точкой, именуемых *набор значений float* и *набор значений double*. Кроме того, реализация языка программирования Java может поддерживать любой (или оба) набор значений с плавающей точкой с расширенным показателем степени. Эти наборы значений могут при определенных обстоятельствах использоваться вместо стандартных для представления значений выражений типа *float* или *double* (§5.1.13, §15.4).

Конечное ненулевое значение из любого набора значений с плавающей точкой может быть выражено в виде $s \cdot m \cdot 2^{(e-N+1)}$, где s равно +1 или –1, m является положительным целым числом, меньшим 2^N, а e — целое число между $E_{min} = -\left(2^{K-1} - 2\right)$ и $E_{max} = 2^{K-1} - 1$ включительно и где N и K — параметры, зависящие от набора значений. Некоторые значения могут быть представлены в этом виде более чем одним способом. Предположим, например, что значение v в наборе значений может быть представлено в этой форме определенными элементами s, m и e. Тогда, если окажется, что m четно, а e меньше 2^{K-1}, можно уменьшить m вдвое и увеличить e на 1, получая при этом второе представление того же самого значения v. Представление в таком виде называется *нормализованным*, если $m \geq 2^{N-1}$; в противном случае представление называется *денормализованным*. Если значение в наборе значений не может быть представлено так, чтобы выполнялось неравенство $m \geq 2^{N-1}$, то такое значение называется *денормализованным значением*, потому что оно не имеет нормализованного представления.

Ограничения на параметры N и K (и производные параметры E_{min} и E_{max}) для двух обязательных и двух дополнительных наборов значений с плавающей точкой приведены в табл. 4.1.

Таблица 4.1. Параметры наборов значений с плавающей точкой

Параметр	float	Расширенный float	double	Расширенный double
N	24	24	53	53
K	8	≥ 11	11	≥ 15
E_{max}	+127	≥ +1023	+1023	≥ +16383
E_{min}	–126	≤ –1022	–1022	≤ –16382

В случае, когда реализация поддерживает один или два расширенных набора значений, для каждого поддерживаемого набора имеется своя константа K, зависящая от реализации и ограниченная так, как показано в табл. 4.1. Это значение K, в свою очередь, диктует значения E_{min} и E_{max}.

Каждый из четырех наборов значений включает не только конечные ненулевые значения, описанные выше, но и значения NaN, а также положительный и отрицательный нули и положительную и отрицательную бесконечности.

Обратите внимание, что ограничения в табл. 4.1 разработаны так, чтобы каждый элемент множества значений обязательно являлся также элементом соответствующего расширенного набора. Каждый расширенный набор имеет больший диапазон значения показателя степени, чем соответствующий стандартный набор, но не бо́льшую точность.

Элементы набора значений float представляют собой в точности те значения, которые могут быть представлены с использованием формата с плавающей точкой одинарной точности, определенного в стандарте IEEE 754. Элементы набора значений double представляют собой в точности те значения, которые могут быть представлены с использованием формата с плавающей точкой двойной точности, определенного в стандарте IEEE 754. Заметьте, однако, что элементы определенных здесь расширенных наборов *не* соответствуют значениям, которые могут быть представлены с использованием форматов с одинарной и двойной точностью IEEE 754.

Все указанные наборы значений не являются типами. Корректным решением является использование реализацией языка программирования Java элемента набора значений float для представления значения типа `float`; однако в ряде областей кода реализации может быть допустимо использовать вместо этого элемент расширенного набора значений float. Аналогично корректным решением является использование реализацией языка программирования Java элемента набора значений double для представления значения типа `double`; однако в ряде областей кода реализации может быть допустимо использовать вместо этого элемент расширенного набора значений double.

За исключением значений NaN, значения с плавающей точкой *упорядочиваемы*; в порядке от наименьшего к наибольшему идут отрицательная бесконечность, отрицательные конечные ненулевые значения, отрицательный и положительный нули, положительные конечные ненулевые значения и положительная бесконечность. Стандарт IEEE 754 допускает наличие нескольких различных значений NaN для каждого из форматов чисел с плавающей точкой одинарной и двойной точности. Хотя каждая аппаратная архитектура возвращает определенный битовый шаблон при генерации нового значения NaN, программист может создавать значения NaN с различными битовыми шаблонами, например, для ретроспективной диагностической информации.

В большинстве случаев платформа Java SE рассматривает значения NaN данного типа так, как если бы они были собраны в одно каноническое значение, а следовательно, данная книга обычно ссылается на произвольные значения NaN так, как если бы это были канонические значения.

> Однако в версии 1.3 платформы Java SE введены методы, позволяющие программисту различать значения NaN: методы `Float.floatToRawIntBits` и `Double.doubleToRawLongBits`. Заинтересовавшийся читатель может для получения дополнительной информации обратиться к спецификации классов `Float` и `Double`.

Положительный и отрицательный нули рассматриваются при сравнении как одинаковое значение; таким образом, результат выражения `0.0 == -0.0` равен `true`, а результат выражения `0.0 > -0.0` равен `false`. Однако другие операции различают положительный и отрицательный нули; например, `1.0/0.0` дает положительную бесконечность, а `1.0/-0.0` — бесконечность отрицательную.

NaN является *неупорядочиваемым* значением, так что выполняется следующее.

- Операторы численного сравнения <, <=, > и >= возвращают `false`, если один или оба операнда являются NaN (§15.20.1).
- Оператор проверки равенства == возвращает значение `false`, если любой из операндов имеет значение NaN. В частности, результатом выражения `(x<y) == !(x>=y)` будет `false`, если `x` или `y` имеет значение NaN.
- Оператор проверки неравенства != возвращает значение `true`, если любой из операндов имеет значение NaN (§15.21.1). В частности, результат выражения `x != x` равен `true` тогда и только тогда, когда `x` равно NaN.

§4.2.4. Операции с плавающей точкой

Язык программирования Java предоставляет ряд операторов, работающих со значениями с плавающей точкой.

- Операторы сравнения, которые дают результат типа `boolean`.
  - ✦ Операторы числового сравнения <, <=, > и >= (§15.20.1).
  - ✦ Операторы числового равенства == и != (§15.21.1).
- Числовые операторы, дающие в результате значения типа `float` или `double`.
  - ✦ Операторы унарных плюса и минуса + и – (§15.15.3, §15.15.4).
  - ✦ Мультипликативные операторы *, / и % (§15.17).
  - ✦ Аддитивные операторы + и – (§15.18.2).
  - ✦ Оператор инкремента ++, как префиксный (§15.15.1), так и постфиксный (§15.14.2).
  - ✦ Оператор декремента --, как префиксный (§15.15.2), так и постфиксный (§15.14.3).
- Условный оператор ? : (§15.25).
- Оператор приведения (§15.16), который может конвертировать значение с плавающей точкой в значение любого указанного числового типа.
- Оператор конкатенации строк + (§15.18.1), который, получая в качестве операндов объект `String` и значение с плавающей точкой, преобразует последнее в объект `String`, представляющий значение в десятичном виде (без потери информации), а затем создает новый объект `String`, являющийся конкатенацией указанных строк.

Другие полезные конструкторы, методы и константы предопределены в классах `Float`, `Double` и `Math`.

Если как минимум один операнд бинарного оператора имеет тип с плавающей точкой, то операция является операцией с плавающей точкой, даже если второй операнд целочисленный.

Если по крайней мере один операнд числового оператора имеет тип `double`, то операция осуществляется с использованием 64-битовой арифметики с плавающей точкой, и результатом является значение типа `double`. Если другой операнд не принадлежит типу

`double`, он сначала расширяется (§5.1.5) до типа `double` с помощью числового повышения (§5.6).

В противном случае операция осуществляется с помощью 32-битовой арифметики с плавающей точкой, и результатом является значение типа `float`. Если один из операндов не принадлежит типу `float`, он сначала расширяется до типа `float` с помощью числового повышения.

Любое значение любого типа с плавающей точкой может быть приведено к любому другому численному типу, как и получено из него. Приведение между типами с плавающей точкой и типом `boolean` невозможно.

> Смотрите в §4.2.5 идиому для преобразования выражений с плавающей точкой в `boolean`.

Операторы, работающие с числами с плавающей точкой, ведут себя так, как указано в стандарте IEEE 754 (за исключением оператора остатка (§15.17.3)). В частности, язык программирования Java требует поддержки *денормализованных* чисел с плавающей точкой и *постепенной потери значимости* IEEE 754, которые облегчают доказательство требуемых свойств определенных численных алгоритмов. Операции с плавающей запятой не выполняют "сброс в нуль", если вычисленный результат оказывается денормализованным числом.

Язык программирования Java требует, чтобы арифметика с плавающей точкой вела себя так, как если бы каждый оператор с плавающей точкой округлял вычисленное им значение до точности результата. *Неточные* результаты должны быть округлены до представимого значения, ближайшего к абсолютно точному результату операции. Если имеется два в равной степени ближайших представимых значения, выбирается то, младший бит которого нулевой. Это режим округления стандарта IEEE 754 по умолчанию, известный как *округление до ближайшего*.

Язык программирования Java использует *округление в сторону нуля* при преобразовании значения с плавающей точкой в целое число (§5.1.3), которое в этом случае действует так, как будто числа усекаются, с отбрасыванием битов мантиссы. Округление в сторону нуля дает в качестве результата значение в новом формате, ближайшее к точному результату, но не превышающее его.

Операция с плавающей точкой, вызывающая переполнение, дает знаковую бесконечность.

Операция с плавающей точкой, приводящая к потере значимости, дает денормализованное значение или знаковый нуль.

Операция с плавающей точкой, которая не имеет математически определенного результата, дает NaN.

Все числовые операции с NaN в качестве операнда дают в результате NaN.

Оператор с плавающей точкой может генерировать исключение (§11) по следующим причинам.

- Любой оператор с плавающей точкой может генерировать исключение `NullPointerException`, если требуется распаковка (§5.1.8) ссылки null.

- Операторы инкремента и декремента ++ (§15.14.2, §15.15.1) и -- (§15.14.3, §15.15.2) могут генерировать исключение `OutOfMemoryError`, если требуется упаковка (§5.1.7) и недостаточно памяти для выполнения этого преобразования.

Пример 4.2.4-1. Операции с плавающей точкой

```
class Test
{
    public static void main(String[] args)
    {
        // Пример переполнения:
        double d = 1e308;
        System.out.print("переполнение дает бесконечность: ");
        System.out.println(d + "*10==" + d*10);
        // Пример постепенной потери значимости:
        d = 1e-305 * Math.PI;
        System.out.print("постепенная потеря значимости: "
                         + d + "\n ");
        for (int i = 0; i < 4; i++)
        {
            System.out.print(" " + (d /= 100000));
        }
        System.out.println();
        // Пример работы с NaN:
        System.out.print("0.0/0.0 является NaN: ");
        d = 0.0/0.0;
        System.out.println(d);
        // Пример неточного результата и округления:
        System.out.print("неточный результат float:");

        for (int i = 0; i < 100; i++)
        {
            float z = 1.0f / i;

            if (z * i != 1.0f)
            {
                System.out.print(" " + i);
            }
        }

        System.out.println();
        // Другой пример неточного результата и округления:
        System.out.print("неточный результат double:");

        for (int i = 0; i < 100; i++)
        {
            double z = 1.0 / i;
```

```
            if (z * i != 1.0)
            {
                System.out.print(" " + i);
            }
        }

        System.out.println();
        // Пример приведения к целому с округлением:
        System.out.print("приведение к int с округлением " +
                         "к нулю: ");
        d = 12345.6;
        System.out.println((int)d + " " + (int)(-d));
    }
}
```

Вывод программы имеет следующий вид.

```
переполнение дает бесконечность: 1.0e+308*10==Infinity
постепенная потеря значимости: 3.14159265358793E-305
3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
0.0/0.0 является NaN: NaN
неточный результат float: 0 41 47 55 61 82 83 94 97
неточный результат double: 0 49 98
приведение к int с округлением к нулю: 12345 -12345
```

Этот пример среди прочего демонстрирует, что постепенная потеря значимости может привести к постепенной потере точности.

Результаты при `i`, равном `0`, приводят к делению на нуль, так что `z` становится равным положительной бесконечности, и `z*0` дает значение NaN, которое не равно `1.0`.

§4.2.5. Тип `boolean` и его значения

Тип `boolean` представляет логическую величину с двумя возможными значениями, указываемыми литералами `true` и `false` (§3.10.3).

Логическими операторами являются следующие.

- Реляционные операторы == и != (§15.21.2).
- Оператор логического дополнения ! (§15.15.6).
- Логические операторы &, ^ и | (§15.22.2).
- Операторы условного И и условного ИЛИ && (§15.23) и || (§15.24).
- Условный оператор ? : (§15.25).
- Оператор конкатенации + (§15.18.1), который, получая в качестве операндов объект `String` и значение типа `boolean`, преобразует последнее в объект `String` (либо `"true"`, либо `"false"`), а затем создает новый объект `String`, являющийся конкатенацией указанных строк.

Логические выражения определяют поток управления в ряде инструкций.

- Инструкция `if` (§14.9).
- Инструкция `while` (§14.12).
- Инструкция `do` (§14.13).
- Инструкция `for` (§14.14).

Выражение типа `boolean` также определяет, какое подвыражение будет вычисляться в условном операторе `? :` (§15.25).

В конструкциях управления потоком и в качестве первого операнда условного оператора `? :` могут использоваться только выражения типов `boolean` и `Boolean`.

С помощью выражения `x != 0` целое выражение или выражение с плавающей точкой `x` может быть преобразовано в тип `boolean` в соответствии с соглашениями языка программирования C о том, что любое ненулевое значение равно `true`.

Ссылка на объект `obj` может быть преобразована в тип `boolean` в соответствии с соглашениями языка программирования C о том, что любая ссылка, не равная `null`, равна `true`, с помощью выражения `obj != null`.

Значение типа `boolean` может быть преобразовано в объект типа `String` преобразованием строк (§5.4).

Разрешается приведение значения типа `boolean` к типу `boolean`, `Boolean` или `Object` (§5.5). Никакие иные приведения типа `boolean` не разрешены.

§4.3. Ссылочные типы и значения

Существует четыре типа *ссылочных типов*: типы классов (§8.1), типы интерфейсов (§9.1), переменные типов (§4.4) и типы массивов (§10.1).

ReferenceType:
    ClassOrInterfaceType
    TypeVariable
    ArrayType

ClassOrInterfaceType:
    ClassType
    InterfaceType

ClassType:
    {Annotation} Identifier [TypeArguments]
    ClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

InterfaceType:
    ClassType

TypeVariable:
    {Annotation} Identifier

ArrayType:
PrimitiveType Dims
ClassOrInterfaceType Dims
TypeVariable Dims

Dims:
{Annotation} [ ] *{{Annotation}* [ ] *}*

Фрагмент кода

```
class Point { int[] metrics; }
interface Move { void move(int deltax, int deltay); }
```

объявляет тип класса `Point`, **тип интерфейса** `Move` **и использует тип массива** `int[]` **(массив значений типа** `int`**) для объявления поля** `metrics` **класса** `Point`.

Тип класса или интерфейса состоит из идентификатора или последовательности идентификаторов, разделенных точками, где за каждым идентификатором (необязательно) следуют аргументы типа (§4.5.1). Если в каком-то месте типа класса или интерфейса имеются аргументы типа, мы имеем дело с параметризованным типом (§4.5).

Каждый идентификатор в типе класса или интерфейса классифицируется как имя пакета или имя типа (§6.5.1). Идентификаторы, классифицируемые как имена типов, могут быть аннотированы. Если тип класса или интерфейса имеет вид *T*.*id* (с необязательными следующими за ним аргументами типов), то *id* должен быть простым именем доступного типа-члена *T* (§6.6, §8.5, §9.5), иначе генерируется ошибка времени компиляции. Тип класса или интерфейса описывает тип этого члена.

§4.3.1. Объекты

Объект представляет собой *экземпляр класса* или *массив*.

Ссылочные значения (часто просто *ссылки*) являются указателями на эти объекты; к ним относится и специальная пустая ссылка, которая ссылается на несуществующий объект.

Экземпляр класса создается явно, с помощью выражения создания экземпляра класса (§15.9).

Массив создается явно, с помощью выражения создания массива (§15.10.1).

Новый экземпляр класса создается неявно, когда оператор конкатенации строк + (§15.18.1) используется в неконстантном выражении (§15.28), давая в результате новый объект типа `String` (§4.3.3).

Новый объект массива неявно создается при вычислении выражения инициализатора массива (§10.6); это может произойти при инициализации класса или интерфейса (§12.4), когда создается новый экземпляр класса (§15.9) или выполняется инструкция объявления локальной переменной (§14.4).

Новые объекты типов `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` и `Double` могут неявно создаваться преобразованием упаковки (§5.1.7).

Пример 4.3.1-1. Создание объектов

```
class Point
{
```

```
    int x, y;
    Point()
    {
        System.out.println("по умолчанию");
    }
    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    /* Экземпляр Point явно создается во
       время инициализации класса: */
    static Point origin = new Point(0,0);
    /* Объект String может быть создан
       неявно оператором +: */
    public String toString()
    {
        return "(" + x + "," + y + ")";
    }
}
class Test
{
    public static void main(String[] args)
    {
        /* Объект Point создается явно
           с использованием newInstance: */
        Point p = null;
        try
        {
            p = (Point)Class.forName("Point").newInstance();
        }
        catch (Exception e)
        {
            System.out.println(e);
        }

        /* Массив создается неявно конструктором массива: */
        Point a[] = { new Point(0,0), new Point(1,1) };
        /* Строки создаются неявно операторами +: */
        System.out.println("p: " + p);
        System.out.println("a: { " + a[0] + ", " +
                           a[1] + " }");
        /* Массив создается явно с помощью
           выражения создания массива: */
        String sa[] = new String[2];
        sa[0] = "при";
        sa[1] = "вет";
```

```
            System.out.println(sa[0] + sa[1]);
        }
}
```

Вот как выглядит вывод этой программы.

```
по умолчанию
p: (0,0)
a: { (0,0), (1,1) }
привет
```

Над ссылками на объекты выполняются следующие операции.

- Доступ к полям с помощью либо квалифицированного имени (§6.6), либо выражения доступа к полю (§15.11).
- Вызов метода (§15.12).
- Оператор приведения (§5.5, §15.16).
- Оператор конкатенации строк + (§15.18.1), который для заданных операндов типа `String` и ссылки конвертирует ссылку в `String` путем вызова метода `toString` объекта, на который указывает ссылка (в строку `"null"`, если ссылка пустая или если результат вызова `toString` является пустой ссылкой), а затем создает объект типа `String`, который представляет собой конкатенацию двух указанных строк.
- Оператор `instanceof` (§15.20.2).
- Операторы проверки равенства ссылок == и != (§15.21.3).
- Условный оператор ? : (§15.25).

Может иметься много ссылок на один и тот же объект. Большинство объектов имеют состояние, хранящееся в полях объектов, которые являются экземплярами классов, или в переменных, которые являются компонентами объекта массива. Если две переменные содержат ссылки на один и тот же объект, состояние объекта можно изменять с помощью одной переменной-ссылки на объект, а затем измененное состояние можно наблюдать с помощью ссылки, хранящейся в другой переменной.

Пример 4.3.1-2. Примитивная и ссылочная тождественность

```
class Value
{
    int val;
}
class Test
{
    public static void main(String[] args)
    {
        int i1 = 3;
        int i2 = i1;
        i2 = 4;
        System.out.print("i1==" + i1);
        System.out.println(" но i2==" + i2);
        Value v1 = new Value();
```

```
        v1.val = 5;
        Value v2 = v1;
        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" и v2.val==" + v2.val);
    }
}
```

Вот как выглядит вывод этой программы.

```
i1==3 но i2==4
v1.val==6 и v2.val==6
```

Это связано с тем, что `v1.val` и `v2.val` ссылаются на одну и ту же переменную экземпляра (§4.12.3) в одном объекте типа `Value`, созданном выражением `new`, в то время как `i1` и `i2` представляют собой различные переменные.

Каждый объект связан с монитором (§17.1), который используется методами и инструкциями `synchronized` (§8.4.3, §14.19) для обеспечения контроля над параллельным доступом к состоянию объекта несколькими потоками (§17).

§4.3.2. Класс `Object`

Класс `Object` является суперклассом (§8.1.4) всех других классов. Все типы классов и массивов наследуют (§8.4.8) методы класса `Object`, которые подытожены далее.

- Метод `clone` используется для создания дубликата объекта.
- Метод `equals` определяет понятие эквивалентности объектов, основанное на сравнении значений, а не ссылок.
- Метод `finalize` выполняется непосредственно перед уничтожением объекта (§12.6).
- Метод `getClass` возвращает объект типа `Class`, который представляет класс объекта.

  Объект типа `Class` существует для каждого ссылочного типа. Он может использоваться, например, для выяснения полного квалифицированного имени класса, его членов, непосредственного суперкласса и реализуемых им интерфейсов.

  Типом выражения вызова метода `getClass` является `Class<? extends |T|>`, где *T* является искомым классом или интерфейсом (§15.12.1) для метода `getClass`.

  Метод класса, объявленный как `synchronized` (§8.4.3.6), синхронизируется с помощью монитора, связанного с объектом типа `Class` класса.
- Метод `hashCode` (вместе с методом `equals`) весьма полезен в хеш-таблицах, таких как `java.util.Hashmap`.
- Методы `wait`, `notify` и `notifyAll` используются в параллельном программировании с использованием потоков (§17.2).
- Метод `toString` возвращает представление объекта в виде строки `String`.

§4.3.3. Класс `String`

Экземпляры класса `String` представляют последовательности кодов Unicode.

Объект типа `String` имеет константное (неизменяемое) значение.

Строковые литералы (§3.10.5) являются ссылками на экземпляры класса `String`.

Оператор конкатенации строк + (§15.18.1) неявно создает новый объект типа `String`, если результат не является константным выражением (§15.28).

§4.3.4. Когда ссылочные типы одинаковы

Два ссылочных типа являются *одним и тем же типом времени компиляции*, если они имеют одно и то же бинарное имя (§13.1) и их аргументы типа (если таковые имеются) также одинаковы (с рекурсивным применением этого определения).

Когда два ссылочных типа одинаковы, они иногда называются *одним и тем же классом* или *одним и тем же интерфейсом*.

Во время выполнения программы несколько ссылочных типов с одним и тем же бинарным именем могут быть одновременно загружены разными загрузчиками классов. Эти типы могут представлять одно и то же объявление типа, но могут и не представлять таковое. Даже если два таких типа представляют одно и то же объявление типа, они рассматриваются как различные.

Два ссылочных типа являются *одним и тем же типом времени выполнения*, если:

- они оба являются типами класса или типами интерфейса, определены одним и тем же загрузчиком класса и имеют одно и то же бинарное имя (§13.1); в этом случае они иногда называются *одним и тем же классом времени выполнения* или *одним и тем же интерфейсом времени выполнения*;
- они оба являются типами массивов, и их типы являются одним и тем же типом времени выполнения (§10).

§4.4. Переменные типа

Переменная типа представляет собой неквалифицированный идентификатор, используемый в качестве типа в телах классов, интерфейсов, методов и конструкторов.

Переменная типа вводится объявлением *параметра типа* обобщенного класса, интерфейса, метода или конструктора (§8.1.2, §9.1.2, §8.4.4, §8.8.4).

TypeParameter:
    {TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
    Annotation

TypeBound:
    `extends` *TypeVariable*
    `extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
& *InterfaceType*

Область видимости переменной типа, объявленной как параметр типа, определена в §6.3.

Каждая переменная типа, объявленная как параметр типа, имеет *границу* (bound). Если для переменной типа граница не объявлена, в ее качестве предполагается `Object`. Если граница объявлена, она состоит

- либо из одной переменной типа T,
- либо из типа класса или интерфейса T, возможно, с последующими типами интерфейсов I_1 & ... & I_n.

Если любой из типов I_1 ... I_n является типом класса или переменной типа, генерируется ошибка времени компиляции.

Затирания (erasures) (§4.6) всех составляющих границу типов должны быть попарно различны, иначе генерируется ошибка времени компиляции.

Переменная типа не должна одновременно быть подтипом двух типов интерфейсов, которые являются различными параметризациями одного и того же обобщенного интерфейса, иначе генерируется ошибка времени компиляции.

Порядок типов в границе важен только в том плане, что затирание переменной типа определяется первым типом ее границы и что тип класса или переменная типа может находиться только в первой позиции.

Члены переменной типа X с границей T & I_1 & ... & I_n являются членами типа пересечения (§4.9) T & I_1 & ... & I_n, появляющегося в точке, где объявляется переменная типа.

Пример 4.4-1. Члены переменной типа

```
package TypeVarMembers;
class C {
    public    void mCPublic() {}
    protected void mCProtected() {}
              void mCPackage() {}
    private   void mCPrivate() {}
}

interface I {
    void mI();
}

class CT extends C implements I {
    public void mI() {}
}

class Test {
    <T extends C & I> void test(T t) {
        t.mI();           // OK
        t.mCPublic();     // OK
```

```
        t.mCProtected(); // OK
        t.mCPackage();   // OK
        t.mCPrivate();   // Ошибка времени компиляции
    }
}
```

Переменная типа `T` имеет те же члены, что и тип пересечения `C & I`, который, в свою очередь, имеет те же члены, что и пустой класс `CT`, определенные в той же области видимости с эквивалентными супертипами. Члены интерфейса всегда являются открытыми (`public`) и поэтому всегда наследуются (если только они не переопределены). Следовательно, `mI` является членом `CT` и `T`. Среди членов `C` все, кроме `mCPrivate`, наследуются `CT` и потому являются членами и `CT`, и `T`.

Если тип `C` был объявлен в пакете, отличном от пакета, где объявлен тип `T`, то вызов `mCPackage` вызовет ошибку времени компиляции, так как этот член будет недоступен в точке объявления `T`.

§4.5. Параметризованные типы

Объявление обобщенного класса или интерфейса (§8.1.2, §9.1.2) определяет набор *параметризованных типов*.

Параметризованный тип представляет собой тип класса или интерфейса вида $C<T_1, \ldots, T_n>$, где C — имя обобщенного типа, а $<T_1, \ldots, T_n>$ — список аргументов типа, который определяет конкретную *параметризацию* обобщенного типа.

Обобщенный тип имеет параметры типа $F_1, \ldots, F_n$ с соответствующими границами $B_1, \ldots, B_n$. Каждый аргумент типа T_i оценивается по всем типам, являющимся подтипами всех перечисленных в соответствующей границе типов. То есть T_i для каждого ограниченного типа S_i в B_i является подтипом типа $S[F_1:=T_1, \ldots, F_n:=T_n]$ (§4.10).

Параметризованный тип $C<T_1, \ldots, T_n>$ является *правильно сформированным*, если истинны все следующие утверждения.

- C представляет собой имя обобщенного типа.
- Количество аргументов типа совпадает с количеством параметров типа в обобщенном объявлении C.
- Результат преобразования при фиксации (§5.1.10) приводит к типу $C<X_1, \ldots, X_n>$, каждый аргумент типа X_i которого является подтипом $S[F_1:=X_1, \ldots, F_n:=X_n]$ для каждого ограниченного типа S в B_i.

Если параметризованный тип не является правильно сформированным, генерируется ошибка времени компиляции.

Когда в книге мы говорим о типе класса или интерфейса, мы включаем сюда и обобщенные классы и интерфейсы, если только явно не указано иное.

Два параметризованных типа *доказуемо различны*, если выполняется любое из следующих условий.

- Они представляют собой конкретизации различных объявлений обобщенных типов.
- Некоторые их аргументы типа доказуемо различны.

Вот некоторые правильно сформированные параметризованные типы для примеров типов из §8.1.2.

- `Seq<String>`
- `Seq<Seq<String>>`
- `Seq<String>.Zipper<Integer>`
- `Pair<String,Integer>`

Вот несколько примеров некорректных конкретизаций обобщенных типов.

- `Seq<int>` неверно, поскольку примитивные типы не могут быть аргументами типа.
- `Pair<String>` неверно, поскольку имеется недостаточное количество аргументов типа.
- `Pair<String,String,String>` неверно, поскольку имеется слишком большое количество аргументов типа.

Параметризованный тип может быть параметризацией обобщенного класса или интерфейса, являющегося вложенным. Например, если необобщенный класс *C* имеет обобщенный класс-член *D<T>*, то *C.D<Object>* является параметризованным типом. Если обобщенный класс *C<T>* имеет не обобщенный класс-член *D*, то тип члена *C<String>.D* является параметризованным типом, несмотря на то что класс *D* не является обобщенным.

§4.5.1. Аргументы типа и символы подстановки

Аргументы типа могут являться ссылочными типами или символами подстановки. Символы подстановки полезны в ситуациях, когда требуется только частичное знание о параметрах типов.

TypeArguments:
    < *TypeArgumentList* >

TypeArgumentList:
    TypeArgument {, *TypeArgument*}

TypeArgument:
    ReferenceType
    Wildcard

Wildcard:
    {*Annotation*} ? [*WildcardBounds*]

WildcardBounds:
    `extends` *ReferenceType*
    `super` *ReferenceType*

Символы подстановки могут иметь явные границы, как и объявления обычных переменных типа. Верхняя граница указывается с помощью следующего синтаксиса, где *B* представляет собой границу.

```
? extends B
```

В отличие от обычных переменных типа, объявленных в сигнатуре метода, при применении символа подстановки не требуется вывод типа. Следовательно, можно указать нижнюю границу символа подстановки с помощью следующего синтаксиса, где *B* представляет собой нижнюю границу.

```
? super B
```

Символ подстановки `? extends Object` эквивалентен неограниченному символу подстановки `?`.

Два аргумента типов *доказуемо различны*, если истинно одно из следующих утверждений.

- Ни один аргумент не является ни переменной типа, ни символом подстановки, и два аргумента не являются одним и тем же типом.
- Один аргумент типа является переменной типа или символом подстановки с верхней границей (если необходимо, из преобразования при фиксации (§5.1.10)) *S*; а второй аргумент типа *T* не является переменной типа или символом подстановки; и кроме того, не выполняется ни `|`*S*`| <: |`*T*`|`, ни `|`*T*`| <: |`*S*`|` (§4.8, §4.10).
- Каждый аргумент типа является переменной типа или символом подстановки с верхней границей (если необходимо, из преобразования при фиксации) *S* и *T*; и кроме того, не выполняется ни `|`*S*`| <: |`*T*`|`, ни `|`*T*`| <: |`*S*`|`.

Говорят, что аргумент типа T_1 называется *содержащим* другой аргумент типа T_2 (записывается как T_2 `<=` T_1), если набор типов, описываемых T_2, доказуемо является подмножеством набора типов, описываемых T_1, при условии рефлексивного и транзитивного замыкания следующих правил (где *<:* обозначает подтипы (§4.10)).

- `? extends` *T* `<= ? extends,` *S* **если** *T* `<:` *S*
- `? extends` *T* `<= ?`
- `? super` *T* `<= ? super` *S*, **если** *S* `<:` *T*
- `? super` *T* `<= ?`
- `? super` *T* `<= ? extends Object`
- *T* `<=` *T*
- *T* `<= ? extends` *T*
- *T* `<= ? super` *T*

> Связь символов подстановки с теорией типов достаточно интересна, чтобы вкратце коснуться ее здесь. Символы подстановки являются ограниченной формой экзистенциальных типов. Для данного объявления обобщенного типа *G<T extends B>*, *G<?>* грубо аналогично *Некоторое X <: B. G<X>*.
>
> Исторически символы подстановки являются прямым потомком работы Ацуси Игараси (Atsushi Igarashi) и Мирко Вироли (Mirko Viroli). Заинтересованным чи

тателям советуем обратиться к работе *On Variance-Based Subtyping for Parametric Types* указанных авторов, представленной на конференции *Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP 2002)*. Эта работа опирается на более ранние работы Крестена Торупа (Kresten Thorup) и Мэдса Торгерсена (Mads Torgersen) (*Unifying Genericity*, ECOOP 99), а также на давние традиции, восходящие к работе Пьера Америки (Pierre America) (представленной на конференции *Conference on Object-Oriented Programming: Systems, Languages, and Applications* (*OOPSLA 89)*).

В ряде деталей символы подстановки отличаются от конструкций, описанных в вышеупомянутых статьях, в частности в использовании преобразования фиксации (§5.1.10) вместо операции `close`, описанной Игараси Вироли. Формальное описание символов подстановки можно найти в докладе *Wild FJ*, представленном Мэдсом Торгерсеном (Mads Torgersen), Эриком Эрнстом (Erik Ernst) и Кристианом Плеснером Хансеном (Christian Plesner Hansen) на 12-м семинаре *Foundations of Object Oriented Programming* (FOOL 2005).

Пример 4.5.1-1. Символы подстановки

```
import java.util.Collection;
import java.util.ArrayList;
class Test
{
    static void printCollection(Collection<?> c)
    {
        for (Object o : c)
        {
            System.out.println(o);
        }
    }
    public static void main(String[] args)
    {
        Collection<String> cs = new ArrayList<String>();
        cs.add("hello");
        cs.add("world");
        printCollection(cs);
    }
}
```

Обратите внимание, что применение `Collection<Object>` в качестве типа входного параметра, *c*, не было бы столь же полезным; метод мог бы использоваться только с выражением аргумента, имеющим тип `Collection<Object>`, что встречается довольно редко. В противоположность этому применение неограниченного символа подстановки позволяет использовать в качестве параметра коллекции любого рода.

Вот пример, в котором тип элемента массива параметризован с помощью символа подстановки.

```
public Method getMethod(Class<?>[] parameterTypes) { ... }
```

Пример 4.5.1-2. Ограниченные символы подстановки

```
boolean addAll(Collection<? extends E> c)
```

Здесь метод объявляется в пределах интерфейса `Collection<E>` и предназначен для добавления всех элементов входной коллекции, для которой он вызывается. Естественное решение заключается в использовании `Collection<E>` как типа `c`, но оно оказывается излишне ограничивающим. В качестве альтернативы можно объявить как обобщенный сам метод.

```
<T> boolean addAll(Collection<T> c)
```

Эта версия является достаточно гибкой, но обратите внимание, что параметр типа используется только один раз в подписи. Это отражает тот факт, что параметр типа не используется для выражения какого-либо рода взаимозависимости между типами аргументов, типом возвращаемого значения и/или типом генерируемого исключения. При отсутствии такой взаимозависимости обобщенные методы считаются плохим стилем, и более предпочтительными являются символы подстановки.

```
Reference(T referent, ReferenceQueue<? super T> queue);
```

Здесь `referent` может быть вставлен в любую очередь, элементы которой имеют тип, являющийся супертипом типа `T`; `T` в данном случае выступает в роли нижней границы для символа подстановки.

§4.5.2. Члены и конструкторы параметризованных типов

Пусть C является объявлением обобщенного класса или интерфейса с параметрами типа $A_1, \ldots, A_n$ и пусть $C<T_1, \ldots, T_n>$ является параметризацией C, где для $1 \le i \le n$ T_i представляют собой типы (а не символы подстановки). Тогда имеем следующее.

- Пусть m — объявление члена или конструктора (§8.2, §8.8.6) в C с объявленным типом `T`.
- Типом m в $C<T_1, \ldots, T_n>$ является $T[A_1 := T_1, \ldots, A_n := T_n]$.
- Пусть m — объявление члена или конструктора в D, где D — класс, расширенный C или интерфейсом, реализованным C. Пусть $D<U_1, \ldots, U_k>$ представляет собой супертип $C<T_1, \ldots, T_n>$, который соответствует D.
- Типом m в $C<T_1, \ldots, T_n>$ является тип m в $D<U_1, \ldots, U_k>$.

Если любой из аргументов типа в параметризации C представляет собой символ подстановки, то имеем следующее.

- Типы полей, методов и конструкторов в $C<T_1, \ldots, T_n>$ представляют собой типы полей, методов и конструкторов в преобразовании при фиксации $C<T_1, \ldots, T_n>$ (§5.1.10).
- Пусть D представляет собой объявление (возможно, обобщенного) класса или интерфейса в C. Тогда типом D в $C<T_1, \ldots, T_n>$ является D, где, если D — обобщенный тип, все аргументы типа представляют собой неограниченные символы подстановки.

> Это не влечет за собой последствий, поскольку доступ к члену параметризованного типа невозможен без выполнения преобразования при фиксации, как невозможно использовать тип с символом подстановки после ключевого слова `new` в выражении создания экземпляра класса (§15.9).
>
> Единственным исключением в предыдущем абзаце является ситуация, когда вложенный параметризованный тип используется как выражение в операторе `instanceof` (§15.20.2), где не применяется преобразование при фиксации.

Ссылки на статический член, описанный в объявлении обобщенного типа, должны использовать необобщенный тип, соответствующий обобщенному типу (§6.1, §6.5.5.2, §6.5.6.2), иначе генерируется ошибка времени компиляции.

> Другими словами, некорректно ссылаться на статический член, описанный в объявлении обобщенного типа, используя параметризованный тип.

§4.6. Затирание типа

Затирание типа представляет собой отображение типов (возможно, включая параметризованные типы и переменные типа) на типы, которые никогда не являются параметризованными типами или переменными типа. Мы записываем затирание типа T как $|T|$. Отображение затирания определяется следующим образом.

- Затиранием параметризованного типа (§4.5) $G<T_1, \ldots, T_n>$ является $|G|$.
- Затиранием вложенного типа $T.C$ является $|T|.C$.
- Затиранием типа массива $T[]$ является $|T|[]$.
- Затиранием переменной типа (§4.4) является затирание ее левой границы.
- Затиранием любого иного типа является сам этот тип.

Затирание типа также отображает сигнатуру (§8.4.2) конструктора или метода на сигнатуру, которая не имеет параметризованных типов или переменных типа. Затирание сигнатуры конструктора или метода s представляет собой сигнатуру, состоящую из того же имени, что и s, и затираний всех формальных типов параметров из s.

Параметры типа обобщенного конструктора или метода (§8.4.4, §8.8.4) и тип возвращаемого значения (§8.4.5) метода также проходят затирание, если затирается сигнатура конструктора или метода.

Затирание сигнатуры обобщенного метода не имеет параметров типа.

§4.7. Доступные при выполнении типы

Поскольку некоторая информация о типе затирается в процессе компиляции, не все типы доступны во время выполнения. Типы, полностью доступные во время выполнения, имеют специальное название (*reifiable types*).

Тип *доступен во время выполнения* тогда и только тогда, когда выполняется одно из следующих условий.

- Он ссылается на необобщенный класс или интерфейс.
- Это параметризованный тип, в котором все аргументы типа являются неограниченными символами подстановки (§4.5.1).
- Это несформированный (raw) тип (§4.8).
- Это примитивный тип (§4.2).
- Это тип массива (§10.1), тип элемента которого доступен во время выполнения.
- Это вложенный тип, где для всех типов *T*, разделенных точкой (.), тип *T* является доступным во время выполнения.

Например, если обобщенный класс `X<T>` имеет обобщенный член-класс `Y<U>`, то тип `X<?>.Y<?>` доступен во время выполнения, потому что и `X<?>` доступен во время выполнения, и `Y<?>` доступен во время выполнения. Тип `X<?>.Y<Object>` не доступен во время выполнения, потому что `Y<Object>` является типом, недоступным во время выполнения.

Тип пересечения не доступен во время выполнения.

Решение не делать все обобщенные типы доступными во время выполнения является одним из наиболее важных и противоречивых проектных решений в системе типов языка программирования Java.

В конечном итоге наиболее важной мотивацией для этого решения является совместимость с существующим кодом. В некотором "наивном" смысле добавление новых конструкций, таких как обобщенные типы, не имеет последствий для уже существующего кода. Язык программирования Java сам по себе совместим с более ранними версиями, пока каждая программа, написанная на предыдущей версии языка программирования, сохраняет свой смысл в новой версии. Однако понятие, которое можно определить как совместимость версий языка, представляет чисто теоретический интерес. Реальные программы (даже столь тривиальные, как пресловутая "Hello World") состоят из нескольких модулей компиляции, некоторые из которых предоставляются платформой Java SE (такие, как элементы `java.lang` или `java.util`). На практике минимальным требованием является совместимость платформ, чтобы любая программа, написанная для предыдущей версии платформы Java SE, продолжала функционировать в новой версии без изменений.

Один из способов обеспечить совместимость платформ — оставить существующую функциональность платформы без изменений, только добавляя новые функции. Например, вместо того чтобы менять существующую иерархию коллекций в `java.util`, можно ввести новую библиотеку, использующую обобщенность.

Недостатки такой схемы заключаются в том, что для уже существующих клиентов библиотеки коллекций чрезвычайно трудно перейти к новой библиотеке. Коллекции используются для обмена данными между независимо разработанными модулями; если производитель решит переключиться на новые, обобщенные библиотеки, то он должен будет поставлять две версии кода, чтобы обеспечить совместимость с клиентами. Библиотеки, которые зависят от сторонних производителей, не могут быть изменены для использования обобщенных классов до тех пор, пока не будет обновлена библиотека стороннего производителя. Если два модуля являются взаимозависимыми, изменения в них должны быть внесены одновременно.

Очевидно, что совместимость платформ, как указывалось выше, не предоставляет реалистичного пути принятия новой функциональности, такой как обобщенность. Таким образом, дизайн системы обобщенных типов стремится поддерживать миграционную совместимость. Миграционная совместимость допускает такую эволюцию существующего кода, которая позволяет воспользоваться преимуществами обобщенности без установления зависимости между независимо разрабатываемыми программными модулями.

Ценой миграционной совместимости является невозможность полной и надежной доступности системы обобщенных типов во время выполнения; по крайней мере, пока имеет место миграция.

§4.8. Несформированные типы

Для облегчения взаимодействия со старым необобщенным кодом можно использовать как тип затирание (§4.6) параметризованного типа (§4.5) или затирание типа массива (§10.1), элементы которого принадлежат параметризованному типу. Такой тип называется *несформированным* (raw).

Более строго несформированный тип определен как один из следующего списка.

- Ссылочный тип, который образуется из имени объявления обобщенного типа без сопровождающего списка аргументов типа.
- Тип массива, элементы которого принадлежат несформированному типу.
- Нестатический член-тип несформированного типа R, который не наследуется от суперкласса или суперинтерфейса R.

Необобщенные классы или интерфейсы не являются несформированными типами.

Чтобы понять, почему нестатический член-тип несформированного типа считается несформированным, рассмотрим следующий пример.

```
class Outer<T>{
    T t;
    class Inner {
        T setOuterT(T t1) { t = t1; return t; }
    }
}
```

Тип членов `Inner` от параметра типа `Outer`. Если `Outer` не сформирован, `Inner` также должен рассматриваться как несформированный, поскольку для `T` нет корректной привязки.

Это правило применимо только к членам типа, которые не наследуются. Наследуемые члены типа, которые зависят от переменных типа, наследуются как несформированные типы вследствие правила о том, что супертипы несформированного типа затираются, описанного далее в этом разделе.

Еще одно следствие приведенных выше правил состоит в том, что обобщенный внутренний класс несформированного типа сам может использоваться только как несформированный тип.

```
class Outer<T>{
    class Inner<S> {
        S s;
    }
}
```

Невозможно обратиться к `Inner` как к частично несформированному типу.

```
Outer.Inner<Double> x = null; // Неверно
Double d = x.s;
```

Поскольку `Outer` сам по себе не сформирован, то же относится и ко всем его внутренним классам, включая `Inner`, а значит, невозможно передать никакой аргумент типа классу `Inner`.

Суперклассы (соответственно, суперинтерфейсы) несформированного типа являются затираниями суперклассов (суперинтерфейсов) любой параметризации обобщенного типа.

Тип конструктора (§8.8), метода экземпляра (§8.4, §9.4) или нестатического поля (§8.3) *M* несформированного типа *C*, который не наследуется от его суперклассов или суперинтерфейсов, представляет собой несформированный тип, который соответствует затиранию его типа в обобщенном объявлении, соответствующем *C*.

Тип статического метода или статического поля несформированного типа *C* совпадает с его типом в обобщенном объявлении, соответствующем *C*.

Передача аргументов типа нестатическому члену-типу несформированного типа, который не наследуется от его суперклассов или суперинтерфейсов, приводит к ошибке времени компиляции.

К ошибке времени компиляции приводит и попытка использовать член-тип параметризованного типа в качестве несформированного типа.

Это означает, что запрет распространяется и на случай, когда квалифицированный тип является параметризованным, но мы пытаемся использовать внутренний класс как несформированный тип.

```
Outer<Integer>.Inner x = null; // Неверно
```

Это ситуация, противоположная рассмотренной выше. Нет никаких практических оснований для такого "полувыпеченного" типа. В устаревшем коде аргументы типа не используются. В современном коде необходимо корректно использовать обобщенные типы и передавать все необходимые аргументы типов.

Супертип класса может быть несформированным типом. Член, обращающийся к классу, рассматривается как нормальный, а член, обращающийся к супертипу, рассматривается как обращающийся к несформированному типу. В конструкторе вызовы `super` рассматриваются как вызовы метода несформированного типа.

Использование несформированных типов допускается только как уступка для совместимости с кодом прежних версий. Использование несформированных типов в коде, написанном после добавления обобщенных типов в язык программирования Java, не рекомендуется. Вполне возможно, что будущие версии языка программирования Java запретят использование несформированных типов.

Чтобы гарантировать, что потенциальные нарушения правил системы типов всегда будут обнаружены, некоторые обращения к членам несформированных типов ведут к предупреждениям времени компиляции о непроверенных типах (compile-time unchecked warnings). Ниже перечислены правила для таких предупреждений времени компиляции при доступе к членам или конструкторам несформированных типов.

- В присваивании полю: если тип *Primary* в выражении обращения к полю (§15.11) является несформированным типом, то, если затирание изменяет тип поля, генерируется предупреждение времени компиляции.
- В вызове метода или конструктора: если тип искомого класса или интерфейса (§15.12.1) является несформированным, то, если затирание изменяет любой из формальных параметров типа метода или конструктора, генерируется предупреждение времени компиляции.
- Предупреждение времени компиляции не генерируется для вызова метода, когда в случае чтения из поля или при создании экземпляра класса несформированного типа типы формальных параметров не изменяются при затирании (даже если меняется возвращаемый тип и/или конструкция `throws`).

> Обратите внимание, что упомянутые выше предупреждения о непроверенных типах отличаются от предупреждений при непроверенных преобразованиях (§5.1.9), приведениях (§5.5.2), объявлениях методов (§8.4.1, §8.4.8.3, §8.4.8.4, §9.4.1.2) и вызовах методов с переменным количеством аргументов (§15.12.4.2).
>
> Упомянутые здесь предупреждения охватывают случаи, когда устаревший код использует обобщенную библиотеку. Например, библиотека объявляет обобщенный класс `Foo<T extends String>`, имеющий поле `f` типа `Vector<T>`, но пользователь присваивает вектор целых чисел переменной `e.f`, где `e` имеет несформированный тип `Foo`. При компиляции будет получено предупреждение, поскольку это может стать причиной засорения памяти (§4.12.2) для пользователей, использующих обобщенную библиотеку с применением соответствующих возможностей языка программирования.
>
> (Обратите внимание, что старый код может присвоить `Vector<String>` из библиотеки собственной переменной типа `Vector` без каких бы то ни было предупреждений. То есть правила создания подтипов (§4.10.2) языка программирования Java позволяют присваивать переменной несформированного типа значение любого экземпляра параметризованного типа.)
>
> Предупреждения от непроверенного преобразования охватывают дуальный случай, когда обобщенный пользовательский код использует устаревшие библиотеки. Например, метод библиотеки имеет несформированный возвращаемый тип `Vector`, но пользовательский код присваивает результат вызова метода переменной типа `Vector<String>`. Это небезопасно, поскольку несформированный вектор может иметь элементы типа, отличного от `String`; тем не менее такое непроверенное преобразование все еще разрешено для взаимодействия с устаревшим кодом. Предупреждение от непроверенного преобразования указывает, что у обобщенного пользовательского кода могут возникнуть проблемы засорения кучи в других местах программы.

Пример 4.8-1. Несформированные типы

```
class Cell<E> {
    E value;
    Cell(E v)      { value = v; }
    E get()        { return value; }
    void set(E v) { value = v; }

    public static void main(String[] args) {
        Cell x = new Cell<String>("abc");
        System.out.println(x.value); // OK, имеет тип Object
        System.out.println(x.get()); // OK, имеет тип Object
        x.set("def");    // Предупреждение о непроверенности
    }
}
```

Пример 4.8-2. Несформированные типы и наследование

```
import java.util.*;
class NonGeneric {
    Collection<Number> myNumbers() { return null; }
}

abstract class RawMembers<T> extends NonGeneric
                              implements Collection<String> {
    static Collection<NonGeneric> cng =
               new ArrayList<NonGeneric>();
    public static void main(String[] args) {
        RawMembers rw = null;
        Collection<Number> cn = rw.myNumbers();
                   // OK
        Iterator<String> is = rw.iterator();
                   // Предупреждение о непроверенности
        Collection<NonGeneric> cnn = rw.cng;
                   // OK, статический член
    }
}
```

В этой (не предназначенной для выполнения) программе `RawMembers<T>` **наследует метод**

```
Iterator<String> iterator()
```

от суперинтерфейса `Collection<String>`. **Несформированный тип** `RawMembers` **наследует** `iterator()` **от** `Collection`, **затирания** `Collection<String>`, **что означает, что возвращаемый тип метода** `iterator()` **в** `RawMembers` **является** `Iterator`. **В результате попытка присвоения** `rw.iterator()` **переменной типа** `Iterator<String>` **требует непроверенного преобразования, что приводит к предупреждению времени компиляции о непроверенности.**

Напротив, `RawMembers` **наследует** `myNumbers()` **от класса** `NonGeneric`, **затиранием которого также является** `NonGeneric`. **Таким образом, возвраща**

емый тип `myNumbers()` в `RawMembers` не затерт, и попытка присваивания `rw.myNumbers()` переменной типа `Collection<Number>` не требует непроверенного преобразования, так что никакого предупреждения времени компиляции о непроверенности нет.

Аналогично статический член `cng` сохраняет свой параметризованный тип даже при доступе через объект несформированного типа. Обратите внимание, что доступ к статическому члену через экземпляр класса считается плохим стилем и не одобряется.

Этот пример показывает, что некоторые члены несформированного типа не затираются, а именно — `static`-члены, типы которых являются параметризованными, и члены, унаследованные от необобщенных супертипов.

Несформированные типы тесно связаны с символами подстановки. И те, и другие основаны на экзистенциальных типах. Несформированные типы можно рассматривать как символы подстановки, правила для типов которых преднамеренно "испорчены", чтобы обеспечить взаимодействие с устаревшим кодом. Исторически несформированные типы предшествовали символам подстановки; впервые они были введены в GJ и описаны в статье *Making the future safe for the past: Adding Genericity to the Java Programming Language* Жилада Брача (Gilad Bracha), Мартина Одерски (Martin Odersky), Дэвида Стутамира (David Stoutamire) и Филипа Уадлера (Philip Wadler) в *Proceedings of the ACM Conference on Object-Oriented Programming. Systems, Languages and Applications (OOPSLA 98)* (октябрь 1998).

§4.9. Типы пересечений

Тип пересечения имеет вид $T_1 \& \ldots \& T_n$ ($n > 0$), где T_i ($1 \le i \le n$) представляют собой типы.

Типы пересечений могут быть выведены из границ параметров типа (§4.4) и выражений приведения (§15.16); они также возникают в процессах преобразования при фиксации (§5.1.10) и вычисления наименьшей верхней границы (§4.10.4).

Значениями типа пересечения являются те объекты, которые являются значениями всех типов T_i для $1 \le i \le n$.

Каждый тип пересечения $T_1 \& \ldots \& T_n$ вводит воображаемый класс или интерфейс с целью идентификации членов типа пересечения следующим образом.

- Пусть для каждого T_i ($1 \le i \le n$) C_i является наиболее определенным типом класса или массива, таким, что $T_i <: C_i$. Тогда должен существовать некоторый $T_k <: C_k$, такой, что $C_k <: C_i$ для любого i ($1 \le i \le n$). В противном случае генерируется ошибка времени компиляции.
- Если для $1 \le j \le n$ T_j является переменной типа, то пусть T_j' представляет собой интерфейс, члены которого такие же, как открытые члены T_j; в противном случае, если T_j является интерфейсом, пусть T_j' представляет собой T_j.
- Если C_k представляет собой `Object`, выводится воображаемый интерфейс; в противном случае выводится воображаемый класс с непосредственным суперклассом C_k.

Этот класс или интерфейс имеет непосредственные суперинтерфейсы $T'_1,\ldots,T'_n$ и объявлен в том же пакете, в котором находится и тип пересечения.

Члены типа пересечения являются членами введенного класса или интерфейса.

> Стоит остановиться на различии между типами пересечения и границами переменных типа. Каждая граница переменной типа неизбежно приводит к типу пересечения. Этот тип пересечения часто является тривиальным (т.е. состоит из одного типа). Вид границ лимитирован (только первый элемент может быть типом класса или переменной типа, и в границе может быть только одна переменная типа), чтобы исключить некоторые неловкие ситуации. Однако преобразование при фиксации может привести к созданию переменных типа, границы которых носят более общий характер (например, типы массивов).

§4.10. Создание подтипов

Отношения подтипа и супертипа являются бинарным отношением над типами.

Супертипы типа получаются путем рефлексивного и транзитивного замыкания над отношением непосредственного (direct) супертипа, что записывается как $S >_1 T$, и определяются правилами, приводимыми ниже в этом разделе. Чтобы указать, что между S и T выполняется отношение супертипа, записываем $S :> T$.

S является *истинным* (proper) *супертипом* T, что записывается как $S > T$, если $S :> T$ и $S \neq T$.

Подтипами типа T являются все типы U, такие, что T является супертипом U, а также тип null. Мы записываем $T <: S$, чтобы указать, что между подтипами T и S имеется соотношение подтипа.

T является *истинным* (proper) *подтипом* S, что записывается как $T < S$, если $T <: S$ и $S \neq T$.

T является *непосредственным подтипом* S, что записывается как $T <_1 S$, если $S >_1 T$.

Отношения типов не распространяются через параметризованные типы: из $T <: S$ не следует, что $C<T> <: C<S>$.

§4.10.1. Подтипы среди примитивных типов

Следующие правила определяют отношения непосредственного супертипа среди примитивных типов.

- `double` $>_1$ `float`
- `float` $>_1$ `long`
- `long` $>_1$ `int`
- `int` $>_1$ `char`
- `int` $>_1$ `short`
- `short` $>_1$ `byte`

§4.10.2. Подтипы среди типов классов и интерфейсов

Для данного объявления необобщенного типа *C непосредственными супертипами C* является все перечисленное далее.

- Непосредственный суперкласс *C* (§8.1.4).
- Непосредственные суперинтерфейсы *C* (§8.1.5).
- Тип `Object`, если *C* представляет собой тип интерфейса без непосредственных суперинтерфейсов (§9.1.3).

Для данного объявления обобщенного типа $C<F_1, \ldots, F_n>$ $(n>0)$ *непосредственными супертипами* несформированного типа *C* (§4.8) является все перечисленное далее.

- Непосредственные суперклассы несформированного типа *C*.
- Непосредственные суперинтерфейсы несформированного типа *C*.
- Тип `Object`, если $C<F_1, \ldots, F_n>$ представляет собой тип обобщенного интерфейса без непосредственных суперинтерфейсов (§9.1.2).

Для данного объявления обобщенного типа $C<F_1, \ldots, F_n>$ $(n>0)$ *непосредственными супертипами* обобщенного типа $C<F_1, \ldots, F_n>$ (§4.8) является все перечисленное далее.

- Непосредственные суперклассы $C<F_1, \ldots, F_n>$.
- Непосредственные интерфейсы $C<F_1, \ldots, F_n>$.
- Тип `Object`, если $C<F_1, \ldots, F_n>$ представляет собой тип обобщенного интерфейса без непосредственных суперинтерфейсов.
- Несформированный тип *C*.

Для данного объявления обобщенного типа $C<F_1, \ldots, F_n>$ $(n>0)$ *непосредственными супертипами* параметризованного типа $C<T_1, \ldots, T_n>$, где T_i $(1 \le i \le n)$ представляют собой типы, является все перечисленное далее.

- $D<U_1\ \theta, \ldots, U_k\ \theta>$, где $D<U_1, \ldots, U_k>$ является непосредственным супертипом $C<T_1, \ldots, T_n>$, а θ является подстановкой $[F_1 := T_1, \ldots, F_n := T_n]$.
- $C<S_1, \ldots, S_n>$, где S_i содержит T_i $(1 \le i \le n)$ (§4.5.1).
- Тип `Object`, если $C<F_1, \ldots, F_n>$ представляет собой тип обобщенного интерфейса без непосредственных суперинтерфейсов.
- Несформированный тип *C*.

Для данного объявления обобщенного типа $C<F_1, \ldots, F_n>$ $(n>0)$ *непосредственными супертипами* параметризованного типа $C<R_1, \ldots, R_n>$, где как минимум одно из R_i $(1 \le i \le n)$ представляет собой аргумент типа с символом подстановки, являются непосредственные супертипы параметризованного типа $C<X_1, \ldots, X_n>$, которые представляют собой результат применения преобразования при фиксации (§5.1.10) к $C<R_1, \ldots, R_n>$.

Непосредственными супертипами типа пересечения $T_1\ \&\ \ldots\ \&\ T_n$ являются T_i $(1 \le i \le n)$.

Непосредственными супертипами переменной типа являются типы, перечисленные в ее границе.

Переменная типа является непосредственным супертипом ее нижней границы.

Непосредственными супертипами типа null являются все ссылочные типы, отличные от самого типа null.

§4.10.3. Подтипы среди типов массивов

Приведенные далее правила определяют отношение непосредственного супертипа среди типов массивов.

- Если и S, и T являются ссылочными типами, то $S[] >_1 T[]$ тогда и только тогда, когда $S >_1 T$.
- `Object` $>_1$ `Object[]`
- `Cloneable` $>_1$ `Object[]`
- `java.io.Serializable` $>_1$ `Object[]`
- Если P представляет собой примитивный тип, то
  - `Object` $>_1$ `P[]`
  - `Cloneable` $>_1$ `P[]`
  - `java.io.Serializable` $>_1$ `P[]`

§4.10.4. Наименьшая верхняя граница

Наименьшая верхняя граница (*least upper bound*, или "lub") множества ссылочных типов представляет собой разделяемый супертип, более конкретный, чем любой иной разделяемый супертип (т.е. ни один другой разделяемый супертип не является подтипом наименьшей верхней границы). Этот тип, $\mathrm{lub}(U_1,\ldots,U_k)$, определяется следующим образом.

Если $k = 1$, то lub представляет собой сам тип: $\mathrm{lub}(U) = U$.

В противном случае

- Для каждого U_i $(1 \le i \le k)$:

  пусть $\mathrm{ST}(U_i)$ представляет собой множество супертипов U_i;

  пусть множество $\mathrm{EST}(U_i)$ затертых супертипов U_i представляет собой $\mathrm{EST}(U_i) = \{|W| \mid W \in \mathrm{ST}(U_i)\}$, где $|W|$ представляет собой затирание W.

> **Причина вычисления множества затертых супертипов — работа с ситуациями, когда множество типов включает несколько различных параметризаций обобщенного типа.**
>
> **Например, для данных `List<String>` и `List<Object>` простое пересечение множеств ST(`List<String>`) = {`List<String>`, `Collection<String>`, `Object`} и ST(`List<Object>`) = {`List<Object>`, `Collection<Object>`, `Object`} дает множество {`Object`}, и при этом теряется тот факт, что можно безопасно принять в качестве верхней границы `List`.**
>
> **И напротив, пересечение EST(`List<String>`) = {`List`, `Collection`, `Object`} и EST(`List<Object>`) = {`List`, `Collection`, `Object`} дает {`List`, `Collection`, `Object`}, что в конечном счете позволяет нам получить `List<?>`.**

- Пусть множество затертых кандидатов EC для $U_1 \ldots U_k$ представляет собой пересечение всех множеств $\mathrm{EST}(U_i)$ $(1 \le i \le k)$.
- Пусть минимальное множество затертых кандидатов MEC для $U_1 \ldots U_k$ представляет собой

  $$\mathrm{MEC} = \{ V \mid V \in \mathrm{EC} \text{ и для всех } W \neq V \text{ из EC не выполняется } W <: V \}$$

  Поскольку мы стремимся вывести более точные типы, мы хотим отфильтровать все кандидаты, являющиеся супертипами других кандидатов. Это вычисление выполняет MEC. В нашем примере EC = {`List`, `Collection`, `Object`}, так что MEC = {`List`}. Следующим шагом является восстановление аргументов типа для затертых типов из MEC.

- Пусть для любого элемента *G* из MEC, который является обобщенным типом, "релевантная" параметризация *G*, $\mathrm{Relevant}(G)$, представляет собой

  $$\mathrm{Relevant}(G) = \{ V \mid 1 \le i \le k : \; V \in \mathrm{ST}(U_i) \text{ и } V = G<\ldots> \}.$$

  В нашем примере единственным обобщенным элементом MEC является `List`, и Inv(`List`) = { `List<String>`, `List<Object>` }. Теперь мы будем искать аргумент типа для `List`, который содержит (§4.5.1) как `String`, так и `Object`.

  Это делается с помощью определенной ниже операции наименьшей содержащей параметризации (least containing parameterization — lcp). Первая строка определяет lcp() на множестве, таком как Relevant(`List`), как операцию над списком элементов множества. Следующая строка определяет операцию над такими списками, как попарное сокращение элементов списка. Третья строка является определением lcp() на парах параметризованных типов, которое, в свою очередь, опирается на понятие наименьшего содержащего аргумента типа (least containing type argument — lcta). lcta() определяется для всех возможных случаев.

Пусть "кандидат" в параметризацию *G*, $\mathrm{Candidate}(G)$, представляет собой наиболее конкретную параметризацию обобщенного типа *G*, которая содержит все релевантные параметризации *G*:

$$\mathrm{Candidate}(G) = \mathrm{lcp}(\mathrm{Relevant}(G))$$

Здесь lcp(), наименьшая содержащая параметризация, представляет собой

- $\mathrm{lcp}(S) = \mathrm{lcp}(e_1,\ldots,e_n)$, где e_i $(1 \le i \le n)$ входят в S
- $\mathrm{lcp}(e_1,\ldots,e_n) = \mathrm{lcp}(\mathrm{lcp}(e_1,e_2),e_3,\ldots,e_n)$
- $\mathrm{lcp}(G<X_1,\ldots,X_n>, G<Y_1,\ldots,Y_n>) = G<\mathrm{lcta}(X_1,Y_1),\ldots,\mathrm{lcta}(X_n,Y_n)>$
- $\mathrm{lcp}(G<X_1,\ldots,X_n>) = G<\mathrm{lcta}(X_1),\ldots,\mathrm{lcta}(X_n)>$

и где lcta(), наименьший содержащий аргумент типа, определяется следующим образом (в предположении, что *U* и *V* представляют собой типы) :

- $\mathrm{lcta}(U,V) = U$, если $U = V$, в противном случае `? extends` $\mathrm{lub}(U,V)$
- $\mathrm{lcta}(U,$`? extends` $V) =$ `? extends` $\mathrm{lub}(U,V)$
- $\mathrm{lcta}(U,$`? super` $V) =$ `? super` $\mathrm{glb}(U,V)$
- $\mathrm{lcta}($`? extends` $U,$ `? extends` $V) =$ `? extends` $\mathrm{lub}(U,V)$

- ✦ lcta(`? extends` *U*, `? super` *V*) = *U*, если *U* = *V*, в противном случае `?`
- ✦ lcta(`? super` *U*, `? super` *V*) = `? super` glb(*U*,*V*)
- ✦ lcta(*U*) = `?`, если верхней границей *U* является `Object`, в противном случае `? extends` lub(*U*,`Object`), и где glb() определено в §5.1.10.

- Пусть $\text{lub}(U_1 \ldots U_k)$ представляет собой

  $\text{Best}(W_1) \& \ldots \& \text{Best}(W_r)$

  Здесь W_i $(1 \le i \le r)$ являются элементами MEC, минимального множества затертых кандидатов $U_1 \ldots U_k$;

  и, если любой из этих элементов обобщенный, мы используем параметризацию-кандидат (как и для восстановления аргументов типа):

  $\text{Best}(X) = \text{Candidate}(X)$, если X — обобщенный; в противном случае — X.

Строго говоря, функция lub() только аппроксимирует наименьшую верхнюю границу. Формально может существовать некоторый другой тип T, такой, что все $U_1 \ldots U_k$ являются подтипами T, и T является подтипом $\text{lub}(U_1,\ldots,U_k)$. Однако компилятор языка программирования Java должен реализовывать lub() как указано выше.

Возможна ситуация, когда функция lub() дает бесконечный тип. Это разрешено, и компилятор языка программирования Java должен распознавать такие ситуации и соответствующим образом представлять их с помощью циклических структур данных.

> Возможность бесконечного типа вытекает из рекурсивных вызовов lub(). Читатели, знакомые с рекурсивными типами, должны заметить, что бесконечный тип — это не то же самое, что и рекурсивный тип.

§4.11. Где используются типы

Типы используются в большинстве видов объявлений и в некоторых видах выражений. Конкретнее, имеется 16 *контекстов типов*, в которых используются типы.

- В объявлениях
  1. Тип в конструкции `extends` или `implements` объявления класса (§8.1.4, §8.1.5, §8.5, §9.5)
  2. Тип в конструкции `extends` объявления интерфейса (§9.1.3, §8.5, §9.5)
  3. Возвращаемый тип метода (включая тип элемента типа аннотации) (§8.4.5, §9.4, §9.6.1)
  4. Тип в конструкции `throws` метода или конструктора (§8.4.6, §8.8.5, §9.4)
  5. Тип в конструкции `extends` объявления параметра типа обобщенного класса, интерфейса, метода или конструктора (§8.1.2, §9.1.2, §8.4.4, §8.8.4)
  6. Тип в объявлении поля класса или интерфейса (включая константу перечисления) (§8.3, §9.3, §8.9.1)
  7. Тип в объявлении формального параметра метода, конструктора или лямбда-выражения (§8.4.1, §8.8.1, §9.4, §15.27.1)

8. Тип параметра-получателя метода (см. §8.4.1 — специальный параметр метода, который ссылается на объект, метод которого вызывается)
9. Тип в объявлении локальной переменной (§14.4, §14.14.1, §14.14.2, §14.20.3)
10. Тип в объявлении параметра исключения (§14.20)

- В выражениях

11. Тип в списке аргументов явных типов инструкции явного вызова конструктора, выражения создания экземпляра класса или выражении вызова метода (§8.8.7.1, §15.9, §15.12)
12. В выражении создания экземпляра неквалифицированного класса в качестве типа инстанцируемого класса (§15.9) или в качестве непосредственного суперкласса или непосредственного суперинтерфейса инстанцируемого анонимного класса (§15.9.5)
13. Тип элемента в выражении создания массива (§15.10.1)
14. Тип в операторе приведения или выражении приведения (§15.16)
15. Тип, следующий за оператором отношения `instanceof` (§15.20.2)
16. В выражении ссылки на метод (§15.13), в качестве типа ссылки для поиска метода-члена, в качестве типа создаваемого класса или массива

Типы также используются следующим образом.

- В качестве типа элемента типа массива в любом из перечисленных выше контекстов.
- В качестве аргумента типа без символов подстановки или границы аргумента типа с символом подстановки, или в качестве параметризированного типа в любом из перечисленных выше контекстов.

Наконец имеется три специальных элемента в языке программирования Java, которые описывают использование типа.

- Неограниченный символ подстановки (§4.5.1).
- Троеточие . . . в типе параметра переменной арности (§8.4.1) для указания типа массива.
- Простое имя типа в объявлении конструктора (§8.8) для указания класса конструируемого объекта.

Значение типов в контексте типа описывается в

- §4.2 для примитивных типов;
- §4.4 для параметров типа;
- §4.5 для параметризованных типов класса или интерфейса либо как аргументы типа в параметризованном типе, либо как границы аргументов типа с символами подстановки в параметризованном типе;
- §4.8 для несформированных типов классов и интерфейсов;
- §4.9 для типов пересечений в границах параметров типов;
- §6.5 для типов классов и интерфейсов в контекстах, где обобщенность не играет роли (§6.1);

- §10.1 для типов массивов.

Некоторые контексты типов ограничивают параметризацию ссылочных типов.

- Приведенные далее контексты типов требуют, чтобы тип, если он является параметризованным ссылочным типом, не имел аргументов типов с символами подстановки.
  - ✦ В конструкциях `extends` или `implements` объявления класса (§8.1.4, §8.1.5).
  - ✦ В конструкции `extends` объявления интерфейса (§9.1.3).
  - ✦ В выражении создания экземпляра неквалифицированного класса, в качестве инстанцируемого типа класса (§15.9) или как непосредственный суперкласс или непосредственный суперинтерфейс инстанцируемого анонимного класса (§15.9.5).
  - ✦ В выражении ссылки на метод (§15.13), в качестве ссылочного типа для поиска метода члена или в качестве типа создаваемого класса или массива.

  Кроме того, не разрешены никакие аргументы типов с символами подстановки в списке явных аргументов типа инструкции явного вызова конструктора, выражения создания экземпляра класса, выражения вызова метода или выражения ссылки на метод (§8.8.7.1, §15.9, §15.12, §15.13).
- Следующие контексты типа требуют, чтобы тип, если он является параметризованным ссылочным типом, имел только неограниченные аргументы типа с символами подстановки (т.е. был типом, доступным во время выполнения):
  - ✦ в качестве типа элемента в выражении создания массива (§15.10.1);
  - ✦ в качестве типа, следующего за оператором отношения `instanceof` (§15.20.2).
- Следующие контексты типа запрещают параметризованный ссылочный тип вообще, поскольку они включают исключения, и тип исключения является необобщенным (§6.1):
  - ✦ в качестве типа исключения, которое может быть сгенерировано методом или конструктором (§8.4.6, §8.8.5, §9.4);
  - ✦ в объявлении параметра исключения (§14.20).

В любом контексте типа, где используется тип, возможно аннотировать ключевое слово, обозначающее примитивный тип, или *Identifier*, обозначающий простое имя ссылочного типа. Возможно также аннотировать тип массива путем написания аннотации слева от открывающей квадратной скобки `[` на желаемом уровне вложенности в типе массива. Аннотации в этих местах называются *аннотациями типов* и определены в §9.7.4. Вот несколько примеров.

- `@Foo int[] f;` аннотирует примитивный тип `int`
- `int @Foo [] f;` аннотирует тип массива `int[]`
- `int @Foo [][] f;` аннотирует тип массива `int[][]`
- `int[] @Foo [] f;` аннотирует тип массива `int[]`, который является типом компонента типа массива `int[][]`

Пять из *контекстов типа*, которые встречаются в объявлениях, занимают ту же синтаксическую нишу, что и ряд *контекстов объявлений* (§9.6.4.1).

- Возвращаемый тип метода (включая тип элемента типа аннотации).

- Тип в объявлении поля класса или интерфейса (включая константы перечисления).
- Тип в объявлении формального параметра метода, конструктора или лямбда-выражения.
- Тип в объявлении локальной переменной.
- Тип в объявлении параметра исключения.

Тот факт, что одно и то же синтаксические местоположение в программе может быть как контекстом типа, так и контекстом объявления, возникает потому, что модификаторы объявления непосредственно предшествуют типу объявляемой сущности. В §9.7.4 объясняется, как разобраться, является ли аннотация в таком месте находящейся в контексте типа или в контексте объявления (или в обоих).

Пример 4.11-1. Применение типа

```
import java.util.Random;
import java.util.Collection;
import java.util.ArrayList;
class MiscMath<T extends Number> {
    int divisor;
    MiscMath(int divisor) { this.divisor = divisor; }
    float ratio(long l) {
        try {
            l /= divisor;
        } catch (Exception e) {
            if (e instanceof ArithmeticException)
                l = Long.MAX_VALUE;
            else
                l = 0;
        }
        return (float)l;
    }
    double gausser() {
        Random r = new Random();
        double[] val = new double[2];
        val[0] = r.nextGaussian();
        val[1] = r.nextGaussian();
        return (val[0] + val[1]) / 2;
    }
    Collection<Number> fromArray(Number[] na) {
        Collection<Number> cn = new ArrayList<Number>();
        for (Number n : na) cn.add(n);
        return cn;
    }
    <S> void loop(S s) { this.<S>loop(s); }
}
```

В этом примере типы использованы в следующих объявлениях.

- Импортированных типов (§7.5); здесь это тип `Random`, импортированный из типа `java.util.Random` пакета `java.util`.
- Полей, которые являются переменными класса и переменными экземпляров классов (§8.3), и констант интерфейсов (§9.3); здесь поле `divisor` класса `MiscMath` объявляется как имеющее тип `int`.
- Параметров методов (§8.4.1); здесь параметр `l` метода `ratio` объявлен как имеющий тип `long`.
- Результатов методов (§8.4); здесь результат метода `ratio` объявлен как имеющий тип `float`, а результат метода `gausser` объявлен как имеющий тип `double`.
- Параметров конструкторов (§8.8.1); здесь параметр конструктора `MiscMath` объявлен как имеющий тип `int`.
- Локальных переменных (§14.4, §14.14); локальные переменные `r` и `val` метода `gausser` объявлены как имеющие тип `Random` и `double[]` (массив `double`).
- Параметров исключений (§14.20); здесь параметр исключения `e` конструкции `catch` объявлен как имеющий тип `Exception`.
- Параметров типа (§4.4); здесь параметр типа `MiscMath` является переменной типа `T` с типом `Number` в качестве объявленной границы.
- В любом объявлении, которое использует параметризованный тип; здесь тип `Number` используется в качестве аргумента типа (§4.5.1) в параметризованном типе `Collection<Number>`.

Кроме того, типы использованы в выражениях следующих видов.

- Создания экземпляра класса (§15.9); здесь локальная переменная `r` метода `gausser` инициализируется путем выражения создания экземпляра класса, которое использует тип `Random`.
- Создания экземпляра (§15.9) обобщенного класса (§8.1.2); здесь `Number` используется в качестве аргумента типа в выражении `new ArrayList<Number>()`.
- Создания массивов (§15.10.1); здесь локальная переменная `val` метода `gausser` инициализируется с помощью выражения создания массива, которое создает массив `double` размером 2.
- Вызова (§15.12) обобщенного метода (§8.4.4) или конструктора (§8.8.4); здесь метод `loop` вызывает сам себя с явным аргументом типа `S`.
- Приведения (§15.16); здесь оператор `return` метода `ratio` использует тип `float` в приведении.
- Оператора `instanceof` (§15.20.2); здесь оператор `instanceof` проверяет, является ли `e` совместимым по присваиванию с типом `ArithmeticException`.

§4.12. Переменные

Переменная представляет собой место в памяти для хранения информации, имеющее связанный с ним тип (иногда именуемый *типом времени компиляции*), являющийся либо примитивным (§4.2), либо ссылочным типом (§4.3).

Значение переменной изменяется с помощью присваивания (§15.26) или с помощью префиксного или постфиксного оператора ++ (инкремент) или -- (декремент) (§15.14.2, §15.14.3, §15.15.1, §15.15.2).

Совместимость значения переменной с ее типом гарантируется дизайном языка программирования Java (пока при компиляции программы не выдается предупреждение о невозможности проверки типа (§4.12.2)). Значения по умолчанию (§4.12.5) являются совместимыми и все присваивания переменной проверяются на совместимость по присваиванию (§5.2), обычно во время компиляции (однако в одном случае с участием массивов проверка осуществляется во время выполнения (§10.5)).

§4.12.1. Переменные примитивного типа

Переменная примитивного типа всегда хранит примитивное значение в точности указанного примитивного типа.

§4.12.2. Переменные ссылочного типа

Переменная типа класса T может содержать пустую ссылку или ссылку на экземпляр класса T или любого класса, который является подклассом T.

Переменная типа интерфейса может содержать пустую ссылку или ссылку на любой экземпляр любого класса, который реализует этот интерфейс.

> Обратите внимание, что не гарантируется, что переменная всегда ссылается на подтип своего объявленного типа, но только на подклассы или подынтерфейсы объявленного типа. Это связано с возможностью замусоривания кучи, обсуждающейся ниже.

Если T является примитивным типом, то переменная типа "массив T" может содержать пустую ссылку или ссылку на любой массив типа "массив T".

Если T является ссылочным типом, то переменная типа "массив T" может содержать пустую ссылку или ссылку на любой массив типа "массив S", такой, что тип S является подклассом или подынтерфейсом типа T.

Переменная типа `Object[]` может содержать ссылку на массив любого ссылочного типа.

Переменная типа `Object` может содержать пустую ссылку или ссылку на любой объект, будь то экземпляр класса или массив.

Возможна ситуация, когда переменная параметризованного типа ссылается на объект, который не принадлежит параметризованному типу. Такая ситуация известна как *замусоривание кучи* (heap pollution).

Замусоривание кучи может произойти, только если программа выполняет некоторые операции с участием несформированного типа, которые приводят к выводу предупрежде-

ния о невозможности проверки времени компиляции (§4.8, §5.1.9, §5.5.2, §8.4.1, §8.4.8.3, §8.4.8.4, §9.4.1.2, §15.12.4.2), или если программа работает с переменной массива, тип элементов которого недоступен во время выполнения, с помощью переменной массива супертипа, который либо не сформирован, либо не является обобщенным.

Например, код

```
List l = new ArrayList<Number>();
List<String> ls = l; // Предупреждение о непроверенности
```

приводит к предупреждению о невозможности проверки во время компиляции, поскольку невозможно установить, как во время компиляции (в пределах правил проверки типов во время компиляции), так и во время выполнения, ссылается ли в действительности переменная `l` на `List<String>`.

При выполнении приведенного выше кода возникает загрязнение кучи, поскольку переменная `ls`, объявленная как `List<String>`, ссылается на значение, которое в действительности `List<String>` не является.

Проблему нельзя идентифицировать и во время выполнения, поскольку переменные типа недоступны во время выполнения, и, таким образом, во время выполнения экземпляры не несут никакой информации об аргументах типа, использованных для их создания.

В простом примере, таком, как приведенный выше, может показаться, что выявить такую ситуацию во время компиляции и сообщить об ошибке проще простого. Однако в общем (и типичном) случае значение переменной `l` может быть результатом вызова отдельно скомпилированного метода или его значение может зависеть от некоторого потока управления. Приведенный выше код поэтому весьма нетипичен и демонстрирует очень плохой стиль.

Кроме того, тот факт, что супертипом всех типов массивов является `Object[]`, означает, что может произойти небезопасное совмещение имен, ведущее к замусориванию кучи. Например, следующий код компилируется, поскольку он статически корректен с точки зрения типов.

```
static void m(List<String>... stringLists) {
    Object[] array = stringLists;
    List<Integer> tmpList = Arrays.asList(42);
    array[0] = tmpList;                    // (1)
    String s = stringLists[0].get(0);  // (2)
}
```

Замусоривание кучи происходит в (1), поскольку компонент массива `stringLists`, который должен ссылаться на `List<String>`, теперь ссылается на `List<Integer>`. Нет никакого способа обнаружить это замусоривание при наличии как универсального супертипа (`Object[]`), так и недоступного во время выполнения типа (объявленный тип формального параметра, `List<String>[]`). В (1) не генерируется никакое предупреждение о непроверенных типах; тем не менее во время выполнения в (2) генерируется исключение `ClassCastException`.

Предупреждение времени компиляции о непроверенных типах будет выдаваться при любом вызове приведенного выше метода, поскольку для создания массива с элементами типа `List<String>`, являющегося недоступным во время выполнения, вызов рассматривается статической системой типов языка программирования Java

(§15.12.4.2). *Тогда и только тогда, когда* тело метода безопасно в смысле типов по отношению к переменному количеству параметров, программист может использовать аннотацию `SafeVarargs`, чтобы убрать предупреждения при вызовах (§9.6.4.7). Поскольку тело метода, как написано выше, приводит к замусориванию кучи, использовать аннотацию для отключения предупреждений совершенно неуместно.

Наконец обратите внимание, что обращение к массиву `stringLists` может осуществляться через переменные типов, отличных от `Object[]`, так что все еще может осуществиться замусоривание кучи. Например, типом переменной `array` может быть `java.util.Collection[]` (несформированный тип элемента), и при этом тело приведенного выше метода будет компилироваться без предупреждений или ошибок времени компиляции и по-прежнему приводить к замусориванию кучи. И если платформа Java SE определяет, скажем, `Sequence` как необобщенный супертип для `List<T>`, то использование `Sequence` в качестве типа `array` также приведет к замусориванию кучи.

Переменная всегда будет ссылаться на объект, являющийся экземпляром класса, который представляет параметризованный тип.

Значение `ls` в приведенном выше примере всегда является экземпляром класса, который предоставляет представление `List`.

Присваивание переменной параметризованного типа выражения несформированного типа должно использоваться только при объединении устаревшего кода, который не использует параметризованные типы, с более современный кодом, который это делает.

Если нет никаких операций, требующих вывода предупреждения времени компиляции о непроверенных типах, и нет небезопасного совмещения имен переменных массивов с типами элементов, недоступными во время выполнения, тогда замусоривание кучи произойти не может. Обратите внимание, что это не означает, что замусоривание кучи происходит только тогда, когда во время компиляции выдается предупреждение о непроверенных типах. Можно запустить программу, в которой некоторые из бинарных файлов сгенерированы компилятором для старой версии Java, или из исходных текстов, в которых явно подавлены предупреждения о непроверенных типах. Такая практика в лучшем случае заслуживает эпитета "нездоровая".

И обратно, вполне возможно, что несмотря на то, что код мог бы привести (а возможно, и привел) к предупреждениям времени компиляции о непроверенных типах, при его работе никакого замусоривания кучи не происходит. В действительности хорошая практика программирования требует, чтобы программист убедился, что несмотря на любые возможные предупреждения код корректен и никакие неприятности наподобие замусоривания кучи при его выполнении не произойдут.

§4.12.3. Виды переменных

Имеется восемь видов переменных.

1. *Переменная класса* представляет собой поля, объявленные с помощью ключевого слова `static` внутри объявления класса (§8.3.1.1) или (с ключевым словом `static` или без него) в объявлении интерфейса (§9.3).

Переменная класса создается, когда подготавливается ее класс или интерфейс (§12.3.2), и инициализируется значением по умолчанию (§4.12.5). Переменная класса перестает фактически существовать при выгрузке (§12.7) ее класса или интерфейса.

2. *Переменная экземпляра* представляет собой поле, объявленное внутри объявления класса без использования ключевого слова `static` (§8.3.1.1).

   Если класс *T* имеет поле *a*, являющееся переменной экземпляра, то новая переменная экземпляра *a* создается и инициализируется значением по умолчанию (§4.12.5) в рамках каждого вновь создаваемого объекта класса *T* или любого класса, который является подклассом *T* (§8.1.4). Переменная экземпляра перестает фактически существовать, когда больше не имеется ссылок на объект, которому принадлежит это поле, после завершения любой необходимой финализации объекта (§12.6).

3. *Компоненты массива* представляют собой неименованные переменные, которые создаются и инициализируются значениями по умолчанию (§4.12.5) всякий раз, когда создается новый объект, представляющий собой массив (§10, §15.10.2). Компоненты массива перестают фактически существовать, когда на массив больше нет ссылок.

4. *Параметры метода* (§8.4.1) представляют собой именованные значения аргументов, передаваемых методу.

   Для каждого параметра, объявленного в объявлении метода, новая переменная создается всякий раз, когда вызывается метод (§15.12). Новая переменная инициализируется соответствующим значением аргумента из вызова метода. Параметр метода перестает фактически существовать после завершения выполнения тела метода.

5. *Параметры конструктора* (§8.8.1) представляют собой именованные значения аргументов, передаваемых конструктору.

   Для каждого параметра, объявленного в объявлении конструктора, новая переменная создается всякий раз, когда этот конструктор вызывается выражением создания экземпляра класса (§15.9) или явным вызовом конструктора (§8.8.7). Новая переменная инициализируется соответствующим значением аргумента из выражения создания экземпляра класса или вызова конструктора. Параметр конструктора перестает фактически существовать по завершении выполнения тела конструктора.

6. *Параметры лямбда-выражений* (§15.27.1) представляют собой именованные значения аргументов, передаваемые телу лямбда-выражения (§15.27.2).

   Для каждого параметра, объявленного в лямбда-выражении, новая переменная параметра создается всякий раз, когда вызывается метод, реализованный телом лямбда-выражения (§15.12). Новая переменная инициализируется соответствующим значением аргумента из вызова метода. Лямбда-параметр перестает фактически существовать, когда завершается выполнение тела лямбда-выражения.

7. *Параметр исключения* создается всякий раз, когда исключение перехватывается инструкцией `catch` конструкции `try` (§14.20).

   Новая переменная инициализируется фактическим объектом, связанным с исключением (§11.3, §14.18). Параметр исключения перестает фактически существовать после завершения выполнения блока, связанного с инструкцией `catch`.

8. *Локальные переменные*, объявленные в конструкции объявления локальных переменных (§14.4).

   При входе потока управления в блок (§14.2) или цикл `for` (§14.14) для каждого объявления локальной переменной в соответствующей конструкции, непосредственно содержащейся в этом блоке или цикле `for`, создается новая переменная.

   Конструкция объявления локальной переменной может содержать выражение, инициализирующее эту переменную. Локальная переменная с инициализирующим выражением не инициализируется до тех пор, пока не будет выполнена конструкция объявления локальной переменной. (Привила определенного присваивания (§16) предохраняют значение локальной переменной от использования до инициализации или иного присваивания значения.) Локальная переменная перестает фактически существовать после завершения выполнения блока или цикла `for`.

Кроме одной исключительной ситуации, локальная переменная всегда может рассматриваться как создаваемая при выполнении соответствующего объявления локальной переменной. Исключительная ситуация связана с конструкцией `switch` (§14.11), где управление может входить в блок, минуя выполнение объявления локальной переменной. Однако в силу ограничений, налагаемых правилами определенного присваивания (§16), локальная переменная, объявленная в таком "обойденном" объявлении локальной переменной, не может использоваться до того, как получит значение с помощью выражения присваивания (§15.26).

Пример 4.12.3-1. Различные виды переменных

```
class Point {
    static int numPoints;  // numPoints – переменная класса
    int x, y;              // x и y - переменные экземпляра
    int[] w = new int[10]; // w[0] - компонент массива
    int setX(int x) {      // x - параметр метода
        int oldx = this.x; // oldx - локальная переменная
        this.x = x;
        return oldx;
    }
}
```

§4.12.4. Переменные `final`

Переменная может быть объявлена с применением ключевого слова `final`. Такой "окончательной" переменной значение может присваиваться только один раз. Если `final`-переменной присваивается значение (за исключением ситуации, когда до этого

значение переменной определенно не присвоено (§16)), генерируется ошибка времени компиляции.

После присваивания значения переменной `final` она всегда содержит одно и то же значение. Если переменная `final` хранит ссылку на объект, то состояние этого объекта может быть изменено с помощью операции над объектом, но переменная всегда будет ссылаться на тот же объект. Это относится также к массивам, поскольку массивы являются объектами. Если переменная `final` содержит ссылку на массив, то компоненты массива могут быть изменены с помощью операций над массивом, но переменная всегда будет ссылаться на один и тот же массив.

Пустой `final`-переменной является переменная, объявленная с ключевым словом `final`, но без инициализатора.

Переменная примитивного типа или типа `String`, которая объявлена как `final` и инициализируется константным выражением времени компиляции (§15.28), называется *константной переменной*. Является ли переменная константной, может иметь последствия по отношению к инициализации класса (§12.4.1), бинарной совместимости (§13.1, §13.4.9) и определенному присваиванию (§16).

Три вида переменных неявно объявляются как `final`: поле интерфейса (§9.3), локальная переменная, которая представляет собой ресурс конструкции `try` с ресурсами (§14.20.3), и параметр исключения блока мульти-`catch` (§14.20). Параметр исключения блока `catch` (§14.20) никогда не объявляется как `final` неявно, но может быть *фактически финальным*.

Пример 4.12.4-1. Переменные `final`

Объявление переменной `final` может служить как полезное документирование того, что ее значение не изменится. Это может помочь избежать ошибок при программировании.

```
class Point {
    int x, y;
    int useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
```

В этой программе класс `Point` объявляет `final`-переменную класса `origin`. Переменная `origin` хранит ссылку на объект, являющийся экземпляром класса `Point` с координатами (0,0). Значение переменной `Point.origin` никогда не изменится, поэтому она всегда ссылается на один и тот же объект `Point`, созданный ее инициализатором. Однако операция над указанным объектом `Point` может изменить его состояние, например изменить значение его поля `useCount` или даже ошибочно изменить его координаты `x` и `y`.

Некоторые переменные, которые не объявлены как `final`, тем не менее могут рассматриваться как *фактически финальные* (effectively final).

Локальная переменная или параметр метода, конструктора, лямбда-выражения или исключения является *фактически финальной*, если она не объявлена как `final`, но никогда не встречается в качестве левого операнда оператора присваивания (§15.26) или в

качестве операнда префиксного или постфиксного оператора инкремента или декремента (§15.14, §15.15).

Кроме того, локальная переменная, в объявлении которой отсутствует итератор, является *фактически финальной*, если истинны все перечисленные ниже условия.

- Она не объявлена как `final`.
- Когда она встречается в качестве левого операнда оператора присваивания, она определенно не присвоена и не является определенно присвоенной перед присваиванием; т.е. она определенно не присвоена и не является определенно присвоенной после правого операнда присваивания (§16).
- Она никогда не встречается в качестве операнда префиксного или постфиксного оператора инкремента или декремента.

Если переменная является фактически финальной, добавление модификатора `final` к ее объявлению не приводит к генерации ошибки времени компиляции. И наоборот, локальная переменная (или параметр), объявленная как `final` в корректной программе, становится фактически финальной при удалении модификатора `final`.

§4.12.5. Начальные значения переменных

Каждая переменная программы перед ее использованием должна иметь значение.

- Каждая переменная класса, переменная экземпляра или компонент массива при создании инициализируется *значением по умолчанию* (§15.9, §15.10.2).
  - ✦ Значение по умолчанию для типа `byte` нулевое, т.е. представляет собой значение `(byte)0`.
  - ✦ Значение по умолчанию для типа `short` нулевое, т.е. представляет собой значение `(short)0`.
  - ✦ Значение по умолчанию для типа `int` нулевое, т.е. представляет собой значение `0`.
  - ✦ Значение по умолчанию для типа `long` нулевое, т.е. представляет собой значение `0L`.
  - ✦ Значение по умолчанию для типа `float` — положительный нуль, т.е. представляет собой значение `0.0f`.
  - ✦ Значение по умолчанию для типа `double` — положительный нуль, т.е. представляет собой значение `0.0d`.
  - ✦ Значение по умолчанию для типа `char` — нулевой символ, т.е. представляет собой значение `'\u0000'`.
  - ✦ Значением по умолчанию для типа `boolean` является `false`.
  - ✦ Для всех ссылочных типов (§4.3) значением по умолчанию является `null`.
- Каждый параметр метода (§8.4.1) инициализируется соответствующим значением аргумента, предоставленным вызывающим метод кодом (§15.12).

- Каждый параметр конструктора (§8.8.1) инициализируется соответствующим значением аргумента, предоставленным выражением создания экземпляра класса (§15.9) или явным вызовом конструктора (§8.8.7).
- Параметр исключения (§14.20) инициализируется объектом, представляющим сгенерированное исключение (§11.3, §14.18).
- Локальной переменной (§14.4, §14.14) должно явным образом быть присвоено значение с помощью инициализации (§14.4) или присваивания (§15.26) таким образом, чтобы в этом можно было убедиться с применением правил определенного присваивания (§16).

Пример 4.12.5-1. Начальные значения переменных

```
class Point {
    static int npoints;
    int x, y;
    Point root;
}
class Test {
    public static void main(String[] args) {
        System.out.println("npoints=" + Point.npoints);
        Point p = new Point();
        System.out.println("p.x=" + p.x + ", p.y=" + p.y);
        System.out.println("p.root=" + p.root);
    }
}
```

Вывод программы

```
npoints=0
p.x=0, p.y=0
p.root=null
```

иллюстрирует инициализацию по умолчанию `npoints`, которая происходит при подготовке класса `Point` (§12), и инициализацию по умолчанию `x`, `y` и `root`, которая выполняется при создании нового экземпляра класса `Point`. Смотрите в §12 полное описание всех аспектов загрузки, связывания и инициализации классов и интерфейсов, а также описание инстанцирования классов (создания новых экземпляров).

§4.12.6. Типы, классы и интерфейсы

В языке программирования Java каждая переменная и каждое выражение имеет тип, который может быть определен во время компиляции. Тип может быть примитивным или ссылочным. Ссылочные типы включают типы классов и типы интерфейсов. Ссылочные типы вводятся *объявлениями типов*, которые включают объявления классов (§8.1) и объявления интерфейсов (§9.1). Термин *тип* часто используется для обозначения класса или интерфейса.

В виртуальной машине Java каждый объект принадлежит некоторому конкретному классу: классу, который был упомянут в выражении создания объекта (§15.9); классу,

объект `Class` которого использовался для вызова рефлексивного метода для создания объекта; классу `String` для объектов, неявно создаваемых оператором конкатенации строк + (§15.18.1). Этот класс называется *классом объекта*. Считается, что объект является экземпляром его класса и всех суперклассов этого класса.

Каждый массив также имеет класс. Метод `getClass`, вызванный для объекта массива, возвращает объект класса (класса `Class`), который представляет *класс массива* (§10.8).

Тип времени компиляции переменной всегда объявлен, а тип времени компиляции выражения может быть выведен в процессе компиляции. Тип времени компиляции ограничивает возможные значения, которые могут храниться в переменной во время выполнения или которые могут возвращаться при выполнении выражения. Если значение времени выполнения представляет собой ссылку, не являющуюся `null`, она ссылается на объект или массив, имеющий класс, и этот класс обязательно будет совместим с типом времени компиляции.

Несмотря на то что переменная или выражение может иметь тип времени компиляции, который является типом интерфейса, не существует экземпляров интерфейсов. Переменная или выражение, тип которого является типом интерфейса, может ссылаться на любой объект, класс которого реализует (§8.1.5) этот интерфейс.

Иногда о переменной или выражении говорят как об имеющем "тип времени выполнения". Это тип класса объекта, на который ссылается значение переменной или выражения во время выполнения, при условии, что это значение не равно `null`.

Соответствие между типами времени компиляции и времени выполнения является неполным по двум причинам.

1. Во время выполнения классы и интерфейсы загружаются виртуальной машиной Java с помощью загрузчика классов. Каждый загрузчик классов определяет собственный набор классов и интерфейсов. В результате возможно, что два загрузчика загружают идентичные определения классов или интерфейсов, но во время выполнения производят различные классы или интерфейсы. Следовательно, корректно скомпилировавшийся код может не скомпоноваться, если загрузчики этого класса несовместимы.

   Более подробную информацию можно найти в статье *Dynamic Class Loading in the Java Virtual Machine* Шенга Лианга (Sheng Liang) и Жилада Брачи (Gilad Bracha), в *Proceedings of OOPSLA'98*, опубликованной в *ACM SIGPLAN Notices*, Volume 33, Number 10, October 1998, pages 36–44, и в *The Java Virtual Machine Specification, Java SE 7 Edition*.

2. Переменные типа (§4.4) и аргументы типа (§4.5.1) недоступны во время выполнения. В результате один и тот же класс или интерфейс во время выполнения представляет несколько параметризованных типов (§4.5) времени компиляции. В частности, все параметризации времени компиляции данного обобщенного типа (§8.1.2, §9.1.2) совместно используют единое представление времени выполнения.

   При определенных условиях вполне возможно, что переменная параметризованного типа ссылается на объект, который не принадлежит параметризованному типу. Такая ситуация известна как *замусоривание кучи* (§4.12.2). Переменная всегда бу-

дет ссылаться на объект, который является экземпляром класса, который представляет параметризованный тип.

Пример 4.12.6-1. Тип переменной, класс и объект

```
interface Colorable {
    void setColor(byte r, byte g, byte b);
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
    byte r, g, b;
    public void setColor(byte rv, byte gv, byte bv) {
        r = rv; g = gv; b = bv;
    }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;
    }
}
```

В этом примере имеем следующее.

- Локальная переменная `p` метода `main` класса `Test` имеет тип `Point` и первоначально получает значение ссылки на новый экземпляр класса `Point`.
- Локальная переменная `cp` аналогично имеет тип `ColoredPoint` и первоначально получает значение ссылки на новый экземпляр класса `ColoredPoint`.
- Присваивание значения `cp` переменной `p` приводит к тому, что `p` хранит ссылку на объект `ColoredPoint`. Это допускается, поскольку `ColoredPoint` является подклассом класса `Point`, так что класс `ColoredPoint` является совместимым по присваиванию (§5.2) с типом `Point`. Объект `ColoredPoint` включает поддержку для всех методов `Point`. В дополнение к собственным полям `r`, `g` и `b` он имеет поля класса `Point`, а именно — `x` и `y`.
- Локальная переменная `c` имеет своим типом тип интерфейса `Colorable`, так что она может содержать ссылку на любой объект, класс которого реализует `Colorable`; в частности, она может содержать ссылку на `ColoredPoint`.

Обратите внимание, что выражение, такое как `new Colorable()`, является недопустимым, поскольку создать можно только экземпляр класса, но не экземпляр интерфейса. Однако выражение `new Colorable() { public void setColor... }` допустимо, поскольку объявляет анонимный класс (§15.9.5), который реализует интерфейс `Colorable`.

ГЛАВА 5

Преобразования и контексты

Каждое выражение в языке программирования Java либо не производит результата (§15.1), либо имеет тип, который может быть выведен во время компиляции (§15.3). При появлении выражения в большинстве контекстов оно должно быть совместимо с *типом*, ожидаемым в этом контексте; этот тип называется *целевым типом*. Для удобства совместимость выражения с окружающим контекстом облегчается двумя путями.

- Во-первых, для некоторых выражений, названных *поливыражениями* (poly expressions) (§15.2), на выводимый тип может влиять целевой тип. Одно и то же выражение может иметь различные типы в разных контекстах.
- Во-вторых, после вывода типа выражения может быть выполнено неявное *преобразование* из типа выражения в целевой тип.

Если ни одна из стратегий не дает подходящий тип, генерируется ошибка времени компиляции.

Правила, определяющие, является ли выражение поливыражением и, если является, его тип и совместимость в определенном контексте, варьируются в зависимости от контекста и формы выражения. Помимо влияния на тип выражения, тип целевого объекта может в некоторых случаях влиять на поведение во время выполнения выражения, чтобы последнее произвело значение подходящего типа.

Аналогично правила, определяющие, допускает ли целевой тип неявное преобразование, варьируются в зависимости от вида контекста, типа выражения и, в одном особом случае, от значения константного выражения (§15.28). Преобразование из типа S в тип T позволяет рассматривать выражение типа S так, как будто оно имеет во время компиляции тип T. В некоторых случаях во время выполнения потребуются соответствующие действия для проверки допустимости преобразования или перевода значения выражения времени выполнения в форму, подходящую для нового типа T.

Пример 5.0-1. Преобразования во время компиляции и во время выполнения

- Преобразование из типа `Object` в тип `Thread` требует проверки времени выполнения, чтобы убедиться, что значение времени выполнения в действительности является экземпляром класса `Thread` или одним из его подклассов; если это не так, генерируется исключение.
- Преобразование из типа `Thread` в тип `Object` не требует действий времени выполнения; `Thread` является подклассом `Object`, поэтому любая ссылка, генерируемая выражением типа `Thread`, является допустимым значением типа `Object`.

- Преобразование из типа `int` в тип `long` требует знакового расширения времени выполнения 32-битового целого значения в 64-битовое представление `long`. При этом никакая информация не теряется.
- Преобразование из типа `double` в тип `long` требует нетривиального перевода 64-битового значения с плавающей точкой в 64-битовое целочисленное представление. В зависимости от фактического значения времени выполнения некоторая информация может быть потеряна.

Преобразования, возможные в языке программирования Java, сгруппированы в несколько больших категорий.

- Тождественные преобразования
- Расширяющие примитивные преобразования
- Сужающие примитивные преобразования
- Расширяющие ссылочные преобразования
- Сужающие ссылочные преобразования
- Преобразования упаковки
- Преобразования распаковки
- Непроверяемые преобразования
- Преобразования при фиксации
- Преобразования строк
- Преобразования множества значений

Имеется шесть видов *контекстов преобразования*, в которых контекст может влиять на поливыражения или может осуществиться неявное преобразование. Каждая разновидность контекста имеет различные правила типизации поливыражений и разрешает преобразования в одних из приведенных выше категорий и запрещает в других. Этими контекстами являются следующие.

- Контекст присваивания (§5.2, §15.26), в котором значение выражения связывается с именованной переменной. Над примитивными и ссылочными типами выполняется расширяющее преобразование, значения могут упаковываться или распаковываться, а некоторые примитивные константные выражения могут подвергаться сужению. Может также выполняться непроверяемое преобразование.
- Контекст строгого вызова (§5.3, §15.9, §15.12), в котором аргумент связывается с формальным параметром конструктора или метода. Могут осуществляться расширяющее примитивное и расширяющее ссылочное преобразования, а также непроверяемое преобразование.
- Контекст свободного вызова (§5.3, §15.9, §15.12), в котором, как и в контексте строгого вызова, аргумент связывается с формальным параметром. Этот контекст может применяться для вызова метода или конструктора, если не удается найти применимое объявление с использованием только строгого контекста вызова. В дополнение к расширяющим и непроверяемым преобразованиям этот контекст допускает преобразования упаковки и распаковки.

- Строковый контекст (§5.4, §15.18.1), в котором значение любого типа преобразуется в объект типа `String`.
- Контекст приведения (§5.5), в котором значение выражения преобразуется в тип, явно указанный оператором приведения (§15.16). Контексты приведения более широкие, чем контексты присваивания или свободного вызова, так как допускают любые конкретные преобразования, отличные от строковых; но некоторые приведения к ссылочному типу проверяются на корректность во время выполнения.
- Числовой контекст (§5.6), в котором операнды числового оператора могут быть расширены до общего типа, так, чтобы можно было выполнить указанную операцию.

Термин "преобразование" (conversion) используется также для описания, без конкретного указания, любых преобразований в определенном контексте. Например, мы говорим, что выражение инициализатора локальной переменной подвергается "преобразованию присваивания", что означает, что конкретное преобразование будет выбрано для этого выражения неявно в соответствии с правилами для контекста присваивания.

Пример 5.0-2. Преобразования в разных контекстах

```
class Test {
    public static void main(String[] args) {
        // Преобразование приведения (5.4) литерала типа float
        // к типу int. Без оператора приведения это приведет к
        // ошибке времени компиляции, поскольку это сужающее
        // преобразование (5.1.3):
        int i = (int)12.5f;
        // Строковое преобразование (5.4) целого значения i:
        System.out.println("(int)12.5f==" + i);
        // Преобразование присваивания (5.2) значения i в тип
        // float. Это расширяющее преобразование (5.1.2):
        float f = i;
        // Строковое преобразование значения f типа float:
        System.out.println("После расширения float: " + f);
        // Числовое повышение (5.6) значения i до типа
        // float. Это бинарное числовое повышение.
        // После повышения выполняется операция float*float:
        System.out.print(f);
        f = f * i;
        // Два строковых преобразования, i и f:
        System.out.println("*" + i + "==" + f);
        // Преобразование вызова (5.3) значения f
        // в тип double, необходимое, так как метод Math.sin
        // принимает только аргумент типа double:
        double d = Math.sin(f);
        // Два строковых преобразования, f и d:
        System.out.println("Math.sin(" + f + ")==" + d);
    }
}
```

Вывод программы имеет следующий вид.

```
(int)12.5f==12
После расширения float: 12.0
12.0*12==144.0
Math.sin(144.0)==-0.49102159389846934
```

§5.1. Виды преобразований

Конкретные виды преобразований в языке программирования Java подразделяются на 13 категорий.

§5.1.1. Тождественное преобразование

Преобразование некоторого типа в тот же тип разрешено для любого типа.

Это правило может показаться тривиальным, но оно имеет два практических следствия. Во-первых, выражению всегда разрешено с самого начала иметь требуемый тип, таким образом, позволяя просто указать правило, которому подчиняется преобразование каждого выражения, но только для тривиального тождественного преобразования. Во-вторых, это означает, что программе разрешено для ясности включать избыточные операторы приведения.

§5.1.2. Расширяющее примитивное преобразование

19 определенных преобразований примитивных типов называются *расширяющими примитивными преобразованиями* (widening primitive conversions).

- `byte` в `short`, `int`, `long`, `float` или `double`
- `short` в `int`, `long`, `float` или `double`
- `char` в `int`, `long`, `float` или `double`
- `int` в `long`, `float` или `double`
- `long` в `float` или `double`
- `float` в `double`

Расширяющее примитивное преобразование не теряет информацию о числовом значении в следующих случаях, где числовое значение сохраняется в точности:

- из целочисленного типа в целочисленный тип;
- из `byte`, `short` или `char` в тип с плавающей точкой;
- из `int` в `double`;
- из `float` в `double` в `strictfp`-выражении (§15.4).

Расширяющее примитивное преобразование `float` в `double`, не являющееся `strictfp`, может терять информацию о полной величине преобразуемого значения.

Расширяющее преобразование значений типа `int` или `long` в тип `float` или значения типа `long` в `double`, может привести к *потере точности*, т.е. результат может

терять некоторые младшие биты значения. В этом случае результирующее значение с плавающей точкой будет представлять собой корректно округленную с использованием режима округления стандарта IEEE 754 к ближайшему значению (§4.2.4) версию целого значения.

Расширяющее преобразование знакового целого значения в целочисленный тип `T` представляет собой простое знаковое расширение представления целочисленного значения в дополнительном до 2 коде, заполняющее более широкий формат.

Расширяющее преобразование `char` в целочисленный тип `T` представляет собой простое беззнаковое (с дополнением нулями) расширение значения типа `char` до более широкого формата.

Несмотря на возможную потерю точности, расширяющее примитивное преобразование никогда не приводит к исключению времени выполнения (§11.1.1).

Пример 5.1.2-1. Расширяющее примитивное преобразование

```
class Test {
    public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```

Вывод программы

```
-46
```

указывает на потерю информации при преобразовании типа `int` в тип `float`, так как значения типа `float` не обладают точностью до девяти значащих цифр.

§5.1.3. Сужающее примитивное преобразование

22 определенных преобразования примитивных типов называются *сужающими примитивными преобразованиями* (narrowing primitive conversions).

- `short` в `byte` или `char`
- `char` в `byte` или `short`
- `int` в `byte`, `short` или `char`
- `long` в `byte`, `short`, `char` или `int`
- `float` в `byte`, `short`, `char`, `int` или `long`
- `double` в `byte`, `short`, `char`, `int`, `long` или `float`

Сужающее примитивное преобразование может терять информацию об общей величине числового значения, а также терять точность и диапазон представления.

Сужающее примитивное преобразование `double` в `float` регулируется правилами округления стандарта IEEE 754 (§4.2.4). Это преобразование может терять точность, но также терять и диапазон представления, преобразовывая в нуль типа `float` ненулевое значение `double` и в бесконечность `float` конечное значение `double`. NaN типа

`double` преобразуется в NaN типа `float`, а бесконечность типа `double` — в бесконечность типа `float`.

Сужающее преобразование знакового целого числа в целочисленный тип *T* просто отбрасывает все, кроме *n* младших битов, где *n* — количество битов, использующихся для представления типа *T*. Помимо возможной потери информации о величине числового значения, такое преобразование может привести к знаку результирующего значения, отличному от знака исходного значения.

Сужающее преобразование типа `char` в целочисленный тип *T* также просто отбрасывает все, кроме *n* младших битов, где *n* — количество битов, использующихся для представления типа *T*. Помимо возможной потери информации о величине числового значения, такое преобразование может привести к тому, что результирующее значение будет отрицательным числом, несмотря на то что символы представляют собой 16-битовые беззнаковые целочисленные значения.

Сужающее преобразование числа с плавающей точкой в целочисленный тип *T* выполняется за два шага.

1. На первом шаге число с плавающей точкой преобразуется либо в `long`, если *T* является типом `long`, либо в `int`, если *T* представляет собой тип `byte`, `short`, `char` или `int`, следующим образом.
   - Если число с плавающей точкой представляет собой NaN (§4.2.3), в результате первого шага преобразования получается нулевое значение типа `int` или `long`.
   - В противном случае, если число с плавающей точкой не является бесконечностью, значение с плавающей точкой округляется до целочисленного значения *V* с помощью округления по направлению к нулю с применением соответствующего режима округления стандарта IEEE 754 (§4.2.3). Имеются два варианта.
     - ✦ Если *T* представляет собой тип `long` и это целочисленное значение может быть представлено как `long`, то результатом первого шага является значение *V* типа `long`.
     - ✦ В противном случае, если это целочисленное значение может быть представлено как `int`, результатом первого шага является значение *V* типа `int`.
   - В противном случае должна осуществляться одна из следующих двух ситуаций.
     - ✦ Значение должно быть слишком малым (отрицательное значение большой величины или отрицательная бесконечность), и результат первого шага представляет собой наименьшее представимое значение типа `int` или `long`.
     - ✦ Значение должно быть слишком большим (положительное значение большой величины или положительная бесконечность), и результат первого шага представляет собой наибольшее представимое значение типа `int` или `long`.
2. На втором шаге выполняется следующее.
   - Если *T* представляет собой `int` или `long`, результатом преобразования является результат первого шага.
   - Если *T* представляет собой тип `byte`, `char` или `short`, результатом преобразования является результат сужающего преобразования в тип *T* (§5.1.3) значения, полученного на первом шаге.

Несмотря на то что при преобразовании могут произойти переполнение, потеря значимости и другие потери информации, сужающее примитивное преобразование никогда не приводит к исключению времени выполнения (§11.1.1).

Пример 5.1.3-1. Сужающее примитивное преобразование

```
class Test {
    public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)fmin +
                           ".." + (long)fmax);
        System.out.println("int: " + (int)fmin +
                           ".." + (int)fmax);
        System.out.println("short: " + (short)fmin +
                           ".." + (short)fmax);
        System.out.println("char: " + (int)(char)fmin +
                           ".." + (int)(char)fmax);
        System.out.println("byte: " + (byte)fmin +
                           ".." + (byte)fmax);
    }
}
```

Вывод программы следующий.

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

В результатах для `char`, `int` и `long` нет ничего удивительного — преобразования дают минимальные и максимальные представимые значения типов.

Результаты для `byte` и `short` наряду с потерей информации о знаке и величине числовых значений теряют также точность. Результаты могут быть поняты после рассмотрения младших битов минимального и максимального значений типа `int`. Минимальное значение типа `int` в шестнадцатеричном формате имеет вид `0x80000000`, а максимальным значением типа `int` является `0x7fffffff`. Это объясняет результаты типа short, которые представляют собой 16 младших битов этих величин, а именно — `0x0000` и `0xffff`. Аналогично объясняются результаты типа `char`, которые также представляют собой 16 младших битов этих величин, а именно — `'\u0000'` и `'\uffff'`; и результаты типа `byte`, которые представляют собой младшие 8 битов этих величин, а именно — `0x00` и `0xff`.

Пример 5.1.3-2. Сужающее примитивное преобразование с потерей информации

```
class Test {
    public static void main(String[] args) {
        // Сужающее примитивное преобразование
        // int в short теряет старшие биты:
        System.out.println("(short)0x12345678==0x" +
```

```
        Integer.toHexString((short)0x12345678));
        // Сужающее примитивное преобразование слишком
        // большого int в byte меняет знак и величину:
        System.out.println("(byte)255==" + (byte)255);
        // Сужающее примитивное преобразование слишком
        // большого float дает наибольшее значение типа int:
        System.out.println("(int)1e20f==" + (int)1e20f);
        // Преобразование NaN в int дает нуль:
        System.out.println("(int)NaN==" + (int)Float.NaN);
        // Слишком большое значение типа double при
        // преобразовании в float дает бесконечность:
        System.out.println("(float)-1e100==" + (float)-1e100);
        // Слишком малое значение типа double при
        // преобразовании в float дает нуль:
        System.out.println("(float)1e-50==" + (float)1e-50);
    }
}
```

Вывод программы следующий.

```
(short)0x12345678==0x5678
(byte)255==-1
(int)1e20f==2147483647
(int)NaN==0
(float)-1e100==-Infinity
(float)1e-50==0.0
```

§5.1.4. Расширяющее и сужающее примитивные преобразования

Следующее преобразование сочетает в себе как расширяющее, так и сужающее примитивные преобразования.

- `byte` в `char`

Сначала значение типа `byte` преобразуется в тип `int` посредством расширяющего примитивного преобразования (§5.1.2), а затем результирующий тип `int` преобразуется в тип `char` посредством сужающего примитивного преобразования (§5.1.3).

§5.1.5. Расширяющее ссылочное преобразование

Расширяющее ссылочное преобразование существует для любого ссылочного типа S, преобразуемого в любой ссылочный тип T, если S является подтипом (§4.10) типа T.

Расширяющее ссылочное преобразование никогда не требует каких-либо специальных действий во время выполнения и потому никогда не генерирует исключений времени выполнения. Оно заключается лишь в рассмотрении ссылки как имеющей некоторый другой тип, таким образом, что его корректность может быть доказана во время компиляции.

§5.1.6. Сужающее ссылочное преобразование

Шесть видов преобразований называются *сужающими ссылочными преобразованиями.*

- Из любого ссылочного типа *S* в любой ссылочный тип *T* при условии, что *S* является истинным супертипом *T* (§4.10).

  Важным частным случаем является сужающее ссылочное преобразование типа класса `Object` в любой другой ссылочный тип (§4.12.4).
- Из любого типа класса *C* в любой тип непараметризованного интерфейса *K* при условии, что *C* не является `final` и не реализует *K*.
- Из любого типа интерфейса *J* в любой тип непараметризованного класса *C*, не являющийся `final`.
- Из любого типа интерфейса *J* в любой тип непараметризованного интерфейса *K* при условии, что *J* не является подынтерфейсом *K*.
- Из типов интерфейса `Cloneable` и `java.io.Serializable` в любой тип массива *T*`[]`.
- Из любого типа массива *SC*`[]` в тип массива *TC*`[]` при условии, что *SC* и *TC* являются ссылочными типами и существует сужающее ссылочное преобразование из *SC* в *TC*.

Такие преобразования требуют проверок времени выполнения, чтобы выяснить, является ли фактическое значение ссылки правомерным для значения нового типа. Если нет, генерируется исключение `ClassCastException`.

§5.1.7. Преобразование упаковки

Преобразование упаковки преобразует выражения примитивного типа в соответствующие выражения ссылочного типа. В частности, *преобразованиями упаковки* (boxing conversions) называются следующие 9 преобразований.

- Из типа `boolean` в тип `Boolean`
- Из типа `byte` в тип `Byte`
- Из типа `short` в тип `Short`
- Из типа `char` в тип `Character`
- Из типа `int` в тип `Integer`
- Из типа `long` в тип `Long`
- Из типа `float` в тип `Float`
- Из типа `double` в тип `Double`
- Из типа null в тип null

> Это правило является необходимым, поскольку условный оператор (§15.25) применяет преобразование упаковки к типам операндов и использует полученный результат в дальнейших вычислениях.

Во время выполнения преобразование упаковки работает следующим образом.

- Если `p` представляет собой значение типа `boolean`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Boolean`, такую, что `r.booleanValue() == p`.
- Если `p` представляет собой значение типа `byte`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Byte`, такую, что `r.byteValue() == p`.
- Если `p` представляет собой значение типа `char`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Character`, такую, что `r.charValue() == p`.
- Если `p` представляет собой значение типа `short`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Short`, такую, что `r.shortValue() == p`.
- Если `p` представляет собой значение типа `int`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Integer`, такую, что `r.intValue() == p`.
- Если `p` представляет собой значение типа `long`, то преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Long`, такую, что `r.longValue() == p`.
- Если `p` представляет собой значение типа `float`, то
  - ✦ если `p` не является NaN, преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Float`, такую, что вычисление `r.floatValue()` дает `p`;
  - ✦ в противном случае преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Float`, такую, что вычисление `r.isNaN()` дает `true`.
- Если `p` представляет собой значение типа `double`, то
  - ✦ если `p` не является NaN, преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Double`, такую, что вычисление `r.doubleValue()` дает `p`;
  - ✦ в противном случае преобразование упаковки преобразует `p` в ссылку `r` класса и типа `Double`, такую, что вычисление `r.isNaN()` дает `true`.
- Если `p` представляет собой значение любого другого типа, преобразование упаковки эквивалентно тождественному преобразованию (§5.1.1).

Если упаковываемое значение *p* представляет собой целочисленный литерал типа `int` между `-128` и `127` включительно (§3.10.1) или булев литерал `true` или `false` (§3.10.3), или символьный литерал в диапазоне от `\u0000` до `\u007f` включительно (§3.30.4), то пусть *a* и *b* представляют собой результаты любых двух преобразований упаковки *p*. В таком случае всегда выполняется равенство *a* == *b*.

> В идеале упаковка заданного примитивного значения *p* всегда дает одинаковые ссылки. На практике применение существующих методов реализации может сделать это утверждение неверным. Приведенные выше правила являются прагматичным компромиссом, требующим, чтобы некоторые распространенные значения всегда упаковывались в неотличимые объекты. Реализация может выполнять их кеширование. Для других значений эта формулировка запрещает какие-либо предположения об упакованных значениях. Тем самым допускается (но не требуется) совместное использование некоторых (или всех) из этих ссылок. Обратите внимание, что интегральные литералы типа `long` могут использоваться совместно, но это не является обязательным требованием.
>
> Гарантируется, что в наиболее распространенных случаях будет обеспечено желаемое поведение без неоправданных потерь производительности, в особенности на

> небольших устройствах. Менее ограниченные по памяти реализации могут, например, кешировать все значения типа `char` и `short`, как и значения `int` и `long` из диапазона от –32K до +32K.

Преобразование упаковки может привести к генерации исключения `OutOfMemoryError`, если новый экземпляр одного из классов-оболочек (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` или `Double`) требует выделения памяти, а количества доступной памяти при этом недостаточно.

§5.1.8. Преобразование распаковки

Преобразование распаковки преобразует выражения ссылочного типа в соответствующие выражения примитивного типа. В частности, *преобразованиями распаковки* называются следующие 8 преобразований.

- Из типа `Boolean` в тип `boolean`
- Из типа `Byte` в тип `byte`
- Из типа `Short` в тип `short`
- Из типа `Character` в тип `char`
- Из типа `Integer` в тип `int`
- Из типа `Long` в тип `long`
- Из типа `Float` в тип `float`
- Из типа `Double` в тип `double`

Во время выполнения преобразование распаковки действует следующим образом.

- Если `r` представляет собой ссылку типа `Boolean`, то преобразование распаковки преобразует `r` в `r.booleanValue()`.
- Если `r` представляет собой ссылку типа `Byte`, то преобразование распаковки преобразует `r` в `r.byteValue()`.
- Если `r` представляет собой ссылку типа `Character`, то преобразование распаковки преобразует `r` в `r.charValue()`.
- Если `r` представляет собой ссылку типа `Short`, то преобразование распаковки преобразует `r` в `r.shortValue()`.
- Если `r` представляет собой ссылку типа `Integer`, то преобразование распаковки преобразует `r` в `r.intValue()`.
- Если `r` представляет собой ссылку типа `Long`, то преобразование распаковки преобразует `r` в `r.longValue()`.
- Если `r` представляет собой ссылку типа `Float`, то преобразование распаковки преобразует `r` в `r.floatValue()`.
- Если `r` представляет собой ссылку типа `Double`, то преобразование распаковки преобразует `r` в `r.doubleValue()`.

- Если `r` равно `null`, то преобразование распаковки генерирует исключение `NullPointerException`.

Тип называется *преобразуемым в числовой тип*, если он имеет числовой тип (§4.2) или если это ссылочный тип, который может быть преобразован в числовой тип преобразованием распаковки.

Тип называется *преобразуемым в целочисленный тип*, если это целочисленный тип или если это ссылочный тип, который может быть преобразован в целочисленный тип преобразованием распаковки.

§5.1.9. Непроверяемое преобразование

Пусть G обозначает объявление обобщенного типа с n параметрами типа.

Существует *непроверяемое преобразование* из несформированного типа класса или интерфейса (§4.8) G в любой параметризованный тип вида $G<T_1, \ldots, T_n>$.

Существует *непроверяемое преобразование* из несформированного типа массива $G[]^k$ в любой тип массива вида $G<T1, \ldots, Tn>[]^k$. (Запись $[]^k$ указывает на k-мерный массив.)

Использование непроверяемого преобразования приводит к *предупреждению времени компиляции о непроверенном преобразовании* типов, за исключением ситуаций, когда `G<...>` является параметризованным типом, в котором все аргументы типа T_i ($1 \le i \le n$) являются неограниченными символами подстановки (§4.5.1), или когда предупреждение подавлено аннотацией `SuppressWarnings` (§9.6.3.5).

> Непроверяемое преобразование используется для обеспечения взаимодействия устаревшего кода, написанного до введения обобщенных типов, с библиотеками, которые были преобразованы с использованием обобщенности. В таких обстоятельствах (прежде всего, это клиенты Collections Framework из `java.util`) устаревший код использует несформированные типы (например, `Collection` вместо `Collection<String>`). Выражения несформированных типов передаются как аргументы методам библиотеки, использующим параметризованные версии этих же типов, в качестве типов их соответствующих формальных параметров.
>
> Такие вызовы не могут быть статически безопасными при использовании системы типов, использующей обобщенность. Если запретить такие вызовы, окажется недействительным огромное количество существующего кода, которому будет запрещено использование новых версий библиотек. Это, в свою очередь, будет препятствовать применению обобщенности разработчиками библиотек. Для предотвращения такого нежелательного развития событий несформированный тип может быть преобразован в произвольной вызов объявления обобщенного типа, на который ссылается этот несформированный тип. Поскольку такое преобразование является ненадежным, оно является не более чем уступкой практичности. В таких случаях компилятор выдает предупреждение времени компиляции о непроверенном преобразовании типов.

§5.1.10. Преобразование при фиксации

Пусть G представляет объявление обобщенного типа (§8.1.2, §9.1.2) с n параметрами типа $A_1, \ldots, A_n$ с соответствующими границами $U_1, \ldots, U_n$.

Существует *преобразование при фиксации* (capture conversion) из параметризованного типа $G<T_1, \ldots, T_n>$ (§4.5) в параметризованный тип $G<S_1, \ldots, S_n>$, где для $1 \le i \le n$ выполняется следующее.

- Если T_i — аргумент типа, представляющий собой символ подстановки (§4.5.1) вида `?`, то S_i является переменной несформированного типа, верхняя граница которого представляет собой $U_i[A_1 := S_1, \ldots, A_n := S_n]$, а нижняя граница — пустой тип (§4.1).
- Если T_i — аргумент типа с символом подстановки вида `? extends` B_i, то S_i — переменная несформированного типа, верхняя граница которой представляет собой glb($B_i, U_i[A_1 := S_1, \ldots, A_n := S_n]$), а нижняя граница — пустой тип.

  glb($V_1, \ldots, V_m$) определяется как V_1 `&` ... `&` V_m.

  Если для любых двух классов (не интерфейсов) V_i и V_j класс V_i не является подклассом V_j или наоборот, генерируется ошибка времени компиляции.
- Если T_i — аргумент типа с символом подстановки вида `? super` B_i, то S_i — переменная несформированного типа, верхняя граница которой представляет собой $U_i[A_1 := S_1, \ldots, A_n := S_n]$, а нижняя граница — B_i.
- В противном случае $S_i = T_i$.

Преобразование при фиксации любого типа, отличного от параметризованного (§4.5), действует как тождественное преобразование (§5.1.1).

Преобразование при фиксации не применяется рекурсивно.

Преобразование при фиксации никогда не требует специальных действий во время выполнения, а потому никогда не генерирует исключений во время выполнения.

> Преобразование при фиксации призвано сделать символы подстановки более полезными. Чтобы понять мотивацию преобразования при фиксации, начнем с рассмотрения метода `java.util.Collections.reverse()`.
>
> ```
> public static void reverse(List<?> list);
> ```
>
> Этот метод обращает список, переданный в качестве параметра. Он работает с любым типом списка, так что вполне уместно использовать тип с символом подстановки `List<?>`.
>
> Теперь рассмотрим, как можно реализовать `reverse()`.
>
> ```
> public static void reverse(List<?> list) { rev(list); }
> private static <T> void rev(List<T> list) {
>     List<T> tmp = new ArrayList<T>(list);
>     for (int i = 0; i < list.size(); i++) {
>         list.set(i, tmp.get(list.size() - i - 1));
>     }
> }
> ```
>
> Реализация должна скопировать список, извлечь элементы из копии и вставить их в оригинал. Чтобы сделать это строго типизированным образом, необходимо пре

доставить имя `T` типа элемента входного списка. Мы делаем это в закрытом служебном методе `rev()`. Это требует от нас передачи в качестве входного аргумента списка типа `List<?>` методу `rev()`. В общем случае `List<?>` представляет собой список неизвестного типа. Это не подтип `List<T>` ни для какого типа `T`. Допустить такое отношение подтипа было бы ошибочным.

При наличии метода

```
public static <T> void fill(List<T> l, T obj)
```

следующий код приведет к нарушениям системы типов.

```
List<String> ls = new ArrayList<String>();
List<?> l = ls;
Collections.fill(l, new Object()); // Некорректно,
                        // но предположим, что так можно!
String s = ls.get(0); // Исключение ClassCastException –
                        // ls содержит Object, а не String
```

Итак, без некоторых специальных разрешений вызов `rev()` из `reverse()` будет запрещен. Но если бы это было можно, автор `reverse()` был бы вынужден записать сигнатуру как

```
public static <T> void reverse(List<T> list)
```

Это нежелательно, поскольку предоставляет информацию о реализации вызывающему коду. Хуже того, разработчик API может сделать вывод о том, что сигнатура с символом подстановки — это именно то, чего требует вызывающий код, и только позже понять, что таким путем безопасная в смысле типов реализация невозможна.

Вызов `rev()` из `reverse()` на самом деле является безвредным, но он не может быть оправдан на основе обобщенного отношения подтипа между `List<?>` и `List<T>`. Вызов безопасен, потому что входной аргумент, несомненно, является списком элементов некоторого типа (хотя и неизвестного). Если мы можем фиксировать этот неизвестный тип в переменной типа `X`, мы можем заключить, что `T` является `X`. Такова суть преобразования при фиксации. Конечно, спецификация должна справиться со сложностями, такими как нетривиальная (а возможно, и рекурсивно определенная) верхняя или нижняя граница, наличие нескольких аргументов и т.п.

Математически подкованные читатели могут захотеть связать преобразование при фиксации с теорией типов. Читатели, не знакомые с этой теорией, могут пропустить приведенный здесь материал или сначала изучить подходящую книгу, например *Types and Programming Languages* Бенджамина Пирса (Benjamin Pierce), а затем вновь обратиться к данному разделу.

Вот краткое резюме отношения преобразования при фиксации с понятиями теории типов. Типы с символами подстановки представляют собой ограниченный вид экзистенциальных типов. Преобразование при фиксации слабо соответствует открытию значения экзистенциального типа. Преобразование при фиксации выражения `e` можно рассматривать как `open` для `e` в области, которая включает выражение верхнего уровня, охватывающее `e`.

Классическая операция `open` над экзистенциалами требует, чтобы фиксированная переменная типа не избегала открытого выражения. Операция `open`, соответст

вующая преобразованию при фиксации, всегда находится в области, достаточно большой для того, чтобы фиксированная переменная типа никогда не была видна вне этой области. Преимущество этой схемы заключается в том, что нет необходимости в операции `close`, как это определено в статье *On Variance-Based Subtyping for Parametric Types* Ацуси Игараси (Atsushi Igarashi) и Мирко Вироли (Mirko Viroli), опубликованной в трудах конференции *16th European Conference on Object Oriented Programming* (ECOOP 2002). Формальный учет символов подстановки можно найти в статье *Wild FJ* Мэдса Торгерсена (Mads Torgersen), Эрика Эрнста (Erik Ernst) и Кристиана Плезнера Хансена (Christian Plesner Hansen) в трудах *12th workshop on Foundations of Object Oriented Programming* (FOOL 2005).

§5.1.11. Строковое преобразование

Любой тип может быть преобразован в тип `String` с помощью *строкового преобразования*.

Значение *x* примитивного типа *T* сначала преобразуется в ссылочное значение, как если бы оно было передано как аргумент выражению создания экземпляра соответствующего класса (§15.9).

- Если *T* представляет собой тип `boolean`, то используется выражение `new Boolean(x)`.
- Если *T* представляет собой тип `char`, то используется выражение `new Character(x)`.
- Если *T* представляет собой тип `byte`, `short` или `int`, то используется выражение `new Integer(x)`.
- Если *T* представляет собой тип `long`, то используется выражение `new Long(x)`.
- Если *T* представляет собой тип `float`, то используется выражение `new Float(x)`.
- Если *T* представляет собой тип `double`, то используется выражение `new Double(x)`.

  Это ссылочное значение преобразуется в тип `String` путем строкового преобразования.
- Если ссылка имеет значение `null`, она преобразуется в строку `"null"` (четыре ASCII-символа `n`, `u`, `l`, `l`).
- В противном случае преобразование выполняется, как если бы был вызван метод `toString` объекта, на который указывает ссылка, без передачи ему аргументов. Но если результат вызова метода `toString` — значение `null`, то в качестве результата используется строка `"null"`.

> Метод `toString` определяется изначальным классом `Object` (§4.3.2). Многие классы переопределяют его, в частности `Boolean`, `Character`, `Integer`, `Long`, `Float`, `Double` и `String`.
>
> Подробная информация о строковом контексте рассматривается в §5.4.

§5.1.12. Запрещенные преобразования

Запрещено любое преобразование, которое явно не разрешено.

§5.1.13. Преобразование набора значений

Преобразование набора значений — это процесс отображения значения с плавающей точкой из одного набора значений в другой без изменения его типа.

В выражении, которое не является FP-строгим (§15.4), преобразование набора значений представляет следующий выбор конкретной реализации языка программирования Java.

- Если значение представляет собой элемент расширенного набора значений float, то реализация может по своему усмотрению отображать значение на ближайший элемент набора значений float. Это преобразование может приводить к переполнению (в этом случае значение заменяется бесконечностью того же знака) или потере значимости (в этом случае значение может терять точность, будучи замененным денормализованным числом или нулем того же знака).
- Если значение представляет собой элемент расширенного набора значений double, то реализация может по своему усмотрению отображать значение на ближайший элемент набора значений double. Это преобразование может приводить к переполнению (в этом случае значение заменяется бесконечностью того же знака) или потере значимости (в этом случае значение может терять точность, будучи замененным денормализованным числом или нулем того же знака).

В рамках FP-строгого выражения (§15.4) преобразование набора значений не предоставляет какого-либо выбора. Каждая реализация должна вести себя одним и тем же образом.

- Если значение имеет тип `float` и не является элементом набора значений float, то реализация обязана отобразить значение на ближайший элемент набора значений float. Это преобразование может приводить к переполнению или потере точности.
- Если значение имеет тип `float` и не является элементом набора значений float, то реализация обязана отобразить значение на ближайший элемент набора значений float. Это преобразование может приводить к переполнению или потере точности.

В рамках FP-строгого выражения отображение значений из расширенного набора значений float или расширенного набора значений double необходимо только тогда, когда вызывается метод, объявление которого не является FP-строгим, а реализация выбирает для представления результата вызова метода элемент расширенного набора значений.

Независимо от того, является ли код FP-строгим, преобразование набора значений всегда оставляет без изменений любое значение, тип которого не является ни `float`, ни `double`.

§5.2. Контексты присваивания

Контексты присваивания позволяют значению выражения быть присвоенным (§15.26) переменной; тип выражения должен быть преобразован в тип переменной.

Контексты присваивания позволяют использовать одно из следующих преобразований.

- Тождественное преобразование (§5.1.1).
- Расширяющее примитивное преобразование (§5.1.2).

- Расширяющее ссылочное преобразование (§5.1.5).
- Преобразование упаковки (§5.1.7), за которым при необходимости следует расширяющее ссылочное преобразование.
- Преобразование распаковки (§5.1.8), за которым при необходимости следует расширяющее примитивное преобразование.

Если после применения перечисленных выше преобразований результирующий тип является несформированным (§4.8), может дополнительно применяться непроверяемое преобразование (§5.1.9).

Кроме того, если выражение является константным выражением (§15.28) типа `byte`, `short`, `char` или `int`, то следует учитывать следующее.

- Если тип переменной — `byte`, `short` или `char` и значение константного выражения представимо типом переменной, может использоваться сужающее примитивное преобразование.
- Если тип и значения переменной соответствуют одному из перечисленных ниже пунктов, может использоваться сужающее примитивное преобразование, за которым следует преобразование упаковки.
  - ✦ Тип переменной — `Byte`, и значение константного выражения представимо типом `byte`.
  - ✦ Тип переменной — `Short`, и значение константного выражения представимо типом `short`.
  - ✦ Тип переменной — `Character`, и значение константного выражения представимо типом `char`.

> Сужение констант времени компиляции означает, что разрешен код наподобие следующего.
>
> ```
> byte theAnswer = 42;
> ```
>
> Без такого сужения тот факт, что целочисленный литерал 42 имеет тип `int`, означает, что требуется приведение к типу `byte`.
>
> ```
> byte theAnswer = (byte)42; // Приведение разрешено, но не обязательно
> ```

Наконец значение типа null (единственным таким значением является `null`) может быть присвоено любому ссылочному типу, давая в результате пустую ссылку данного типа.

Если цепочка преобразований содержит два параметризованных типа, которые не связаны отношением подтипа, генерируется ошибка времени компиляции.

> Примером такой некорректной цепочки может служить
>
> ```
> Integer, Comparable<Integer>, Comparable, Comparable<String>
> ```
>
> Первые три элемента цепочки связаны расширяющим ссылочным преобразованием, в то время как последний элемент является производным от своего предшественника с помощью непроверяемого преобразования. Однако такое преобразование не является корректным преобразованием присваивания, потому что цепочка содержит два параметризованных типа, `Comparable<Integer>` и `Comparable<String>`, которые не являются подтипами.

Если тип выражения не может быть преобразован в тип переменной с помощью разрешенных в контексте присваивания преобразований, генерируется ошибка времени компиляции.

Если тип выражения может быть преобразован в тип переменной путем преобразования присваивания, мы говорим, что выражение (или его значение) *присваиваемо* (assignable) переменной или, что эквивалентно, что тип выражения является *совместимым по присваиванию* с типом переменной.

Если тип переменной — `float` или `double`, то преобразование набора значений (§5.1.13) применимо к значению *v*, которое является результатом преобразования типов.

- Если `v` имеет тип `float` и является элементом расширенного набора значений float, то реализация должна отображать *v* на ближайший элемент набора значений float. Это преобразование может привести к переполнению или потере точности.
- Если `v` имеет тип `double` и является элементом расширенного набора значений double, то реализация должна отображать *v* на ближайший элемент набора значений double. Это преобразование может привести к переполнению или потере точности.

Преобразование присваивания может вызвать генерацию только следующих исключений.

- `ClassCastException`, если после применения перечисленных выше преобразований типов результирующее значение представляет собой объект, который не является экземпляром подкласса или подынтерфейса затирания (§4.6) типа переменной.

  Эта ситуация может возникать только в результате замусоривания кучи (§4.12.2). На практике реализации должны выполнять приведения только при доступе к полю или методу объекта параметризованного типа, когда затертый тип поля или затертый тип результата метода отличается от их незатертых типов.

- `OutOfMemoryError` в результате преобразования упаковки.
- `NullPointerException` в результате преобразования распаковки пустой ссылки.
- `ArrayStoreException` в особых случаях, связанных с доступом к элементам массива или к полю (§10.5, §15.26.1).

Пример 5.2-1. Преобразование присваивания для примитивных типов

```
class Test {
    public static void main(String[] args) {
        short s = 12;           // Сужение 12 до short
        float f = s;            // Расширение short до float
        System.out.println("f=" + f);
        char c = '\u0123';
        long l = c;             // Расширение char до long
        System.out.println("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f;           // Расширение float до double
        System.out.println("d=" + d);
    }
}
```

Вывод программы имеет следующий вид.

```
f=12.0
l=0x123
d=1.2300000190734863
```

Однако приведенная далее программа приводит к ошибкам времени компиляции.

```
class Test {
    public static void main(String[] args) {
        short s = 123;
        char c = s; // Ошибка: требуется приведение
        s = c;      // Ошибка: требуется приведение
    }
}
```

Дело в том, что не все значения `short` являются значениями `char` и не все значения `char` являются значениями `short`.

Пример 5.2-2. Преобразование присваивания для ссылочных типов

```
class Point { int x, y; }
class Point3D extends Point { int z; }
interface Colorable { void setColor(int color); }

class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        // Присваивания переменным типа класса:
        Point p = new Point();
        p = new Point3D();
            // OK, поскольку Point3D
            // является подклассом Point
        Point3D p3d = p;
            // Ошибка: требует приведения, поскольку Point
            // может не быть Point3D (несмотря на то, что
            // в этом примере это так)

        // Присваивания переменным типа Object:
        Object o = p;         // OK: любой объект в Object
        int[] a = new int[3];
        Object o2 = a;        // OK: массив в Object

        // Присваивания переменным типа интерфейса:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp;
            // OK: ColoredPoint реализует Colorable
```

```
        // Присваивания переменным типа массива:
        byte[] b = new byte[4];
        a = b;
            // Ошибка: это массивы разных примитивных типов
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da;
            // OK: поскольку можно присваивать Point3D
            // переменной типа Point
        p3da = pa;
            // Ошибка: (требуется приведение) поскольку Point
            // не может быть присвоен переменной Point3D
    }
}
```

Следующая тестовая программа иллюстрирует преобразования присваивания для ссылочных значений, но она не компилируется, как описано в комментариях. Этот пример следует сравнить с предыдущим.

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // OK, поскольку ColoredPoint является
        // подклассом Point:
        p = cp;
        // OK, поскольку ColoredPoint реализует Colorable:
        Colorable c = cp;
        // Приведенный далее код приводит к ошибке времени
        // компиляции, поскольку нельзя гарантировать его
        // успешность в зависимости от типа времени
        // выполнения p; для требуемого сужающего
        // преобразования необходима проверка времени
        // выполнения, и она должна быть указана путем
        // включения приведения:
        cp = p;  // p может не быть ни ColoredPoint,
                 // ни подклассом ColoredPoint
        c = p;   // p может не реализовывать Colorable
    }
}
```

Пример 5.2-3. Преобразование присваивания для типов массивов

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
    public static void main(String[] args) {
        long[] veclong = new long[100];
        Object o = veclong;     // OK
        Long l = veclong;       // Ошибка времени компиляции
        short[] vecshort
                 = veclong;     // Ошибка времени компиляции
        Point[] pvec = new Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec;           // OK
        pvec[0] = new Point(); // Все в порядке при
                               // компиляции, но при выполнении
                               // генерируется исключение
        cpvec = pvec;           // Ошибка времени компиляции
    }
}
```

В этом примере имеем следующее.

- Значение `veclong` не может быть присвоено переменной типа `Long`, поскольку `Long` является типом класса, отличным от `Object`. Массив можно присвоить только переменной совместимого типа массива или переменной типа `Object`, `Cloneable` или `java.io.Serializable`.
- Значение `veclong` не может быть присвоено переменной `vecshort`, поскольку они представляют собой массивы примитивного типа, и `short` и `long` представляют собой различные примитивные типы.
- Значение `cpvec` может быть присвоено переменной `pvec`, поскольку любая ссылка, которая может быть значением выражения типа `ColoredPoint`, может быть значением переменной типа `Point`. Последующее присваивание нового объекта `Point` компоненту `pvec` приведет к генерации исключения `ArrayStoreException`, поскольку массив `ColoredPoint` не может иметь экземпляр `Point` в качестве значения компонента.
- Значение `pvec` не может быть присвоено переменной `cpvec`, поскольку не всякая ссылка, которая может быть значением выражения типа `ColoredPoint`, может быть корректным значением переменной типа `Point`. Если значение `pvec` во время выполнения является ссылкой на экземпляр `Point[]` и присваивание переменной `cpvec` разрешено, то простая ссылка на компонент `cpvec`, скажем, `cpvec[0]`, может вернуть `Point`, а `Point` не является `ColoredPoint`. Таким образом, позволив такое присваивание, мы нарушаем систему типов. Можно использовать приведение (§5.5, §15.16), чтобы гарантировать, что `pvec` ссылается на `ColoredPoint[]`.

```
cpvec = (ColoredPoint[])pvec; // OK, но может сгенерировать
                              // исключение времени выполнения
```

§5.3. Контексты вызова

Контексты вызова позволяют значению аргумента вызова метода или конструктора (§8.8.7.1, §15.9, §15.12) быть присвоенным соответствующему формальному параметру.

Строгий контекст вызова позволяет использовать одно из следующих преобразований.

- Тождественное преобразование (§5.1.1).
- Расширяющее примитивное преобразование (§5.1.2).
- Расширяющее ссылочное преобразование (§5.1.5).

Свободный контекст вызова допускает более широкий набор преобразований, так как они используются только для конкретных вызовов, если применимое объявление не может быть найдено в рамках строгого контекста вызова. Свободный контекст вызова позволяет использовать одно из следующих преобразований.

- Тождественное преобразование (§5.1.1).
- Расширяющее примитивное преобразование (§5.1.2).
- Расширяющее ссылочное преобразование (§5.1.5).
- Преобразование упаковки (§5.1.7), за которым при необходимости следует расширяющее ссылочное преобразование.
- Преобразование распаковки (§5.1.8), за которым при необходимости следует расширяющее примитивное преобразование.

Если после применения перечисленных для контекста вызова преобразований результирующий тип является несформированным (§4.8), может дополнительно применяться непроверяемое преобразование (§5.1.9).

Значение типа null (единственным таким значением является `null`) может быть присвоено любому ссылочному типу, давая в результате пустую ссылку данного типа.

Если цепочка преобразований содержит два параметризованных типа, которые не связаны отношением подтипа (§4.10), генерируется ошибка времени компиляции.

Если тип выражения не может быть преобразован в тип параметра с помощью разрешенных в свободном контексте вызова преобразований, генерируется ошибка времени компиляции.

Если тип выражения аргумента — `float` или `double`, то после преобразования типа применяется преобразование набора значений (§5.1.13).

- Если значение аргумента имеет тип `float` и является элементом расширенного набора значений float, то реализация должна отображать это значение на ближайший элемент набора значений float. Это преобразование может привести к переполнению или потере точности.
- Если значение аргумента имеет тип `double` и является элементом расширенного набора значений double, то реализация должна отображать это значение на ближайший элемент набора значений double. Это преобразование может привести к переполнению или потере точности.

В контексте вызова могут генерироваться только следующие исключения.

- `ClassCastException`, если после применения перечисленных выше преобразований типов результирующее значение представляет собой объект, который не является экземпляром подкласса или подынтерфейса затирания (§4.6) типа соответствующего формального параметра.
- `OutOfMemoryError` в результате преобразования упаковки.
- `NullPointerException` в результате преобразования распаковки пустой ссылки.

Ни строгий, ни свободный контекст вызова не включают неявное сужение целочисленных констант, которое позволено контекстом присваивания. Разработчики языка программирования Java почувствовали, что включение этих неявных сужающих преобразований добавит дополнительные сложности в процесс выбора одного из перегруженных методов (§15.12.2).

Таким образом, программа

```
class Test {
    static int m(byte a, int b) { return a+b; }
    static int m(short a, short b) { return a-b; }
    public static void main(String[] args) {
        System.out.println(m(12,2)); // Ошибка времени
                                     // компиляции
    }
}
```

приводит к ошибке времени компиляции, потому что целочисленные литералы 12 и 2 имеют тип `int`, так что ни один метод `m` не подходит под правила из раздела §15.12.2. Язык, который включает неявное сужение целочисленных констант, требует дополнительных правил для разрешения подобных ситуаций.

§5.4. Строковые контексты

Строковые контексты применяются только к операнду бинарного оператора +, который не является объектом типа `String`, если второй операнд принадлежит типу `String`.

В этом контексте целевым типом всегда является `String`, и всегда выполняется строковое преобразование (§5.1.11) операнда бинарного оператора +, не являющегося объектом `String`. Вычисление оператора + выполняется, как указано в §15.18.1.

§5.5. Контекст приведения

Контекст приведения позволяет операнду оператора приведения (§15.16) быть преобразованным в тип, явно указанный оператором приведения.
Контексты приведения позволяют использовать одно из следующих преобразований.

- Тождественное преобразование (§5.1.1).
- Расширяющее примитивное преобразование (§5.1.2).
- Сужающее примитивное преобразование (§5.1.3).

- Расширяющее и сужающее примитивные преобразования (§5.1.4).
- Расширяющее ссылочное преобразование (§5.1.5), за которым при необходимости следует либо преобразование распаковки (§5.1.8), либо непроверяемое преобразование (§5.1.9).
- Сужающее ссылочное преобразование (§5.1.6), за которым при необходимости следует либо преобразование распаковки (§5.1.8), либо непроверяемое преобразование (§5.1.9).
- Преобразование упаковки (§5.1.7), за которым при необходимости следует расширяющее ссылочное преобразование.
- Преобразование распаковки (§5.1.8), за которым при необходимости следует расширяющее примитивное преобразование.

Преобразование набора значений (§5.1.13) применяется после преобразования типа.

Корректность преобразования приведения времени компиляции определяется следующим образом.

- Выражение примитивного типа может пройти преобразование приведения к другому примитивному типу путем тождественного преобразования (если типы одинаковы), расширяющего примитивного преобразования, сужающего примитивного преобразования или с помощью расширяющего и сужающего примитивных преобразований.
- Выражение примитивного типа может пройти преобразование приведения к ссылочному типу без ошибок с помощью преобразования упаковки.
- Выражение ссылочного типа может пройти преобразование приведения к примитивному типу без ошибок с помощью преобразования распаковки.
- Выражение ссылочного типа может пройти преобразование приведения к другому ссылочному типу, если не произойдет ошибки времени компиляции согласно правилам из §5.5.1.

В приведенных далее табл. 5.1 и 5.2 перечислено, какие преобразования используются в некоторых преобразованиях приведения. Каждое преобразование обозначается соответствующим символом:

- – означает недопустимость преобразования приведения;
- ≈ означает тождественное преобразование (§5.1.1);
- ω означает расширяющее примитивное преобразование (§5.1.2);
- η означает сужающее примитивное преобразование (§5.1.3);
- ωη означает расширяющее и сужающее примитивные преобразования (§5.1.4);
- ⇑ означает расширяющее ссылочное преобразование (§5.1.5);
- ⇓ означает сужающее ссылочное преобразование (§5.1.6);
- ⊕ означает преобразование упаковки (§5.1.7);
- ⊗ означает преобразование распаковки (§5.1.8).

В таблицах запятая между символами указывает, что преобразование приведения использует одно преобразование со следующим за ним другим преобразованием. Тип `Object`

означает любой ссылочный тип, отличный от восьми классов-оберток `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.

Таблица 5.1. Преобразование приведения к примитивным типам

В → **Из ↓**	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**boolean**
byte	$\approx$	ω	$\omega\eta$	ω	ω	ω	ω	–
short	η	$\approx$	η	ω	ω	ω	ω	–
char	η	η	$\approx$	ω	ω	ω	ω	–
int	η	η	η	$\approx$	ω	ω	ω	–
long	η	η	η	η	$\approx$	ω	ω	–
float	η	η	η	η	η	$\approx$	ω	–
double	η	η	η	η	η	η	$\approx$	–
boolean	–	–	–	–	–	–	–	$\approx$
Byte	$\otimes$	$\otimes,\omega$	–	$\otimes,\omega$	$\otimes,\omega$	$\otimes,\omega$	$\otimes,\omega$	–
Short	–	$\otimes$	–	$\otimes,\omega$	$\otimes,\omega$	$\otimes,\omega$	$\otimes,\omega$	–
Character	–	–	$\otimes$	$\otimes,\omega$	$\sqcup,\omega$	$\otimes,\omega$	$\otimes,\omega$	–
Int	–	–	–	$\otimes$	$\otimes,\omega$	$\otimes,\omega$	$\sqcup,\omega$	–
Long	–	–	–	–	$\otimes$	$\otimes,\omega$	$\otimes,\omega$	–
Float	–	–	–	–	–	$\otimes$	$\otimes,\omega$	–
Double	–	–	–	–	–	–	$\otimes$	–
Boolean	–	–	–	–	–	–	–	$\otimes$
Object	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$	$\Downarrow,\otimes$

Таблица 5.2. Преобразование приведения к ссылочным типам

В → **Из ↓**	**Byte**	**Short**	**Character**	**Int**	**Long**	**Float**	**Double**	**Boolean**	**Object**
byte	$\oplus$	–	–	–	–	–	–	–	$\oplus,\Uparrow$
short	–	$\oplus$	–	–	–	–	–	–	$\oplus,\Uparrow$
char	–	–	$\oplus$	–	–	–	–	–	$\oplus,\Uparrow$
int	–	–	–	$\oplus$	–	–	–	–	$\oplus,\Uparrow$
long	–	–	–	–	$\oplus$	–	–	–	$\oplus,\Uparrow$
float	–	–	–	–	–	$\oplus$	–	–	$\oplus,\Uparrow$
double	–	–	–	–	–	–	$\oplus$	–	$\oplus,\Uparrow$
boolean	–	–	–	–	–	–	–	$\oplus$	$\oplus,\Uparrow$
Byte	$\approx$	–	–	–	–	–	–	–	$\Uparrow$
Short	–	$\approx$	–	–	–	–	–	–	$\Uparrow$

Окончание табл. 5.2

В → / **Из ↓**	**Byte**	**Short**	**Character**	**Int**	**Long**	**Float**	**Double**	**Boolean**	**Object**
Character	–	–	≈	–	–	–	–	–	⇑
Int	–	–	–	≈	–	–	–	–	⇑
Long	–	–	–	–	≈	–	–	–	⇑
Float	–	–	–	–	–	≈	–	–	⇑
Double	–	–	–	–	–	–	≈	–	⇑
Boolean	–	–	–	–	–	–	–	≈	⇑
Object	⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓	≈

§5.5.1. Приведение ссылочных типов

Для данных ссылочного типа времени компиляции S (исходный) и ссылочного типа времени компиляции T (целевой) существует преобразование приведения S к T, если в соответствии с приведенными далее правилами не возникает ошибка времени компиляции.

Если S представляет собой тип класса.

- Если T является типом класса, то либо $|S| <: |T|$, либо $|T| <: |S|$. В противном случае генерируется ошибка времени компиляции.

  Кроме того, если существуют супертип X типа T и супертип Y типа S, такие, что и X, и Y являются доказуемо различными параметризованными типами (§4.5), и при этом затирания X и Y одинаковы, генерируется ошибка времени компиляции.

- Если T является типом интерфейса.

  ✦ Если S не является классом, объявленным как `final` (§8.1.1), то, если существуют супертип X типа T и супертип Y типа S, такие, что и X, и Y являются доказуемо различными параметризованными типами, и при этом затирания X и Y одинаковы, генерируется ошибка времени компиляции.

  В противном случае приведение времени компиляции всегда корректно (поскольку, даже если S не реализует T, это может делать подкласс S).

  ✦ Если S является классом, объявленным как `final` (§8.1.1), то S должен реализовывать T, иначе генерируется ошибка времени компиляции.

- Если T является переменной типа, то данный алгоритм применяется рекурсивно, с использованием вместо T верхней границы T.
- Если T является типом массива, то S должен быть классом `Object`, иначе генерируется ошибка времени компиляции.
- Если T является типом пересечения, T_1 & ... & T_n, то если существует T_i ($1 \le i \le n$), такой, что S не может быть приведено к T_i этим алгоритмом, генерируется ошибка вре-

мени компиляции. То есть успех приведения определяется наиболее ограничивающим компонентом типа пересечения.

Если *S* представляет собой тип интерфейса.

- Если *T* является типом массива, то *S* должен быть типом `java.io.Serializable` или `Cloneable` (единственные интерфейсы, реализованные массивами), иначе генерируется ошибка времени компиляции.
- Если *T* является типом класса или интерфейса, не объявленным как `final` (§8.1.1), то, если существуют супертип *X* типа *T* и супертип *Y* типа *S*, такие, что и *X*, и *Y* являются доказуемо различными параметризованными типами, и при этом затирания *X* и *Y* одинаковы, генерируется ошибка времени компиляции.

  В противном случае приведение времени компиляции всегда корректно (поскольку, даже если *T* не реализует *S*, это может делать подкласс *T*).
- Если *T* является типом класса, объявленным как `final`.
  - ✦ Если *S* не является параметризованным или несформированным типом, то *T* должен реализовывать *S*, иначе генерируется ошибка времени компиляции.
  - ✦ В противном случае *S* представляет собой либо параметризованный тип, являющийся конкретизацией объявления некоторого обобщенного типа *G*, либо несформированным типом, соответствующим объявлению обобщенного типа *G*. Тогда должен существовать супертип *X* типа *T*, такой, что *X* представляет собой конкретизацию *G*, иначе генерируется ошибка времени компиляции.

    Кроме того, если *S* и *T* являются доказуемо различными параметризованными типами, генерируется ошибка времени компиляции.
- Если *T* является переменной типа, то этот алгоритм применяется рекурсивно, с использованием верхней границы *T* вместо *T*.
- Если *T* является типом пересечения, T_1 & ... & T_n, то, если существует T_i ($1 \le i \le n$), такой, что *S* не может быть приведено к T_i этим алгоритмом, генерируется ошибка времени компиляции.

Если *S* представляет собой переменную типа, то данный алгоритм применяется рекурсивно, с использованием вместо *S* верхней границы *S*.

Если *S* является типом пересечения A_1 &...& A_n, то, если существует A_i ($1 \le i \le n$), такой, что *S* не может быть приведен к A_i данным алгоритмом, генерируется ошибка времени компиляции. То есть успех приведения определяется наиболее ограничивающим типом пересечения.

Если *S* является типом массива *SC*`[]`, т.е. массивом компонентов типа *SC*.

- Если *T* является типом класса, то, если *T* не является `Object`, генерируется ошибка времени компиляции (поскольку `Object` — единственный тип класса, которому может быть присвоен массив).
- Если *T* является типом интерфейса, то, если только *T* не является типом `java.io.Serializable` или `Cloneable` (единственные интерфейсы, реализуемые массивами), генерируется ошибка времени компиляции.

- Если T является переменной типа, то этот алгоритм применяется рекурсивно, с использованием верхней границы T вместо T.
- Если T является типом массива `TC[]`, т.е. массивом компонентов типа TC, то ошибка времени компиляции генерируется во всех случаях, кроме ситуаций, когда истинно одно из следующих условий.
  - ✦ TC и SC являются одним и тем же примитивным типом.
  - ✦ TC и SC являются ссылочными типами, и тип SC может быть преобразован в TC преобразованием приведения.
- Если T является типом пересечения, T_1 & ... & T_n, то, если существует T_i ($1 \le i \le n$), такой, что S не может быть приведен к T_i этим алгоритмом, генерируется ошибка времени компиляции.

Пример 5.5.1-1. Преобразование приведения для ссылочных типов

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point {}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
        // Приведенный далее код может вызвать ошибки
        // времени выполнения, поскольку мы не можем
        // гарантировать его успешность; такая возможность
        // предполагается приведениями:
        cp = (ColoredPoint)p;   // p может не ссылаться
                    // на объект, являющийся ColoredPoint
                    // или подклассом ColoredPoint
        c = (Colorable)p;       // p может не быть Colorable
        // Приведенный далее код не компилируется, поскольку
        // не может быть успешно выполнен ни при каких
        // условиях (см. текст раздела):
        Long l = (Long)p;   // Ошибка времени компиляции #1
        EndPoint e = new EndPoint();
        c = (Colorable)e;   // Ошибка времени компиляции #2
    }
}
```

Здесь первая ошибка времени компиляции возникает из-за того, что типы классов `Long` и `Point` не связаны (т.е. они не одинаковы, и один из них не является подклассом другого), так что приведение между ними всегда будет приводить к ошибке.

Вторая ошибка времени компиляции возникает, поскольку переменная типа `EndPoint` не может ссылаться на значение, которое реализует интерфейс `Colorable`. Это связано с тем, что `EndPoint` представляет собой тип, объявленный как `final`, а переменная такого типа всегда содержит значение того же типа времени выполнения, что и тип времени компиляции. Таким образом, тип времени выполнения переменной `e` должен быть в точности типом `EndPoint`, а тип `EndPoint` не реализует `Colorable`.

Пример 5.5.1-2. Преобразование приведения для типов массивов

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}

class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
            System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" }");
    }
}
```

Эта программа компилируется без ошибок и дает следующий вывод.

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

§5.5.2. Проверяемые и непроверяемые приведения

Приведение типа S к типу T является *приведением со статически известной корректностью* тогда и только тогда, когда $S <: T$ (§4.10).

Приведение типа S к параметризованному типу (§4.5) T является *непроверяемым* (unchecked), если только не выполняется хотя бы одно из приведенных далее условий.

- S `<:` T.
- Все аргументы типа (§4.5.1) T представляют собой неограниченные символы подстановки.
- T `<:` S и S не имеет подтипа X, отличного от T, где аргументы типа X не содержатся в аргументах типа T.

Приведение типа S к переменной типа T является непроверяемым, если только не выполняется условие S `<:` T.

Приведение типа S к типу пересечения T_1 & ... & T_n является непроверяемым, если существует T_i ($1 \le i \le n$), такой, что приведение S к T_i является непроверяемым.

Непроверяемое приведение типа S к типу T, не являющемуся пересечением, *полностью непроверяемое*, если приведение |S| к |T| — приведение со статически известной корректностью. В противном случае оно является *частично непроверяемым*.

Непроверяемое приведение типа S к типу пересечения T_1 & ... & T_n является *полностью непроверяемым*, если для всех i ($1 \le i \le n$) приведение S к T_i является либо приведением со статически известной корректностью, либо полностью непроверяемым. В противном случае такое приведение является *частично непроверяемым*.

Непроверяемое приведение вызывает предупреждение времени компиляции о непроверенном преобразовании типов, если только оно не подавляется аннотацией `SuppressWarnings` (§9.6.3.5).

Приведение является *проверяемым*, если оно не является приведением со статически известной корректностью и не является непроверяемым приведением.

Если приведение к ссылочному типу не является ошибкой времени компиляции, возможны несколько ситуаций.

- Приведение является приведением со статически известной корректностью.

  При выполнении такого приведения никакие действия времени выполнения не осуществляются.
- Приведение является полностью непроверяемым приведением.

  При выполнении такого приведения никакие действия времени выполнения не осуществляются.
- Приведение является частично непроверяемым или проверяемым приведением к типу пересечения.

  Если типом пересечения является T_1 & ... & T_n, то для всех i ($1 \le i \le n$) любая проверка времени выполнения, требующаяся для приведения S к T_i, требуется и для приведения к типу пересечения.
- Приведение является частично непроверяемым приведением к типу, не являющемуся типом пересечения.

  Такое приведение требует проверки корректности во время выполнения. Проверка выполняется, как если бы приведение было проверяемым приведением между |S| и |T|, как описано ниже.
- Приведение является проверяемым приведением к типу, не являющемуся типом пересечения.

Такое приведение требует проверки корректности во время выполнения. Если значение времени выполнения равно `null`, приведение разрешено. В противном случае пусть *R* является классом объекта, на который ссылается ссылочное значение времени выполнения, а *T* представляет собой затирание (§4.6) типа, указанного в операции приведения. Преобразование приведения должно проверить во время выполнения, что класс *R* совместим по присваиванию с типом *T*, воспользовавшись алгоритмом из §5.5.3.

Обратите внимание, что *R* не может быть интерфейсом при первом применении этих правил к любому заданному приведению; но *R* может быть интерфейсом, если правила применяются рекурсивно, потому что значение ссылки времени выполнения может ссылаться на массив, тип элементов которого является типом интерфейса.

§5.5.3. Проверяемые приведения времени выполнения

Вот как выглядит алгоритм проверки, является ли тип времени выполнения *R* объекта совместимым по присваиванию с типом *T*, который представляет собой затирание (§4.6) типа, указанного в операторе приведения. В случае генерации исключения времени выполнения генерируется `ClassCastException`.

Если *R* является обычным классом (не классом массива).

- Если *T* представляет собой тип класса, то *R* должен быть либо тем же классом (§4.3.4), что и *T*, либо подклассом *T*, либо генерируется исключение времени выполнения.
- Если *T* представляет собой тип интерфейса, то *R* должен реализовывать (§8.1.5) интерфейс *T*, иначе генерируется исключение времени выполнения.
- Если *T* представляет собой тип массива, генерируется исключение времени выполнения.

Если *R* является интерфейсом.

- Если *T* представляет собой тип класса, то *T* должен быть `Object` (§4.3.2), иначе генерируется исключение времени выполнения.
- Если *T* представляет собой тип интерфейса, то *R* должен быть либо тем же интерфейсом, что и *T*, либо подынтерфейсом T, иначе генерируется исключение времени выполнения.
- Если *T* представляет собой тип массива, генерируется исключение времени выполнения.

  Если *R* — класс, представляющий тип массива *RC*`[]`, т.е. массив компонентов типа *RC*.
- Если *T* представляет собой тип класса, то *T* должен быть `Object` (§4.3.2), иначе генерируется исключение времени выполнения.
- Если *T* представляет собой тип интерфейса, то генерируется исключение времени выполнения, если только *T* не является типом `java.io.Serializable` или `Cloneable` (единственные интерфейсы, реализуемые массивами).

> Этот случай может пройти проверку времени компиляции, если, например, ссылка на массив была сохранена в переменной типа `Object`.

- Если *T* представляет собой тип массива *TC*[], т.е. массив компонентов типа *TC*, то генерируется исключение времени выполнения, если только не выполняется одно из следующих условий.
  - ✦ *TC* и *RC* являются одним и тем же примитивным типом.
  - ✦ *TC* и *RC* являются ссылочными типами, и тип *RC* может быть приведен к типу *TC* с помощью рекурсивного применения этих правил времени выполнения для приведения.

Пример 5.5.3-1. Несовместимые типы времени выполнения

```
class Point { int x, y; }
interface Colorable { void setColor(int color); }
class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];

        // Следующая строка генерирует
        // исключение ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;

        // Следующая строка генерирует
        // исключение ClassCastException:
        Colorable c = (Colorable)o;
        c.setColor(0);
    }
}
```

Эта программа использует приведения для компиляции, однако из-за несовместимости типов во время выполнения генерируются исключения.

§5.6. Числовые контексты

Числовые контексты применяются к операндам арифметического оператора. Числовые контексты позволяют использовать следующие преобразования.

- Тождественное преобразование (§5.1.1)
- Расширяющее примитивное преобразование (§5.1.2)
- Преобразование распаковки (§5.1.8), за которым необязательно следует расширяющее примитивное преобразование

Числовое повышение является процессом, с помощью которого для данного арифметического оператора и выражений его аргументов аргументы преобразуются в выведенный целевой тип T. Тип T выбирается в процессе повышения так, что каждое выражение аргумента может быть преобразовано в T и для значений типа T определена данная арифметическая операция.

Числовые повышения подразделяются на два вида: унарное числовое повышение (§5.6.1) и бинарное числовое повышение (§5.6.2).

§5.6.1. Унарное числовое повышение

Некоторые операторы применяют *унарное числовое повышение* к единственному операнду, который должен порождать значение числового типа.

- Если операнд имеет тип времени компиляции `Byte`, `Short`, `Character` или `Integer`, к нему применяется преобразование распаковки (§5.1.8). Затем результат повышается до значения типа `int` расширяющим примитивным преобразованием (§5.1.2) или тождественным преобразованием (§5.1.1).
- В противном случае, если операнд имеет тип `Long`, `Float` или `Double`, применяется преобразование распаковки (§5.1.8).
- В противном случае, если операнд имеет тип времени компиляции `byte`, `short` или `char`, он повышается до значения типа `int` расширяющим примитивным преобразованием (§5.1.2).
- В противном случае унарный числовой операнд остается неизменным.

После преобразований, если таковые имели место, применяется преобразование набора значений (§5.1.13).

Унарное числовое повышение выполняется над выражениями в следующих ситуациях.

- Каждое выражение измерения в выражении создания массива (§15.10).
- Выражение индекса в выражении обращения к массиву (§15.13).
- Операнд унарного оператора “плюс” (+) (§15.15.3).
- Операнд унарного оператора “минус” (−) (§15.15.4)
- Операнд оператора побитового дополнения (~) (§15.15.5)
- Каждый операнд операторов сдвига >>, >>> и << по отдельности (§15.19). Величина сдвига типа `long` (правый операнд) не повышает сдвигаемую величину (левый операнд) до `long`.

Пример 5.6.1-1. Унарное числовое повышение

```
class Test {
    public static void main(String[] args) {
        byte b = 2;
        int a[] = new int[b]; // Повышение выражения измерения
        char c = '\u0001';
        a[c] = 1;             // Повышение выражения индекса
```

```
        a[0] = -c;                  // Повышение унарного минуса (-)
        System.out.println("a: " + a[0] + "," + a[1]);
        b = -1;
        int i = ~b;             // Повышение побитового дополнения
        System.out.println("~0x" + Integer.toHexString(b)
                           + "==0x" + Integer.toHexString(i));
        i = b << 4L;            // Повышение сдвига (левый операнд)
        System.out.println("0x" + Integer.toHexString(b)
                              + "<<4L==0x" + Integer.toHexString(i));
    }
}
```

Вывод программы имеет следующий вид.

```
a: -1,1
~0xffffffff==0x0
0xffffffff<<4L==0xfffffff0
```

§5.6.2. Бинарное числовое повышение

Когда оператор применяет *бинарное числовое повышение* к паре операндов, каждый из которых должен представлять значение, преобразуемое в числовой тип, применяются следующие правила (в указанном порядке).

1. Если какой-либо из операндов имеет ссылочный тип, к нему применяется преобразование распаковки (§5.1.8).
2. К одному или обоим операндам в соответствии с приведенными далее правилами может применяться расширяющее примитивное преобразование (§5.1.2).
   - Если один из операндов имеет тип `double`, второй операнд преобразуется в `double`.
   - В противном случае, если один из операндов имеет тип `float`, второй операнд преобразуется во `float`.
   - В противном случае, если один из операндов имеет тип `long`, второй операнд преобразуется в `long`.
   - В противном случае оба операнда преобразуются в `int`.

После преобразований, если таковые имели место, применяется преобразование набора значений (§5.1.13).

Бинарное числовое повышение применяется к операндам определенных операторов.

- Мультипликативные операторы *, / и % (§15.17)
- Операторы сложения и вычитания для числовых типов + и – (§15.18.2)
- Операторы числового сравнения <, <=, > и >= (§15.20.1)
- Операторы числового равенства == и != (§15.21.1)
- Целочисленные побитовые операторы &, ^ и | (§15.22.1)
- В определенных ситуациях условный оператор ? : (§15.25)

Пример 5.6.2-1. Бинарное числовое повышение

```
class Test {
    public static void main(String[] args) {
        int i = 0;
        float f = 1.0f;
        double d = 2.0;

        // Сначала int*float повышается до float*float,
        // затем float==double повышается до double==double:
        if (i * f == d) System.out.println("oops");

        // char&byte повышается до int&int:
        byte b = 0x1f;
        char c = 'G';
        int control = c & b;
        System.out.println(Integer.toHexString(control));

        // Здесь int:float повышается до float:float:
        f = (b==0) ? i : 4.0f;
        System.out.println(1.0/f);
    }
}
```

Вывод программы имеет следующий вид.

```
7
0.25
```

В примере выполняется преобразование ASCII-символа `G` в символ ASCII Ctrl-G (BEL) путем маскировки всех, кроме пяти младших, битов символа. Числовое значение этого управляющего символа равно `7`.

ГЛАВА 6

Имена

Имена используются для обращения к сущностям, объявленным в программе.

Объявленная сущность (§6.1) представляет собой пакет, тип класса (обычный или перечисление), тип интерфейса (обычный или аннотация типа), член (класс, интерфейс, поле или метод) ссылочного типа, параметр типа (класса, интерфейса, метода или конструктора), параметр (метода, конструктора или обработчика исключения) или локальную переменную.

Имена в программах являются либо *простыми*, состоящими из одного идентификатора, либо *квалифицированными*, состоящими из последовательности идентификаторов, разделенных токенами "." (§6.2).

Каждое объявление, которое вводит имя, имеет *область видимости* (scope) (§6.3), представляющую собой часть текста программы, в рамках которой к объявленной сущности можно обратиться посредством простого имени.

Квалифицированное имя `N.x` может использоваться для обозначения *члена* пакета или ссылочного типа, где `N` — простое или квалифицированное имя, а `x` — это идентификатор. Если `N` именует пакет, то `x` именует член этого пакета — тип класса или интерфейса, или подпакет. Если `N` именует ссылочный тип или переменную ссылочного типа, то `x` именует член этого типа, который представляет собой класс, интерфейс, поле или метод.

При определении значения имени (§6.5) для разрешения неоднозначности при наличии пакетов, типов, переменных и методов с одним и тем же именем используется контекст его применения.

Управление доступом (§6.6) может быть указано в объявлении класса, интерфейса, метода или поля для определения, когда разрешен *доступ* к члену. Доступ — концепция, отличная от концепции области видимости. Доступ указывает часть исходного текста программы, в рамках которой к объявленной сущности можно обратиться с помощью квалифицированного имени. Доступ к объявленной сущности также актуален в выражении доступа к полю (§15.11), выражении вызова метода, в котором метод указан не с помощью простого имени (§15.12), выражении ссылки на метод (§15.13) или выражении создания экземпляра квалифицированного класса (§15.9). При отсутствии модификатора доступа большинство объявлений имеют доступ на уровне пакета, при котором обращение к члену разрешено откуда угодно из пакета, содержащего его объявление. Другими возможностями являются `public`, `protected` и `private`.

В этой главе рассматриваются также полностью квалифицированные и канонические имена (§6.7).

§6.1. Объявления

Объявление вводит в программу некоторую сущность и включает идентификатор (§3.8), который может использоваться в имени, ссылающемся на эту сущность.

Объявленная сущность представляет собой одно из следующего списка.

- Пакет, описанный в объявлении `package` (§7.4).
- Импортируемый тип, описанный в объявлении единственного импортируемого типа (§7.5.1) или в объявлении импортирования типа по требованию (§7.5.2).
- Импортированный `static`-член, объявленный в объявлении единственного статического импорта или статического импорта по требованию (§7.5.3, §7.5.4).
- Класс, описанный в объявлении типа класса (§8.1).
- Интерфейс, описанный в объявлении типа интерфейса (§9.1).
- Параметр типа, объявленный как часть объявления обобщенного класса, интерфейса, метода или конструктора (§8.1.2, §9.1.2, §8.4.4, §8.8.1).
- Член ссылочного типа (§8.2, §9.2, §8.9.3, §9.6, §10.7), один из следующего списка.
  - ✦ Класс-член (§8.5, §9.5).
  - ✦ Интерфейс-член (§8.5, §9.5).
  - ✦ Константа перечисления (§8.9).
  - ✦ Поле, одно из следующего списка.
    - — Поле, объявленное в типе класса (§8.3, §8.9.2).
    - — Поле, объявленное в типе интерфейса (§9.3, §9.6.1).
    - — Поле `length`, неявно являющееся членом каждого типа массива (§10.7).
  - ✦ Метод, один из следующего списка.
    - — Метод (объявленный, как `abstract` или иной), объявленный в типе класса или типе перечисления (§8.4, §8.9.2).
    - — Метод (всегда объявленный, как `abstract`), объявленный в типе интерфейса или аннотации (§9.4, §9.6.1).
- Параметр, один из следующего списка.
  - ✦ Формальный параметр метода или конструктора типа класса или перечисления (§8.4.1, §8.8.1, §8.9.2) или лямбда-выражения (§15.27.1).
  - ✦ Формальный параметр абстрактного метода типа интерфейса или аннотации (§9.4, §9.6.1).
  - ✦ Параметр обработчика исключения, объявленный в выражении `catch` конструкции `try` (§14.20).
- Локальная переменная, одна из следующего списка.
  - ✦ Локальная переменная, объявленная в блоке (§14.4).
  - ✦ Локальная переменная, объявленная в заголовке цикла `for` (§14.14).

Конструкторы (§8.8) также вводятся объявлениями, но не вводят новое имя, а используют имя класса, в котором объявлены.

Конструкторы (§8.8) также вводятся объявлениями, но не вводят новое имя, а используют имя класса, в котором они объявлены.

Объявление типа, не являющегося обобщенным (`class C ...`), объявляет единственную сущность: необобщенный тип (`C`). Необобщенный тип не является несформированным типом, несмотря на синтаксическую схожесть. Напротив, объявление обобщенного типа (`class C<T> ...` или `interface C<T> ...`) объявляет две сущности: обобщенный тип (`C<T>`) и соответствующий необобщенный тип (`C`). В этом случае смысл элемента `C` зависит от контекста, в котором он находится.

- Если обобщенность не важна, как в *необобщенном контексте*, определенном ниже, идентификатор `C` обозначает необобщенный тип `C`.
- Если обобщенность важна, как во всех контекстах, начиная с §6.5, за исключением необобщенных контекстов, идентификатор `C` обозначает
  - ✦ либо несформированный тип `C`, который представляет собой затирание (§4.6) обобщенного типа `C<T>`;
  - ✦ либо параметризованный тип, который представляет собой частичную параметризацию (§4.5) обобщенного типа `C<T>`.

  Существует 13 необобщенных контекстов.

  1. В объявлении импорта единственного импорта (§7.5.1).
  2. Слева от точки `.` в объявлении единственного статического импорта (§7.5.3).
  3. Слева от точки `.` в объявлении статического импорта по требованию (§7.5.4).
  4. Слева от `(` в объявлении конструктора (§8.8).
  5. После знака `@` в аннотации (§9.7).
  6. Слева от `.class` в литерале класса (§15.8.2).
  7. Слева от `.this` в квалифицированном выражении `this` (§15.8.4).
  8. Слева от `.super` в выражении доступа к полю квалифицированного суперкласса (§15.11.2).
  9. Слева от *`.Identifier`* или `.super`*`.Identifier`* в выражении вызова квалифицированного метода (§15.12).
  10. Слева от `.super::` в выражении ссылки на метод (§15.13).
  11. В квалифицированном имени выражения в постфиксном выражении (§15.14.1).
  12. В конструкции `throws` метода или конструктора (§8.4.6, §8.8.5, §9.4).
  13. В объявлении параметра исключения (§14.20).

Первые десять необобщенных контекстов соответствуют первым десяти синтаксическим контекстам *TypeName* в §6.5.1. Одиннадцатым необобщенным контекстом является постфиксное выражение, в котором квалифицированное имя *ExpressionName*, такое как `C.x`, может включать *TypeName* `C` для обозначения доступа к статическому члену. Распространенное использование *TypeName* является важным: оно указывает, что эти контексты включают менее чем первоклассное использование типа. Двенадцатый же и тринадцатый необобщенные контексты, напротив, используют *ClassType*, указывая, что конструкции `throws` и `catch` используют типы как элементы первого класса (т.е., на-

пример, создание объекта во время выполнения, передача в качестве параметра и т.д.), в строке со, скажем, объявлением поля. Характеристика этих двух контекстов как необобщенных связана с тем фактом, что тип исключения не может быть параметризован.

> Заметим, что продукция *ClassType* допускает аннотации, так что можно аннотировать применение типа в конструкциях `throws` или `catch`, в то время как продукция *TypeName* запрещает аннотации, так что невозможно аннотировать имя типа в, скажем, объявлении импорта единственного типа.

Соглашения по именованию

Библиотеки классов платформы Java SE пытаются использовать (где это возможно) имена, выбранные в соответствии с представленными ниже соглашениями. Эти соглашения помогают сделать код более удобочитаемым и избежать некоторых видов конфликтов имен.

Мы рекомендуем использовать эти соглашения во всех программах, написанных на языке программирования Java. Однако этим соглашениям не надо следовать рабски, в особенности если давние традиции диктуют иное. Так, например, методы `sin` и `cos` класса `java.lang.Math` имеют удобные математические имена, несмотря на то что эти имена нарушают приведенные ниже соглашения (они являются короткими словами, не являющимися глаголами).

Имена пакетов

Разработчикам следует принять меры, чтобы избежать возможного совпадения имен двух опубликованных пакетов, выбирая *уникальные имена пакетов* для широко распространяемых пакетов. Это позволяет легко и автоматически инсталлировать и каталогизировать пакеты. В данном разделе предлагается соглашение для генерации таких уникальных имен пакетов. Предполагается, что реализации платформы Java SE должны обеспечивать автоматическую поддержку преобразования локальных и случайных имен пакетов в описанный здесь формат уникальных имен.

Если уникальные имена пакетов не используются, то конфликт имен пакетов может возникнуть достаточно далеко от точки создания каждого из пакетов. Это может создать ситуацию, которую пользователю или программисту будет трудно (или вовсе невозможно) решить. В тех случаях, когда пакеты ограниченно взаимодействуют один с другим, для изоляции пакетов с одинаковыми именами может применяться класс `ClassLoader`; но этот способ не подходит для простых программ.

Уникальное имя пакета получается путем обращения доменного имени организации, такого как `oracle.com` (в этом примере получается `com.oracle`), и его использования в качестве префикса для ваших имен пакетов, полученных с помощью соглашений, разработанных в вашей организации для управления именами пакетов. Такое соглашение может указывать, например, что определенные компоненты имени пакета должны быть отделами, проектами, именами машин или учетных записей.

Пример 6.1-1. Уникальные имена пакетов

```
com.nighthacks.java.jag.scrabble
org.openjdk.tools.compiler
net.jcip.annotations
```

```
ua.kiev.kiv.fidonet
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
```

Первый компонент уникального имени пакета всегда записывается строчными буквами ASCII и должен представлять собой имя домена верхнего уровня, такого как `com`, `edu`, `gov`, `mil`, `net` или `org`, или один из двухбуквенных кодов, идентифицирующих страну в соответствии со стандартом *ISO Standard 3166*.

Имя пакета не подразумевает место его хранения в Интернете. Предлагаемое соглашение для генерации уникального имени пакета — не более чем использование широко известного реестра уникальных имен вместо создания нового.

Например, пакет с именем `edu.cmu.cs.bovik.cheese` не обязательно хранится по интернет-адресу `cmu.edu` или `cs.cmu.edu`, или `bovik.cs.cmu.edu`.

В некоторых случаях имя домена может не быть допустимым именем пакета. Ниже приведены некоторые предлагаемые соглашения для таких ситуаций.

- Если имя домена содержит дефис или любой другой специальный символ, не разрешенный в идентификаторах (§3.8), преобразуйте его в подчеркивание.
- Если любой компонент получающегося в результате имени представляет собой ключевое слово (§3.9), добавьте к нему подчеркивание.
- Если любой компонент получающегося в результате имени начинается с цифры или любого другого символа, который не допускается в качестве первого символа идентификатора, добавьте в качестве префикса этого компонента знак подчеркивания.

Имена пакетов, предназначенных только для локального использования, должны иметь первый идентификатор, начинающийся со строчной буквы, но этот первый идентификатор не должен быть `java`; имена пакетов, которые начинаются с идентификатора `java`, зарезервированы для пакетов платформы Java SE.

Имена типов классов и интерфейсов

Имена типов классов должны носить описательный характер и быть существительными словами или фразами, не слишком длинными, в смешанном регистре (первые буквы каждого слова — прописные, остальные — строчные).

Пример 6.1-2. Описательные имена классов

```
ClassLoader
SecurityManager
Thread
Dictionary
BufferedInputStream
```

Аналогично имена типов интерфейса должны быть краткими и носить описательный характер, быть слишком длинными, в смешанном регистре (первые буквы каждого слова — прописные, остальные — строчные). Имя может быть описательным существительным или фразой — такие имена подходят, когда интерфейс используется как абстрактный суперкласс, например интерфейсы `java.io.DataInput` и `java.io.DataOutput`. Имя может быть прилагательным, описывающим поведение, как в случае интерфейсов `Runnable` и `Cloneable`.

Имена переменных типа

Имена переменных типа должны быть краткими (если можно — один символ) и не должны содержать буквы нижнего регистра. Это позволяет легко отличать их от обычных классов и интерфейсов.

Типы контейнеров должны использовать имя `E` для типа их элементов. Ассоциативные контейнеры должны использовать `K` для типа ключей и `V` для типа значений. Имя `X` должно использоваться для произвольных типов исключений. Мы используем `T` для имени типа, когда нет никакой более конкретной информации о типе, чтобы отличать его от других типов. (Это частая ситуация в случае обобщенных методов.)

Если есть несколько параметров типа, которые означают произвольные типы, следует использовать буквы, соседние с `T` в алфавите, такие как `S`. В качестве альтернативы допускается применение числовых индексов (например, `T1`, `T2`), чтобы различать разные переменные типов. В таких случаях все переменные должны быть индексируемыми и иметь один и тот же префикс.

Если обобщенный метод появляется внутри обобщенного класса, то стоит избегать применения одних и тех же имен параметров типа у метода и класса, чтобы избежать путаницы. То же относится и к вложенным обобщенным классам.

Пример 6.1-3. Имена переменных типа, соответствующие соглашению

```
public class HashSet<E> extends AbstractSet<E> { ... }
public class HashMap<K,V> extends AbstractMap<K,V> { ... }
public class ThreadLocal<T> { ... }
public interface Functor<T, X extends Throwable> {
    T eval() throws X;
}
```

Если параметры типа не попадают ни в одну из упомянутых категорий, имена должны быть выбраны значимыми, насколько это возможно в пределах одной буквы. Имена, приведенные выше (`E`, `K`, `V`, `X`, `T`), не должны использоваться для параметров типов, которые не попадают в описанные выше категории.

Имена методов

Имена методов должны быть глаголами или глагольными фразами в смешанном регистре (первая буква имени и первая буква каждого последующего слова — прописные). Вот некоторые дополнительные соглашения для именования методов.

- Методы для получения и установки атрибутов, которые можно рассматривать как переменную `V`, должны именоваться `getV` и `setV`. Примерами таких методов являются методы `getPriority` и `setPriority` класса `Thread`.
- Метод, возвращающий длину чего-либо, должен иметь имя `length`, как в классе `String`.
- Метод, проверяющий логическое условие `V`, связанное с объектом, должен иметь имя `isV`. Примером может служить метод `isInterrupted` класса `Thread`.
- Метод, преобразующий объект в некоторый формат `F`, должен иметь имя `toF`. Примерами таких методов являются метод `toString` класса `Object` и методы `toLocaleString` и `toGMTString` класса `java.util.Date`.

Где это возможно и целесообразно, основывайте имена методов нового класса на именах методов существующего класса, в особенности класса API платформы Java SE, что обеспечит б*о*льшую простоту и легкость их применения.

Имена полей

Имена полей, не объявленных как final, должны быть в смешанном регистре (первая буква имени и первая буква каждого последующего слова — прописные). Обратите внимание, что хорошо спроектированные классы имеют очень немного открытых (`public`) или защищенных (`protected`) полей, за исключением полей-констант (статические поля, объявленные как `final`).

Поля должны иметь имена, которые являются существительными, фразами или их аббревиатурами.

Примерами имен, соответствующих данному соглашению, являются имена полей `buf`, `pos` и `count` класса `java.io.ByteArrayInputStream` и поле `bytesTransferred` класса `java.io.InterruptedIOException`.

Имена констант

Имена констант в типах интерфейсов должны быть (а имена переменных `final` в типах классов могут быть) последовательностью из одного или нескольких слов или аббревиатур, все буквы в которых — прописные, а компоненты разделены символами подчеркивания "_". Имена констант должны быть описательными и не быть излишне сокращенными. По соглашению они могут быть любыми подходящими частями речи.

Примеры имен констант включают `MIN_VALUE`, `MAX_VALUE`, `MIN_RADIX` и `MAX_RADIX` класса `Character`.

Группа констант, которые представляют собой альтернативные значения набора, или, реже, маскирующая биты в целочисленных значениях, зачастую именуется с некоторой аббревиатурой, являющейся общим префиксом имен группы.

Вот пример.

```
interface ProcessStates {
    int PS_RUNNING   = 0;
    int PS_SUSPENDED = 1;
}
```

Имена локальных переменных и параметров

Имена локальных переменных и параметров должны быть краткими, но осмысленными. Зачастую они представляют собой последовательности строчных символов, не являющиеся словами, как показано в следующем списке.

- Аббревиатуры, т.е. первые буквы последовательностей слов, как `cp` для переменной, хранящей ссылку на объект класса `ColoredPoint`.
- Сокращения, как, например, переменная `buf` для хранения указателя на буфер некоторого вида.
- Мнемонические термины, организованные таким образом, чтобы облегчить запоминание и понимание, обычно при использовании наборов локальных переменных по образцу имен полей распространенных классов, например:

— `in` и `out`, некоторые виды ввода и вывода;

— `off` и `len`, представляющие смещение и длину.

Односимвольных имен локальных переменных или параметров следует избегать, за исключением временных переменных и переменных циклов или когда переменная содержит неважное значение. Обычно используемыми односимвольными именами являются:

- `b` — для переменной типа `byte`;
- `c` — для переменной типа `char`;
- `d` — для переменной типа `double`;
- `e` — для исключения `Exception`;
- `f` — для переменной типа `float`;
- `i`, `j` и `k` — для целочисленных значений;
- `l` — для переменной типа `long`;
- `o` — для переменной типа `Object`;
- `s` — для переменной типа `String`;
- `v` — для произвольной переменной некоторого типа.

Имена локальных переменных или параметров, состоящие из двух или трех строчных букв, не должны конфликтовать с двухбуквенными кодами стран и доменными именами, являющимися первыми компонентами уникальных имен пакетов.

§6.2. Имена и идентификаторы

Имя используется для обращения к сущности, объявленной в программе.

Имеются две формы имен: простые имена и квалифицированные имена.

Простое имя представляет собой отдельный идентификатор.

Квалифицированное имя состоит из имени, токена “ . ” и идентификатора.

При определении значения имени (§6.5) следует учитывать контекст, в котором встречается имя. Правила §6.5 различают контексты, где имя должно обозначать (ссылаться на) пакет (§6.5.3), тип (§6.5.5), переменную или значение в выражении (§6.5.6), или метод (§6.5.7).

Пакеты и ссылочные типы имеют *члены*, которые могут быть доступны посредством квалифицированных имен. В качестве основы для обсуждения квалифицированных имен и определения смысла имен рекомендуется обратиться к описанию членов в §4.4, §4.5.2, §4.8, §4.9, §7.1, §8.2, §9.2 и §10.7.

Не все идентификаторы в программе являются частью имени. Идентификаторы используются также в следующих ситуациях.

- В объявлениях (§6.1), где идентификатор может указывать имя, под которым будет известна объявленная сущность.
- Как метки в инструкциях с метками (§14.7) и в инструкциях `break` и `continue` (§14.15, §14.16), которые ссылаются на метки.

Идентификаторы, используемые в помеченных инструкциях и связанных с ними инструкциях `break` и `continue`, полностью изолированы от используемых в объявлениях.

- В выражениях доступа к полю (§15.11), где идентификатор, расположенный после токена "`.`", обозначает член объекта, описываемого выражением перед токеном "`.`", или объекта, описываемого `super` или *TypeName*`.super` перед токеном "`.`".
- В некоторых выражениях вызова метода (§15.12), где идентификатор может встречаться после токена "`.`" и перед токеном "`(`" для указания вызываемого метода объекта, описываемого выражением перед токеном "`.`", или типа, описываемого *TypeName* перед токеном "`.`", или объекта, описываемого `super` или *TypeName*`.super` перед токеном "`.`".
- В некоторых выражениях ссылки на метод (§15.13), где идентификатор встречается после токена "`::`" для указания метода объекта, описываемого выражением перед токеном "`::`", или типа, описываемого *TypeName* перед токеном "`::`", или объекта, описываемого `super` или *TypeName*`.super` перед токеном "`::`".
- В выражении создания экземпляра квалифицированного класса (§15.9), в котором идентификатор располагается непосредственно справа от крайнего слева токена `new` и указывает тип, который является членом типа времени компиляции выражения, предшествующего токену `new`.
- В парах "элемент–значение" аннотаций (§9.7.1) для обозначения элемента соответствующего типа аннотации.

В программе

```
class Test {
    public static void main(String[] args) {
        Class c = System.out.getClass();
        System.out.println(c.toString().length() +
                           args[0].length() + args.length);
    }
}
```

идентификаторы `Test`, `main` и первые вхождения `args` и `c` не являются именами. Вместо этого они используются в объявлениях для указания имен объявленных сущностей. В этом примере встречаются имена `String`, `Class`, `System.out.getClass`, `System.out.println`, `c.toString`, `args` и `args.length`.

Слово `length` в `args.length` является именем, поскольку `args.length` представляет собой квалифицированное имя (§6.5.6.2) и не является выражением доступа к полю (§15.11). Выражение доступа к полю, подобно выражению вызова метода, выражению ссылки на метод и выражению создания экземпляра квалифицированного класса, использует для указания интересующего члена идентификатор, а не имя. Таким образом, находящееся в `args[0].length()` слово `length` является *не* именем, а идентификатором в выражении вызова метода (§15.12).

Можно задаться вопросом, почему эти виды выражений используют идентификаторы, а не просто имя, которое в конечном итоге представляет собой не что иное, как просто идентификатор. Причина в том, что имя простого выражения опреде-

ляется в терминах лексической среды; т.е. имя простого выражения должно быть в области видимости объявления переменной (§6.5.6.1). С другой стороны, доступ к полю, вызов квалифицированного метода, ссылка на метод и создание экземпляра квалифицированного класса ссылаются на члены, имена которых не находятся в лексической среде. По определению такие имена ограничиваются только контекстом, предоставляемым *Primary* выражения доступа к полю, выражения вызова метода, выражения ссылки на метод или выражения создания экземпляра класса. Таким образом, мы описываем эти члены с идентификаторами, а не простыми именами.

Для дальнейшего усложнения выражение доступа к полю не является единственным способом обозначения поля объекта. По причинам, связанным с синтаксическим анализом, квалифицированное имя используется для обозначения поля внутренней переменной. (Сама переменная обозначается простым именем.) Для управления доступом (§6.6) необходимы оба механизма обозначения поля.

§6.3. Область видимости объявления

Область видимости объявления представляет собой область программы, в которой к объявленной в данном объявлении сущности можно обратиться с применением простого имени, что делает ее видимой (§6.4.1).

Объявление оказывается *в области видимости* в определенной точке программы тогда и только тогда, когда область видимости объявления включает эту точку.

Областью видимости объявления видимого (§7.4.3) пакета верхнего уровня являются все видимые модули компиляции (§7.3).

Объявление невидимого пакета никогда не находится в области видимости.

Объявление подпакета никогда не находится в области видимости.

Пакет `java` всегда находится в области видимости.

Область видимости типа, импортированного объявлением импорта одного типа (§7.5.1) или объявлением импорта типа по требованию (§7.5.2), представляет собой все объявления типов классов и интерфейсов (§7.6) в модуле компиляции, в котором находится объявление `import`, а также любые аннотации к объявлению пакета (если таковое имеется) модуля компиляции.

Область видимости члена, импортированного объявлением единого статического импорта (§7.5.3) или объявлением статического импорта по требованию (§7.5.4), представляет собой все объявления типов классов и интерфейсов (§7.6) в модуле компиляции, в котором находится объявление `import`, а также любые аннотации к объявлению пакета (если таковое имеется) модуля компиляции.

Область видимости типа верхнего уровня (§7.6) представляет собой все объявления типов в пакете, в котором объявлен тип верхнего уровня.

Область видимости объявления члена *m*, объявленного в типе класса *C* (§8.1.6) или унаследованного им, представляет собой все тело класса *C*, включая все вложенные объявления типов.

Область видимости объявления члена *m* объявленного в типе интерфейса *I* (§9.1.4) или унаследованного им представляет собой все тело интерфейса *I*, включая все вложенные объявления типов.

Область видимости константы перечисления *C*, объявленной в типе перечисления *T*, представляет собой тело типа *T* и любую метку `case` инструкции `switch`, выражение которой принадлежит типу перечисления *T*.

Область видимости формального параметра метода (§8.4.1), конструктора (§8.8.1) или лямбда-выражения (§15.27) представляет собой все тело метода, конструктора или лямбда-выражения.

Область видимости параметра типа класса (§8.1.2) представляет собой раздел параметра типа объявления класса, раздел параметра типа любого суперкласса или суперинтерфейса объявления класса и тело класса.

Область видимости параметра типа интерфейса (§9.1.2) представляет собой раздел параметра типа объявления интерфейса, раздел параметра типа любого суперинтерфейса объявления интерфейса и тело интерфейса.

Область видимости параметра типа метода (§8.4.4) представляет собой все объявление метода, включая раздел параметра типа, но исключая модификаторы метода.

Область видимости параметра типа конструктора (§8.8.4) представляет собой все объявление конструктора, включая раздел параметра типа, но исключая модификаторы конструктора.

Область видимости объявления локального класса, непосредственно вложенного в блок (§14.2), представляет собой остальную часть непосредственно охватывающего блока, включая объявление его собственного класса.

Область видимости объявления локального класса, непосредственно вложенного в блок инструкции `switch` (§14.11), представляет собой остальную часть непосредственно охватывающего блока `switch`, включая объявление его собственного класса.

Область видимости объявления локальной переменной в блоке (§14.4) представляет собой остальную часть блока, в котором находится объявление, начиная с ее инициализатора и включая все дальнейшие объявления справа от инструкции объявления локальной переменной.

Область видимости локальной переменной, объявленной в части *ForInit* базовой инструкции `for` (14.14.1), включает все перечисленное ниже.

- Ее собственный инициализатор.
- Все дальнейшие объявления справа от объявления данной локальной переменной в части *ForInit* инструкции `for`.
- Части *Expression* и *ForUpdate* инструкции `for`.
- Содержащуюся в инструкции `for` часть *Statement*.

Область видимости объявления локальной переменной в части *FormalParameter* расширенной инструкции `for` (§14.14.2) представляет собой часть *Statement* инструкции `for`.

Область видимости параметра обработчика исключения, объявленного в инструкции `catch` конструкции `try` (§14.20), представляет собой весь блок, связанный с данной инструкцией `catch`.

Область видимости переменной, объявленной в части *ResourceSpecification* инструкции `try` с ресурсами (§14.20.3), простирается от объявления вправо по всей части *ResourceSpecification* и всему блоку `try`, связанному с инструкцией `try` с ресурсами.

Указанное выше правило вытекает из трансляции инструкции `try` с ресурсами.

Пример 6.3-1. Область видимости объявлений типов

Из этих правил следует, что объявления типов классов и интерфейсов не обязаны находиться до использования этих типов. В приведенной далее программе использование класса `PointList` в классе `Point` является корректным, поскольку область видимости объявления класса `PointList` включает и класс `Point`, и класс `PointList`, как и все прочие объявления типов в других модулях компиляции пакета `points`.

```
package points;
class Point {
    int x, y;
    PointList list;
    Point next;
}
class PointList {
    Point first;
}
```

Пример 6.3-2. Область видимости объявления локальной переменной

Эта программа вызывает ошибку времени компиляции, потому что инициализация локальной переменной `x` выполняется в области видимости объявления локальной переменной `x`, но локальная переменная `x` пока что не имеет значения и использоваться не может. Поле `x` имеет значение `0` (присвоенное при инициализации `Test1`), но это отвлекающий маневр, так как оно затеняется (§6.4.1) локальной переменной `x`.

```
class Test1 {
    static int x;
    public static void main(String[] args) {
        int x = x;
    }
}
```

Без ошибок компилируется следующая программа.

```
class Test2 {
    static int x;
    public static void main(String[] args) {
        int x = (x=2)*2;
        System.out.println(x);
    }
}
```

В ней локальной переменной `x` определенно присваивается значение (§16) до ее использования. Вывод программы имеет вид

```
4
```

В следующей программе инициализатор переменной `three` без проблем может обращаться к переменной `two`, объявленной ранее в той же инструкции, а вызов метода в следующей строке может без проблем обращаться к переменной `three`, объявленной в блоке ранее.

```
class Test3 {
    public static void main(String[] args) {
        System.out.print("2+1=");
        int two = 2, three = two + 1;
        System.out.println(three);
    }
}
```

Вывод программы имеет вид

```
2+1=3
```

§6.4. Затенение и затемнение

Обращаться к локальной переменной (§14.4), формальному параметру (§8.4.1, §15.27.1), параметру исключения (§14.20) и локальному классу (§14.3) можно только с использованием простого имени (§6.2), но не квалифицированного имени (§6.6).

Некоторые объявления в области видимости локальной переменной, формального параметра, параметра исключения или объявления локального класса не разрешены, потому что они приводят к тому, что при использовании только простых имен объявленные сущности будет невозможно различить.

> Например, если бы имя формального параметра метода было повторно объявлено как имя локальной переменной в теле метода, то локальная переменная затеняла бы формальный параметр, который перестал бы быть видимым в теле метода, что, конечно же, крайне нежелательно.

Если имя формального параметра используется для объявления новой переменной в теле метода, конструктора или лямбда-выражения, генерируется ошибка времени компиляции (если только новая переменная не объявлена в объявлении класса, содержащегося в методе, конструкторе или лямбда-выражении).

Если имя локальной переменной `v` используется для объявления новой переменной в области видимости `v`, генерируется ошибка времени компиляции (если только новая переменная не объявлена в классе, объявление которого находится в области видимости `v`).

Если имя параметра исключения используется для объявления новой переменной в пределах части *Block* инструкции `catch`, генерируется ошибка времени компиляции (если только новая переменная не объявлена в объявлении класса, содержащемся в пределах части *Block* инструкции `catch`).

Если имя локального класса *C* используется для объявления нового локального класса в области видимости *C*, генерируется ошибка времени компиляции (если только новый локальный класс не объявлен в другом классе, объявление которого находится в области видимости *C*).

> Эти правила допускают повторное объявление переменной или локального класса в объявлениях вложенных классов (локальных классов (§14.3) и анонимных классов (§15.9)), которые находятся в области видимости переменной или локального класса. Таким образом, объявление формального параметра, локальной переменной или локального класса может быть затенено в объявлении класса, вложенном

в метод, конструктор или лямбда-выражение; а объявление параметра исключения может быть затенено в объявлении класса, вложенном в *Block* конструкции `catch`.

Имеются два варианта обработки конфликтов имен, создаваемые лямбда-параметрами и другими переменными, объявленными в лямбда-выражениях. Один из них заключается в имитации объявления класса: подобно локальным классам, лямбда-выражения вводят новый "уровень" для имен, и все имена переменных вне выражения могут быть объявлены повторно. Другой представляет собой "локальную" стратегию: подобно конструкциям `catch`, циклам `for` и блокам, лямбда-выражения работают на том же "уровне", что и охватывающий контекст, и локальные переменные вне выражения не могут быть затенены. Приведенные выше правила используют локальную стратегию; не требуется никакого специального разрешения, позволяющего переменной, объявленной в лямбда-выражении, затенять переменную, объявленную в охватывающем методе.

Обратите внимание, что правило для локальных классов не делает исключения для класса с тем же именем, объявленным в самом локальном классе. Однако этот случай запрещен отдельным правилом: класс не может иметь то же имя, что и класс, охватывающий его (§8.1).

Пример 6.4-1. Попытка затенения локальной переменной

Поскольку объявление идентификатора как локальной переменной метода, конструктора или блока инициализации не должно находиться в области видимости параметра или локальной переменной с тем же именем, компиляция следующей программы завершается с ошибкой.

```
class Test1 {
    public static void main(String[] args) {
        int i;
        for (int i = 0; i < 10; i++)
            System.out.println(i);
    }
}
```

Это ограничение помогает обнаруживать некоторые ошибки, которые в противном случае было бы весьма нелегко найти. Аналогичное ограничение на затенение членов локальными переменными было сочтено непрактичным, поскольку тогда добавление члена в суперкласс может привести к необходимости переименования локальных переменных в подклассах. Похожие соображения делают непривлекательными и ограничения на затенение локальных переменных членами вложенных классов или на затенение локальных переменных локальными переменными, объявленными во вложенных классах.

Следовательно, приведенная далее программа будет скомпилирована без ошибок.

```
class Test2 {
    public static void main(String[] args) {
        int i;
        class Local {
            {
                for (int i = 0; i < 10; i++)
```

```
                    System.out.println(i);
                }
            }
            new Local();
        }
}
```

С другой стороны, локальные переменные с одним и тем же именем могут быть объявлены в двух разных блоках или инструкциях `for`, не содержащих друг друга.

```
class Test3 {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
        for (int i = 10; i > 0; i--)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

Эта программа компилируется без ошибок и при выполнении дает следующий вывод.

```
0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1
```

§6.4.1. Затенение

Некоторые объявления могут быть *затенены* (shadowing) в части их области видимости другим объявлением с тем же именем, и в этом случае простое имя не может использоваться для обращения к объявленной сущности.

Затенение отличается от сокрытия (§8.3, §8.4.8.2, §8.5, §9.3, §9.5), которое применимо только к членам, которые в противном случае были бы унаследованы, но не являются таковыми в силу объявления в подклассе. Затенение отличается также от затемнения (§6.4.2).

Объявление d называется *видимым в точке p программы*, если область видимости d включает p и если d не затеняется в p никаким другим объявлением.

Когда рассматриваемая точка программы очевидна из контекста, мы зачастую говорим просто о том, что объявление является *видимым*.

Объявление d типа с именем n затеняет объявления любых других типов с именем n, которые находятся в области видимости в точке, где находится d и далее во всей области видимости d.

Объявление d поля или формального параметра с именем n затеняет во всей области видимости d объявления всех других переменных с именем n, которые находятся в области видимости в точке, где находится d.

Объявление d локальной переменной или параметра исключения с именем n затеняет во всей области видимости d (а) объявления любых других полей с именем n, которые находятся в области видимости в точке, где располагается d, и (б) объявления любых других переменных с именем n, которые находятся в области видимости в точке, где располагается d, но *не* объявлены в самом внутреннем классе, где объявлено d.

Объявление d метода с именем n затеняет объявления любых других методов с именем n, которые находятся в охватывающей области видимости в точке, где располагается d, во всей области видимости d.

Объявление пакета никогда не затеняет никакое другое объявление.

Объявление импорта типа по требованию никогда не приводит к затенению никаких других объявлений.

Объявление статического импорта по требованию никогда не приводит к затенению никаких других объявлений.

Объявление импорта единственного типа d в модуле компиляции c пакета p, которое импортирует тип с именем n, затеняет во всем c следующие объявления.

- Любого типа верхнего уровня с именем n, объявленного в другом модуле компиляции p.
- Любого типа с именем n, импортированного объявлением импорта типа по требованию в c.
- Любого типа с именем n, импортированного объявлением статического импорта по требованию в c.

Объявление единственного статического импорта d в модуле компиляции c пакета p, которое импортирует поле с именем n, затеняет объявление любого статического поля с именем n, импортированного объявлением статического импорта по требованию в c во всем модуле c.

Объявление единственного статического импорта d в модуле компиляции c пакета p, которое импортирует метод с именем n и сигнатурой s, затеняет объявление любого статического метода с именем n и сигнатурой s, импортированного объявлением статического импорта по требованию в c во всем модуле c.

Объявление единственного статического импорта d в модуле компиляции c пакета p, которое импортирует тип с именем n, затеняет во всем модуле c следующие объявления.

- Любого статического типа с именем n, импортированного объявлением статического импорта по требованию в модуле c.
- Любого типа верхнего уровня (§7.6) с именем n, объявленного в другом модуле компиляции (§7.3) пакета p.
- Любого типа с именем n, импортированного объявлением импорта типа по требованию (§7.5.2) в модуле c.

Пример 6.4.1-1. Затенение объявления поля объявлением локальной переменной

```
class Test {
    static int x = 1;
    public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println(", Test.x=" + Test.x);
    }
}
```

Вывод программы имеет вид

```
x=0, Test.x=1
```

В программе объявлены

- класс `Test`;
- переменная класса (`static`) `x`, являющаяся членом класса `Test`;
- метод класса `main`, являющийся членом класса `Test`;
- параметр `args` метода `main`;
- локальная переменная `x` метода `main`.

Поскольку область видимости переменной класса включает все тело класса (§8.2), переменная класса `x` обычно доступна во всем теле метода `main`. В данном примере, однако, переменная класса `x` затенена в теле метода `main` объявлением локальной переменной `x`.

Областью видимости локальной переменной является остаток блока, в котором она объявлена (§6.3); в данном случае это остаток тела метода `main`, а именно — инициализатор "0" и вызовы `System.out.print` и `System.out.println`.

Это означает, что

- выражение `x` в вызове `print` ссылается на (обозначает) значение локальной переменной `x`;
- вызов `println` использует квалифицированное имя (§6.6) `Test.x`, которое использует имя класса `Test` для доступа к переменной класса `x`, поскольку объявление `Test.x` затенено в данной точке и к этой переменной нельзя обратиться посредством ее простого имени.

Для доступа к затененному полю `x` можно также использовать ключевое слово `this`, используя запись `this.x`. Фактически эта идиома часто используется в конструкторах (§8.8).

```
class Pair {
    Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
}
```

Здесь конструктор получает параметры, имеющие те же имена, что и инициализируемые поля. Использовать `this` оказывается проще, чем изобретать разные имена для параметров, при этом в таком стилизованном контексте запись оказывается не слишком запутанной. В общем случае, однако, считается плохим стилем иметь локальные переменные с именами, совпадающими с именами полей.

Пример 6.4.1-2. Затенение объявления типа другим объявлением типа

```
import java.util.*;
class Vector {
    int val[] = { 1 , 2 };
}
```

```
class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

Эта программа компилируется и выводит

```
1
```

предпочитая класс `Vector`, объявленный в этом фрагменте, обобщенному классу `java.util.Vector` (§8.1.2), который может импортироваться по требованию.

§6.4.2. Затемнение

Простое имя может встречаться в контекстах, где оно потенциально может быть интерпретировано как имя переменной, типа или пакета. В таких ситуациях правила из §6.5 указывают, что переменная будет предпочтительным выбором по сравнению с типом, а тип будет предпочтительным выбором по сравнению с пакетом. Таким образом, иногда может оказаться невозможно сослаться на видимое объявление типа или пакета с помощью его простого имени. Мы говорим, что такое объявление оказывается *затемненным* (obscured).

Затемнение отличается от затенения (§6.4.1) и сокрытия (§8.3, §8.4.8.2, §8.5, §9.3, §9.5).

Соглашения именования в §6.1 помогают уменьшить затемнение, но если оно все же происходит, обратитесь к этим заметкам, чтобы узнать, что вы можете сделать, чтобы избежать его.

Когда имена пакетов встречаются в выражениях.

- Если имя пакета затемняется объявлением поля, то для того, чтобы сделать имена типов, объявленных в этом пакете, доступными, обычно можно прибегнуть к объявлению `import` (§7.5).
- Если имя пакета затемняется объявлением параметра или локальной переменной, то имя параметра или локальной переменной можно изменить, не затрагивая остальной код.

Первый компонент имени пакета обычно не так легко спутать с именем типа, поскольку имя типа обычно начинается с одной буквы в верхнем регистре. (В действительности язык программирования Java не полагается на различия в регистре символов, чтобы определить, является ли имя именем пакета или именем типа.)

Затемнение с участием имен типов класса и интерфейсов встречается редко. Имена полей, параметров и локальных переменных обычно не затемняют имена типов, потому что по соглашению они начинаются со строчной буквы, тогда как имена типов обычно начинаются с прописной буквы.

Имена методов не могут затемнить или быть затемненными другими именами (§6.5.7).

Затемнение с участием имен полей — редкое явление. Тем не менее рассмотрите следующие советы.

- Если имя поля затемняет имя пакета, то, чтобы сделать имена типов, объявленных в этом пакете, доступными, обычно можно воспользоваться объявлением `import` (§7.5).
- Если имя поля затемняет имя типа, то можно использовать квалифицированное имя типа, если только имя типа не обозначает локальный класс (§14.3).
- Имена полей не могут затемнить имена методов.
- Если имя поля затенено объявлением параметра или локальной переменной, то имя параметра или локальной переменной можно изменить, не затрагивая остальной код.
- Затемнение с участием имен констант — явление редкое.
- Имена констант обычно не содержат строчных букв, поэтому они обычно не затемняют имена типов или пакетов и, как правило, не затеняют поля, имена которых обычно содержат по крайней мере одну строчную букву.
- Имена констант не могут затемнять имена методов, потому что они различаются синтаксически.

§6.5. Определение значения имени

Значение имени зависит от контекста его использования. Определение значения имени требует выполнения трех шагов.

- Во-первых, контекст приводит к тому, что имя синтаксически попадает в одну из шести категорий: *PackageName*, *TypeName*, *ExpressionName*, *MethodName*, *PackageOr TypeName* или *AmbiguousName*.
- Во-вторых, имя, которое изначально классифицируется контекстом как *Ambiguous Name* или как *PackageOrTypeName*, затем переклассифицируется как *PackageName*, *TypeName* или *ExpressionName*.
- В-третьих, затем полученная категория диктует окончательное определение значения имени (или приводит к генерации ошибки времени компиляции, если имя не имеет значения).

PackageName:
    Identifier
    PackageName . *Identifier*

TypeName:
    Identifier
    PackageOrTypeName . *Identifier*

PackageOrTypeName:
    Identifier
    PackageOrTypeName . *Identifier*

ExpressionName:

Identifier
AmbiguousName . *Identifier*

MethodName:
Identifier

AmbiguousName:
Identifier
AmbiguousName . *Identifier*

Использование контекста помогает свести к минимуму конфликты имен между сущностями различных видов. Такие конфликты будут редкими, если следовать соглашениям именования, описанным в §6.1. Тем не менее конфликты могут возникнуть случайно, если типы разрабатываются различными программистами или различными организациями. Например, типы, методы и поля могут иметь одно и то же имя. Всегда возможно различить метод и поле с тем же именем, так как контекст использования всегда в состоянии указать, подразумевается ли применение метода.

§6.5.1. Синтаксическая классификация имен в соответствии с контекстом

Имя синтаксически классифицируется как *TypeName* в следующих контекстах.

- В первых десяти необобщенных контекстах (§6.1).
  1. В объявлении импорта единственного импорта (§7.5.1).
  2. Слева от точки `.` в объявлении единственного статического импорта (§7.5.3).
  3. Слева от точки `.` в объявлении статического импорта по требованию (§7.5.4).
  4. Слева от `(` в объявлении конструктора (§8.8).
  5. После знака `@` в аннотации (§9.7).
  6. Слева от `.class` в литерале класса (§15.8.2).
  7. Слева от `.this` в квалифицированном выражении `this` (§15.8.4).
  8. Слева от `.super` в выражении доступа к полю квалифицированного суперкласса (§15.11.2).
  9. Слева от *.Identifier* или `.super` *.Identifier* в выражении вызова квалифицированного метода (§15.12).
  10. Слева от `.super::` в выражении ссылки на метод (§15.13).
- Как *Identifier* или последовательность *Identifier* с точками, которая составляет любой *ReferenceType* (включая *ReferenceType* слева от квадратных скобок в типе массива или слева от < в параметризованном типе, или аргумент типа без символов подстановки параметризованного типа, или в конструкциях `extends` или `super` аргумента с символами подстановки параметризованного типа) в 16 контекстах, где используются типы (§4.11).

1. Тип в конструкции `extends` или `implements` объявления класса (§8.1.4, §8.1.5, §8.5, §9.5).
2. Тип в конструкции `extends` объявления интерфейса (§9.1.3, §8.5, §9.5).
3. Возвращаемый тип метода (включая тип элемента типа аннотации) (§8.4.5, §9.4, §9.6.1).
4. Тип в конструкции `throws` метода или конструктора (§8.4.6, §8.8.5, §9.4).
5. Тип в конструкции `extends` объявления параметра типа обобщенного класса, интерфейса, метода или конструктора (§8.1.2, §9.1.2, §8.4.4, §8.8.4).
6. Тип в объявлении поля класса или интерфейса (включая константу перечисления) (§8.3, §9.3, §8.9.1).
7. Тип в объявлении формального параметра метода, конструктора или лямбда-выражения (§8.4.1, §8.8.1, §9.4, §15.27.1).
8. Тип параметра-получателя метода (§8.4.1).
9. Тип в объявлении локальной переменной (§14.4, §14.14.1, §14.14.2, §14.20.3).
10. Тип в объявлении параметра исключения (§14.20).
11. Тип в списке аргументов явных типов инструкции явного вызова конструктора, выражения создания экземпляра класса или выражении вызова метода (§8.8.7.1, §15.9, §15.12).
12. В выражении создания экземпляра неквалифицированного класса в качестве типа инстанцируемого класса (§15.9) или в качестве непосредственного суперкласса или непосредственного суперинтерфейса инстанцируемого анонимного класса (§15.9.5).
13. Тип элемента в выражении создания массива (§15.10.1).
14. Тип в операторе приведения или выражении приведения (§15.16).
15. Тип, следующий за оператором отношения `instanceof` (§15.20.2).
16. В выражении ссылки на метод (§15.13), в качестве типа ссылки для поиска метода-члена, в качестве типа создаваемого класса или массива.

Выделение *TypeName* из идентификаторов *ReferenceType* в 16-ти приведенных выше контекстах предназначено для рекурсивного применения ко всем подчиненным элементам *ReferenceType*, таким как тип элемента и любые аргументы типа.

Предположим, например, что объявление поля использует тип `p.q.Foo[]`. Квадратные скобки типа массива игнорируются, элемент `p.q.Foo` выделяется как разделенная точками последовательность *Identifiers* слева от квадратных скобок в типе массива, и классифицируется как *TypeName*. Последующие шаги определяют, что из `p`, `q` и `Foo` является именем типа или именем пакета.

В качестве другого примера предположим, что оператор приведения использует тип `p.q.Foo<? extends String>`. Элемент `p.q.Foo` вновь выделяется как разделенная точками последовательность *Identifiers* слева от < в параметризованном типе и классифицируется как *TypeName*. Элемент `String` выделяется как *Identifier* в конструкции `extends` аргумента типа с символами подстановки параметризованного типа, и классифицируется как *TypeName*.

Имя синтаксически классифицируется как *ExpressionName* в следующих контекстах:

- как квалифицированное выражение в вызове конструктора квалифицированного суперкласса (§8.8.7.1);
- как квалифицированное выражение в выражении создания экземпляра квалифицированного класса (§15.9);
- как выражение ссылки на массив в выражении доступа к массиву (§15.13);
- как *PostfixExpression* (§15.14);
- как левый операнд оператора присваивания (§15.26).

Имя синтаксически классифицируется как *MethodName* в следующем контексте:

- перед “ (” в выражении вызова метода (§15.12).

Имя синтаксически классифицируется как *PackageOrTypeName* в следующих контекстах:

- слева от точки . в квалифицированном *TypeName*;
- в объявлении импорта типа по требованию (§7.5.2).

Имя синтаксически классифицируется как *AmbiguousName* в следующих контекстах:

- слева от точки . в квалифицированном *ExpressionName*;
- слева от крайней справа точки . перед (в выражении вызова метода;
- слева от точки . в квалифицированном *AmbiguousName*;
- в выражении значения по умолчанию в объявлении элемента типа аннотации (§9.6.2);
- справа от знака = в паре значений элемента аннотации (§9.7.1);
- слева от : : в выражении ссылки на метод (§15.13).

> Результат синтаксической классификации состоит в ограничении некоторых видов сущностей для определенных частей выражений.
>
> - Имя поля, параметра или локальной переменной может использоваться как выражение (§15.14.1).
> - Имя метода может появиться в выражении только как часть выражения вызова метода (§15.12).
> - Имя типа класса или интерфейса может появиться в выражении только как часть литерала класса (§15.8.2), квалифицированного `this` выражения (§15.8.4), выражения создания экземпляра класса (§15.9), выражения создания массива (§15.10.1), выражения приведения (§15.16), выражения `instanceof` (§15.20.2), константы перечисления (§8.9) или как часть квалифицированного имени поля или метода.
> - Имя пакета может появиться в выражении только как часть квалифицированного имени типа класса или интерфейса.

§6.5.2. Переклассификация контекстуально неоднозначных имен

Имена, классифицированные как *AmbiguousName*, затем переклассифицируются следующим образом.

Если *AmbiguousName* представляет собой простое имя, состоящее из одного *Identifier*.

- Если *Identifier* находится в области видимости (§6.3) объявления локальной переменной (§14.4), объявления параметра (§8.4.1, §8.8.1, §14.20) или объявления поля (§8.3) с данным именем, то *AmbiguousName* переклассифицируется как *ExpressionName*.
- В противном случае, если поле с данным именем объявлено в модуле компиляции (§7.3), содержащем *Identifier*, с помощью объявления единственного статического импорта (§7.5.3) или с помощью объявления статического импорта по требованию (§7.5.4), то *AmbiguousName* переклассифицируется как *ExpressionName*.
- В противном случае, если *Identifier* находится в области видимости (§6.3) объявления типа класса верхнего уровня (§8) или интерфейса (§9), объявления локального класса (§14.3) или объявления типа-члена (§8.5, §9.5) с этим именем, *AmbiguousName* переклассифицируется как *TypeName*.
- В противном случае, если тип с этим именем объявлен в модуле компиляции (§7.3), содержащем *Identifier*, либо с помощью объявления импорта единственного типа (§7.5.1), либо с помощью объявления импорта типа по требованию (§7.5.2), либо с помощью объявления единственного статического импорта (§7.5.3), либо с помощью объявления статического импорта по требованию (§7.5.4), *AmbiguousName* переклассифицируется как *TypeName*.
- В противном случае *AmbiguousName* переклассифицируется как *PackageName*. Последующий шаг определяет, существует ли в действительности пакет с данным именем.

Если *AmbiguousName* представляет собой квалифицированное имя, состоящее из имени, точки . и *Identifier*, то сначала переклассифицируется имя слева от точки, если оно само является *AmbiguousName*.

- Если имя слева от точки переклассифицировано как *PackageName*, то
  - ✦ если имеется пакет, имя которого совпадает с именем слева от точки и этот пакет содержит объявление типа, имя которого совпадает с *Identifier*, то рассматриваемое *AmbiguousName* переклассифицируется как *TypeName*;
  - ✦ в противном случае рассматриваемое *AmbiguousName* переклассифицируется как *PackageName*. Последующий шаг определяет, существует ли в действительности пакет с данным именем.
- Если имя слева от точки переклассифицировано как *TypeName*, то
  - ✦ если *Identifier* представляет собой то же имя, что и имя метода или поля типа, обозначаемого посредством *TypeName*, рассматриваемое *AmbiguousName* переклассифицируется как *ExpressionName*;
  - ✦ в противном случае, если *Identifier* представляет собой имя типа-члена типа, обозначаемого посредством *TypeName*, то рассматриваемое *AmbiguousName* переклассифицируется как *TypeName*;
  - ✦ в противном случае генерируется ошибка времени компиляции.
- Если имя слева от точки переклассифицировано как *ExpressionName*, то пусть T представляет собой тип выражения, обозначаемого этим *ExpressionName*.

- Если *Identifier* представляет собой имя метода или поля типа, описываемого T, то рассматриваемое *AmbiguousName* переклассифицируется как *ExpressionName*.
- В противном случае, если *Identifier* представляет собой имя типа-члена (§8.5, §9.5) типа, описываемого T, рассматриваемое *AmbiguousName* переклассифицируется как *TypeName*.
- В противном случае генерируется ошибка времени компиляции.

Пример 6.5.2-1. Переклассификация контекстуально неоднозначных имен

Рассмотрим следующий надуманный "библиотечный код".

```
package org.rpgpoet;
import java.util.Random;
public interface Music { Random[] wizards = new Random[4]; }
```

Затем рассмотрим пример кода из другого пакета.

```
package bazola;
class Gabriel {
    static int n = org.rpgpoet.Music.wizards.length;
}
```

Прежде всего, имя `org.rpgpoet.Music.wizards.length` классифицируется как *ExpressionName*, потому что оно функционирует как *PostfixExpression*. Таким образом, каждое из имен

```
org.rpgpoet.Music.wizards
org.rpgpoet.Music
org.rpgpoet
org
```

изначально классифицируется как *AmbiguousName*. Затем они переклассифицируются.

- Простое имя `org` переклассифицируется как *PackageName* (из-за отсутствия в области видимости переменной или типа с именем `org`).
- Затем, в предположении, что ни в одном модуле компиляции пакета `org` нет класса или интерфейса с именем `rpgpoet` (и мы знаем, что такого класса или интерфейса нет, так как пакет `org` имеет подпакет с именем `rpgpoet`), квалифицированное имя `org.rpgpoet` переклассифицируется как *PackageName*.
- Затем, поскольку пакет `org.rpgpoet` имеет доступный (§6.6) тип интерфейса с именем `Music`, квалифицированное имя `org.rpgpoet.Music` переклассифицируется как *TypeName*.
- Наконец, поскольку имя `org.rpgpoet.Music` представляет собой *TypeName*, квалифицированное имя `org.rpgpoet.Music.wizards` переклассифицируется как *ExpressionName*.

§6.5.3. Значение имен пакетов

Значение имени, классифицированного как *PackageName*, определяется следующим образом.

§6.5.3.1. Простые имена пакетов

Если имя пакета состоит из единственного *Identifier*, то этот идентификатор обозначает пакет верхнего уровня с именем, определяемым данным идентификатором.

Если в области видимости (§6.3) нет пакета верхнего уровня с этим именем, генерируется ошибка времени компиляции.

§6.5.3.2. Квалифицированные имена пакетов

Если имя пакета имеет вид `Q.Id`, то `Q` также должно быть именем пакета. Имя пакета `Q.Id` именует пакет, который является членом с именем `Id` внутри пакета с именем `Q`.

Если Q не именует наблюдаемый пакет (§7.4.3) или `Id` не является простым именем наблюдаемого подпакета данного пакета, то генерируется ошибка времени компиляции.

§6.5.4. Значение *PackageOrTypeNames*

§6.5.4.1. Простой *PackageOrTypeNames*

Если *PackageOrTypeName* с именем `Q` встречается в области видимости типа с именем `Q`, то *PackageOrTypeName* переклассифицируется как *TypeName*.

В противном случае *PackageOrTypeName* переклассифицируется как *PackageName*. Значением *PackageOrTypeName* является значение переквалифицированного имени.

§6.5.4.2 Квалифицированный *PackageOrTypeNames*

В случае заданного квалифицированного *PackageOrTypeName* вида `Q.Id`, если тип или пакет, обозначенный `Q`, имеет тип-член с именем `Id`, то квалифицированное имя *PackageOrTypeName* переклассифицируется как *TypeName*.

В противном случае имя переклассифицируется как *PackageName*. Значением квалифицированного *PackageOrTypeName* является значение переклассифицированного имени.

§6.5.5. Значение имен типов

Значение имени, классифицированного как *TypeName*, определяется следующим образом.

§6.5.5.1. Простые имена типов

Если имя типа состоит из единственного *Identifier*, то идентификатор должен находиться в области видимости ровно одного видимого объявления типа с этим именем, иначе генерируется ошибка времени компиляции. Значением имени типа является этот тип.

§6.5.5.2. Квалифицированные имена типов

Если имя типа имеет вид `Q.Id`, то `Q` должно быть либо именем типа, либо именем пакета.

Если `Id` именует ровно один доступный тип (§6.6), который является членом типа или пакета, обозначаемого `Q`, то квалифицированное имя типа обозначает этот тип.

Если *Id* не именует тип-член (§8.5, §9.5) в *Q* или тип-член с именем *Id* в *Q* недоступен (§6.6) , или *Id* именует более чем один тип-член в *Q*, генерируется ошибка времени компиляции.

Пример 6.5.5.2-1. Квалифицированные имена типов

```
class Test {
    public static void main(String[] args) {
        java.util.Date date =
            new java.util.Date(System.currentTimeMillis());
        System.out.println(date.toLocaleString());
    }
}
```

При первом запуске данная программа сгенерировала следующий вывод.

```
Sun Jan 21 22:56:29 1996
```

В этом примере имя `java.util.Date` должно обозначать тип, так что мы сначала рекурсивно используем процедуру для определения, представляет ли `java.util` доступный тип или пакет (как и есть на самом деле), а затем смотрим, является ли тип `Date` доступным в данном пакете.

§6.5.6. Значение имен выражений

Значение имени, классифицированного как *ExpressionName*, определяется следующим образом.

§6.5.6.1. Простые имена выражений

Если имя выражения состоит из единственного *Identifier*, то должно иметься ровно одно объявление, описывающее локальную переменную, параметр или поле, видимое (§6.4.1) в точке, в которой встречается *Identifier*. В противном случае генерируется ошибка времени компиляции.

Если объявление описывает переменную экземпляра (§8.3), имя выражения должно находиться в объявлении метода экземпляра (§8.4), объявлении конструктора (§8.8), инициализаторе экземпляра (§8.6) или инициализаторе переменной экземпляра (§8.3.2.2). Если имя выражения находится в статическом методе (§8.4.3.2), статическом инициализаторе (§8.7) или инициализаторе статической переменной (§8.3.2.1, §12.4.2), то генерируется ошибка времени компиляции.

Если объявление объявляет `final`-переменную, для которой выполняется определенное присваивание до простого выражения, то значением имени является значение этой переменной. В противном случае значением имени выражения является переменная, объявленная в объявлении.

Если имя выражения встречается в контексте присваивания, вызова или приведения, то тип имени выражения представляет собой объявленный тип поля, локальной переменной или параметра после преобразования при фиксации (§5.1.10).

В противном случае тип имени выражения представляет собой объявленный тип поля, локальной переменной или параметра.

Иначе говоря, если имя выражения появляется “в правой части”, его тип подвергается преобразованию при фиксации. Если имя выражения представляет собой переменную “из левой части”, ее тип преобразованию при фиксации не подвергается.

Пример 6.5.6.1-1. Простые имена выражений

```
class Test {
    static int v;
    static final int f = 3;
    public static void main(String[] args) {
        int i;
        i = 1;
        v = 2;
        f = 33;         // Ошибка времени компиляции
        System.out.println(i + " " + v + " " + f);
    }
}
```

В этой программе имена, использованные в левой части присваиваний переменным `i`, `v` и `f`, обозначают локальную переменную `i`, поле `v` и значение `f` (не переменную `f`, поскольку `f` объявлена как `final`). Таким образом, при компиляции данного примера генерируется ошибка времени компиляции из-за того, что последнее присваивание не имеет переменной в левой части. Если удалить это некорректное присваивание, код компилируется и при выполнении выводит

```
1 2 3
```

§6.5.6.2. Квалифицированные имена выражений

Если имя выражения имеет вид *`Q.Id`*, то *`Q`* уже является классифицированным как имя пакета, типа или выражения.

Если *`Q`* является именем пакета, генерируется ошибка времени компиляции.

Если *`Q`* является именем типа, которое именует тип класса (§8), то выполняется следующее.

- Если имеется не ровно один доступный (§6.6) член типа класса, который представляет собой поле с именем *`Id`*, то генерируется ошибка времени компиляции.
- В противном случае, если единственный доступный член-поле не является переменной класса (т.е. не объявлен как `static`), генерируется ошибка времени компиляции.
- В противном случае, если переменная класса объявлена как `final`, *`Q.Id`* означает значение переменной класса.

  Типом выражения *`Q.Id`* является объявленный тип переменной класса после преобразования при фиксации (§5.1.10).

  Если *`Q.Id`* находится в контексте, который требует переменную, а не значение, генерируется ошибка времени компиляции.
- В противном случае *`Q.Id`* обозначает переменную класса.

  Типом выражения *`Q.Id`* является объявленный тип переменной класса после преобразования при фиксации (§5.1.10).

Заметим, что этот пункт охватывает использование констант перечисления `enum` (§8.9), поскольку они всегда имеют соответствующую переменную класса, объявленную как `final`.

Если *Q* представляет собой имя типа, обозначающее тип интерфейса (§9), то выполняется следующее.

- Если нет ровно одного доступного (§6.6) члена типа интерфейса, являющегося полем с именем *Id*, то генерируется ошибка времени компиляции.
- В противном случае *Q*`.`*Id* описывает значение этого поля.
- Типом выражения *Q*`.`*Id* является объявленный тип поля после преобразования при фиксации (§5.1.10).
- Если *Q*`.`*Id* находится в контексте, который требует переменную, а не значение, генерируется ошибка времени компиляции.

Если *Q* является именем выражения, то пусть *T* представляет собой тип выражения *Q*.

- Если *T* не является ссылочным типом, генерируется ошибка времени компиляции.
- Если нет ровно одного доступного (§6.6) члена типа *T*, являющегося полем с именем *Id*, генерируется ошибка времени компиляции.
- В противном случае, если это поле является одним из следующего списка:
  - ✦ поле типа интерфейса;
  - ✦ поле типа класса, объявленное как `final` (которое может быть как переменной класса, так и переменной экземпляра);
  - ✦ поле `length` типа массива, объявленное как `final` (§10.7);
- *Q*`.`*Id* описывает значение поля, если только оно не появляется в контексте, который требует переменную, и поле представляет собой определенное неинициализированное поле типа `final`; в этом случае оно представляет собой переменную.
- Типом выражения *Q*`.`*Id* является объявленный тип поля после преобразования при фиксации (§5.1.10).
- Если *Q*`.`*Id* находится в контексте, который требует переменную, а не значение, и поле, описываемое *Q*`.`*Id*, является определенно присвоенным, генерируется ошибка времени компиляции.
- В противном случае *Q*`.`*Id* описывает переменную, поле *Id* класса *T*, и может быть либо переменной класса, либо переменной экземпляра.
- Типом выражения *Q*`.`*Id* является тип поля-члена после преобразования при фиксации (§5.1.10).

Пример 6.5.6.2-1. Квалифицированные имена выражений

```
class Point {
    int x, y;
    static int nPoints;
}
class Test {
    public static void main(String[] args) {
```

```
        int i = 0;
        i.x++;              // Ошибка времени компиляции
        Point p = new Point();
        p.nPoints();        // Ошибка времени компиляции
    }
}
```

В этой программе имеются два источника ошибок времени компиляции, поскольку переменная `i` типа `int` не имеет членов и поскольку `nPoints` не является методом класса `Point`.

Пример 6.5.6.2-2. Квалификация выражения именем типа

Заметим, что выражение может быть квалифицировано именами типов, но не типами в общем случае. Следствием этого является невозможность доступа к переменной класса посредством параметризованного типа.

Пусть имеется следующий код.

```
class Foo<T> {
    public static int classVar = 42;
}
```

Тогда приведенное далее присваивание будет некорректным.

```
Foo<String>.classVar = 91; // Неверно
```

Вместо этого следует записать

```
Foo.classVar = 91;
```

Это не ограничивает язык программирования Java сколь-нибудь существенно. Параметры типа не могут использоваться в типах статических переменных, а потому аргументы типа параметризованного типа не могут влиять на тип статической переменной. Таким образом, выразительная сила языка не теряется. Имя типа `Foo` кажется несформированным типом, но это не так; на самом деле это имя необобщенного типа `Foo`, статический член которого доступен (§6.1). Поскольку здесь нет использования несформированных типов, нет и вывода предупреждения компилятором.

§6.5.7. Значение имен методов

Значение имени, классифицированного как *MethodName*, определяется следующим образом.

§6.5.7.1. Простые имена методов

Простое имя метода находится в контексте выражения вызова метода (§15.12). Простое имя метода состоит из единственного *Identifier*, который определяет имя вызываемого метода. Правила вызова метода требуют, чтобы *Identifier* обозначал либо метод, видимый в точке вызова метода, либо метод, импортированный объявлением единственного статического импорта либо статического импорта по требованию (§7.5.3, §7.5.4).

Пример 6.5.7.1-1. Простые имена методов и видимость

Приведенная далее программа демонстрирует роль видимости методов при определении того, какой из методов вызывается.

```
class Super {
    void f2(String s) {}
    void f3(String s) {}
    void f3(int i1, int i2) {}
}

class Test {
    void f1(int i) {}
    void f2(int i) {}
    void f3(int i) {}
    void m() {
        new Super() {
            {
                f1(0); // Разрешается в Test.f1(int)
                f2(0); // Ошибка времени компиляции
                f3(0); // Ошибка времени компиляции
            }
        };
    }
}
```

В вызове `f1(0)` видимым является только один метод по имени `f1`. Это метод `Test.f1(int)`, объявление которого находится в области видимости во всем теле `Test`, включая объявление анонимного класса. В §15.12.1 выбирается поиск в классе `Test`, поскольку объявление анонимного класса не имеет члена с именем `f1`. В конечном итоге вызов `Test.f1(int)` оказывается разрешенным.

В вызове `f2(0)` видимыми являются два метода с именем `f2`. Первый — объявление метода `Super.f2(String)` в области видимости объявления анонимного класса. Второй — объявление метода `Test.f2(int)` в области видимости тела `Test`, включая объявление анонимного класса. В §15.12.1 выбирается поиск в классе `Super`, поскольку он имеет член с именем `f2`. Однако `Super.f2(String)` не применим для вызова `f2(0)`, так что генерируется ошибка времени компиляции. Заметим, что в классе `Test` поиск не выполняется.

В вызове `f3(0)` видимыми являются три метода с именем `f3`. Первый и второй — объявления методов `Super.f3(String)` и `Super.f3(int,int)` — находятся в области видимости объявления анонимного класса. Третий, объявление метода `Test.f3(int)`, находится в области видимости во всем теле `Test`, включая объявление анонимного класса. В §15.12.1 выбирается поиск в классе `Super`, поскольку он имеет член с именем `f3`. Однако `Super.f3(String)` и `Super.f3(int,int)` неприменимы для вызова `f3(0)`, так что генерируется ошибка времени компиляции. Заметим, что в классе `Test` поиск не выполняется.

Выбор поиска в иерархии суперклассов вложенного класса до лексически охватывающей области видимости называется “правилом гребенки” (§15.12.1).

§6.6. Управление доступом

Язык программирования Java предоставляет механизмы для *управления доступом*, чтобы оградить пользователей пакета или класса от зависимости от излишних деталей реализации этого пакета или класса. Если доступ к чему-либо разрешен, такая сущность называется *доступной* (accessible).

Обратите внимание, что доступность является статическим свойством, которое можно определить во время компиляции. Оно зависит только от модификаторов типов и объявлений.

Квалифицированные имена являются средством доступа к членам пакетов и ссылочным типам. Управление доступом применяется при классификации имени такого члена, исходя из его контекста (§6.5.1), как квалифицированного имени типа (обозначающего член пакета или ссылочный тип, §6.5.5.2) или квалифицированного имени выражения (обозначающего член ссылочного типа, §6.5.6.2).

> Например, инструкция импорта единственного типа (§7.5.1) использует квалифицированное имя типа, так что имя импортируемого типа должно быть доступно из модуля компиляции, содержащего инструкцию импорта. В качестве другого примера объявление класса может использовать квалифицированное имя типа для суперкласса (§8.1.5), и снова квалифицированное имя типа должно быть доступно.
>
> Некоторые очевидные выражения "отсутствуют" в классификации контекста в §6.5.1: доступ к полю в *Primary* (§15.11.1), вызов метода в *Primary* (§15.12) и инстанцируемый класс при создании экземпляра квалифицированного класса (§15.9). Каждое из этих выражений использует идентификаторы, а не имена, по причине, указанной в §6.2. Следовательно, управление доступом к членам (полям, методам, типам) применяется *явно* с помощью выражений доступа к полю, выражений вызова метода и выражений создания экземпляра квалифицированного класса. (Обратите внимание, что доступ к полю может также описываться квалифицированным именем, представляющим собой постфиксное выражение).
>
> Кроме того, многие инструкции и выражения позволяют использовать типы, а не имена типов. Например, объявление класса может использовать параметризованный тип (§4.5) для обозначения суперкласса. Поскольку параметризованный тип не является квалифицированным именем типа, объявление класса должно явно выполнить управление доступом к указанному суперклассу. Следовательно, большинство инструкций и выражений, которые предоставляют контексты в §6.5.1 для классификации *TypeName*, должны также выполнять собственные проверки управления доступом.
>
> Помимо доступа к членам пакета или ссылочного типа встает вопрос доступа к конструкторам ссылочного типа. Управление доступом должно проверяться и при явном, и при неявном вызове конструктора. Следовательно, управление доступом проверяется как в инструкции явного вызова конструктора (§8.8.7.1), так и в выражении создания экземпляра класса (§15.9.3). Эти проверки "вручную" необходимы, поскольку §6.5.1 игнорирует инструкции явного вызова конструктора (потому что они обращаются к именам конструкторов косвенно) и не осведомлен о различии между типом класса, обозначаемым выражением создания экземпляра

неквалифицированного класса, и конструктором этого типа класса. Кроме того, конструкторы не имеют квалифицированных имен, так что мы не можем полагаться на управление доступом, проверяемое во время классификации квалифицированных имен типов.

Доступность влияет на наследование членов класса (§8.2), включая сокрытие и перекрытие методов (§8.4.8.1).

§6.6.1. Определение доступности

- Пакет всегда доступен.
- Если тип класса или интерфейса объявлен как `public`, то он доступен любому коду в модуле компиляции (§7.3), в котором он объявлен как видимый.

  Если тип класса или интерфейса объявлен с доступом пакета, то он может быть доступен только из пакета, в котором объявлен.

  Тип класса или интерфейса, объявленный без модификатора доступа, неявно имеет доступ пакета.
- Тип массива доступен тогда и только тогда, когда доступен тип его элементов.
- Член (класс, интерфейс, поле или метод) ссылочного типа (класса, интерфейса или массива) или конструктор типа класса доступен, только если тип доступен и член или конструктор объявлен с разрешением доступа.

  ✦ Если член или конструктор объявлен как `public`, доступ разрешен.

  Все члены интерфейсов без модификаторов доступа неявно являются `public`.

  ✦ В противном случае, если член или конструктор объявлен как `protected`, доступ разрешен, только когда выполняется одно из следующих утверждений.

  — Доступ к члену или конструктору осуществляется из пакета, содержащего класс, в котором объявлен защищенный член или конструктор.

  — Доступ корректен, как описано в §6.6.2.

  ✦ В противном случае, если член или конструктор объявлен с доступом пакета, доступ разрешен, только когда он осуществляется из пакета, в котором объявлен этот тип.

  Член класса или конструктор, объявленный без модификатора доступа, неявно имеет доступ пакета.

  ✦ В противном случае, если член или конструктор объявлен как `private`, доступ разрешен тогда и только тогда, когда он осуществляется из тела класса верхнего уровня (§7.6), который охватывает объявление члена или конструктора.

Пример 6.6-1. Управление доступом.

Рассмотрим модули компиляции

```
package points;
class PointVec { Point[] vec; }
```

и

```
package points;
public class Point {
    protected int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

которые объявляют два типа класса в пакете `points`.

- Тип класса `PointVec` не является `public` и не является частью `public` интерфейса пакета `points`, так что может использоваться только другими классами в этом пакете.
- Тип класса `Point` объявлен как `public` и доступен другим пакетам. Он является частью `public`-интерфейса пакета `points`.
- Методы `move`, `getX` и `getY` класса `Point` объявлены как `public` и, таким образом, доступны любому коду, использующему объекты типа `Point`.
- Поля `x` и `y` объявлены как `protected` и доступны извне пакета `points` только в подклассах класса `Point` и только когда они являются полями объектов, реализованных кодом, который к ним обращается.

Смотрите пример ограничения доступа модификатором `protected` в §6.6.2.

Пример 6.6-2. Доступ к открытым полям, методам и конструкторам

Член класса или конструктор, описанный как `public`, доступен в пакете, в котором он объявлен, и в любом другом пакете, в котором он объявлен как видимый (§7.4.3). Например, в модуле компиляции

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
        moves++;
    }
    public static int moves = 0;
}
```

`public`-класс `Point` имеет `public`-члены: метод `move` и поле `moves`. Эти `public`-члены доступны любым другим пакетам, которые имеют доступ к пакету `points`. Поля `x` и `y` не являются `public` и, таким образом, доступны только из пакета `points`.

Пример 6.6-3. Доступ к классам

Если у класса отсутствует модификатор `public`, доступ к объявлению класса ограничен пакетом, в котором он объявлен (§6.6).

```
package points;
public class Point {
    public int x, y;
```

```
    public void move(int dx, int dy) { x += dx; y += dy; }
}
class PointList {
    Point next, prev;
}
```

В приведенном примере в модуле компиляции объявлены два класса. Класс `Point` доступен извне пакета `points`, в то время как класс `PointList` доступен только внутри пакета. Таким образом, модуль компиляции в другом пакете может обращаться к `points.Point` либо с помощью полностью квалифицированного имени:

```
package pointsUser;
class Test1 {
    public static void main(String[] args) {
        points.Point p = new points.Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

либо с помощью объявления импорта единственного типа (§7.5.1) с упоминанием полностью квалифицированного имени, так что после этого может применяться простое имя:

```
package pointsUser;
import points.Point;
class Test2 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

Однако этот модуль компиляции не может использовать или импортировать `points.PointList`, который не объявлен как `public`, а следовательно, недоступен вне пакета `points`.

Пример 6.6-4. Доступ к полям, методам и конструкторам с доступом пакета

Если не указан ни один из модификаторов доступа `public`, `protected` или `private`, член класса или конструктор имеет доступ пакета: они доступны в пакете, содержащем объявление класса, в котором объявлен этот член класса, но член класса или конструктор недоступен ни в каком другом пакете.

Если класс, объявленный как `public`, имеет метод или конструктор с доступом пакета, то этот метод или конструктор недоступен для доступа или наследования подклассом, объявленным вне данного пакета.

Например, если имеется код

```
package points;
public class Point {
    public int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
```

```
    public void moveAlso(int dx, int dy) { move(dx, dy); }
}
```

то подкласс в другом пакете может объявить несвязанный метод `move` с теми же сигнатурой (§8.4.2) и типом возвращаемого значения. Поскольку исходный метод `move` недоступен в пакете `morepoints`, использовать ключевое слово `super` нельзя.

```
package morepoints;
public class PlusPoint extends points.Point {
    public void move(int dx, int dy) {
        super.move(dx, dy); // Ошибка времени компиляции
        moveAlso(dx, dy);
    }
}
```

Поскольку метод `move` класса `Point` не перекрывается методом `move` в `PlusPoint`, метод `moveAlso` в `Point` никогда не вызывает метод `move` из `PlusPoint`. Таким образом, если вы удалите вызов `super.move` из `PlusPoint` и выполните следующую тестовую программу

```
import points.Point;
import morepoints.PlusPoint;
class Test {
    public static void main(String[] args) {
        PlusPoint pp = new PlusPoint();
        pp.move(1, 1);
    }
}
```

то она завершится без ошибок. Если бы метод `move` класса `Point` был перекрыт методом `move` в `PlusPoint`, то данная программа вошла бы в бесконечную рекурсию, которая продолжалась бы до переполнения стека (`StackOverflowError`).

Пример 6.6-5. Доступ к `private`-полям, методам и конструкторам

Член класса или конструктор, объявленный как `private`, доступен только в теле класса верхнего уровня (§7.6), который охватывает объявление члена или конструктора. Он не наследуется подклассами. В примере

```
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; }
}
```

закрытые члены `ID`, `masterID` и `setMasterID` могут использоваться только в теле класса `Point`. Они не могут быть доступны с помощью квалифицированных имен, выражений доступа к полям или выражений вызовов методов вне тела объявления `Point`.

Пример использования закрытого конструктора приведен в §8.8.8.

§6.6.2. Детали защищенного доступа

Доступ к защищенному (`protected`) члену или конструктору объекта может осуществляться извне пакета, в котором он объявлен, только кодом, ответственным за реализацию этого объекта.

§6.6.2.1. Доступ к защищенному члену

Пусть *C* — класс, в котором объявлен `protected`-член. Доступ к нему разрешен только в теле подкласса *S* класса *C*.

Кроме того, если *Id* обозначает поле экземпляра или метод экземпляра, то выполняется следующее.

- Если доступ выполняется посредством квалифицированного имени *Q*`.`*Id* или выражения ссылки на метод *Q*`::`*Id* (§15.13), где *Q* представляет собой *ExpressionName*, то доступ разрешен тогда и только тогда, когда типом выражения *Q* является *S* или подкласс *S*.
- Если доступ выполняется с помощью выражения доступа к полю *E*`.`*Id* или с помощью выражения вызова метода *E*`.`*Id*`(...)`, или с помощью выражения ссылки на метод *E*`::`*Id*, где *E* представляет собой выражение *Primary*, то доступ разрешен тогда и только тогда, когда типом *E* является *S* или подкласс *S*.
- Если доступ выполняется с помощью выражения ссылки на метод *T*`::`*Id*, где *T* представляет собой *ReferenceType*, то доступ разрешен тогда и только тогда, когда тип *T* представляет собой *S* или подкласс *S*.

> Более полную информацию о доступе к членам, объявленным как `protected`, можно найти в статье Алессандро Коглио (Alessandro Coglio) *Checking Access to Protected Members in the Java Virtual Machine* в *Journal of Object Technology* (октябрь 2005).

§6.6.2.2. Квалифицированный доступ к защищенному конструктору

Пусть *C* представляет собой класс, в котором объявлен защищенный конструктор, и пусть *S* — наиболее глубоко вложенный класс, в объявлении которого встречается использование этого защищенного конструктора. Тогда выполняется следующее.

- Если доступ осуществляется вызовом конструктора суперкласса `super(...)` или вызовом конструктора квалифицированного суперкласса вида *E*`.super(...)`, где *E* представляет собой выражение *Primary*, то доступ разрешен.
- Если доступ осуществляется выражением создания экземпляра анонимного класса вида `new` *C*`(...){...}` или выражением создания экземпляра квалифицированного анонимного класса вида *E*`.new` *C*`(...){...}`, где *E* представляет собой выражение *Primary*, то доступ разрешен.
- Если доступ осуществляется выражением создания экземпляра простого класса вида `new` *C*`(...)` или выражением создания экземпляра квалифицированного класса вида *E*`.new` *C*`(...)`, где *E* представляет собой выражение *Primary*, то доступ не разрешен. Выражение создания экземпляра класса (который не объявлен как анонимный)

может получить доступ к конструктору, объявленному как `protected`, только из пакета, в котором он определен.

Пример 6.6.2-1. Доступ к защищенным полям, методам и конструкторам

Рассмотрим следующий пример, в котором в пакете `points` имеется объявление

```
package points;
public class Point {
    protected int x, y;
    void warp(threePoint.Point3d a) {
        if (a.z > 0)          // Ошибка времени компиляции:
            a.delta(this);    // нельзя обратиться к a.z
    }
}
```

а в пакете `threePoint` **— объявление**

```
package threePoint;
import points.Point;
public class Point3d extends Point {
    protected int z;
    public void delta(Point p) {
        p.x += this.x; // Ошибка времени компиляции:
                       // нельзя обратиться к p.x
        p.y += this.y; // Ошибка времени компиляции:
                       // нельзя обратиться к p.y
    }
    public void delta3d(Point3d q) {
        q.x += this.x;
        q.y += this.y;
        q.z += this.z;
    }
}
```

В методе `delta` генерируется ошибка времени компиляции: он не может обратиться к членам `x` и `y` своего параметра `p`, объявленным как `protected`, поскольку в то время как `Point3d` (класс, в котором выполняется обращение к полям `x` и `y`) является подклассом `Point` (класс, в котором объявлены поля `x` и `y`), он не включен в реализацию `Point` (тип параметра `p`). Метод `delta3d` может обращаться к членам своего параметра `q`, объявленным как `protected`, поскольку класс `Point3d` является подклассом `Point` и включен в реализацию `Point3d`.

Метод `delta` может попытаться выполнить приведение (§5.5, §15.16) своего параметра к `Point3d`, но такое приведение приведет к генерации исключения, если классом `p` времени выполнения не является `Point3d`.

Ошибка времени компиляции генерируется также и в методе `warp`: он не может обратиться к защищенному члену `z` своего параметра `a`, поскольку в то время как класс `Point` (класс, в котором выполняется обращение к полю `z`) включен в реализацию `Point3d` (типа его параметра `a`), он не является подклассом `Point3d` (класса, в котором объявлен член `z`).

§6.7. Полностью квалифицированные и канонические имена

Каждый примитивный тип, именованный пакет, класс и интерфейс верхнего уровня имеет *полностью квалифицированное имя*.

- Полностью квалифицированное имя примитивного типа представляет собой ключевое слово для данного примитивного типа, а именно — `byte`, `short`, `char`, `int`, `long`, `float`, `double` и `boolean`.
- Полностью квалифицированное имя именованного пакета, не являющегося подпакетом именованного пакета, представляет собой его простое имя.
- Полностью квалифицированное имя именованного пакета, являющегося подпакетом другого именованного пакета, состоит из полностью квалифицированного имени содержащего пакета, за которым следуют точка `.` и простое имя подпакета.
- Полностью квалифицированное имя класса или интерфейса верхнего уровня, объявленного в неименованном пакете, представляет собой простое имя этого класса или интерфейса.
- Полностью квалифицированное имя класса или интерфейса верхнего уровня, объявленного в именованном пакете, состоит из полностью квалифицированного имени пакета, за которым следуют точка `.` и простое имя класса или интерфейса.

Каждый класс-член, интерфейс-член и тип массива *может* иметь полностью квалифицированное имя.

- Класс-член или интерфейс-член *M* другого класса или интерфейса *C* имеет полностью квалифицированное имя тогда и только тогда, когда *C* имеет полностью квалифицированное имя.

  В этом случае полностью квалифицированное имя *M* состоит из полностью квалифицированного имени *C*, за которым следуют точка `.` и простое имя *M*.
- Тип массива имеет полностью квалифицированное имя тогда и только тогда, когда тип его элементов имеет полностью квалифицированное имя.

  В этом случае полностью квалифицированное имя типа массива состоит из полностью квалифицированного имени типа компонента данного типа массива, за которым следует `[]`.

Локальный класс полностью квалифицированного имени не имеет.

Каждый примитивный тип, именованный пакет, класс верхнего уровня и интерфейс верхнего уровня имеет *каноническое имя*.

- Для каждого примитивного типа, именованного пакета, класса верхнего уровня и интерфейса верхнего уровня каноническое имя совпадает с полностью квалифицированным именем.

Каждый класс-член, интерфейс-член и тип массива *может* иметь каноническое имя.

- Класс-член или интерфейс-член *M*, объявленный в другом классе *C*, имеет каноническое имя тогда и только тогда, когда каноническое имя имеет *C*.

  В этом случае каноническое имя *M* состоит из канонического имени *C*, за которым следуют точка . и простое имя *M*.

- Тип массива имеет каноническое имя тогда и только тогда, когда тип его компонентов имеет каноническое имя.

  В этом случае каноническое имя типа массива состоит из канонического имени типа компонентов типа массива, за которым следует `[]`.

Локальный класс канонического имени не имеет.

Пример 6.7-1. Полностью квалифицированные имена

- Полностью квалифицированным именем типа `long` является "`long`".
- Полностью квалифицированным именем пакета `java.lang` является "`java.lang`", поскольку он представляет собой подпакет `lang` пакета `java`.
- Полностью квалифицированным именем класса `Object`, определенного в пакете `java.lang`, является "`java.lang.Object`".
- Полностью квалифицированным именем интерфейса `Enumeration`, определенного в пакете `java.util`, является "`java.util.Enumeration`".
- Полностью квалифицированным именем типа "массив `double`" является "`double[]`".
- Полностью квалифицированным именем типа "массив массивов массивов массивов элементов типа `String`" является "`java.lang.String[][][][]`".

В коде

```
package points;
class Point { int x, y; }
class PointVec { Point[] vec; }
```

полностью квалифицированным именем типа `Point` является "`points.Point`", полностью квалифицированным именем типа `PointVec` является "`points.PointVec`", а полностью квалифицированным именем типа поля `vec` класса `PointVec` является "`points.Point[]`".

Пример 6.7-2. Полностью квалифицированные и канонические имена

Различие между полностью квалифицированными и каноническими именами можно увидеть на примере следующего кода.

```
package p;
class O1 { class I {} }
class O2 extends O1 {}
```

И `p.O1.I`, и `p.O2.I` представляют собой полностью квалифицированные имена, которые обозначают класс-член `I`, но только `p.O1.I` является его каноническим именем.

ГЛАВА 7

Пакеты

Программы организованы как наборы пакетов. Каждый пакет имеет собственный набор имен для типов, что помогает предотвратить конфликты имен.

Тип верхнего уровня является доступным (§6.6) вне пакета, в котором он объявлен, только если этот тип объявлен как `public`.

Структура имен пакетов является иерархической (§7.1). Членами пакета являются типы классов и интерфейсов (§7.6), объявленные в модулях компиляции пакета, и подпакеты, которые могут содержать собственные модули компиляции и подпакеты.

Пакет может храниться в файловой системе или в базе данных (§7.2). Пакеты, хранящиеся в файловой системе, могут иметь определенные ограничения на организацию их модулей, которые обеспечат простую реализацию легкого поиска классов.

Пакет состоит из ряда модулей компиляции (§7.3). Модуль компиляции автоматически получает доступ ко всем типам, объявленным в его пакете, а также автоматически импортирует все `public`-типы, объявленные в предопределенном пакете `java.lang`.

Для небольших программ и случайных задач пакет может быть безымянным (§7.4.2) или иметь простое имя, но если код предназначен для широкого распространения, требуется с помощью квалифицированных имен выбирать уникальные имена пакетов. Это может предотвратить конфликты, которые имели бы место в противном случае, в ситуации, когда две группы разработчиков выбрали бы одно и то же имя для пакетов, которые позже могли бы использоваться в одной программе.

§7.1. Члены пакетов

Членами пакета являются его подпакеты, все типы классов верхнего уровня (§7.6, §8) и все типы интерфейсов верхнего уровня (§9), объявленные во всех модулях компиляции (§7.3) пакета.

> Например, в API платформы Java SE:
>
> - пакет `java` имеет подпакеты `awt`, `applet`, `io`, `lang`, `net` и `util`, но не имеет модулей компиляции;
> - пакет `java.awt` имеет подпакет `image`, а также ряд модулей компиляции, содержащих объявления типов классов и интерфейсов.

Если полностью квалифицированное имя (§6.7) пакета — *P*, а *Q* является подпакетом *P*, то *P*.*Q* представляет собой полностью квалифицированное имя подпакета, а кроме того, описывает пакет.

Пакет не может содержать два члена с одним и тем же именем; в противном случае генерируется ошибка времени компиляции.

Вот некоторые примеры.

- Поскольку пакет `java.awt` имеет подпакет `image`, он не может содержать (и не содержит) объявление типа класса или интерфейса с именем `image`.
- Если имеется пакет `mouse`, а в этом пакете есть тип-член `Button` (к которому после этого можно обратиться как к `mouse.Button`), то не может быть никакого пакета с полностью квалифицированным именем `mouse.Button` или `mouse.Button.Click`.
- Если `com.nighthacks.java.jag` является полностью квалифицированным именем типа, то не может быть пакета с полностью квалифицированным именем `com.nighthacks.java.jag` или `com.nighthacks.java.jag.scrabble`.

Однако члены разных пакетов могут иметь одно и то же простое имя. Например, можно объявить пакет

```
package vector;
public class Vector { Object[] vec; }
```

который имеет в качестве члена `public`-класс `Vector`, несмотря на то что в пакете `java.util` также объявлен класс с именем `Vector`. Эти два класса различны, что отражено в факте наличия у них разных полностью квалифицированных имен (§6.7). Полностью квалифицированное имя класса `Vector` в данном примере — `vector.Vector`, в то время как полностью квалифицированное имя класса `Vector`, включенного в платформу Java SE, — `java.util.Vector`. Поскольку пакет `vector` содержит класс с именем `Vector`, он не может содержать одновременно подпакет с тем же именем `Vector`.

Иерархическая структура именования пакетов предназначена для удобства организации связанных пакетов. Сама по себе она не предоставляет никаких преимуществ, имея ограничение, что пакет не может иметь подпакет с простым именем, совпадающим с именем типа верхнего уровня (§7.6), объявленного в этом пакете.

Например, нет какого-то особого отношения доступа между пакетами `oliver` и `oliver.twist` или между пакетами `evelyn.wood` и `evelyn.waugh`. То другими словами, никакой разницы между доступом кода из пакета `oliver.twist` к типам в пакете `oliver` по сравнению с кодом из любого иного пакета.

§7.2. Реализация пакетов в разных системах

Каждая вычислительная система определяет, как именно создаются и хранятся пакеты и модули компиляции.

Каждая система также определяет, какие модули компиляции наблюдаемы (§7.3) при конкретной компиляции. Наблюдаемость модуля компиляции, в свою очередь, определяет, какие пакеты наблюдаемы, а какие находятся в области видимости.

В простой реализации платформы Java SE пакеты и модули компиляции могут храниться в локальной файловой системе. Другие реализации могут хранить их с помощью распределенной файловой системы или той или иной формы базы данных.

Если система хранит пакеты и модули компиляции в базе данных, то эта база данных не должна налагать дополнительные ограничения (§7.6) на модули компиляции, допустимые при реализации на основе файлов.

> Например, система, в которой для хранения пакетов используется база данных, не может требовать наличия максимум одного открытого класса или интерфейса в модуле компиляции.

Однако использующие базы данных системы должны обеспечивать возможность конвертации программы в форму, подчиняющуюся этим ограничениям, для экспорта в реализации на основе файловых систем.

> В качестве чрезвычайно простого примера хранения пакетов в файловой системе все пакеты, исходные тексты и бинарный код проекта могут храниться в одном каталоге и его подкаталогах. Каждый непосредственный подкаталог этого каталога будет представлять пакет верхнего уровня, т.е. пакет, полностью квалифицированное имя которого состоит из одного простого имени. Каждый подкаталог следующего уровня будет представлять подпакет пакета, представленного содержащим подкаталог каталогом, и т.д.
>
> Каталог может содержать следующие непосредственные подкаталоги.
>
> ```
> com
> gls
> jag
> java
> wnj
> ```
>
> Здесь каталог `java` содержит пакеты платформы Java SE; каталоги `jag`, `gls` и `wnj` могут содержать пакеты, которые три автора данной книги создали для личного использования и совместно используют их в своей небольшой группе; а каталог `com` будет содержать пакеты компаний, которые используют описанное в §6.1 соглашение по генерации уникальных имен их пакетов.
>
> Каталог `java` может, помимо прочих, содержать следующие подкаталоги.
>
> ```
> applet
> awt
> io
> lang
> net
> util
> ```
>
> Они соответствуют пакетам `java.applet`, `java.awt`, `java.io`, `java.lang`, `java.net` и `java.util`, определенным как часть API платформы Java SE.
>
> Если мы взглянем в каталог `util`, то сможем увидеть там следующие файлы.

```
BitSet.java    Observable.java
BitSet.class   Observable.class
Date.java      Observer.java
Date.class     Observer.class
...
```

Здесь каждый `.java`-файл содержит исходный текст модуля компиляции (§7.3) с определением класса или интерфейса, скомпилированная бинарная форма которого хранится в соответствующем `.class`-файле.

При этой простой организации пакетов реализация платформы Java SE трансформирует имя пакета в файловый путь посредством конкатенации компонентов имени пакета, помещая разделитель (указатель каталога) между соседними компонентами.

Например, если эта простая организация используется в операционной системе, в которой разделителем является символ "/", то имя пакета

```
jag.scrabble.board
```

будет преобразовано в имя каталога

```
jag/scrabble/board
```

Компонент имени пакета или класса может содержать символ, который запрещено использовать в имени обычного каталога файловой системы, например символ Unicode в системе, которая допускает в именах файлов только ASCII-символы. Можно принять соглашение, по которому использовать управляющий символ (скажем, "@"), за которым следуют четыре шестнадцатеричные цифры, дающие числовое значение символа, как в управляющих последовательностях `\uxxxx` (§3.3).

В таком случае имя пакета

```
children.activities.crafts.papierM\u00e2ch\u00e9
```

которое можно записать с использованием полного набора символов Unicode как

```
children.activities.crafts.papierMâché
```

будет отображаться на имя каталога

```
children/activities/crafts/papierM@00e2ch@00e9
```

Если символ "@" является недопустимым в имени файла символом в некоторой файловой системе, можно воспользоваться каким-либо иным символом, который нельзя использовать в идентификаторе.

§7.3. Модули компиляции

CompilationUnit является целевым символом (§2.1) синтаксической грамматики (§2.3) программ на языке программирования Java. Он определяется следующей продукцией.

CompilationUnit:
[PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

Модуль компиляции состоит из трех частей, каждая из которых является необязательной.

- Объявление `package` (§7.4), дающее полностью квалифицированное имя (§6.7) пакета, которому принадлежит модуль компиляции.

Модуль компиляции, в котором нет объявления `package`, является частью безымянного пакета (§7.4.2).

- Объявления `import` (§7.5), которые позволяют ссылаться на типы из других пакетов и члены типов, объявленные как `static`, с использованием их простых имен.
- Объявления верхнего уровня (§7.6) типов класса или интерфейса.

Каждый модуль компиляции неявно импортирует каждое имя типа, объявленное как `public` в предопределенном пакете `java.lang`, как если бы в начале каждого модуля компиляции непосредственно за инструкцией `package` следовало объявление `import java.lang.*`. В результате имена всех этих типов доступны как простые имена в каждом модуле компиляции.

Все модули компиляции предопределенного пакета `java` и его подпакетов `lang` и `io` всегда *наблюдаемы*.

Для всех прочих пакетов наблюдаемость определяется вычислительной системой.

Наблюдаемость модуля компиляции влияет на наблюдаемость его пакета (§7.4.3).

Типы, объявленные в разных модулях компиляции, могут циклически зависеть друг от друга. Компилятор Java должен принять меры для одновременной компиляции всех таких типов.

§7.4. Объявления пакетов

Объявление `package` находится внутри модуля компиляции для указания пакета, которому принадлежит этот модуль компиляции.

§7.4.1. Именованные пакеты

Объявление пакета в модуле компиляции определяет имя (§6.2) пакета, которому принадлежит этот модуль компиляции.

PackageDeclaration:
{PackageModifier} `package` *Identifier {. Identifier}* `;`

PackageModifier:
Annotation

Имя пакета, упомянутое в объявлении `package`, должно быть полностью квалифицированным именем (§6.7) пакета.

Область видимости и затенение объявления пакета рассматриваются в §6.3 и §6.4.

Для данного пакета может иметься не более одного аннотированного объявления `package`.

Способ обеспечения этого ограничения при необходимости варьируется от реализации к реализации. Для реализаций на основе файловых систем настоятельно рекомендуется следующая схема: единственное аннотированное объявление `package`, если таковое существует, помещается в исходном файле с именем

`package-info.java` в каталог, содержащий исходные файлы пакета. Этот файл не содержит исходный текст класса с именем `package-info.java`; это попросту невозможно, поскольку `package-info` не является допустимым идентификатором. Обычно `package-info.java` содержит только объявление пакета, которому непосредственно предшествует аннотация пакета. Хотя технически этот файл может содержать исходный код для одного или нескольких классов с уровнем доступа пакета, это было бы очень нездоровой практикой.

Рекомендуется, чтобы `package-info.java`, если таковой присутствует, занял место `package.html` для `javadoc` и других аналогичных систем генерации документации. Если этот файл присутствует, инструмент генерации документации должен искать комментарий документации пакета непосредственно перед (возможно, аннотированным) объявлением `package` в `package-info.java`. Таким образом, `package-info.java` становится единственным хранилищем аннотаций и документации уровня пакета. Если в будущем потребуется добавить любую другую информацию уровня пакета, ее хранилищем должен стать именно этот файл.

§7.4.2. Безымянные пакеты

Модуль компиляции, не имеющий объявления `package`, является частью *безымянного пакета*.

Безымянные пакеты предоставляются платформой Java SE главным образом для удобства при разработке малых или временных приложений или на первых этапах разработки.

Безымянный пакет не может иметь подпакеты, так как синтаксис объявления `package` всегда включает ссылку на именованный пакет верхнего уровня.

Реализация платформы Java SE должна поддерживать по крайней мере один безымянный пакет; она может поддерживать и несколько безымянных пакетов, но не обязана делать это. Какой модуль компиляции находится в том или ином безымянном пакете, определяется вычислительной системой.

Например, модуль компиляции

```
class FirstCall {
    public static void main(String[] args) {
        System.out.println("Мистер Ватсон, подойдите. "
                           + "Вы мне нужны.");
    }
}
```

определяет очень простой модуль компиляции как часть безымянного пакета.

В реализациях платформы Java SE, которые для хранения пакетов используют иерархическую файловую систему, одна из распространенных стратегий состоит в связывании безымянного пакета с каждым каталогом; одновременно наблюдаемым является только один безымянный пакет, а именно — тот, который связан с "текущим рабочим каталогом". Точное значение понятия "текущий рабочий каталог" зависит от конкретной вычислительной системы.

§7.4.3. Наблюдаемость пакетов

Пакет *наблюдаем* (observable) тогда и только тогда, когда наблюдаемым является

- либо модуль компиляции (§7.3), который содержит объявление пакета;
- либо подпакет этого пакета.

Пакеты `java`, `java.lang` и `java.io` всегда наблюдаемы.

> Этот вывод можно сделать, исходя из приведенного выше правила и правил наблюдаемости модулей компиляции, следующим образом. Предопределенный пакет `java.lang` объявляет класс `Object`, так что модуль компиляции для `Object` всегда наблюдаемый (§7.3). Следовательно, пакет `java.lang` является наблюдаемым (§7.4.3), а значит, и пакет `java` тоже. Кроме того, поскольку `Object` является наблюдаемым, неявно существует тип массива `Object[]`. Существует также его суперинтерфейс `java.io.Serializable` (§10.1), следовательно, пакет `java.io` наблюдаемый.

§7.5. Объявления импорта

Объявление импорта позволяет обращаться к именованному типу или члену, объявленному как `static`, через простое имя (§6.2), состоящее из одного идентификатора.

Без использования соответствующего объявления `import` единственный способ обращения к типу, объявленному в другом пакете, — с помощью полностью квалифицированного имени (§6.7).

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration
SingleStaticImportDeclaration
StaticImportOnDemandDeclaration

- Объявление импорта единственного типа (§7.5.1) импортирует единственный именованный тип с помощью его канонического имени (§6.7).
- Объявление импорта типа по требованию (§7.5.2) импортирует все доступные типы (§6.6) именованного типа или именованного пакета при необходимости с помощью канонического имени типа или пакета.
- Объявление единственного статического импорта (§7.5.3) импортирует все доступные `static`-члены с данным именем из типа, задаваемого каноническим именем.
- Объявление статического импорта по требованию (§7.5.4) импортирует все доступные `static`-члены именованного типа при необходимости с помощью канонического имени типа.

Область видимости и затенение типа или членов, импортированных этими объявлениями, определяются в §6.3 и §6.4.

Объявление `import` делает типы или члены доступными по их простым именам только в пределах модуля компиляции, который содержит это объявление импорта. Область видимости типа (типов) или члена (членов), введенных объявлением импорта, в частности, не включает *PackageName* объявления `package`, другие объявления `import` в текущем модуле компиляции или другие модули компиляции в том же пакете.

§7.5.1. Объявления импорта единственного типа

Объявление импорта единственного типа импортирует единственный именованный тип с помощью его канонического имени, делая его доступным посредством простого имени в объявлениях класса и интерфейса модуля компиляции, в котором находится это объявление импорта.

SingleTypeImportDeclaration:
`import` *TypeName* `;`

TypeName должно представлять собой каноническое имя (§6.7) типа класса, типа интерфейса, типа перечисления или типа аннотации.

Имя должно быть квалифицированным (§6.5.5.2), иначе генерируется ошибка времени компиляции.

Если именованный тип недоступен (§6.6), генерируется ошибка времени компиляции.

Если два объявления импорта единственного типа в одном и том же модуле компиляции пытаются импортировать типы с одинаковыми простыми именами, то генерируется ошибка времени компиляции, если только эти два типа не являются одним и тем же (в таком случае повторное объявление игнорируется).

Если тип, импортированный объявлением импорта единственного типа, объявляется в модуле компиляции, который содержит объявление `import`, это объявление `import` игнорируется.

Если объявление импорта единственного типа импортирует тип, простое имя которого — n, а модуль компиляции также объявляет тип верхнего уровня (§7.6), простое имя которого — n, генерируется ошибка времени компиляции.

Если модуль компиляции содержит как объявление импорта единственного типа, которое импортирует тип с простым именем n, так и объявление единственного статического импорта (§7.5.3), которое импортирует тип с простым именем n, генерируется ошибка времени компиляции.

Пример 7.5.1-1. Импорт единственного типа

```
import java.util.Vector;
```

Это объявление приводит к тому, что в объявлениях класса и интерфейса в модуле компиляции доступно простое имя `Vector`. Таким образом, простое имя `Vector` относится к объявлению типа `Vector` в пакете `java.util` во всех местах, где оно не затенено (§6.4.1) или затемнено (§6.4.2) объявлением поля, параметра, локальной переменной или объявлением вложенного типа с тем же именем.

Обратите внимание, что фактическое объявление `java.util.Vector` является обобщенным (§8.1.2). После импорта имя `Vector` может использоваться без

квалификации в параметризованном типе, как, например, тип `Vector<String>` или как несформированный тип `Vector`. Это подчеркивает ограничение объявления `import`: тип, вложенный в объявление обобщенного типа, может быть импортирован, но внешний тип всегда затерт.

Пример 7.5.1-2. Повторные объявления типов

Программа

```
import java.util.Vector;
class Vector { Object[] vec; }
```

приводит к генерации ошибки времени компиляции из-за повторного объявления `Vector`, как и в следующей программе

```
import java.util.Vector;
import myVector.Vector;
```

Здесь `myVector` представляет собой пакет, содержащий модуль компиляции:

```
package myVector;
public class Vector { Object[] vec; }
```

Пример 7.5.1-3. Запрет импорта подпакета

Обратите внимание, что инструкция `import` может импортировать только тип, но не подпакет.

Например, попытка импортировать `java.util`, а затем использовать имя `util.Random` для обращения к типу `java.util.Random` работать не будет.

```
import java.util;
class Test { util.Random generator; }
// Неверно: ошибка времени компиляции
```

Пример 7.5.1-4. Импорт имени типа, являющегося также именем пакета

Имена пакетов и имена типов при использовании соглашений, описанных в §6.1, обычно различны. Тем не менее в надуманном примере с не соответствующим соглашению именем пакета `Vector`, который объявляет открытый класс `Mosquito`

```
package Vector;
public class Mosquito { int capacity; }
```

и модулем компиляции

```
package strange;
import java.util.Vector;
import Vector.Mosquito;
class Test {
    public static void main(String[] args) {
        System.out.println(new Vector().getClass());
        System.out.println(new Mosquito().getClass());
    }
}
```

объявление импорта единственного типа, импортирующее класс `Vector` из пакета `java.util`, не мешает корректно распознать имя пакета `Vector` в следующем

за ним объявлении `import`. Пример успешно компилируется и выводит на экран следующие строки.

```
class java.util.Vector
class Vector.Mosquito
```

§7.5.2. Объявление импорта типа по требованию

Объявление импорта типа по требованию позволяет при необходимости импортировать все доступные типы именованного пакета или типа.

TypeImportOnDemandDeclaration:
`import` *PackageOrTypeName* `. * ;`

PackageOrTypeName должен быть каноническим именем (§6.7) пакета, типа класса, типа интерфейса, типа перечисления или типа аннотации.

Если *PackageOrTypeName* обозначает тип (§6.5.4), то имя должно быть квалифицированным (§6.5.5.2), иначе генерируется ошибка времени компиляции.

Если имя пакета или типа недоступно (§6.6), генерируется ошибка времени компиляции.

Для имен текущего пакета или пакета `java.lang` в объявлении импорта типа по требованию ошибка времени компиляции не генерируется; в таких случаях объявление импорта типа по требованию игнорируется.

Одно и то же имя пакета или типа может использоваться в двух и более объявлениях импорта типа по запросу в одном модуле компиляции. В таком случае все объявления, кроме одного, считаются излишними; результат выглядит так, как если бы этот тип был импортирован только один раз.

Если модуль компиляции содержит и объявление импорта типа по требованию, и объявление статического импорта по требованию (§7.5.4), которые именуют один и тот же тип, результат выглядит так, как если бы типы `static` членов этого типа (§8.5, §9.5) были импортированы только один раз.

Пример 7.5.2-1. Импорт типа по требованию

```
import java.util.*;
```

Это объявление импорта приводит к тому, что простые имена всех открытых (`public`) типов, объявленных в пакете `java.util`, становятся доступными в пределах объявлений классов и интерфейсов модуля компиляции. Таким образом, простое имя `Vector` ссылается на тип `Vector` из пакета `java.util` во всех местах модуля компиляции, где объявление этого типа не затеняется (§6.4.1) или затемняется (§6.4.2).

Объявление может затеняться объявлением импорта единственного типа, простое имя которого — `Vector`, типом с именем `Vector`, объявленным в пакете, к которому принадлежит модуль компиляции, или любыми вложенными классами или интерфейсами.

Объявление может быть затенено объявлением поля, параметра или локальной переменной с именем `Vector`.

(Любая из этих ситуаций, впрочем, достаточно необычна на практике.)

§7.5.3. Объявления единственного статического импорта

Объявление единственного статического импорта импортирует все доступные `static`-члены типа с заданным простым именем. Такое объявление делает все эти `static`-члены доступными по их простому имени в объявлениях классов и интерфейсов модуля компиляции, в котором находится данное объявление единственного статического импорта.

SingleStaticImportDeclaration:
`import static` *TypeName* `.` *Identifier* `;`

TypeName должен быть каноническим именем (§6.7) типа класса, типа интерфейса, типа перечисления или типа аннотации.

Имя должно быть квалифицированным (§6.5.5.2), иначе генерируется ошибка времени компиляции.

Если именованный тип является недоступным (§6.6), генерируется ошибка времени компиляции.

Identifier должен именовать по крайней мере один `static`-член именованного типа. Если нет никаких статических членов с таким именем или если все именованные члены являются недоступными, генерируется ошибка времени компиляции.

Допускается импортирование одним объявлением единственного статического импорта нескольких полей или типов с одним и тем же именем или нескольких методов с одним и тем же именем и сигнатурой.

Если объявление единственного статического импорта импортирует тип, простое имя которого — n, и модуль компиляции также объявляет тип верхнего уровня (§7.6), простое имя которого — n, генерируется ошибка времени компиляции.

Если модуль компиляции содержит как объявление единственного статического импорта, которое импортирует тип с простым именем n, так и объявление единственного импорта типа (§7.5.1), которое импортирует тип с тем же простым именем n, генерируется ошибка времени компиляции.

§7.5.4. Объявления статического импорта по требованию

Объявление статического импорта по требованию позволяет при необходимости импортировать все доступные `static`-члены именованного типа.

StaticImportOnDemandDeclaration:
`import static` *TypeName* `. * ;`

TypeName должен быть каноническим именем (§6.7) типа класса, типа интерфейса, типа перечисления или типа аннотации.

Имя должно быть квалифицированным (§6.5.5.2), иначе генерируется ошибка времени компиляции.

Если именованный тип является недоступным (§6.6), генерируется ошибка времени компиляции.

Два или более объявлений статического импорта по требованию в одном и том же модуле компиляции могут именовать один и тот же тип; результат выглядит так, как если бы имелось только одно такое объявление.

Два или более объявлений статического импорта по требованию в одном и том же модуле компиляции могут именовать один и тот же член; результат выглядит так, как если бы член импортировался только одним таким объявлением.

Допускается импортирование одним объявлением статического импорта по требованию нескольких полей или типов с одним и тем же именем или нескольких методов с одним и тем же именем и сигнатурой.

Если модуль компиляции содержит как объявление статического импорта по требованию, так и объявление импорта типа по требованию (§7.5.2), которые именуют один и тот же тип, результат выглядит так, как если бы статические члены импортируемого типа (§8.5, §9.5) импортировались только один раз.

§7.6. Объявления типа верхнего уровня

Объявление типа верхнего уровня объявляет тип класса верхнего уровня (§8) или тип интерфейса верхнего уровня (§9).

TypeDeclaration:
    ClassDeclaration
    InterfaceDeclaration
    ;

> Дополнительные токены “*;*” на уровне объявлений типов в модуле компиляции не влияют на значение модуля компиляции. Лишние точки с запятой разрешены в языке программирования Java исключительно как уступка программистам на C++, которые привыкли к символам “*;*” после объявления класса. Они не должны использоваться в новых исходных текстах на Java.

При отсутствии модификатора доступа типы верхнего уровня имеют доступ пакета: они доступны только в пределах модулей компиляции пакета, в котором объявлены (§6.6.1). Тип может быть объявлен как `public`, чтобы предоставить доступ к нему коду из других пакетов.

Если объявление типа верхнего уровня содержит любой из модификаторов доступа `protected`, `private` или `static`, генерируется ошибка времени компиляции.

Если имя типа верхнего уровня встречается как имя любого другого типа класса или интерфейса верхнего уровня, объявленного в том же пакете, генерируется ошибка времени компиляции.

Область видимости и затенение типа верхнего уровня описываются в §6.3 и §6.4.

Полностью квалифицированное имя типа верхнего уровня описывается в §6.7.

Пример 7.6-1. Конфликты объявлений типа верхнего уровня

```
package test;
import java.util.Vector;
class Point {
```

```
    int x, y;
}
interface Point {                    // Ошибка времени компиляции 1
    int getR();
    int getTheta();
}
class Vector { Point[] pts; } // Ошибка времени компиляции 2
```

Здесь первая ошибка времени компиляции вызвана двойным объявлением имени `Point` и как класса, и как интерфейса в одном и том же пакете. Вторая ошибка времени компиляции представляет собой попытку объявить имя `Vector` как объявлением типа класса, так и объявлением импорта единственного типа.

Заметим, однако, что не является ошибочной ситуация, когда имя класса является также именем типа, который в противном случае мог бы быть импортирован объявлением импорта типа по требованию (§7.5.2) в модуле компиляции (§7.3), содержащем объявление класса. Так, в программе

```
package test;
import java.util.*;
class Vector {} // Ошибки времени компиляции нет
```

объявление класса `Vector` допустимо несмотря на то, что имеется также класс `java.util.Vector`. В этом модуле компиляции простое имя `Vector` ссылается на класс `test.Vector`, а не на `java.util.Vector` (к которому в коде этого модуля компиляции можно обратиться с помощью полностью квалифицированного имени).

Пример 7.6-2. Область видимости типов верхнего уровня

```
package points;
class Point {
    int x, y;                  // Координаты
    PointColor color;          // Цвет точки
    Point next;                // Следующая точка этого цвета
    static int nPoints;
}
class PointColor {
    Point first;               // Первая точка этого цвета
    PointColor(int color) { this.color = color; }
    private int color;         // Компоненты цвета
}
```

В этой программе определены два класса, которые используют друг друга в объявлениях их членов. Поскольку типы классов `Point` и `PointColor` объявлены в пакете `points`, в текущем модуле компиляции, являющемся их областью видимости, программа компилируется корректно, т.е. опережающие ссылки проблемой не являются.

Пример 7.6-3. Полностью квалифицированные имена

```
class Point { int x, y; }
```

В этом коде класс `Point` объявлен в модуле компиляции без инструкции `package`, и, таким образом, `Point` является его полностью квалифицированным именем, в то время как в коде

```
package vista;
class Point { int x, y; }
```

полностью квалифицированным именем класса `Point` является `vista.Point`. (Имя пакета `vista` подходит для локального или персонального применения; если пакет предназначен для широкого распространения, лучше дать ему уникальное имя пакета (§6.1).)

Реализация платформы Java SE должна отслеживать типы в пакетах по их бинарным именам (§13.1). Многочисленные способы именования типов должны проходить этап преобразования в бинарные имена, чтобы гарантировать, что такие имена корректно распознаются при обращении к одному и тому же типу.

Например, если модуль компиляции содержит объявление импорта единственного типа (§7.5.1)

```
import java.util.Vector;
```

то в пределах модуля компиляции простое имя `Vector` и полностью квалифицированное имя `java.util.Vector` ссылаются на один и тот же тип.

Тогда и только тогда, когда пакеты хранятся в файловой системе (§7.2), базовая вычислительная система может наложить ограничение, заключающееся в генерации ошибки времени компиляции, если тип не найден в файле с именем, составленным из имени типа плюс расширение (такое, как `.java` или `.jav`), и при этом выполняется одно из условий:

- на тип ссылается код в другом модуле компиляции пакета, в котором этот тип объявлен;
- тип объявлен как `public` (а следовательно, потенциально доступен коду из других пакетов).

Это ограничение означает, что должно существовать не более одного такого типа на модуль компиляции. Это ограничение упрощает компилятору языка программирования Java поиск именованного класса в пакете. На практике многие программисты предпочитают размещать каждый класс или интерфейс в собственном модуле компиляции, независимо от того, является ли он открытым или ссылается ли на него код из других модулей компиляции.

Например, исходный код `public`-типа `wet.sprocket.Toad` будет находиться в файле `Toad.java` в каталоге `wet/sprocket`, а соответствующий объектный код будет находиться в файле `Toad.class` в том же каталоге.

ГЛАВА 8

Классы

Объявления классов определяют новые ссылочные типы и описывают, как они реализованы (§8.1).

Класс верхнего уровня представляет собой класс, не являющийся вложенным классом.

Вложенный класс представляет собой любой класс, объявление которого находится в теле другого класса или интерфейса.

В этой главе обсуждается общая семантика всех классов — верхнего уровня (§7.6) и вложенных (включая классы-члены (§8.5, §9.5), локальные классы (§14.3) и анонимные классы (§15.9.5)). Детали, специфичные для конкретных видов классов, рассматриваются в разделах, посвященных этим конструкциям.

Именованный класс может быть объявлен как `abstract` (§8.1.1.1) и должен быть объявлен абстрактным, если он реализован не полностью; такой класс не может быть инстанцирован, но может быть расширен подклассами. Класс может быть объявлен как `final` (§8.1.1.2), в этом случае он не может иметь подклассы. Если класс объявлен как `public`, то к нему можно обращаться из других пакетов. Каждый класс, за исключением `Object`, является расширением (т.е. подклассом) единственного существующего класса (§8.1.4) и может реализовывать интерфейсы (§8.1.5). Классы могут быть *обобщенными* (§8.1.2), т.е. могут объявлять переменные типа, значения которых могут отличаться у разных экземпляров класса.

Классы могут сопровождаться аннотациями (§9.7) так же, как и объявления любого другого вида.

Тело класса объявляет члены (поля и методы, а также вложенные классы и интерфейсы), инициализаторы экземпляра и статические, а также конструкторы (§8.1.6). Областью видимости (§6.3) члена (§8.2) является все тело объявления класса, которому принадлежит член. Объявления полей, методов, классов-членов, интерфейсов-членов и конструкторов могут включать модификаторы доступа (§6.6) `public`, `protected` или `private`. Члены класса включают как объявленные, так и унаследованные члены (§8.2). Вновь объявленные поля могут скрывать поля, объявленные в суперклассе или суперинтерфейсе. Вновь объявленные члены класса и члены интерфейса могут скрывать члены класса или интерфейса, объявленные в суперклассе или суперинтерфейсе. Вновь объявленные методы могут скрывать, реализовать или перекрывать методы, объявленные в суперклассе или суперинтерфейсе.

Объявления полей (§8.3) описывают переменные класса, которые создаются только один раз, и переменные экземпляров, которые создаются для каждого экземпляра класса.

Поле может быть объявлено как `final` (§8.3.1.2), в этом случае присвоение ему значения может быть выполнено только один раз. Любое объявление поля может включать инициализатор.

Объявления классов-членов (§8.5) описывают вложенные классы, которые являются членами охватывающего класса. Классы-члены могут быть объявлены как `static`, и в этом случае они не имеют доступа к переменным экземпляра охватывающего класса; либо они могут быть внутренними классами (§8.1.3).

Объявления интерфейсов-членов (§8.5) описывают вложенные интерфейсы, которые являются членами охватывающего класса.

Объявления методов (§8.4) описывают код, который может быть вызван выражениями вызова метода (§15.12). Метод класса связан с самим типом класса; метод экземпляра относится к конкретному объекту, являющемуся экземпляром типа класса. Метод, объявление которого не указывает, как он реализован, должен быть объявлен как `abstract`. Метод может быть объявлен как `final` (§8.4.3.3), и в этом случае он не может быть скрыт или перекрыт. Метод может быть реализован зависящим от конкретной платформы `native`-кодом (§8.4.3.4). Метод, объявленный как `synchronized` (§8.4.3.6), автоматически блокирует объект перед выполнением его тела и автоматически разблокирует при возврате из метода, как если бы использовалась инструкция `synchronized` (§14.19), таким образом обеспечивая синхронизацию действий с действиями других потоков (§17).

Имена методов могут быть перегружены (§8.4.9).

Инициализаторы экземпляров (§8.6) представляют собой блоки выполняемого кода, который может использоваться для инициализации экземпляра при его создании (§15.9).

Статические инициализаторы (§8.7) представляют собой блоки выполняемого кода, который может использоваться для инициализации класса.

Конструкторы (§8.8) аналогичны методам, но не могут быть вызваны непосредственно вызовом метода; они используются для инициализации новых экземпляров класса. Как и методы, они могут быть перегружены (§8.8.8).

8.1. Объявления классов

Объявление класса определяет новый именованный ссылочный тип.

Имеется два вида объявлений классов: *объявления обычных классов* и *объявления перечислений*.

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration

NormalClassDeclaration:
{ClassModifier} `class` *Identifier [TypeParameters]*
[Superclass] [Superinterfaces] ClassBody

Правила из этого раздела применимы ко всем объявлениям классов, включая объявления перечислений. Однако для объявления перечислений применяются специальные правила, касающиеся модификаторов классов, внутренних классов и суперклассов; эти правила перечислены в §8.9.

Identifier в объявлении класса определяет имя класса.

Если класс имеет то же простое имя, что и любой из охватывающих его классов или интерфейсов, генерируется ошибка времени компиляции.

Область видимости и затенение объявления класса рассматриваются в §6.3 и §6.4.

§8.1.1. Модификаторы класса

Объявление класса может включать *модификаторы класса*.

ClassModifier: одно из
Annotation `public protected private`
`abstract static final strictfp`

Правила для модификаторов аннотаций в объявлениях классов определены в §9.7.4 и §9.7.5.

Модификатор доступа `public` (§6.6) применим только к классам верхнего уровня (§7.6) и классам-членам (§8.5), но не к локальным (§14.3) или анонимным (§15.9.5) классам.

Модификаторы доступа `protected` и `private` (§6.6) применимы только к классам-членам в непосредственно охватывающем классе или к объявлениям перечислений (§8.5).

Модификатор `static` применим только к классам-членам (§8.5.1), но не к классам верхнего уровня, локальным или анонимным.

Если один и тот же модификатор встречается в объявлении класса более одного раза, генерируется ошибка времени компиляции.

> Если в объявлении класса встречаются два или более (различных) модификаторов класса, то обычно, хотя и необязательно, они располагаются в порядке, согласующемся с приведенным выше в продукции для *ClassModifier*.

§8.1.1.1. Абстрактные классы

Абстрактный класс (объявленный как `abstract`) представляет собой незавершенный класс или класс, рассматривающийся как незавершенный.

При попытке создания экземпляра абстрактного класса с помощью выражения создания экземпляра класса (§15.9.1) генерируется ошибка времени компиляции.

Подкласс абстрактного класса, который сам по себе не является абстрактным, может быть инстанцирован, что приводит к выполнению конструктора абстрактного класса и, следовательно, к выполнению инициализаторов полей для переменных экземпляра этого класса.

Обычные классы могут иметь методы, объявленные как `abstract`, т.е. методы, которые объявлены, но еще не реализованы (§8.4.3.1), только если эти классы объявлены как `abstract`. Если обычный класс, не объявленный как `abstract`, содержит метод, объявленный как `abstract`, генерируется ошибка времени компиляции.

Класс `C` имеет абстрактные методы, если выполняется одно из следующих условий.

- Любой из методов-членов (§8.2) `C` — объявленный или унаследованный — является `abstract` (§8.4.3).

- Некоторый из суперклассов *C* имеет метод, объявленный как `abstract` с доступом на уровне пакета, и не имеется метода, который перекрывает абстрактный метод из *C* или суперкласса *C*.

Если тип абстрактного класса объявлен так, что невозможно создать подкласс, реализующий все его абстрактные методы, генерируется ошибка времени компиляции. Эта ситуация может осуществиться в случае, когда класс в качестве членов пытается иметь два абстрактных метода с одной и той же сигнатурой (§8.4.2), но с возвращаемыми типами, для которых нет типа, являющегося взаимозаменяемым (§8.4.5) с ними обоими.

Пример 8.1.1.1-1. Объявление абстрактного класса

```
abstract class Point {
    int x = 1, y = 1;
    void move(int dx, int dy) {
        x += dx;
        y += dy;
        alert();
    }
    abstract void alert();
}
abstract class ColoredPoint extends Point {
    int color;
}
class SimplePoint extends Point {
    void alert() { }
}
```

Здесь класс `Point` обязан быть объявлен как `abstract`, поскольку он содержит объявление абстрактного метода `alert`. Подкласс `Point` с именем `ColoredPoint` наследует абстрактный метод `alert`, так что он также должен быть объявлен как `abstract`. С другой стороны, подкласс `Point` с именем `SimplePoint` предоставляет реализацию `alert`, так что он не обязан быть объявлен как `abstract`.

Код

```
Point p = new Point();
```

приводит к генерации ошибки времени компиляции; класс `Point` не может быть инстанцирован, так как он объявлен как `abstract`. Однако переменная типа `Point` может быть корректно инициализирована ссылкой на любой подкласс `Point`, а так как класс `SimplePoint` абстрактным не является, то инструкция

```
Point p = new SimplePoint();
```

вполне корректна. Инстанцирование `SimplePoint` приводит к вызову конструктора по умолчанию и выполнению инициализаторов для полей `x` и `y` класса `Point`.

Пример 8.1.1.1-2. Объявление абстрактного класса, который запрещает создание подклассов

```
interface Colorable {
    void setColor(int color);
```

```
}
abstract class Colored implements Colorable {
    public abstract int setColor(int color);
}
```

Эти объявления приводят к ошибке времени компиляции: ни в каком подклассе класса `Colored` невозможно обеспечить реализацию метода с именем `setColor`, принимающим один аргумент типа `int`, который может удовлетворить спецификациям обоих абстрактных методов, потому что один (в интерфейсе `Colorable`) не должен возвращать какого-либо значения, в то время как другой (в классе `Colored`) должен возвращать значение типа `int` (§8.4).

Тип класса должен быть объявлен как `abstract`, только если его цель состоит в создании подклассов для завершения реализации. Если целью является простое предотвращение создания экземпляров класса, то корректным способом достичь этого является объявление конструктора (§8.8.10) без аргументов. Такой конструктор можно объявить как `private`, никогда его не вызывать и не объявлять другие конструкторы. Класс такого вида обычно содержит методы и переменные класса.

Класс `Math` является примером класса, который не может быть инстанцирован; его объявление имеет следующий вид.

```
public final class Math {
    private Math() { } // Класс не инстанцируется
    ... объявления методов и переменных класса ...
}
```

§8.1.1.2. Классы `final`

Класс может быть объявлен как `final`, если его определение завершено и никакие его подклассы в дальнейшем не могут потребоваться.

Если имя класса, объявленного как `final`, находится в инструкции `extends` (§8.1.4) объявления другого класса, генерируется ошибка времени компиляции. Отсюда вытекает, что класс, объявленный как `final`, не может иметь подклассов.

Если класс объявлен и как `final`, и как `abstract`, генерируется ошибка времени компиляции, поскольку реализация такого класса не может быть завершена (§8.1.1.1).

Поскольку класс, объявленный как `final`, не может иметь подклассов, методы такого класса никогда не перекрываются (§8.4.8.1).

§8.1.1.3. Классы `strictfp`

Воздействие модификатора `strictfp` состоит в том, что все `float`- или `double`-выражения в объявлении класса (включая выражения в инициализаторах переменных, инициализаторах экземпляров, статических инициализаторах и конструкторах) явным образом становятся FP-строгими (используют переносимые механизмы для вычислений с плавающей точкой) (§15.4).

Отсюда вытекает, что все методы, объявленные в таком классе, и все вложенные типы, объявленные в классе, неявно являются `strictfp`.

§8.1.2. Обобщенные классы и параметры типа

Класс является *обобщенным*, если он объявляет одну или несколько переменных типа (§4.4).

Эти переменные типа известны как *параметры типа* класса. Раздел параметров типа следует за именем класса и ограничивается угловыми скобками.

TypeParameters:
    < *TypeParameterList* >

TypeParameterList:
    TypeParameter {, *TypeParameter*}

Далее для удобства приведены продукции из §4.4.

TypeParameter:
    {TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
    Annotation

TypeBound:
    `extends` *TypeVariable*
    `extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
    & *InterfaceType*

Правила для модификаторов аннотаций в параметре типа определены в §9.7.4 и §9.7.5.

В разделе параметров типа класса переменная типа T *непосредственно зависит* от переменной типа S, если S является границей T, в то время как T зависит от S, если либо T непосредственно зависит от S, либо T непосредственно зависит от переменной типа U, которая зависит от S (с рекурсивным применением данного определения). Если переменная типа в разделе параметров типа класса зависит от себя самой, генерируется ошибка времени компиляции.

Область видимости и затенение параметра типа класса рассматриваются в §6.3 и §6.4.

Объявление обобщенного класса определяет множество параметризованных типов (§4.5), по одному для каждого возможного заполнения набора значений раздела параметров типов аргументами типа. Все эти параметризованные типы во время выполнения программы совместно используют один и тот же класс.

Например, при выполнении кода

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

переменная `b` получает значение `true`.

Если обобщенный класс является прямым или непрямым подклассом класса `Throwable` (§11.1.1), генерируется ошибка времени компиляции.

Это ограничение связано с тем, что механизм перехвата исключений Java Virtual Machine работает только с необобщенными классами.

При обращении к параметру типа обобщенного класса *C* где-либо в приведенном ниже списке генерируется ошибка времени компиляции.

- Объявление статического члена *C* (§8.3.1.1, §8.4.3.2, §8.5.1).
- Объявление статического члена объявления любого типа, вложенного в *C*.
- Статический инициализатор *C* (§8.7).
- Статический инициализатор любого объявления класса, вложенного в *C*.

Пример 8.1.2-1. Взаимно рекурсивные границы переменных типа

```
interface ConvertibleTo<T> {
    T convert();
}
class ReprChange<T extends ConvertibleTo<S>,
                 S extends ConvertibleTo<T>> {
    T t;
    void set(S s) { t = s.convert(); }
    S get()       { return t.convert(); }
}
```

Пример 8.1.2-2. Вложенные обобщенные классы

```
class Seq<T> {
    T head;
    Seq<T> tail;

    Seq() { this(null, null); }
    Seq(T head, Seq<T> tail) {
        this.head = head;
        this.tail = tail;
    }
    boolean isEmpty() { return tail == null; }

    class Zipper<S> {
        Seq<Pair<T,S>> zip(Seq<S> that) {
            if (isEmpty() || that.isEmpty()) {
                return new Seq<Pair<T,S>>();
            } else {
                Seq<T>.Zipper<S> tailZipper =
                    tail.new Zipper<S>();
                return new Seq<Pair<T,S>>(
                    new Pair<T,S>(head, that.head),
                    tailZipper.zip(that.tail));
            }
        }
    }
}
```

```
class Pair<T, S> {
    T fst; S snd;
    Pair(T f, S s) { fst = f; snd = s; }
}
class Test {
    public static void main(String[] args) {
        Seq<String> strs =
            new Seq<String>(
                "a",
                new Seq<String>("b",
                                new Seq<String>()));
        Seq<Number> nums =
            new Seq<Number>(
                new Integer(1),
                new Seq<Number>(new Double(1.5),
                                new Seq<Number>()));
        Seq<String>.Zipper<Number> zipper =
            strs.new Zipper<Number>();
        Seq<Pair<String,Number>> combined =
            zipper.zip(nums);
    }
}
```

§8.1.3. Внутренние классы и охватывающие экземпляры

Внутренний класс является вложенным классом, который (явно или неявно) не объявлен как `static`.

Внутренний класс может быть нестатическим членом-классом (§8.5), локальным (§14.3) или анонимным (§15.9.5) классом.

Внутренние классы не могут объявлять статические инициализаторы (§8.7), иначе генерируется ошибка времени компиляции.

Внутренние классы не могут объявлять явные или неявные статические члены, если только они не являются константными переменными (§4.12.4), иначе генерируется ошибка времени компиляции.

Внутренние классы могут наследовать статические члены, которые не являются константными переменными, даже несмотря на то, что они не могут их объявлять.

Вложенные классы, которые не являются внутренними классами, могут свободно объявлять статические члены в соответствии с обычными правилами языка программирования Java.

Пример 8.1.3-1. Объявления внутренних классов и статические члены

```
class HasStatic {
    static int j = 100;
}
class Outer {
    class Inner extends HasStatic {
```

```
        static final int x = 3;     // OK:
        // константная переменная
        static int y = 4;           // Ошибка
        // времени компиляции: внутренний класс
    }
    static class NestedButNotInner{
        static int z = 5;           // OK:
        // не внутренний класс
    }
    interface NeverInner {}         // Интерфейсы
    // не являются внутренними классами
}
```

Инструкция или выражение *находится в статическом контексте* тогда и только тогда, когда наиболее глубоко вложенный метод, конструктор, инициализатор экземпляра, статический инициализатор, инициализатор поля или инструкция явного вызова конструктора, охватывающий инструкцию или выражение, является статическим методом, статическим инициализатором, инициализатором статической переменной или инструкцией явного вызова конструктора (§8.8.7.1).

Внутренний класс *C* является *непосредственным внутренним классом класса или интерфейса O*, если *O* является непосредственно лексически охватывающим *C* объявлением типа и если объявление *C* не находится в статическом контексте.

Внутренний класс *C* является *внутренним классом класса или интерфейса O*, если он является либо непосредственным внутренним классом *O*, либо внутренним классом внутреннего для *O* класса.

> Возможно, хотя и необычно, что непосредственно охватывающим объявлением типа внутреннего класса является интерфейс. Это происходит, только если класс объявлен в теле метода по умолчанию (§9.4). В частности, это происходит, если анонимный или локальный класс объявлен в теле метода по умолчанию или класс-член объявлен в теле анонимного класса, который объявлен в теле метода по умолчанию.

Класс или интерфейс *O* является *нулевым лексически охватывающим типом для себя самого.*

Класс *O* является *n-м лексически охватывающим типом для класса C*, если его непосредственно охватывающий класс является -м лексически охватывающим типом для класса *C*.

Экземпляр `i` непосредственного внутреннего класса *C* класса или интерфейса *O* связан с экземпляром *O*, известным как *непосредственно охватывающий экземпляр* для экземпляра `i`. Непосредственно охватывающий экземпляр объекта, если таковой существует, определяется при создании объекта (§15.9.2).

Объект `o` является *нулевым лексически охватывающим экземпляром для себя самого.*

Объект `o` является *n-м лексически охватывающим экземпляром для экземпляра* `i`, если его непосредственно охватывающий экземпляр является $n-1$-м лексически охватывающим экземпляром для экземпляра `i`.

Экземпляр внутреннего класса `I`, объявление которого находится в статическом контексте, не имеет лексически охватывающих экземпляров. Однако, если `I` объявлен непосредственно внутри статического метода или статического инициализатора, то `I` имеет *охватывающий блок*, который представляет собой наиболее глубоко вложенную блочную инструкцию, лексически охватывающую объявление `I`.

Для каждого суперкласса `S` класса `C`, который сам является непосредственным внутренним классом для класса или интерфейса `SO`, существует экземпляр `SO`, связанный с экземпляром `i` класса `C`, известный как *непосредственно охватывающий экземпляр для экземпляра `i` по отношению к `S`*. Непосредственно охватывающий экземпляр объекта (если таковой существует) по отношению к непосредственному суперклассу его класса определяется при вызове конструктора суперкласса посредством инструкции явного вызова конструктора (§8.8.7.1).

Когда внутренний класс (объявление которого не находится в статическом контексте) обращается к переменной экземпляра, которая является членом лексически охватывающего объявления типа, используется переменная соответствующего лексически охватывающего экземпляра.

Любая локальная переменная, формальный параметр или параметр исключения, используемый внутренним классом, но не объявленный в нем, должен быть объявлен как `final` или быть фактически финальным (§4.12.4), иначе при попытке его использования генерируется ошибка времени компиляции.

Любая локальная переменная, используемая внутренним классом, но не объявленная в нем, должна быть определенно присвоена (§16) до использования в теле внутреннего класса, иначе генерируется ошибка времени компиляции.

Пустое `final`-поле (§4.12.4) лексически охватывающего объявления типа не может быть связано с внутренним классом, иначе генерируется ошибка времени компиляции.

Пример 8.1.3-2. Объявления внутренних классов

```
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i; // Ошибка времени компиляции
            int m = l; // OK
        }
    }
    void foo() {
        class Local {  // Локальный класс
            int j = i;
        }
    }
}
```

Объявление класса `LocalInStaticContext` находится в статическом контексте, так как оно располагается внутри статического метода `classMethod`. Переменные экземпляра класса `Outer` недоступны в теле статического метода.

В частности, переменные экземпляра `Outer` недоступны внутри тела `LocalInStaticContext`. Тем не менее локальные переменные из охватывающего метода могут быть переданы без ошибки (если они объявлены как `final`).

Внутренние классы, объявления которых не находятся в статическом контексте, могут свободно обращаться к переменным экземпляров охватывающего их объявления типа. Переменная экземпляра всегда определяется по отношению к экземпляру. В случае переменных экземпляра охватывающего объявления типа переменная экземпляра должна быть определена по отношению к охватывающему экземпляру этого объявленного типа. Например, класс `Local` выше имеет охватывающий экземпляр класса `Outer`. Вот еще один пример.

```
class WithDeepNesting {
    boolean toBe;
    WithDeepNesting(boolean b) { toBe = b; }

    class Nested {
        boolean theQuestion;
        class DeeplyNested {
            DeeplyNested(){
                theQuestion = toBe || !toBe;
            }
        }
    }
}
```

Здесь каждый экземпляр `WithDeepNesting.Nested.DeeplyNested` имеет охватывающий экземпляр класса `WithDeepNesting.Nested` (непосредственно охватывающий экземпляр) и охватывающий экземпляр класса `WithDeepNesting` (его второй лексически охватывающий экземпляр).

§8.1.4. Суперклассы и подклассы

Необязательная инструкция `extends` в объявлении обычного класса определяет *непосредственный суперкласс* текущего класса.

Superclass:
    `extends` *ClassType*

Конструкция `extends` не должна появляться в определении класса `Object`, иначе генерируется ошибка времени компиляции, связанная с тем, что это изначальный класс, непосредственного суперкласса не имеющий.

ClassType должен именовать доступный (§6.6) тип класса, иначе генерируется ошибка времени компиляции.

Если определенный *ClassType* именует класс, являющийся `final` (§8.1.1.2), генерируется ошибка времени компиляции, поскольку классы, объявленные как `final`, не могут быть суперклассами.

Если *ClassType* именует класс `Enum` или его производные (§8.9), генерируется ошибка времени компиляции.

Если *ClassType* имеет аргументы типа, он должен обозначать правильно сформированный параметризованный тип (§4.5), и никакой из аргументов типа не может быть аргументом типа с символами подстановки, иначе генерируется ошибка времени компиляции.

Для заданного объявления (возможно, обобщенного) класса $C<F_1,\ldots,F_n>$ ($n \geq 0$, $C \neq$ `Object`) *непосредственным суперклассом* типа класса $C<F_1,\ldots,F_n>$ является тип, указанный в инструкции `extends` объявления C, если таковая инструкция `extends` есть в наличии, или `Object` в противном случае.

Для заданного объявления (возможно, обобщенного) класса $C<F_1,\ldots,F_n>$ ($n > 0$) *непосредственным суперклассом* типа параметризованного класса $C<T_1,\ldots,T_n>$, где T_i ($1 \leq i \leq n$) представляет собой тип, является $D<U_1\ \theta,\ldots,U_k\ \theta>$, где $D<U_1,\ldots,U_k>$ — непосредственный суперкласс класса $C<F_1,\ldots,F_n>$, а θ представляет собой подстановку $[F_1:=T_1,\ldots,F_n:=T_n]$.

Класс является *непосредственным подклассом* своего непосредственного суперкласса. Непосредственный суперкласс представляет собой класс, из реализации которого получается реализация текущего класса.

Отношение *подкласса* является транзитивным замыканием отношения непосредственного подкласса. Класс A является подклассом класса C, если истинно одно из следующих высказываний.

- A является непосредственным подклассом класса C.
- Существует класс B, такой, что A является подклассом класса B, а B подклассом класса C, с рекурсивным применением этого определения.

Класс C называется *суперклассом* класса A, когда A является подклассом класса C.

Пример 8.1.4-1. Непосредственные суперклассы и подклассы

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; } // Ошибка
```

Этот пример характеризуется следующими взаимоотношениями.

- Класс `Point` является непосредственным подклассом класса `Object`.
- Класс `Object` является непосредственным суперклассом класса `Point`.
- Класс `ColoredPoint` является непосредственным подклассом класса `Point`.
- Класс `Point` является непосредственным суперклассом класса `ColoredPoint`.

Объявление класса `Colored3dPoint` приводит к генерации ошибки времени компиляции, поскольку оно пытается расширить класс `ColoredPoint`, объявленный как `final`.

Пример 8.1.4-2. Суперклассы и подклассы

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

Этот пример характеризуется следующими взаимоотношениями.

- Класс `Point` является суперклассом класса `ColoredPoint`.

- Класс `Point` является суперклассом класса `Colored3dPoint`.
- Класс `ColoredPoint` является подклассом класса `Point`.
- Класс `ColoredPoint` является суперклассом класса `Colored3dPoint`.
- Класс `Colored3dPoint` является подклассом класса `ColoredPoint`.
- Класс `Colored3dPoint` является подклассом класса `Point`.

Класс *C непосредственно зависит* от типа *T*, если *T* упоминается в инструкции `extends` или `implements` класса *C* как суперкласс, суперинтерфейс или как квалификатор имени суперкласса или суперинтерфейса.

Класс *C зависит* от ссылочного типа *T*, если выполняется любое из следующих условий.

- *C* непосредственно зависит от *T*.
- *C* непосредственно зависит от интерфейса *I*, который зависит (§9.1.3) от *T*.
- *C* непосредственно зависит от класса *D*, который зависит от *T* (с рекурсивным применением этого определения).

Если класс зависит от себя самого, генерируется ошибка времени компиляции.

Если цикличность в объявлении класса обнаруживается во время выполнения при загрузке классов (§12.2.1), генерируется исключение `ClassCircularityError`.

Пример 8.1.4-3. Класс, зависящий от самого себя

```
class Point extends ColoredPoint { int x, y; }
class ColoredPoint extends Point { int color; }
```

Эта программа приводит к генерации ошибки времени компиляции.

§8.1.5. Суперинтерфейсы

Необязательная конструкция `implements` в объявлении класса перечисляет имена интерфейсов, которые являются непосредственными суперинтерфейсами объявляемого класса.

Superinterfaces:
    `implements` *InterfaceTypeList*

InterfaceTypeList:
    InterfaceType {`,` *InterfaceType*}

Каждый *InterfaceType* должен именовать доступный (§6.6) тип интерфейса, иначе генерируется ошибка времени компиляции.

Если *InterfaceType* имеет аргументы типов, он должен означать корректно сформированный параметризованный тип (§4.5), и ни один из аргументов типа не может быть символом подстановки, иначе генерируется ошибка времени компиляции.

Если один и тот же интерфейс упоминается как непосредственный суперинтерфейс два или большее число раз в одной инструкции `implements`, генерируется ошибка времени компиляции. Это происходит, даже если интерфейс назван разными способами.

Пример 8.1.5-1. Некорректные суперинтерфейсы

```
class Redundant implements java.lang.Cloneable, Cloneable {
    int x;
}
```

Данная программа генерирует ошибку времени компиляции, поскольку имена `java.lang.Cloneable` и `Cloneable` ссылаются на один и тот же интерфейс.

Для заданного объявления (возможно, обобщенного) класса $C<F_1,\ldots,F_n>$ ($n \geq 0$, $C \neq$ `Object`) *непосредственными суперинтерфейсами* типа класса $C<F_1,\ldots,F_n>$ являются типы, указанные в инструкции `implements` объявления C, если таковая инструкция `implements` имеется в наличии.

Для заданного объявления обобщенного класса $C<F_1,\ldots,F_n>$ ($n > 0$) *непосредственными суперинтерфейсами* типа параметризованного класса $C<T_1,\ldots,T_n>$, где T_i ($1 \leq i \leq n$) представляет собой тип, являются все типы $I<U_1\ \theta,\ldots,U_k\ \theta>$, где $I<U_1,\ldots,U_k>$ — непосредственный суперинтерфейс класса $C<F_1,\ldots,F_n>$, а θ представляет собой подстановку $[F_1:=T_1,\ldots,F_n:=T_n]$.

Тип интерфейса I является *суперинтерфейсом* типа класса C, если выполняется любое из приведенных условий.

- I является непосредственным суперинтерфейсом C.
- C имеет некоторый непосредственный суперинтерфейс J, для которого I является суперинтерфейсом, с использованием определения "суперинтерфейс интерфейса", данного в §9.1.3.
- I является суперинтерфейсом непосредственного суперкласса C.

Класс может иметь суперинтерфейс более чем одним способом.

Говорят, что класс *реализует* все свои суперинтерфейсы.

Класс не может быть одновременно подтипом двух типов интерфейсов, которые являются различными параметризациями одного и того же обобщенного интерфейса (§9.1.2), или подтипом параметризации обобщенного интерфейса и несформированного типа, именующего тот же обобщенный интерфейс, иначе генерируется ошибка времени компиляции.

Это требование введено для поддержки трансляции затиранием типа (§4.6).

Пример 8.1.5-2. Суперинтерфейсы

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
enum Finish { MATTE, GLOSSY }
interface Paintable extends Colorable {
    void setFinish(Finish finish);
    Finish getFinish();
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
```

```
    int color;
    public void setColor(int color) { this.color = color; }
    public int getColor() { return color; }
}
class PaintedPoint extends ColoredPoint implements Paintable {
    Finish finish;
    public void setFinish(Finish finish) {
        this.finish = finish;
    }
    public Finish getFinish() { return finish; }
}
```

Этот пример характеризуется следующими взаимоотношениями.

- Интерфейс `Paintable` является суперинтерфейсом класса `PaintedPoint`.
- Интерфейс `Colorable` является суперинтерфейсом класса `ColoredPoint` и класса `PaintedPoint`.
- Интерфейс `Paintable` является суперинтерфейсом интерфейса `Colorable`, а `Colorable` является суперинтерфейсом интерфейса `Paintable`, как определено в §9.1.3.

Класс `PaintedPoint` имеет в качестве суперинтерфейса интерфейс `Colorable` и как суперинтерфейс класса `ColoredPoint`, и как суперинтерфейс интерфейса `Paintable`.

Пример 8.1.5-3. Некорректное множественное наследование интерфейса

```
interface I<T> {}
class B implements I<Integer> {}
class C extends B implements I<String> {}
```

Класс `C` приводит к ошибке времени компиляции, поскольку он пытается быть подтипом как *I*`<Integer>`, так и *I*`<String>`.

Если только класс не объявлен как `abstract`, все абстрактные методы-члены каждого непосредственного суперинтерфейса должны быть реализованы (§8.4.8.1) либо путем объявления в этом классе, либо существующим объявлением метода, унаследованным от непосредственного суперкласса или суперинтерфейса, поскольку не может иметь абстрактные методы класс, не являющийся абстрактным (§8.1.1.1).

Каждый метод по умолчанию (§9.4.3) суперинтерфейса класса может быть необязательно перекрыт методом класса; если это не так, метод по умолчанию обычно наследуется и его поведение таково, как указано в его теле по умолчанию.

Одно объявление метода в классе может реализовывать методы более чем одного суперинтерфейса.

Пример 8.1.5-4. Реализация методов суперинтерфейса

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
class Point { int x, y; };
```

```
class ColoredPoint extends Point implements Colorable {
    int color;
}
```

При компиляции этой программы генерируется ошибка времени компиляции, поскольку `ColoredPoint` не является классом, объявленным как `abstract`, и при этом в нем отсутствуют объявления методов `setColor` и `getColor` интерфейса `Colorable`.

```
interface Fish  { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
    public int getNumberOfScales() { return 91; }
}
```

В этом коде метод `getNumberOfScales` класса `Tuna` имеет имя, сигнатуру и возвращаемый тип, соответствующие как методу, объявленному в интерфейсе `Fish`, так и методу, объявленному в интерфейсе `Piano`; таким образом, этот метод реализует оба интерфейса.

```
interface Fish       { int    getNumberOfScales(); }
interface StringBass { double getNumberOfScales(); }
class Bass implements Fish, StringBass {
    // Это объявление не может быть корректным,
    // независимо от используемого типа
    public ?? getNumberOfScales() { return 91; }
}
```

С другой стороны, в приведенном выше коде невозможно объявить метод с именем `getNumberOfScales`, сигнатура и возвращаемый тип которого совместимы с таковыми у обоих методов, объявленных в интерфейсах `Fish` и `StringBass`, поскольку класс не может иметь несколько методов с одинаковой сигнатурой и разными примитивными возвращаемыми типами (§8.4). Следовательно, один класс не в состоянии реализовать одновременно и интерфейс `Fish`, и интерфейс `StringBass` (§8.4.8).

§8.1.6. Тело класса и объявления членов

Тело класса может содержать объявления членов класса, т.е. полей (§8.3), методов (§8.4), классов (§8.5) и интерфейсов (§8.5).

Тело класса может также содержать инициализаторы экземпляров (§8.6), статические инициализаторы (§8.7) и объявления конструкторов (§8.8) класса.

ClassBody:
    { *{ClassBodyDeclaration}* }

ClassBodyDeclaration:
    ClassMemberDeclaration
    InstanceInitializer
    StaticInitializer
    ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

Область видимости и затенение объявления члена *m*, объявленного в типе класса *C* или унаследованного им, определяются в §6.3 и §6.4.

> Если *C* сам является вложенным классом, в охватывающих областях видимости могут существовать определения того же вида (переменная, метод или тип) и с тем же именем, что и *m*. (Области видимости могут быть блоками, классами или пакетами.) Во всех таких случаях член *m*, объявленный в классе *C* или им унаследованный, затеняет (§6.4.1) другие определения того же вида и с тем же именем.

§8.2. Члены классов

Члены типа класса представляют собой следующее.

- Члены, унаследованные из непосредственных суперклассов (§8.1.4), за исключением класса `Object`, не имеющего непосредственного суперкласса.
- Члены, унаследованные из любых непосредственных суперинтерфейсов (§8.1.5).
- Члены, объявленные в теле класса (§8.1.6).

Члены класса, объявленные как `private`, не наследуются подклассами этого класса.

Подклассами, объявленными в пакете, отличном от того, в котором объявлен сам класс, наследуются только члены класса, объявленные как `protected` или `public`.

Конструкторы, статические инициализаторы и инициализаторы экземпляров не являются членами и, следовательно, не наследуются.

Мы используем фразу *тип члена* для обозначения

- в случае поля — его типа;
- в случае метода — упорядоченной четверки, состоящей из
  - ✦ параметров типа: объявлений любых параметров типа метода-члена;
  - ✦ типов аргументов: списка типов аргументов, передаваемых методу-члену;
  - ✦ возвращаемого типа: типа, возвращаемого методом-членом;
  - ✦ конструкции `throws`: типов исключений, объявленных в инструкции `throws` метода-члена.

Поля, методы и типы-члены класса могут иметь одно и то же имя, поскольку используются в разных контекстах и неоднозначность разрешается с помощью разных процедур поиска (§6.5). Однако такое именование не приветствуется с точки зрения стиля.

Пример 8.2-1. Использование членов классов

```
class Point {
    int x, y;
    private Point() { reset(); }
    Point(int x, int y) { this.x = x; this.y = y; }
    private void reset() { this.x = 0; this.y = 0; }
}
class ColoredPoint extends Point {
    int color;
    void clear() { reset(); }                          // Ошибка
}
class Test {
    public static void main(String[] args) {
        ColoredPoint c = new ColoredPoint(0, 0); // Ошибка
        c.reset();                                         // Ошибка
    }
}
```

Эта программа приводит к четырем ошибкам времени компиляции.

Одна ошибка связана с тем, что `ColoredPoint` не имеет конструктора, объявленного с двумя параметрами типа `int`, требуемого в методе `main`. Это иллюстрирует тот факт, что `ColoredPoint` не наследует конструктор своего суперкласса `Point`.

Еще одна ошибка возникает из-за того, что класс `ColoredPoint` не объявляет конструкторы, а потому для него неявно объявляется конструктор по умолчанию (§8.8.9), и этот конструктор по умолчанию эквивалентен записи

```
ColoredPoint() {super();}
```

в которой вызывается конструктор без аргументов непосредственного суперкласса класса `ColoredPoint`. Ошибка заключается в том, что конструктор `Point`, который не принимает аргументов, объявлен как `private`, а потому недоступен за пределами класса `Point` даже путем вызова конструктора суперкласса (§8.8.7).

Еще две ошибки связаны с тем, что метод `reset` класса `Point` является закрытым (`private`), а потому не наследуется классом `ColoredPoint`. Таким образом, вызовы этого метода в методе `clear` класса `ColoredPoint` и в методе `main` класса `Test` некорректны.

Пример 8.2-2. Наследование членов класса с доступом пакета

Рассмотрим пример, в котором пакет `points` объявляет два модуля компиляции:

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
```

и

```
package points;
public class Point3d extends Point {
```

```
    int z;
    public void move(int dx, int dy, int dz) {
        x += dx; y += dy; z += dz;
    }
}
```

А третий модуль компиляции, в другом пакете, имеет вид

```
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        x += dx; y += dy; z += dz; w += dw;   // Ошибка
    }
}
```

Оба класса в пакете `points` компилируются без ошибок. Класс `Point3d` наследует поля `x` и `y` класса `Point`, потому что он находится в том же пакете, что и класс `Point`. Класс `Point4d`, который находится в другом пакете, не наследует поля `x` и `y` класса `Point` или поле `z` класса `Point3d`, а потому генерируется ошибка времени компиляции.

Третий модуль компиляции следовало бы записать иначе:

```
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
```

используя метод `move` суперкласса `Point3d` для работы с `dx`, `dy` и `dz`. Если записать класс `Point4d` таким способом, он будет компилироваться без ошибок.

Пример 8.2-3. Наследование членов класса, объявленных как `public` и `protected`

Пусть имеется класс `Point`

```
package points;
public class Point {
    public int x, y;
    protected int useCount = 0;
    static protected int totalUseCount = 0;
    public void move(int dx, int dy) {
        x += dx; y += dy; useCount++; totalUseCount++;
    }
}
```

Объявленные как `public` и `protected` поля `x`, `y`, `useCount` и `totalUseCount` наследуются всеми подклассами класса `Point`.

Следовательно, приведенная ниже тестовая программа из другого пакета будет успешно скомпилирована.

```
class Test extends points.Point {
    public void moveBack(int dx, int dy) {
        x -= dx; y -= dy; useCount++; totalUseCount++;
    }
}
```

Пример 8.2-4. Наследование членов класса, объявленных как `private`

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
    void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz; totalMoves++; // Ошибка
    }
}
```

Здесь переменная класса `totalMoves` может использоваться только в пределах класса `Point`; она не наследуется подклассом `Point3d`. Ошибка времени компиляции возникает, когда метод `move` класса `Point3d` пытается увеличить значение `totalMoves`.

Пример 8.2-5. Доступ к членам недоступных классов

Несмотря на то что класс может не быть объявлен как открытый (`public`), экземпляры этого класса могут быть доступны во время выполнения для кода за пределами пакета, в котором он объявлен, с помощью открытого суперкласса или суперинтерфейса. Экземпляр класса может быть присвоен переменной такого `public`-типа. Вызов открытого метода объекта, обращение к которому осуществляется описанным выше способом, может вызывать метод класса, если он реализует или перекрывает метод открытого суперкласса или суперинтерфейса. (В этой ситуации метод обязательно должен быть объявлен как `public`, несмотря на то что он объявлен в классе, который сам по себе `public` не является.)

Рассмотрим модуль компиляции

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}
```

и другой модуль компиляции другого пакета:

```
package morePoints;
class Point3d extends points.Point {
    public int z;
    public void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz;
    }
    public void move(int dx, int dy) {
        move(dx, dy, 0);
    }
}
public class OnePoint {
    public static points.Point getOne() {
        return new Point3d();
    }
}
```

Вызов `morePoints.OnePoint.getOne()` в некотором третьем пакете вернет `Point3d`, который может использоваться как `Point`, несмотря на то что тип `Point3d` недоступен за пределами пакета `morePoints`. После этого для данного объекта может быть вызвана версия метода `move` с двумя аргументами. Это возможно, поскольку метод `move` класса `Point3d` объявлен как `public` (как и должно быть: любой метод, который перекрывает `public`-метод, сам должен быть объявлен как `public`, так что такие ситуации, как рассматриваемая, вполне корректно работают). Поля `x` и `y` этого объекта также могут быть доступны из третьего пакета.

Хотя поле `z` класса `Point3d` объявлено как `public`, имея только ссылку на экземпляр класса `Point3d` в переменной `p` типа `Point`, доступ к этому полю из кода вне пакета `morePoints` получить невозможно. Дело в том, что выражение `p.z` некорректное, так как `p` имеет тип `Point`, а класс `Point` не имеет поля с именем `z`; не является корректным и выражение `((Point3d)p).z`, потому что к типу класса `Point3d` нельзя обращаться извне пакета `morePoints`.

Однако объявление поля `z` как `public` не является бесполезным. Если в пакете `morePoints` имеется открытый подкласс `Point4d` класса `Point3d`

```
package morePoints;
public class Point4d extends Point3d {
    public int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
```

то класс `Point4d` наследует поле `z`, которое, будучи открытым, допускает обращение к нему кода из пакетов, отличных от пакета `morePoints` через переменные и выражения `public`-типа `Point4d`.

§8.3. Объявления полей

Переменные типа класса вводятся с помощью *объявлений полей.*

FieldDeclaration:
{FieldModifier} UnannType VariableDeclaratorList ;

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

VariableInitializer:
Expression
ArrayInitializer

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
`boolean`

UnannReferenceType:
UnannClassOrInterfaceType
UnannTypeVariable
UnannArrayType

UnannClassOrInterfaceType:
UnannClassType
UnannInterfaceType

UnannClassType:
Identifier [TypeArguments]
UnannClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

UnannInterfaceType:
UnannClassType

UnannTypeVariable:
Identifier

UnannArrayType:
UnannPrimitiveType Dims
UnannClassOrInterfaceType Dims
UnannTypeVariable Dims

Для удобства далее приведена продукция из §4.3.

Dims:
{Annotation} [ ] *{{Annotation}* [ ]*}*

FieldModifiers описан в §8.3.1.

Identifier в *FieldDeclarator* может использоваться в имени для обращения к полю.

В одном объявлении поля может быть объявлено несколько полей с помощью более чем одного декларатора; *FieldModifiers* и *UnannType* применимы ко всем декларациям в объявлении.

Объявленный тип поля описывается частью *UnannType* в объявлении поля, за которой следует любое количество пар квадратных скобок, за которыми следует *Identifier* в декларации.

Если тело объявления класса объявляет два поля с одним и тем же именем, генерируется ошибка времени компиляции.

Область видимости и затенение объявления поля описываются в §6.3 и §6.4.

Если класс объявляет поле с некоторым именем, то объявление этого поля *скрывает* (hide) все доступные объявления полей с тем же именем в суперклассах и суперинтерфейсах класса.

> В этом отношении сокрытие полей отличается от сокрытия методов (§8.4.8.3), поскольку нет разницы при выборе между статическими и нестатическими полями в сокрытии полей, в то время как в случае сокрытия методов статические и нестатические методы различны.

Обратиться к скрытому полю можно с помощью квалифицированного имени (§6.5.6.2), если оно является статическим, или с помощью выражения доступа к полю, содержащего ключевое слово `super` (§15.11.2) или приведение к типу суперкласса.

> В этом отношении сокрытие полей подобно сокрытию методов.

Если объявление поля скрывает объявление другого поля, эти два поля не должны иметь один и тот же тип.

Класс наследует из своего непосредственного суперкласса и непосредственных суперинтерфейсов все не закрытые поля суперкласса и суперинтерфейсов, которые доступны для кода класса и не скрыты объявлением в классе.

Поле суперкласса, объявленное как `private`, может быть доступно для подкласса, например если оба класса являются членами одного и того же класса. Тем не менее `private`-поле никогда не наследуется подклассом.

Класс может наследовать больше одного поля с одним и тем же именем. Сама по себе эта ситуация не вызывает генерацию ошибки времени компиляции. Однако любая попытка сослаться в теле класса на такое поле по его простому имени приведет к ошибке времени компиляции, потому что такое обращение является неоднозначным.

Возможны несколько путей, которыми одно и то же объявление поля может быть унаследовано от интерфейса. В такой ситуации поле рассматривается унаследованным только один раз, и к нему можно обращаться по простому имени без какой-либо неоднозначности.

Значение, хранящееся в поле типа `float`, всегда является элементом набора значений float (§4.2.3); аналогично значение, хранящееся в поле типа `double`, всегда является элементом набора значений double. Поле типа `float` не может содержать элемент набора значений float с расширенным показателем степени, который не является одновременно элементом набора значений float, как и поле типа `double` не может содержать элемент набора значений double с расширенным показателем степени, который не является одновременно элементом набора значений double.

Пример 8.3-1. Множественное наследование полей

Класс может наследовать два и более полей с одним и тем же именем либо из двух интерфейсов, либо из суперкласса и интерфейса. Попытка обращения к любому неоднозначно унаследованному полю по его простому имени ведет к генерации ошибки времени компиляции. Для беспроблемного доступа к такому полю можно использовать квалифицированное имя или выражение доступа к полю, которое содержит ключевое слово `super` (§15.11.2).

```
interface Frob  { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() { System.out.println(v); }
}
```

В приведенной программе класс `Test` наследует два поля с именем `v`: одно — из суперкласса `SuperTest`, а другое — из его суперинтерфейса `Frob`. Само по себе это разрешено; ошибка времени компиляции связана с использованием простого имени `v` в методе `printV`: метод не может определить, какое именно `v` имеется в виду.

В приведенном ниже варианте исходного текста для обращения к полю `v`, объявленному в классе `SuperTest`, использовано выражение доступа к полю `super.v`, а для обращения к полю `v`, объявленному в интерфейсе `Frob`, использовано выражение доступа к полю `Frob.v`.

```
interface Frob  { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() {
        System.out.println((super.v + Frob.v)/2);
    }
}
```

Эта программа успешно компилируется и выводит

```
2.5
```

Даже если два различных унаследованных поля имеют один и тот же тип, одно и то же значение и оба объявлены как `final`, любое обращение к любому из этих полей по его простому имени рассматривается как неоднозначное и приводит к генерации ошибки времени компиляции.

```
interface Color         { int RED=0, GREEN=1, BLUE=2; }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN); // Ошибка времени компиляции
        System.out.println(RED);   // Ошибка времени компиляции
    }
}
```

В приведенной выше программе не вызывает удивления тот факт, что обращение к `GREEN` рассматривается как неоднозначное, поскольку класс `Test` наследует два различных объявления `GREEN` с разными значениями. Цель данного примера — показать, что обращение к `RED` также рассматривается как неоднозначное, поскольку наследуются два разных объявления. Тот факт, что два поля с именем `RED` имеют один и тот же тип и одно и то же неизменное значение, никак не влияет на вывод о неоднозначности.

Пример 8.3-2. Повторное наследование полей

Если одно и то же объявление поля наследуется от интерфейса разными путями, поле рассматривается как наследованное только один раз. К нему можно обращаться по простому имени без неоднозначности. Например, рассмотрим следующий код.

```
interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {}
class PaintedPoint extends ColoredPoint implements Paintable {
    int p = RED;
}
```

Здесь поля `RED`, `GREEN` и `BLUE` наследуются классом `PaintedPoint` как от непосредственного суперкласса `ColoredPoint`, так и через непосредственный суперинтерфейс `Paintable`. Простые имена `RED`, `GREEN` и `BLUE` могут, тем не менее, использоваться без неоднозначности в классе `PaintedPoint` для обращения к полям, объявленным в интерфейсе `Colorable`.

§8.3.1. Модификаторы полей

FieldModifier: одно из
Annotation `public protected private`
`static final transient volatile`

Правила для модификаторов аннотаций в объявлениях полей определены в §9.7.4 и §9.7.5.

Если одно и то же ключевое слово появляется в объявлении поля в качестве модификатора более одного раза, генерируется ошибка времени компиляции.

> Если в объявлении поля имеется два или более (различных) модификаторов полей, то, как правило (хотя и не обязательно) они находятся в порядке, согласующемся с показанным выше в продукции для *FieldModifier*.

§8.3.1.1. Статические поля

Если поле объявлено как `static`, существует ровно один экземпляр этого поля, независимо от того, сколько экземпляров (возможно, нуль) класса может быть создано в конечном итоге.

Статическое поле, иногда именуемое переменной класса, создается при инициализации класса (§12.4).

Поле, которое не объявлено как `static` (иногда его называют нестатическим), называется *переменной экземпляра*. При создании нового экземпляра класса (§12.5) для каждой переменной экземпляра, объявленной в этом классе или любом из его суперклассов, создается новая переменная, связанная с этим экземпляром.

Пример 8.3.1.1-1. Статические поля

```
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    static final Point origin = new Point(0, 0);
}
class Test {
    public static void main(String[] args) {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3;
        p.y = 3;
        p.useCount++;
        p.origin.useCount++;
        System.out.println("(" + q.x + "," + q.y + ")");
        System.out.println(q.useCount);
        System.out.println(q.origin == Point.origin);
        System.out.println(q.origin.useCount);
    }
}
```

Вывод программы имеет вид

```
(2,2)
0
true
1
```

показывающий, что изменение полей `x`, `y` и `useCount` объекта `p` не влияет на поля объекта `q`, поскольку эти поля являются переменными экземпляра в различных объектах. В этом примере переменная класса `origin` класса `Point` используется как с помощью имени класса в качестве квалификатора, в `Point.origin`, так и с использованием переменных типа класса в выражениях доступа к полю (§15.11), как в `p.origin` и `q.origin`. Эти два способа обращения к переменной класса `origin` обращаются к одному и тому же объекту, что подтверждается результатом проверки равенства значений (§15.21.3):

```
q.origin==Point.origin
```

Еще одним подтверждением является то, что после увеличения

```
p.origin.useCount++;
```

значение `q.origin.useCount` равно `1`; это объясняется тем, что и `p.origin`, и `q.origin` ссылаются на одну и ту же переменную.

Пример 8.3.1.1-2. Сокрытие переменных класса

```
class Point {
    static int x = 2;
}
class Test extends Point {
    static double x = 4.7;
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
}
```

Вывод этой программы имеет вид

```
4.7 2
```

поскольку объявление `x` в классе `Test` скрывает определение `x` в классе `Point`, так что класс `Test` не наследует поле `x` из своего суперкласса `Point`. В объявлении класса `Test` простое имя `x` ссылается на объявление поля, объявленного в классе `Test`. Код в классе `Test` может ссылаться на поле `x` класса `Point` как на `super.x` (или, поскольку `x` объявлено как `static`, как на `Point.x`). Ниже приведен код, в котором объявление `Test.x` удалено.

```
class Point {
    static int x = 2;
}
class Test extends Point {
    public static void main(String[] args) {
        new Test().printX();
```

```
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
}
```

Здесь поле `x` класса `Point` больше не скрывается в классе `Test`; теперь простое имя `x` ссылается на поле `Point.x`. Код в классе `Test` может обращаться к этому полю как к `super.x`. Таким образом, вывод этой измененной программы имеет вид

```
2 2
```

Пример 8.3.1.1-3. Сокрытие переменных экземпляра

```
class Point {
    int x = 2;
}
class Test extends Point {
    double x = 4.7;
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

Вывод этой программы имеет вид

```
4.7 2
4.7 2
```

поскольку объявление `x` в классе `Test` скрывает определение `x` в классе `Point`, так что класс `Test` не наследует поле `x` от своего суперкласса `Point`. Следует отметить, однако, что хотя поле `x` класса `Point` не наследуется классом `Test`, оно *реализуется* экземплярами класса `Test`. Другими словами, каждый экземпляр класса `Test` содержит два поля, одно — типа `int`, а второе — типа `double`. Оба поля носят имя `x`, но в пределах объявления класса `Test` простое имя `x` всегда относится к полю, объявленному внутри класса `Test`. Код в методах экземпляра класса `Test` может обращаться к переменной экземпляра `x` класса `Point` как к `super.x`.

Код, использующий выражения доступа к полю для обращения к полю `x`, ссылается на поле с именем `x` в классе, указанном типом ссылочного выражения. Таким образом, выражение `sample.x` обращается к значению `double`, переменной экземпляра, объявленной в классе `Test`, поскольку типом переменной `sample` является `Test`; но выражение `((Point)sample).x` обращается к значению `int`, переменной экземпляра, объявленной в классе `Point`, из-за наличия приведения к типу `Point`.

Если объявление `x` удалить из класса `Test`, получим следующую программу.

```
class Point {
    static int x = 2;
}
class Test extends Point {
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

В ней поле `x` класса `Point` больше не скрывается в классе `Test`. В методах экземпляра в объявлении класса `Test` простое имя `x` теперь относится к полю, объявленному внутри класса `Point`. Код в классе `Test` может по-прежнему обращаться к этому полю как к `super.x`. Выражение `sample.x` по-прежнему относится к полю `x` в типе `Test`, но это поле теперь является наследуемым полем, и, таким образом, выражение ссылается на поле `x`, объявленное в классе `Point`.

Вывод этого варианта программы имеет вид

```
2 2
2 2
```

§8.3.1.2. Поля `final`

Поле может быть объявлено как `final` (§4.12.4). Как `final` могут быть объявлены и переменные класса, и переменные экземпляра (статические и нестатические поля).

Пустая переменная класса, объявленная как `final`, должна быть определенно присвоенной статическим инициализатором класса, в котором она объявлена, иначе генерируется ошибка времени компиляции (§8.7, §16.8).

Пустая `final`-переменная экземпляра должна быть определенно присвоенной в конце каждого конструктора класса, в котором она объявлена; в противном случае генерируется ошибка времени компиляции (§8.8, §16.9).

§8.3.1.3. Поля `transient`

Переменные могут быть помечены как `transient`, чтобы указать, что они не являются частью сохраняемого (persistent) состояния объекта.

Пример 8.3.1.3-1. Сохраняемость полей `transient`

```
class Point {
    int x, y;
    transient float rho, theta;
}
```

Если экземпляр класса `Point` сохраняется в постоянном хранилище системной службой, то будут сохранены только поля `x` и `y`. Эта спецификация не определяет детали работы таких служб; в качестве примера службы такого рода можете рассмотреть спецификацию `java.io.Serializable`.

§8.3.1.4. Поля `volatile`

Язык программирования Java позволяет потокам обращаться к совместно используемым переменным (§17.1). Как правило, чтобы обеспечить согласованное и надежное обновление совместно используемых переменных, поток должен гарантировать их исключительное использование путем блокировки, которая по соглашению обеспечивает взаимное исключение для таких переменных.

Язык программирования Java предоставляет второй механизм — поля `volatile`, — который для ряда целей оказывается удобнее блокировки.

Поле может быть объявлено как `volatile`, и в этом случае модель памяти Java гарантирует, что все потоки будут видеть согласованное значение этой переменной (§17.4).

Если переменная `final` одновременно объявлена как `volatile`, генерируется ошибка времени компиляции.

Пример 8.3.1.4-1. Поля `volatile`

```
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

Если в приведенном коде один поток будет постоянно вызывать метод `one` (но всего не более `Integer.MAX_VALUE` раз), а второй поток будет постоянно вызывать метод `two`, то метод `two` иногда может вывести значение `j`, большее, чем значение `i`, поскольку пример не включает синхронизацию, и согласно правилам, описанным в §17.4, совместно используемые переменные `i` и `j` могут обновляться в любом порядке.

Одним из способов предотвращения этого неупорядоченного поведения является объявление методов `one` и `two` как `synchronized` (§8.4.3.6).

```
class Test {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

Тем самым предотвращается одновременное выполнение методов `one` и `two` и, кроме того, гарантируется, что оба совместно используемых значения, `i` и `j`, обновляются до возврата из метода `one`. Поэтому метод `two` никогда не увидит значение `j` б*о*льшим, чем значение `i`. Этот метод всегда будет наблюдать одинаковые значения `i` и `j`.

Другой подход может заключаться в объявлении `i` и `j` как `volatile`.

```
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
```

```
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

Этот код позволяет и методу `one`, и методу `two` выполняться одновременно, но гарантирует, что доступ к совместно используемым значениям `i` и `j` происходит именно столько раз и именно в том порядке, в котором они находятся в исходном тексте каждого потока. Таким образом, совместно используемое значение `j` никогда не будет большим, чем `i`, потому что каждое обновление `i` будет отражено в совместно используемом значении `i` до того, как произойдет обновление `j`. Возможно, однако, что в любом конкретном вызове метода `two` значение `j` окажется больше значения `i`, потому что метод `one` может выполняться неоднократно между моментом, когда метод `two` выполняет выборку значения `i`, и моментом, когда метод `two` выполняет выборку значения `j`.

Дополнительное обсуждение этой темы и примеры вы найдете в §17.4.

§8.3.2. Инициализация полей

Если декларатор в объявлении поля имеет *инициализатор переменной*, то декларатор имеет семантику присваивания (§15.26) значения объявленной переменной.

Если декларатор предназначен для переменной класса (т.е. поля `static`), то к его инициализатору применимы следующие правила.

- Если в инициализаторе осуществляется обращение к любой переменной экземпляра по ее простому имени, генерируется ошибка времени компиляции.
- Если в инициализаторе встречается ключевое слово `this` (§15.8.3) или ключевое слово `super` (§15.11.2, §15.12), генерируется ошибка времени компиляции.
- Во время выполнения инициализатор вычисляется, а присваивание выполняется ровно один раз, при инициализации класса (§12.4.2).

  Заметим, что `static`-поля, которые являются константными переменными (§4.12.4), инициализируются до других `static`-полей (§12.4.2). Это правило применимо также к интерфейсам (§9.3.1). У таких полей никогда не будут наблюдаться их значения по умолчанию (§4.12.5), даже в специальным образом составленных программах.

Если декларатор предназначен для переменной экземпляра (т.е. поля, не являющегося `static`), то к его инициализатору применимы следующие правила.

- Инициализатор может обращаться к любой переменной класса или унаследованной классом по ее простому имени, даже если ее объявление текстуально располагается после инициализатора.
- Инициализатор может обращаться к текущему объекту с помощью ключевого слова `this` (§15.8.3) и использовать ключевое слово `super` (§15.11.2, §15.12).
- Во время выполнения инициализатор вычисляется, а присваивание выполняется всякий раз при создании экземпляра класса (§12.5).

Проверка исключений в инициализаторе переменной в объявлении поля определена в §11.2.3.

Инициализаторы переменных также используются в инструкциях объявлений локальной переменной (§14.4), где вычисление инициализатора и присваивание выполняются каждый раз, когда выполняется инструкция объявления локальной переменной.

Пример 8.3.2-1. Инициализация поля

```
class Point {
    int x = 1, y = 5;
}
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + ", " + p.y);
    }
}
```

Вывод данной программы имеет вид

```
1, 5
```

поскольку присваивание x и y выполняется при создании нового объекта класса `Point`.

Пример 8.3.2-2. Опережающая ссылка на переменную класса

```
class Test {
    float f = j;
    static int j = 1;
}
```

Эта программа компилируется без ошибок; она инициализирует `j` значением `1` при инициализации класса `Test` и инициализирует `f` текущим значением `j` всякий раз при создании экземпляра класса `Test`.

§8.3.3. Опережающие ссылки во время инициализации поля

Использование переменных класса, объявления которых текстуально располагаются после использования, несколько ограничено, несмотря на то, что переменные класса находятся в области видимости (§6.3). В частности, ошибка времени компиляции генерируется, если выполняются все приведенные условия.

- Объявление переменной класса в классе или интерфейсе *C* текстуально располагается после использования переменной класса.
- Использование представляет собой простое имя либо в инициализаторе переменной класса *C*, либо в статическом инициализаторе *C*.
- Использование осуществляется не в левой части присваивания.

- *C* является наиболее глубоко вложенным классом или интерфейсом, охватывающим использование.

Использование переменных экземпляра, объявления которых текстуально располагаются после использования, несколько ограничено, несмотря на то, что переменные экземпляра находятся в области видимости. В частности, ошибка времени компиляции генерируется, если выполняются все приведенные условия.

- Объявление переменной экземпляра в классе или интерфейсе *C* текстуально располагается после использования переменной экземпляра.
- Использование представляет собой простое имя либо в инициализаторе переменной экземпляра *C*, либо в инициализаторе экземпляра *C*.
- Использование осуществляется не в левой части присваивания.
- *C* является наиболее глубоко вложенным классом или интерфейсом, охватывающим использование.

Пример 8.3.3-1. Ограничения, накладываемые на инициализацию поля

```
class Test1 {
    int i = j; // Ошибка времени компиляции:
               // неверная опережающая ссылка
    int j = 1;
}
```

При компиляции приведенной программы генерируется ошибка времени компиляции, в то время как приведенная ниже программа компилируется без ошибок, несмотря на то что конструктор (§8.8) класса `Test2` обращается к полю `k`, объявленному тремя строками ниже.

```
class Test2 {
    Test2() { k = 2; }
    int j = 1;
    int i = j;
    int k;
}
```

Описанное выше ограничение призвано отлавливать во время компиляции циклические или иные неверные инициализации. Таким образом, при компиляции как кода

```
class Z {
    static int i = j + 2;
    static int j = 4;
}
```

так и кода

```
class Z {
    static { i = j + 2; }
    static int i, j;
    static { j = 4; }
}
```

генерируется ошибка времени компиляции. Обращение к методам подобным образом не проверяется, так что вывод кода

```
class Z {
    static int peek() { return j; }
    static int i = peek();
    static int j = 1;
}
class Test {
    public static void main(String[] args) {
        System.out.println(Z.i);
    }
}
```

имеет вид

```
0
```

Это связано с тем, что инициализатор переменной для `i` использует метод класса `peek` для доступа к значению переменной `j` до того, как `j` инициализируется своим инициализатором переменной, и в этой точке она все еще имеет значение по умолчанию (§4.12.5).

Вот более сложный пример.

```
class UseBeforeDeclaration {
    static {
        x = 100;
            // OK - присваивание
        int y = x + 1;
            // Ошибка - чтение до объявления
        int v = x = 3;
            // OK - x в левой части присваивания
        int z = UseBeforeDeclaration.x * 2;
            // OK - доступ не с помощью простого имени

        Object o = new Object() {
            void foo() { x++; }
                // OK - находится в другом классе
            { x++; }
                // OK - находится в другом классе
        };
    }

    {
        j = 200;
            // OK - присваивание
        j = j + 1;
            // Ошибка - чтение правой частью до объявления
        int k = j = j + 1;
            // Ошибка - неверная опережающая ссылка на j
        int n = j = 300;
            // OK - j в левой части присваивания
```

```
int h = j++;
    // Ошибка - чтение до объявления
int l = this.j * 3;
    // OK - доступ не через простое имя

Object o = new Object() {
void foo(){ j++; }
    // OK - находится в другом классе
{ j = j + 1; }
    // OK - находится в другом классе
};
}

int w = x = 3;
    // OK - x в левой части присваивания
int p = x;
    // OK - инициализаторы экземпляра могут
    // обращаться к статическим полям
static int u =
    (new Object() { int bar() { return x; } }).bar();
    // OK - находится в другом классе

static int x;


    int m = j = 4;
        // OK - j в левой части присваивания
    int o =
        (new Object() { int bar() { return j; } }).bar();
        // OK - находится в другом классе
    int j;
}
```

§8.4. Объявления методов

Метод объявляет выполнимый код, который может быть вызван с передачей фиксированного количества значений в качестве аргументов.

MethodDeclaration:
    {MethodModifier} MethodHeader MethodBody

MethodHeader:
    Result MethodDeclarator [Throws]
    TypeParameters {Annotation} Result MethodDeclarator [Throws]

MethodDeclarator:
    Identifier (*[FormalParameterList]*) *[Dims]*

Для удобства далее приведена продукция из §4.3.

Dims:
{Annotation} [ ] *{{Annotation}* [ ]*}*

FormalParameterList описывается в §8.4.1, *MethodModifier* — в §8.4.3, *TypeParameters* — в §8.4.4, *Result* — в §8.4.5, *Throws* — в §8.4.6, а *MethodBody* — в §8.4.7.

Identifier в *MethodDeclarator* может использоваться в имени для ссылки на метод (§6.5.7.1, §15.12).

Если в теле класса объявлены как члены два метода с эквивалентными в смысле перекрытия сигнатурами (§8.4.2), генерируется ошибка времени компиляции.

Область видимости и затенение объявления метода описываются в §6.3 и §6.4.

Для совместимости с более старыми версиями платформы Java SE в объявлении метода, который возвращает массив, разрешено размещать пустые пары квадратных скобок, которые образуют объявление типа массива после списка формальных параметров. Крайне не рекомендуется использовать этот синтаксис в новом коде.

§8.4.1. Формальные параметры

Формальные параметры метода или конструктора, если таковые имеются, указываются в списке разделенных запятыми спецификаторов параметров. Каждый спецификатор параметра состоит из типа (которому могут предшествовать необязательный модификатор `final` и/или одна или несколько аннотаций) и идентификатора (за которым могут следовать необязательные квадратные скобки), который указывает имя параметра.

Если метод или конструктор не имеет формальных параметров, то в его объявлении участвует только пустая пара скобок.

FormalParameterList:
FormalParameters , *LastFormalParameter*
LastFormalParameter

FormalParameters:
FormalParameter {, *FormalParameter}*
ReceiverParameter {, *FormalParameter}*

FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId

VariableModifier: одно из
Annotation `final`

ReceiverParameter:
{Annotation} UnannType [Identifier .] `this`

LastFormalParameter:
{VariableModifier} UnannType {Annotation} `...` *VariableDeclaratorId*
FormalParameter

Приведенные далее продукции взяты из §4.3 и §8.3 для упрощения понимания.

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} [ ] *{{Annotation}* [ ] *}*

Последний формальный параметр метода или конструктора особый: он может быть *параметром переменной арности*, на что указывает многоточие, следующее за типом.

> Обратите внимание, что троеточие (. . .) само по себе является токеном (§3.11). Между ним и типом можно поместить пробельный символ, но это не приветствуется с точки зрения стиля.

Если последний формальный параметр является параметром переменной арности, то и сам метод является *методом с переменным количеством аргументов* (переменной арности). В противном случае это *метод с фиксированным количеством аргументов*.

Параметр-получатель (receiver parameter) представляет собой необязательный синтаксический механизм для метода экземпляра или конструктора внутреннего класса. В случае метода экземпляра параметр-получатель представляет объект, для которого вызывается метод. В случае конструктора внутреннего класса параметр-получатель представляет непосредственно охватывающий экземпляр вновь создаваемого объекта. В любом случае параметр-получатель существует исключительно для того, чтобы позволить типу представленного объекта быть описанным в исходном тексте, так чтобы тип мог быть аннотирован. Параметр-получатель не является формальным параметром; точнее говоря, он не является объявлением ни одной из разновидностей переменных (§4.12.3), никогда не связан с некоторым значением, передаваемым в выражении вызова метода или создания экземпляра класса как аргумент, и ни на что не влияет во время выполнения.

Правила для модификаторов аннотаций в объявлениях формального параметра и параметра-получателя определены в §9.7.4 и §9.7.5.

Если `final` в качестве модификатора в объявлении формального параметра встречается более одного раза, генерируется ошибка времени компиляции.

Применение смешанного обозначения массива (§10.2) для параметра переменной арности ведет к генерации ошибки времени компиляции.

Область видимости и затенение формального параметра описываются в §6.3 и §6.4.

Если метод или конструктор объявляет два формальных параметра с одним и тем же именем (т.е. в их объявлениях используется один и тот же *Identifier*), генерируется ошибка времени компиляции.

Если в теле метода или конструктора выполняется присваивание формальному параметру, объявленному как `final`, генерируется ошибка времени компиляции.

Параметр-получатель может находиться только в части *FormalParameters* метода экземпляра или конструктора внутреннего класса; в противном случае генерируется ошибка времени компиляции.

Там, где допускается параметр-получатель, его имя и тип определяются следующим образом.

- В методе экземпляра тип параметра-получателя должен быть классом или интерфейсом, в котором объявлен метод, а имя параметра-получателя должно быть `this`; в противном случае генерируется ошибка времени компиляции.
- В конструкторе внутреннего класса тип параметра-получателя должен быть классом или интерфейсом, который представляет собой непосредственно охватывающее внутренний класс объявление типа, а имя параметра-получателя должно быть *Identifier* `.this`, где *Identifier* является простым именем класса или интерфейса, который представляет собой непосредственно охватывающее внутренний класс объявление типа; в противном случае генерируется ошибка времени компиляции.

Объявленный тип формального параметра описывает нетерминал *UnannType*, который находится в спецификаторе параметра; за ним идет некоторое количество пар квадратных скобок, а после них — *Identifier*. Исключением является параметр переменной арности, объявленный тип которого — тип массива, тип компонентов которого представляет собой *UnannType* из спецификатора данного параметра.

Если объявленный тип параметра переменной арности имеет элемент типа, недоступного во время выполнения (§4.7), то для объявления метода с переменным числом параметров генерируется предупреждение о непроверенных типах (если только метод не аннотирован с помощью аннотации `@SafeVarargs` (§9.6.4.7) или если предупреждение о непроверенных типах не подавлено аннотацией `@SuppressWarnings` (§9.6.4.5)).

При вызове метода или конструктора (§15.12) перед выполнением его тела вновь создаваемые переменные параметров (объявленного типа) инициализируются значениями выражений фактических аргументов. Для обращения к формальному параметру в теле метода или конструктора в качестве простого имени может использоваться *Identifier*, находящийся в *DeclaratorId*.

Вызовов метода с переменным количеством аргументов может содержать больше выражений фактических аргументов, чем формальных параметров. Все выражения фактических аргументов, соответствующие предшествующим параметру переменной арности формальным параметрам, вычисляются, а результаты вычислений сохраняются в массиве, который передается вызову метода (§15.12.4.2).

Параметр метода или конструктора типа `float` всегда содержит элемент набора значений float (§4.2.3); аналогично параметр метода или конструктора типа `double` всегда содержит элемент набора значений double. Параметр метода или конструктора типа `float` не может содержать элемент набора значений float с расширенным показателем степени, который не является одновременно элементом набора значений float; аналогично параметр метода или конструктора типа `double` не может содержать элемент набора значений double с расширенным показателем степени, который не является одновременно элементом набора значений `double`.

В случае, когда выражение фактического аргумента, соответствующее переменной параметра, не является FP-строгим (§15.4), вычисление этого выражения фактического аргумента может использовать промежуточные значения из соответствующих наборов значений с расширенным показателем степени. Прежде чем быть сохраненным в переменной параметра, результат такого выражения отображается на ближайшее значение из стандартного набора значений с помощью преобразования вызова (§5.3).

Вот несколько примеров параметров-получателей в методах экземпляра и конструкторах внутренних классов.

```
class Test {
    Test(/* ?? ?? */) {}
        // Параметр-получатель не разрешен в конструкторе
        // класса верхнего уровна, так как не имеется
        // никакого мыслимого типа или имени.
    void m(Test this) {}
        // OK: параметр-получатель в методе экземпляра.
    static void n(Test this) {}
        // Неверно: параметр-получатель в статическом методе.
    class A {
        A(Test Test.this) {}
            // OK: параметр-получатель представляет
            // Test, непосредственно охватывающий
            // создаваемый экземпляр A.
        void m(A this) {}
            // OK: параметр-получатель представляет
            // экземпляр A, для которого вызывается A.m().
        class B {
            B(Test.A A.this) {}
                // OK: параметр-получатель представляет
                // экземпляр A, который непосредственно
                // охватывает создаваемый экземпляр B.
                void m(Test.A.B this) {}
                // OK: параметр-получатель представляет
                // экземпляр B, для которого вызывается
                // B.m().
        }
    }
}
```

Конструктор `B` и метод экземпляра показывают, что тип параметра-получателя может быть описан с помощью квалифицированного *TypeName* подобно любому другому типу; но что имя параметра-получателя в конструкторе внутреннего класса должно использовать простое имя охватывающего класса.

§8.4.2. Сигнатура метода

Два метода или конструктора, M и N, имеют *одинаковые сигнатуры*, если они имеют одно и то же имя, одни и те же параметры типов (если таковые имеются) (§8.4.4) и, после применения формальных типов параметров N к параметрам типов M, одинаковые типы формальных параметров.

Сигнатура метода m_1 является *подсигнатурой* сигнатуры метода m_2, если

- либо m_2 имеет ту же сигнатуру, что и m_1,
- либо сигнатура m_1 совпадает с затиранием (§4.6) сигнатуры m_2.

Две сигнатуры методов m_1 и m_2 являются *эквивалентными в смысле перекрытия* (override-equivalent) тогда и только тогда, когда m_1 является подсигнатурой m_2 или m_2 является подсигнатурой m_1.

При объявлении в классе двух методов с эквивалентными в смысле перекрытия сигнатурами генерируется ошибка времени компиляции.

Пример 8.4.2-1. Сигнатуры, эквивалентные в смысле перекрытия

```
class Point {
    int x, y;
    abstract void move(int dx, int dy);
    void move(int dx, int dy) { x += dx; y += dy; }
}
```

Эта программа вызывает ошибку времени компиляции, так как объявляет два метода `move` с одинаковыми (а следовательно, эквивалентными в смысле перекрытия) сигнатурами. Ошибка генерируется, несмотря на то что в одном из объявлений имеется модификатор `abstract`.

Понятие подсигнатуры предназначено для выражения отношения между двумя методами, сигнатуры которых не идентичны, но один метод может перекрывать другой. В частности, это позволяет методу, сигнатура которого не использует обобщенные типы, перекрывать любые обобщенные версии этого метода. Это важно, так как разработчики библиотек могут свободно обобщать методы независимо от клиентов, которые определяют подклассы или подынтерфейсы библиотеки.

Рассмотрим пример.

```
class CollectionConverter {
    List toList(Collection c) {...}
}
class Overrider extends CollectionConverter {
    List toList(Collection c) {...}
}
```

Теперь предположим, что код был написан до добавления обобщенности, и теперь автор класса `CollectionConverter` решил обобщить этот код следующим образом.

```
class CollectionConverter {
    <T> List<T> toList(Collection<T> c) {...}
}
```

Без описанного специального разрешения `Overrider.toList` не перекрывал бы `CollectionConverter.toList`. Вместо этого такой код был бы просто некорректным. Это значительно ограничило бы использование обобщенности, так как создатели библиотек колебались бы при принятии решения об обобщении существующего кода.

§8.4.3. Модификаторы метода

MethodModifier: одно из

Annotation `public protected private abstract`
`static final synchronized native strictfp`

Правила для модификаторов аннотаций в объявлении метода определены в §9.7.4 и §9.7.5.

Если одно и то же ключевое слово встречается в объявлении метода в качестве модификатора больше одного раза, генерируется ошибка времени компиляции.

Если объявление метода, содержащее ключевое слово `abstract`, дополнительно содержит любое из ключевых слов `private`, `static`, `final`, `native`, `strictfp` или `synchronized`, генерируется ошибка времени компиляции.

Если объявление метода, содержащее ключевое слово `native`, дополнительно содержит ключевое слово `strictfp`, генерируется ошибка времени компиляции.

> Если в объявлении метода имеется два или более (различных) модификаторов метода, то, как правило (хотя и не обязательно) они находятся в порядке, согласующемся с показанным выше в продукции для *MethodModifier*.

§8.4.3.1. Абстрактные методы

Объявление метода с модификатором `abstract` вводит метод как член, предоставляя его сигнатуру (§8.4.2), возвращаемый тип (§8.4.5) и раздел `throws` (если таковой имеется) (§8.4.6), но не предоставляет его реализации (§8.4.7). Метод, не являющийся абстрактным, называется конкретным.

Объявление метода *m* как `abstract` должно находиться непосредственно в классе, объявленном как `abstract` (назовем его *A*), если только он не находится в перечислении (§8.9); в противном случае генерируется ошибка времени компиляции.

Каждый подкласс класса *A*, не являющийся `abstract` (§8.1.1.1), должен предоставлять реализацию метода *m*; в противном случае генерируется ошибка времени компиляции.

Абстрактный класс может перекрыть абстрактный метод, предоставляя объявление другого абстрактного метода.

> Таким образом предоставляется место для комментария для документации, для уточнения возвращаемого типа или для объявления того, что множество проверяемых исключений, которые могут генерироваться этим методом при реализации подклассами, является более ограниченным.

Метод экземпляра, не являющийся абстрактным, может быть перекрыт абстрактным методом.

Пример 8.4.3.1-1. Перекрытие абстрактного метода абстрактным методом

```
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
    BufferError() { super(); }
    BufferError(String s) { super(s); }
}
interface Buffer {
    char get() throws BufferEmpty, BufferError;
}
```

```
abstract class InfiniteBuffer implements Buffer {
    public abstract char get() throws BufferError;
}
```

Перекрывающее объявление метода `get` в классе `InfiniteBuffer` указывает, что метод `get` в любом подклассе класса `InfiniteBuffer` никогда не генерирует исключения `BufferEmpty`, предположительно потому что он генерирует данные буфера, и, таким образом, они никогда не могут исчерпаться.

Пример 8.4.3.1-2. Перекрытие абстрактным методом метода, не являющегося абстрактным

Мы можем объявить абстрактный класс `Point`, который требует от своих подклассов реализации метода `toString` для того, чтобы стать завершенными, инстанцируемыми классами.

```
abstract class Point {
    int x, y;
    public abstract String toString();
}
```

Это объявление метода `toString` как `abstract` перекрывает не являющийся абстрактным метод `toString` класса `Object`. (Класс `Object` является неявным непосредственным суперклассом класса `Point`.) Добавим следующий код.

```
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return super.toString() + ": color " + color; // Ошибка
    }
}
```

В результате мы получим ошибку времени компиляции, поскольку вызов `super.toString()` обращается к методу `toString` класса `Point`, который объявлен как `abstract` и, следовательно, вызван быть не может. Метод `toString` класса `Object` может быть сделан доступным классу `ColoredPoint`, только если класс `Point` явно сделает его доступным через некоторый другой метод, как в приведенном ниже коде.

```
abstract class Point {
    int x, y;
    public abstract String toString();
    protected String objString() { return super.toString(); }
}
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return objString() + ": color " + color; // Верно
    }
}
```

§8.4.3.2. Статические методы

Метод, объявленный как `static`, называется *методом класса.*

Использование имени параметра типа любого охватывающего объявления в заголовке или теле метода класса приводит к ошибке времени компиляции.

Метод класса всегда вызывается без ссылки на конкретный объект. При попытке обратиться к текущему объекту с помощью ключевого слова `this` (§15.8.3) или `super` (§15.11.2) генерируется ошибка времени компиляции.

Метод, не объявленный как `static`, называется *методом экземпляра*, а иногда — нестатическим методом.

Метод экземпляра всегда вызывается в связи с некоторым объектом, который становится текущим объектом, доступным при выполнении тела метода с помощью ключевых слов `this` и `super`.

§8.4.3.3. Методы `final`

Метод можно объявить как `final` для предотвращения его перекрытия и сокрытия подклассами.

При попытке перекрытия и сокрытия метода, объявленного как `final`, генерируется ошибка времени компиляции.

Метод, объявленный как `private`, и все методы, объявленные непосредственно в классе, объявленном как `final` (§8.1.1.2), ведут себя так, как если бы они были объявлены как `final`, поскольку невозможно их перекрыть.

> Во время выполнения генератор машинного кода или оптимизатор может "встроить" тело `final`-метода, заменив его вызов кодом его тела. Процесс встраивания должен сохранять семантику вызова метода. В частности, если объект вызова метода экземпляра имеет значение `null`, то должно быть сгенерировано исключение `NullPointerException`, даже если метод оказывается встроенным. Компилятор Java должен гарантировать, что исключение будет сгенерировано в нужной точке, так, чтобы фактические аргументы метода были вычислены в правильном порядке до вызова метода.
>
> Рассмотрим пример.

```
final class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class Test {
    public static void main(String[] args) {
        Point[] p = new Point[100];
        for (int i = 0; i < p.length; i++) {
            p[i] = new Point();
            p[i].move(i, p.length-1-i);
        }
    }
}
```

Встраивание метода `move` класса `Point` в методе `main` приведет цикл `for` к следующему виду.

```
for (int i = 0; i < p.length; i++) {
    p[i] = new Point();
    Point pi = p[i];
    int j = p.length-1-i;
    pi.x += i;
    pi.y += j;
}
```

Затем цикл может быть дополнительно оптимизирован.

Такое встраивание не может быть выполнено во время компиляции, если только нельзя гарантировать, что классы `Test` и `Point` всегда будут перекомпилированы вместе, так что если класс `Point` — и, в частности, его метод `move` — изменится, то будет обновлен и код `Test.main`.

§8.4.3.4. Методы `native`

Метод, объявленный как `native`, реализуется платформо-зависимым кодом, обычно написанным на другом языке программирования, таком как C. Тело метода `native` представлено не блоком (§8.4.7), а точкой с запятой, указывая, что реализация метода опущена.

Например, класс `RandomAccessFile` пакета `java.io` может объявлять следующие `native`-методы.

```
package java.io;
public class RandomAccessFile
    implements DataOutput, DataInput {
    . . .
    public native void open(String name, boolean writeable)
        throws IOException;
    public native int readBytes(byte[] b, int off, int len)
        throws IOException;
    public native void writeBytes(byte[] b, int off, int len)
        throws IOException;
    public native long getFilePointer() throws IOException;
    public native void seek(long pos) throws IOException;
    public native long length() throws IOException;
    public native void close() throws IOException;
}
```

§8.4.3.5. Методы `strictfp`

Влияние модификатора `strictfp` заключается в том, что все выражения типов `float` и `double` в теле метода явно делаются FP-строгими (§15.4).

§8.4.3.6. Методы `synchronized`

Метод, объявленный как `synchronized`, перед своим выполнением захватывает монитор (§17.1).

В случае метода класса (`static`) используемый монитор связывается с объектом `Class` класса метода.

В случае метода экземпляра используемый монитор связывается с `this` (объектом, для которого вызывается этот метод).

Пример 8.4.3.6-1. Мониторы `synchronized`

Инструкция `synchronized` и `synchronized`-метод используют одни и те же мониторы (§14.19). Таким образом, код

```
class Test {
    int count;
    synchronized void bump() {
        count++;
    }
    static int classCount;
    static synchronized void classBump() {
        classCount++;
    }
}
```

выполняется в точности так же, как и код

```
class BumpTest {
    int count;
    void bump() {
        synchronized (this) { count++; }
    }
    static int classCount;
    static void classBump() {
        try {
            synchronized (Class.forName("BumpTest")) {
                classCount++;
            }
        } catch (ClassNotFoundException e) {}
    }
}
```

Пример 8.4.3.6-2. Методы `synchronized`

```
public class Box {
    private Object boxContents;
    public synchronized Object get() {
        Object contents = boxContents;
        boxContents = null;
        return contents;
    }
    public synchronized boolean put(Object contents) {
        if (boxContents != null) return false;
        boxContents = contents;
        return true;
    }
}
```

> В этой программе определен класс, разработанный для параллельного использования. Каждый экземпляр класса `Box` имеет переменную экземпляра `boxContents`, которая может хранить ссылку на любой объект.
>
> Вы можете поместить объект в `Box` с помощью вызова `put`, который возвращает `false`, если объект класса `Box` уже заполнен. Вы можете получить нечто из `Box` с помощью вызова `get`, который вернет ссылку `null`, если объект класса `Box` пуст.
>
> Если бы методы `put` и `get` не были описаны как `synchronized` и два потока выполняли эти методы для одного и того же экземпляра `Box` одновременно, код мог бы вести себя некорректно. Он мог бы, например, просто потерять объект из-за двух одновременных вызовов метода `put`.

§8.4.4. Обобщенные методы

Метод является *обобщенным*, если он объявлен с одной или более переменными типа (§4.4).

Эти переменные типа известны как *параметры типа* метода. Вид раздела параметров типа обобщенного метода идентичен виду раздела параметров типа обобщенного класса (§8.1.2).

Объявление обобщенного метода определяет множество методов, по одному для каждой возможной конкретизации раздела параметров типов аргументами типа. Аргументы типа не обязаны быть явно указаны в вызове обобщенного метода, так как зачастую они могут быть выведены (§18).

Область видимости и затенение параметров типа метода описаны в §6.3.

Два метода или конструктора, M и N, имеют *одинаковые параметры типа*, если справедливы оба приведенные ниже условия.

- M и N имеют одно и то же количество параметров типа (возможно, нулевое).
- Пусть A_1, ..., A_n являются параметрами типа M, а B_1, ..., B_n — параметрами типа N, и пусть $\theta=[B_1:=A_1, ..., B_n:=A_n]$. Тогда для всех i ($1 \le i \le n$) границей A_i является тот же тип, что и θ, примененный к границе B_i.

Если два метода или конструктора, M и N, имеют одни и те же параметры типов, тип, упомянутый в N, может быть *адаптирован к параметрами типа* M путем применения к типу определенной ранее подстановки θ.

§8.4.5. Возвращаемый тип метода

Результат в объявлении метода либо указывает тип значения, которое возвращает метод (*возвращаемый тип*), либо использует ключевое слово `void` для указания, что данный метод не возвращает значения.

Result:
    UnannType
    `void`

Возвращаемые типы могут различаться у методов, которые перекрывают один другой, если возвращаемые типы являются ссылочными типами. Понятие заменимости возвращаемого типа (return-type-substitutability) поддерживает ковариантный возврат (covariant returns), т.е. специализацию возвращаемого типа подтипом.

Объявление метода d_1 с типом возвращаемого значения R_1 является *заменяемым по возвращаемому типу* (return-type-substitutable) для другого метода d_2 с типом возвращаемого значения R_2 тогда и только тогда, когда выполняется любое из следующих условий.

- Если R_1 является `void`, то R_2 является `void`.
- Если R_1 является примитивным типом, то R_2 идентичен R_1.
- Если R_1 является ссылочным типом, то справедливо одно из условий.
  - ✦ R_1, адаптированный к параметрам типа d_2 (§8.4.4), является подтипом R_2.
  - ✦ R_1 может быть преобразован в подтип R_2 непроверяемым преобразованием (§5.1.9).
  - ✦ d_1 имеет отличную от d_2 сигнатуру (§8.4.2) и $R_1 = |R_2|$.

> В определении для обеспечения плавного перехода от необобщенного к обобщенному коду допускается непроверяемое преобразование. Если непроверяемое преобразование используется для определения того, что R_1 является заменяемым возвращаемым типом для R_2, то R_1 с необходимостью не является подтипом R_2, и правила перекрытия (§8.4.8.3, §9.4.1) требуют вывода предупреждения времени компиляции о непроверенном типе.

§8.4.6. Конструкция `throws` метода

Конструкция `throws` используется для объявления классов проверяемых исключений (§11.1.1), которые могут быть сгенерированы телом метода или конструктора (§11.2.2).

Throws:
`throws` *ExceptionTypeList*

ExceptionTypeList:
ExceptionType {`,` *ExceptionType*}

ExceptionType:
ClassType
TypeVariable

Если любой *ExceptionType*, упомянутый в конструкции `throws`, не является подтипом (§4.10) `Throwable`, генерируется ошибка времени компиляции.

В конструкции `throws` допустимы переменные типа, хотя в конструкции `catch` они запрещены (§14.20).

В конструкции `throws` разрешено, но не требуется в обязательном порядке упомянуть классы непроверяемых исключений (§11.1.1).

Взаимоотношения между конструкцией `throws` и проверкой исключений для тела метода или конструктора рассматриваются в §11.2.3.

По сути, для каждого проверяемого исключения, которое может быть сгенерировано в результате выполнения тела метода или конструктора, генерируется ошибка времени компиляции, если только соответствующий тип исключения или супертип этого типа исключения не упомянут в конструкции `throws` в объявлении метода или конструктора.

Требование объявления проверяемых исключений позволяет компилятору Java гарантировать включение кода для обработки этих исключений. Методы или конструкторы, которые не обрабатывают сгенерированные в их телах проверяемые исключения, обычно вызывают ошибки времени компиляции, если соответствующие типы исключений отсутствуют в конструкциях `throws`. Язык программирования Java таким образом поощряет стиль программирования, в котором именно так документируются редкие и в противном случае действительно исключительные условия.

Отношения между конструкцией `throws` метода и конструкциями `throws` перекрытых или сокрытых методов рассматриваются в §8.4.8.3.

Пример 8.4.6-1. Переменные типа в качестве типов генерируемых исключений

```
import java.io.FileNotFoundException;
interface PrivilegedExceptionAction<E extends Exception> {
    void run() throws E;
}
class AccessController {
    public static <E extends Exception>
    Object doPrivileged(PrivilegedExceptionAction<E> action)
            throws E {
        action.run();
        return "success";
    }
}
class Test {
    public static void main(String[] args) {
      try {
        AccessController.doPrivileged(
          new
          PrivilegedExceptionAction<FileNotFoundException>() {
              public void run() throws FileNotFoundException {
                  // ... удаление файла ...
              }
          });
      } catch (FileNotFoundException f) {
              /* Некоторые действия */ }
    }
}
```

§8.4.7. Тело метода

Тело метода представляет собой либо блок кода, который реализует метод, либо просто точку с запятой, указывающую на отсутствие реализации.

MethodBody:
    Block
    ;

Тело метода должно быть точкой с запятой, если метод объявлен как abstract или native (§8.4.3.1, §8.4.3.4). Точнее говоря,

- если метод объявлен как `abstract` или `native` и имеет в качестве тела блок кода, генерируется ошибка времени компиляции;
- если метод не объявлен ни как `abstract`, ни как `native` и имеет в качестве тела точку с запятой, генерируется ошибка времени компиляции.

> Если реализация должна быть представлена для метода с возвращаемым типом `void` и его реализация не требует выполнения кода, то тело такого метода должно быть записано как блок без инструкций: "`{ }`".

Правила для инструкций `return` в теле метода определены в §14.17.

Если метод объявлен как имеющий возвращаемый тип, то, если тело метода может завершаться обычным образом (§14.1), генерируется ошибка времени компиляции.

> Другими словами, выход из метода с возвращаемым типом должен осуществляться только с помощью инструкции `return` с возвращаемым значением; выход "по достижении конца тела метода" запрещен.
>
> Заметим, что возможна ситуация, когда метод с объявленным возвращаемым типом не содержит инструкций `return`. Вот пример такого метода:
>
> ```
> class DizzyDean {
>     int pitch() { throw new RuntimeException("90 mph?!"); }
> }
> ```

§8.4.8. Наследование, перекрытие и сокрытие

Класс *C наследует* из его непосредственного суперкласса все конкретные методы *m* (как статические, так и экземпляра) суперкласса, для которых выполняются все перечисленные условия.

- *m* является членом непосредственного суперкласса *C*.
- *m* объявлен как `public`, `protected` или с доступом пакета в том же пакете, что и *C*.
- Никакой метод в *C* не имеет сигнатуру, которая является подсигнатурой (§8.4.2) сигнатуры *m*.

Класс *C наследует* из его непосредственного суперкласса и непосредственных суперинтерфейсов все абстрактные методы и методы по умолчанию (§9.4) *m*, для которых выполняются все перечисленные условия.

- m является членом непосредственного суперкласса или непосредственного суперинтерфейса, D, класса C.
- m объявлен как `public`, `protected` или с доступом пакета в том же пакете, что и C.
- Никакой метод в C не имеет сигнатуру, которая является подсигнатурой (§8.4.2) сигнатуры m.
- C не наследует из непосредственного суперкласса никакой конкретный метод, который имеет сигнатуру, являющуюся подсигнатурой m.
- Не существует метода m', который является членом непосредственного суперкласса или суперинтерфейса, D', класса C (m отличен от m', D отличен от D'), такого, что m' из D' перекрывает объявление метода m.

Класс не наследует статические методы из суперинтерфейсов.

> Заметим, что унаследованный конкретный метод может препятствовать наследованию абстрактного метода или метода по умолчанию. (Позже мы уточним, что конкретный метод перекрывает абстрактный метод или метод по умолчанию "из C".) Возможно также, что один метод супертипа препятствует наследованию другого метода супертипа, если первый "уже" перекрывает последний; это точно такое же правило, как и для интерфейсов (§9.4.1), и предотвращает конфликты, когда наследуется несколько методов по умолчанию и одна реализация явно предназначена для замены другой.
>
> Обратите внимание, что методы перекрываются или скрываются на основе соответствия сигнатур. Если, например, класс объявляет два `public`-метода с одним и тем же именем (§8.4.9), а подкласс перекрывает один из них, то подкласс все равно наследует другой метод.

§8.4.8.1. Перекрытие (методами экземпляров)

Метод экземпляра m_C, объявленный в классе C, *перекрывает* в C другой метод экземпляра m_A, объявленный в классе A, тогда и только тогда, когда выполнены все приведенные ниже условия.

- A является суперклассом C.
- C не наследует m_A.
- Сигнатура m_C является подсигнатурой (§8.4.2) сигнатуры m_A.
- Выполняется одно из условий:
  - m_A является `public`;
  - m_A является `protected`;
  - m_A объявлен с доступом пакета в том же пакете, что и C, и либо C объявляет m_C, либо m_A является членом непосредственного суперкласса C.
  - m_A объявлен с доступом пакета и m_C перекрывает m_A из некоторого суперкласса C.
  - m_A объявлен с доступом пакета и m_C перекрывает метод m' из C (m'отличен от m_C и m_A), так что m' перекрывает m_A из некоторого суперкласса C.

Если не абстрактный метод m_C перекрывает абстрактный метод m_A из класса C, то говорят, что m_C *реализует* m_A *из* C.

Метод экземпляра m_C, объявленный или унаследованный классом C, *перекрывает из* C другой метод m_I, объявленный в интерфейсе I, тогда и только тогда, когда выполняются все следующие условия.

- I является суперинтерфейсом C.
- m_I является абстрактным методом или методом по умолчанию.
- Сигнатура m_C является подсигнатурой (§8.4.2) сигнатуры m_I.

> Сигнатура перекрывающего метода может отличаться от сигнатуры перекрытого метода, если формальный параметр в одном из методов имеет несформированный тип, в то время как соответствующий параметр в другом имеет параметризованный тип. Это допускает миграцию существующего кода для использования преимуществ обобщенных типов.
>
> Понятие перекрытия включает методы, которые перекрывают другие методы из некоторого подкласса объявляющего их класса. Это может произойти двумя способами.
>
> - Конкретный метод в обобщенном суперклассе может при определенных параметризациях иметь ту же сигнатуру, что и абстрактный метод в этом классе. В таком случае конкретный метод наследуется, а абстрактный — нет (как описано выше). Унаследованный метод должен в таком случае рассматриваться как перекрывающий абстрактный из C. (Этот сценарий усложняется доступом пакета: если C находится в другом пакете, то m_A не должен наследоваться никоим образом и не должен рассматриваться как перекрытый.)
> - Метод, унаследованный от класса, может перекрывать метод суперинтерфейса. (К счастью, проблем с доступом пакета в данном случае нет.)

Если метод экземпляра перекрывает метод, объявленный как `static`, генерируется ошибка времени компиляции.

> В этом отношении перекрытие методов отличается от сокрытия полей (§8.3), при котором переменная экземпляра может скрывать `static`-переменную.

Перекрытый метод может быть доступен с помощью выражения вызова метода (§15.12), содержащего ключевое слово `super`. Квалифицированное имя или приведение к типу суперкласса не позволяет получить доступ к перекрытому методу.

> В этом отношении перекрытие методов отличается от сокрытия полей (§15.12.4.4).

Наличие или отсутствие модификатора `strictfp` абсолютно не влияет на правила перекрытия методов и реализацию абстрактных методов. Например, как методу, не являющемуся FP-строгим, разрешено перекрывать FP-строгий метод, так и методу, являющемуся FP-строгим, разрешено перекрывать не FP-строгий метод.

Пример 8.4.8.1-1. Перекрытие

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
```

```
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
        return d > limit ? limit : d < -limit ? -limit : d;
    }
}
```

Здесь класс `SlowPoint` перекрывает объявление метода `move` класса `Point` своим методом `move`, который ограничивает расстояние перемещения точки при вызове метода. Когда метод `move` вызывается для экземпляра класса `SlowPoint`, всегда вызывается перекрывающее определение в классе `SlowPoint`, даже если ссылка на объект `SlowPoint` взята из переменной, тип которой — `Point`.

Пример 8.4.8.1-2. Перекрытие

Перекрытие упрощает для подкласса расширение поведения существующего класса, как показано в приведенном примере.

```
import java.io.OutputStream;
import java.io.IOException;

class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o = o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws IOException {
        if (pos == buf.length) flush();
        buf[pos++] = (byte)c;
    }
    public void putstr(String s) throws IOException {
        for (int i = 0; i < s.length(); i++)
            putchar(s.charAt(i));
    }
    public void flush() throws IOException {
        o.write(buf, 0, pos);
        pos = 0;
    }
}
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n') flush();
    }
}
```

```
class Test {
    public static void main(String[] args) throws IOException {
        LineBufferOutput lbo = new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}
```

Вывод программы имеет вид

```
lbo
print
lbo
```

Класс `BufferOutput` реализует очень простую буферизованную версию `OutputStream`, сбрасывающую буфер при его заполнении или вызове `flush`. Подкласс `LineBufferOutput` объявляет только конструктор и единственный метод `putchar`, который перекрывает метод `putchar` класса `BufferOutput`. Он наследует методы `putstr` и `flush` от класса `BufferOutput`.

В методе `putchar` объекта `LineBufferOutput` для аргумента, который представляет собой символ новой строки, вызывается метод `flush`. Важным моментом этого примера является то, что метод `putstr`, объявленный в классе `BufferOutput`, вызывает метод `putchar`, определяемый текущим объектом `this`, и этот метод в результате не обязательно является методом `putchar`, объявленным в классе `BufferOutput`.

Таким образом, когда метод `putstr` вызывается в методе `main` с использованием объекта `lbo` типа `LineBufferOutput`, вызов метода `putchar` в теле метода `putstr` представляет собой вызов метода `putchar` объекта `lbo`, перекрывающего объявления `putchar`, которое проверяет наличие символа новой строки. Это позволяет подклассу `BufferOutput` изменить поведение метода `putstr` без переопределения последнего.

Документация такого класса, как `BufferOutput`, который предназначен для расширения, должна ясно указывать соглашение между этим классом и его подклассами, в том числе указывать, что подклассы могут перекрывать метод `putchar` таким способом. Разработчик класса `BufferOutput`, таким образом, не должен изменять реализацию метода `putstr` в будущих версиях `BufferOutput` так, чтобы она перестала использовать метод `putchar`, поскольку это может привести к нарушению работоспособности более ранних версий подклассов. Обсуждение бинарной совместимости рассматривается в главе 13, в частности в §13.2.

§8.4.8.2. Сокрытие (методами класса)

Если класс *C* объявляет статический метод *m*, то говорят, что объявление *m скрывает* любой метод *m′*, когда сигнатура *m* является подсигнатурой (§8.4.2) сигнатуры *m′*, в суперклассах и суперинтерфейсах класса *C*, который в противном случае был бы доступен коду в классе *C*.

Если метод, объявленный как `static`, скрывает метод экземпляра, генерируется ошибка времени компиляции.

В этом отношении сокрытие методов отличается от сокрытия полей (§8.3), когда переменная, объявленная как `static`, может скрывать переменную экземпляра. Сокрытие также отличается от затенения (§6.4.1) и затемнения (§6.4.2).

Доступ к сокрытому методу может осуществляться с помощью квалифицированного имени или выражения вызова метода (§15.12), которое содержит ключевое слово `super` или приведение к типу суперкласса.

В этом отношении сокрытие методов подобно сокрытию полей.

Пример 8.4.8.2-1. Вызов сокрытых методов класса

Метод класса (объявленный как `static`), являющийся сокрытым, может быть вызван с помощью ссылки, тип которой представляет собой тип класса, в действительности содержащий объявление этого метода. В этом отношении сокрытие `static`-методов отличается от перекрытия методов экземпляров.

```
class Super {
    static String greeting() { return "Goodnight"; }
    String name() { return "Richard"; }
}
class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}
```

Приведенный выше код генерирует следующий вывод:

```
Goodnight, Dick
```

Это связано с тем, что вызов `greeting` для выяснения во время компиляции того, какого именно класса метод должен быть вызван, использует тип `s`, а именно — `Super`, в то время как вызов `name` использует класс `s`, а именно — `Sub`, для выяснения во время выполнения, какой именно метод экземпляра будет вызван.

§8.4.8.3. Требования к перекрытию и сокрытию

Если объявление метода d_1 с возвращаемым типом R_1 перекрывает или скрывает объявление другого метода d_2 с возвращаемым типом R_2, то d_1 должен быть заменяемым по возвращаемому типу (§8.4.5) для d_2, иначе генерируется ошибка времени компиляции.

Это правило обеспечивает ковариантность возвращаемых типов — уточнение возвращаемого типа метода при его перекрытии.

Если R_1 не является подтипом R_2, то выводится предупреждение времени компиляции о непроверенном типе, если только его вывод не подавлен аннотацией `SuppressWarnings` (§9.6.4.5).

Метод, который перекрывает или скрывает другой метод, включая методы, реализующие абстрактные методы, определенные в интерфейсах, не может быть объявлен как генерирующий больше проверяемых исключений, чем перекрытый или скрытый метод.

> В этом отношении перекрытие методов отличается от сокрытия полей (§8.3), когда поле может скрывать поле другого типа.

Говоря более строго, предположим, что B представляет собой класс или интерфейс, а A — суперкласс или суперинтерфейс B, и объявление метода m_2 в B перекрывает или скрывает объявление метода m_1 в A. Тогда справедливо следующее.

- Если m_2 имеет конструкцию `throws`, которая упоминает некоторые типы проверяемых исключений, то m_1 должен иметь конструкцию `throws`, иначе генерируется ошибка времени компиляции.
- Для каждого типа проверяемого исключения, перечисленного в конструкции `throws` объявления m_2, тот же самый класс исключения или один из его супертипов должен находиться в затирании (§4.6) конструкции `throws` объявления m_1; в противном случае генерируется ошибка времени компиляции.
- Если незатертая конструкция `throws` объявления m_1 не содержит супертипа каждого типа исключения из конструкции `throws` объявления m_2 (адаптированная при необходимости к параметрам типов m_1), выводится предупреждение времени компиляции о непроверенном типе.

Если объявление типа T имеет метод-член m_1 и имеется метод m_2, объявленный в T или супертипе типа T, такой, что выполняются все перечисленные ниже условия, то генерируется ошибка времени компиляции.

- m_1 и m_2 имеют одно и то же имя.
- m_2 доступен из типа T.
- Сигнатура m_1 не является подсигнатурой (§8.4.2) сигнатуры m_2.
- Сигнатура метода m_1 или некоторого перекрывающего (непосредственно или опосредованно) m_1 метода имеет то же затирание, что и сигнатура метода m_2 или некоторого перекрывающего (непосредственно или опосредованно) m_2 метода.

> Эти ограничения необходимы, поскольку обобщенность реализуется через затирания. Из приведенного выше правила вытекает, что методы, объявленные в одном и том же классе с одним и тем же именем, должны иметь разные затирания. Это также означает, что объявление типа не может реализовывать или расширять две различные конкретизации одного и того же обобщенного интерфейса.

Модификатор доступа (§6.6) перекрывающего или скрывающего метода должен предоставлять как минимум такой же доступ, как и перекрытый или сокрытый метод, следующим образом.

- Если перекрытый или сокрытый метод объявлен как `public`, то перекрывающий или скрывающий метод должен быть объявлен как `public`; в противном случае генерируется ошибка времени компиляции.

- Если перекрытый или сокрытый метод объявлен как `protected`, то перекрывающий или скрывающий метод должен быть объявлен как `protected` или `public`; в противном случае генерируется ошибка времени компиляции.
- Если перекрытый или сокрытый метод имеет доступ пакета, то перекрывающий или скрывающий метод *не* должен быть объявлен как `private`; в противном случае генерируется ошибка времени компиляции.

Обратите внимание, что метод, объявленный как `private`, не может быть сокрыт или перекрыт в техническом смысле этих терминов. Это означает, что подкласс может объявить метод с той же сигнатурой, что и метод, объявленный в одном из его суперклассов как `private`, и при этом не имеется требования к возвращаемому типу или конструкции `throws` этого метода находиться в каких-либо взаимоотношениях с таковыми для метода, объявленного в суперклассе как `private`.

Пример 8.4.8.3-1. Ковариантные возвращаемые типы

```
class C implements Cloneable {
    C copy() throws CloneNotSupportedException {
        return (C)clone();
    }
}
class D extends C implements Cloneable {
    D copy() throws CloneNotSupportedException {
        return (D)clone();
    }
}
```

Приведенные объявления являются корректными в языке программирования Java, начиная с Java SE 5.0.

Ослабленное правило для перекрытия позволяет также ослабить условия, накладываемые на абстрактные классы, реализующие интерфейсы.

Пример 8.4.8.3-2. Предупреждения о непроверенных возвращаемых типах

Рассмотрим код

```
class StringSorter {
    // Превращаем коллекцию строк в отсортированный список
    List toList(Collection c) {...}
}
```

И пусть кто-то создал подкласс `StringSorter`:

```
class Overrider extends StringSorter {
    List toList(Collection c) {...}
}
```

Пусть теперь в некоторый момент автор `StringSorter` решил сделать код обобщенным:

```
class StringSorter {
    // Превращаем коллекцию строк в отсортированный список
    List<String> toList(Collection<String> c) {...}
}
```

При компиляции `Overrider` с новым определением `StringSorter` будет выведено предупреждение о непроверенном типе, поскольку возвращаемым типом `Overrider.toList` является `List`, который не является подтипом возвращаемого типа перекрытого метода `List<String>`.

Пример 8.4.8.3-3. Некорректная конструкция `throws` при перекрытии

В приведенной программе в объявлении класса `BadPointException` используется обычная форма объявления нового типа исключения.

```
class BadPointException extends Exception {
    BadPointException() { super(); }
    BadPointException(String s) { super(s); }
}
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
    void move(int dx, int dy) throws BadPointException {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

Программа приводит к ошибке времени компиляции, потому что перекрытие метода `move` в классе `CheckedPoint` объявляет, что он может генерировать проверяемое исключение, которое в методе `move` класса `Point` не объявлено. Если не считать это ошибкой, то код, вызывающий метод `move` посредством ссылки типа `Point`, может обнаружить нарушение контракта между ним и классом `Point` при генерации указанного исключения.

Удаление конструкции `throws` не помогает.

```
class CheckedPoint extends Point {
    void move(int dx, int dy) {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

Теперь генерируется другая ошибка времени компиляции, связанная с тем, что тело метода `move` не может генерировать проверяемые исключения, а именно — `BadPointException`, не перечисленные в конструкции `throws` метода `move`.

Пример 8.4.8.3-4. Влияние затирания на перекрытие

Класс не может иметь два метода-члена с одним и тем же именем и затиранием типа.

```
class C<T> {
    T id (T x) {...}
}
class D extends C<String> {
    Object id(Object x) {...}
}
```

Это неверный код, потому что `D.id(Object)` **является членом** `D`, `C<String>.id(String)` **объявлен как супертип** `D`, **и**

- **два метода имеют одно и то же имя** `id`;
- `C<String>.id(String)` **доступен классу** `D`;
- **сигнатура** `D.id(Object)` **не является подсигнатурой сигнатуры** `C<String>.id(String)`;
- **эти два метода имеют одно и то же затирание.**

Два различных метода класса не могут перекрывать методы с тем же затиранием:

```
class C<T> {
    T id(T x) {...}
}
interface I<T> {
    T id(T x);
}
class D extends C<String> implements I<Integer> {
    public String id(String x) {...}
    public Integer id(Integer x) {...}
}
```

Этот код также неверен, поскольку `D.id(String)` **является членом** `D`, `D.id(Integer)` **объявлен в** `D`, **и**

- **эти два метода имеют одно и то же имя** `id`;
- `D.id(Integer)` **доступен классу** `D`;
- **эти два метода имеют разные сигнатуры (и ни одна из них не является подсигнатурой другой);**
- `D.id(String)` **перекрывает** `C<String>.id(String)` **и** `D.id(Integer)` **перекрывает** `I.id(Integer)` **и два перекрытых метода имеют одно и то же затирание.**

§8.4.8.4. Наследование методов с сигнатурами, эквивалентными в смысле перекрытия

Возможна ситуация, когда класс наследует несколько методов с сигнатурами, эквивалентными в смысле перекрытия (§8.4.2).

Если класс `C` наследует конкретный метод, сигнатура которого является эквивалентной в смысле перекрытия сигнатуре другого метода, унаследованного *C*, генерируется ошибка времени компиляции.

Если класс `C` наследует метод по умолчанию, сигнатура которого является эквивалентной в смысле перекрытия сигнатуре другого метода, унаследованного *C*, то если только существует абстрактный метод, объявленный в суперклассе *C* и унаследованный

C, который эквивалентен в смысле перекрытия этим двум методам, генерируется ошибка времени компиляции.

> Это исключение из строгих правил конфликтов "по умолчанию–абстрактный" и "по умолчанию–по умолчанию" возникает, когда абстрактный метод объявлен в суперклассе: утверждение, что абстрактность, происходящая из иерархии суперклассов, по сути, превосходит метод по умолчанию, делает метод по умолчанию действующим так, как будто он является абстрактным. Однако абстрактный метод из класса не перекрывает метод (или методы) по умолчанию, так как интерфейсы позволяют уточнять *сигнатуру* абстрактного метода, происходящего из иерархии классов.
>
> Обратите внимание, что исключение неприменимо, если все эквивалентные в смысле перекрытия абстрактные методы, унаследованные *C*, были объявлены в интерфейсах.

В противном случае множество методов, эквивалентных в смысле перекрытия, состоит из как минимум одного абстрактного метода и нуля или большего количества методов по умолчанию; тогда класс с необходимостью является абстрактным и рассматривается как унаследовавший все методы.

Один из унаследованных методов должен быть заменяемым по возвращаемому типу для каждого из других унаследованных методов; в противном случае генерируется ошибка времени компиляции. (В этом случае конструкции `throws` к ошибке не приводят.)

Может быть несколько путей, которыми одно и то же объявление метода наследуется из интерфейса. Этот факт не приводит к сложностям и сам по себе никогда не вызывает ошибку времени компиляции.

§8.4.9. Перегрузка

Если два метода класса (объявленные ли оба в одном и том же классе или оба унаследованные классом, или один объявлен, а другой унаследован) имеют одно и то же имя, но их сигнатуры не являются эквивалентными в смысле перекрытия, такие методы называются *перегруженными* (overloaded).

Этот факт не приводит к сложностям и сам по себе никогда не вызывает ошибку времени компиляции. Не требуется никаких необходимых отношений между возвращаемыми типами или конструкциями `throws` двух методов с одним и тем же именем, если только их сигнатуры не являются эквивалентными в смысле перекрытия.

При вызове метода (§15.12) для определения во время компиляции сигнатуры метода, который будет вызван (§15.12.2), используются количество фактических аргументов (и любые явные аргументы типа) и типы аргументов времени компиляции. Если вызываемый метод является методом экземпляра, фактический вызываемый метод определяется во время выполнения с применением динамического поиска метода (§15.12.4).

Пример 8.4.9-1. Перегрузка

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
```

```
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+","+y+")"; }
}
```

Здесь класс `Point` имеет два члена, которые являются методами с одним и тем же именем `move`. Перегруженный метод `move` класса `Point`, выбираемый для любого конкретного вызова метода, определяется во время компиляции процедурой разрешения перегрузки, описанной в §15.12.

В целом членами класса `Point` являются переменные экземпляра `x` и `y` типа `float`, два объявленных метода `move`, объявленный метод `toString` и члены `Point`, унаследованные от его неявного непосредственного суперкласса `Object` (§4.3.2), такие, как метод `hashCode`. Обратите внимание, что `Point` не наследует метод `toString` класса `Object`, потому что этот метод перекрыт объявлением метода `toString` в классе `Point`.

Пример 8.4.9-2. Перегрузка, перекрытие и сокрытие

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
}
```

Здесь класс `RealPoint` скрывает объявления переменных экземпляра `x` и `y` типа `int` класса `Point` собственными переменными экземпляра `x` и `y` типа `float` и перекрывает метод `move` класса `Point` собственным методом `move`. Он также перегружает имя `move` другим методом с отличающейся сигнатурой (§8.4.2).

В этом примере члены класса `RealPoint` включают переменную экземпляра `color`, унаследованную от класса `Point`, переменные экземпляра `x` и `y` типа `float`, объявленные в классе `RealPoint`, и два метода `move`, объявленные в классе `RealPoint`.

Какой из этих перегруженных методов `move` класса `RealPoint` будет выбран в конкретном вызове метода, определяется во время компиляции процедурой разрешения перегрузки, описанной в §15.12.

Приведенная далее программа представляет собой расширенный вариант предыдущей программы.

```
class Point {
    int x = 0, y = 0, color;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
```

```
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
}
```

Здесь класс `Point` предоставляет методы `getX` и `getY`, которые возвращают значения их полей `x` и `y`; затем класс `RealPoint` перекрывает эти методы объявлениями методов с теми же сигнатурами. В результате мы получаем две ошибки времени компиляции, по одной для каждого метода, поскольку они имеют другие возвращаемые типы: методы в классе `Point` возвращают значения типа `int`, но предлагаемые в качестве перекрывающих методы в классе `RealPoint` возвращают значения типа `float`.

В приведенной далее программе эти ошибки исправлены.

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    int getX() { return (int)Math.floor(x); }
    int getY() { return (int)Math.floor(y); }
}
```

Здесь перекрывающие методы `getX` и `getY` в классе `RealPoint` имеют те же возвращаемые типы, что и перекрываемые ими методы класса `Point`, так что данный код успешно компилируется.

Теперь рассмотрим следующую тестовую программу.

```
class Test {
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        Point p = rp;
        rp.move(1.71828f, 4.14159f);
        p.move(1, -1);
        show(p.x, p.y);
        show(rp.x, rp.y);
        show(p.getX(), p.getY());
        show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
        System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
```

```
            System.out.println("(" + x + ", " + y + ")");
        }
}
```

Вывод этой программы имеет следующий вид.

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

Первая строка вывода иллюстрирует тот факт, что экземпляр `RealPoint` в действительности содержат два целочисленных поля, объявленных в классе `Point`; просто их имена скрыты от кода в объявлении класса `RealPoint` (а тем самым и от всех подклассов, которые он может иметь). Когда для обращения к полю `x` используется ссылка на экземпляр класса `RealPoint` в переменной типа `Point`, обращение выполняется к целочисленному полю `x`, объявленному в классе `Point`. Тот факт, что это значение равно нулю, указывает, что вызов `p.move(1,-1)` вызывает не метод `move` класса `Point`, а перекрытый метод `move` класса `RealPoint`.

Вторая строка вывода показывает, что обращение к полю `rp.x` обращается к полю `x`, объявленному в классе `RealPoint`. Это поле имеет тип `float`, и вторая строка вывода, соответственно, выводит значения с плавающей точкой. Кстати, это также иллюстрирует тот факт, что имя метода `show` перегружено; типы аргументов в вызове метода определяют, какое именно из двух определений будет вызвано.

Последние две строки вывода показывают, что каждый из вызовов методов `p.getX()` и `rp.getX()` вызывает метод `getX`, объявленный в классе `RealPoint`. В действительности нет способа вызвать метод `getX` класса `Point` для экземпляра класса `RealPoint` извне тела `RealPoint`, независимо от того, какой тип переменной используется для хранения ссылки на объект. Таким образом, мы видим, что поля и методы ведут себя по-разному: сокрытие отличается от перекрытия.

§8.5. Объявления типов-членов

Класс-член представляет собой класс, объявление которого размещено непосредственно в объявлении другого класса или интерфейса (§8.1.6, §9.1.4).

Интерфейс-член представляет собой интерфейс, объявление которого размещено непосредственно в объявлении другого класса или интерфейса (§8.1.6, §9.1.4).

Доступность типа-члена в объявлении класса описана в §6.6.

Если объявление типа-члена содержит одно и то же ключевое слово в качестве модификатора для типа-члена более одного раза, генерируется ошибка времени компиляции.

Область видимости и затенение типа-члена описаны в §6.3 и §6.4.

Если класс объявляет тип-член с определенным именем, то говорят, что объявление этого типа *скрывает* любые доступные объявления типов-членов с тем же именем в суперклассах и суперинтерфейсах класса.

В этом отношении сокрытие типов-членов подобно сокрытию полей (§8.3).

Класс наследует из своих непосредственных суперклассов и непосредственных суперинтерфейсов все не объявленные как `private` типы-члены суперклассов и суперинтерфейсов, которые доступны коду класса и не сокрыты объявлениями в классе.

Класс может наследовать два или более объявлений типов с одним и тем же именем либо из двух интерфейсов, либо из его суперкласса и интерфейса. При попытке обращения к любому неоднозначно унаследованному классу или интерфейсу по его простому имени генерируется ошибка времени компиляции.

Если объявление одного и того же типа наследуется из интерфейса несколькими путями, класс или интерфейс рассматривается как унаследованный только один раз. К нему можно обращаться по его простому имени без какой-либо неоднозначности.

§8.5.1. Объявления статических типов-членов

Ключевое слово `static` может модифицировать объявление типа-члена *C* в теле класса или интерфейса *T*, не являющегося внутренним. Результатом его применения является объявление, что *C* не является внутренним классом. Так же, как метод *T*, объявленный как `static`, не имеет текущего экземпляра *T* в своем теле, так и *C* не имеет ни текущего экземпляра *T*, ни каких-то лексически охватывающих экземпляров.

Если в классе, объявленном как `static`, содержится использование нестатического члена охватывающего класса, генерируется ошибка времени компиляции.

Интерфейс-член неявно является статическим (`static`) (§9.1.1). Тем не менее объявление интерфейса-члена может избыточно использовать модификатор `static`.

§8.6. Инициализаторы экземпляра

Инициализатор экземпляра, объявленный в классе, выполняется при создании экземпляра класса (§12.5, §15.9, §8.8.7.1).

InstanceInitializer:
    Block

Если инициализатор экземпляра не может нормально завершиться (§14.21), генерируется ошибка времени компиляции.

Если где-либо в инициализаторе экземпляра встречается инструкция `return` (§14.17), генерируется ошибка времени компиляции.

Инициализаторы экземпляров могут обращаться к текущему объекту с помощью ключевого слова `this` (§15.8.3), использовать ключевое слово `super` (§15.11.2, §15.12) и любые переменные типа в области видимости.

> Использование переменных экземпляра, объявления которых текстуально располагаются после их использования, частично ограничено, несмотря на то что эти переменные экземпляра находятся в области видимости. Смотрите детальное описание этого вопроса в §8.3.3.

Вопросы проверки исключений инициализаторов экземпляра рассматриваются в §11.2.3.

§8.7. Статические инициализаторы

Статический инициализатор, объявленный в классе, вызывается при инициализации класса (§12.4.2). Статические инициализаторы могут использоваться для инициализации переменных класса наряду с инициализаторами полей переменных класса (§8.3.2).

StaticInitializer:
`static` *Block*

Если статический инициализатор не может завершить работу нормально (§14.21), генерируется ошибка времени компиляции.

Если где-либо в статическом инициализаторе встречается инструкция `return` (§14.17), генерируется ошибка времени компиляции.

Если в статическом инициализаторе встречается ключевое слово `this` (§15.8.3), ключевое слово `super` (§15.11.2, §15.12) или любые переменные типа, объявленные вне этого статического инициализатора, генерируется ошибка времени компиляции.

> Использование переменных класса, объявления которых текстуально располагаются после их использования, частично ограничено, несмотря на то что эти переменные класса находятся в области видимости. Смотрите детальное описание этого вопроса в §8.3.2.3.

Вопросы проверки исключений статических инициализаторов рассматриваются в §11.2.3.

§8.8. Объявления конструкторов

Конструктор используется при создании объекта, являющегося экземпляром класса (§12.5, §15.9).

ConstructorDeclaration:
{ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

ConstructorDeclarator:
[TypeParameters] SimpleTypeName (*[FormalParameterList]*)

SimpleTypeName:
Identifier

Правила в этом разделе применимы к конструкторам во всех объявлениях классов, включая объявления перечислений. Однако для объявления перечислений применяются специальные правила, касающиеся модификаторов конструкторов, тел конструкторов и конструкторов по умолчанию; эти правила перечислены в §8.9.2.

SimpleTypeName в *ConstructorDeclarator* должно быть простым именем класса, содержащего объявление конструктора; в противном случае генерируется ошибка времени компиляции.

Во всех прочих аспектах объявление конструктора выглядит в точности так же, как объявление метода, не имеющего возвращаемого типа (§8.4.5).

Объявления конструкторов не являются членами. Они никогда не наследуются, а следовательно, не могут быть перекрыты или скрыты.

Конструкторы вызываются выражениями создания экземпляра класса (§15.9), преобразованиями и конкатенациями при использовании оператора конкатенации строк + (§15.18.1) и явными вызовами конструкторов из других конструкторов (§8.8.7). Доступ к конструкторам управляется модификаторами доступа (§6.6), так что возможно предотвращение инстанцирования путем объявления недоступного конструктора (§8.8.10).

Конструкторы никогда не вызываются выражениями вызова метода (§15.12).

Пример 8.8-1. Объявления конструкторов

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

§8.8.1. Формальные параметры и параметры типа

Формальные параметры конструктора синтаксически и семантически идентичны таковым для метода (§8.4.1).

Конструктор внутреннего класса-члена, не объявленного как `private`, неявно объявляет в качестве первого формального параметра переменную, представляющую непосредственно охватывающий экземпляр класса (§15.9.2, §15.9.3).

Пояснение, почему только класс этого вида имеет неявно объявленный параметр конструктора, достаточно тонкое. Приведенное далее пояснение может оказаться полезным.

1. В §15.9.2 определяется непосредственно охватывающий экземпляр члена-класса в выражении создания экземпляра класса для внутреннего члена-класса, не являющегося `private`. Байт-код класса-члена может быть произведен компилятором, отличным от компилятора, который обрабатывает выражение создания экземпляра класса. Таким образом, должен иметься стандартный способ, которым компилятор, обрабатывающий выражение создания, передает ссылку (представляющую непосредственно охватывающий экземпляр) конструктору класса-члена. Следовательно, язык программирования Java в этом разделе полагает, что конструктор внутреннего класса-члена, не являющегося `private`, неявно объявляет начальный параметр для немедленно охватывающего экземпляра. В §15.9.3 указано, что этот экземпляр передается конструктору.
2. В выражении создания экземпляра класса для локального класса (не в статическом контексте) или анонимного класса §15.9.2 определяет непосредственно охватывающий экземпляр локального/анонимного класса. Локальный/анонимный класс с необходимостью производится (создается байт-код класса) тем же компилятором, что и выражение создания экземпляра класса. Этот компилятор может представлять непосредственно охватывающий экземпляр так, как ему заблагорассудится. Языку программирования Java нет необходимости неявно объявлять параметр в конструкторе локального/анонимного класса.

3. В выражении создания экземпляра анонимного класса, если суперкласс анонимного класса либо внутренний, либо локальный (не в статическом контексте) §15.9.2 определяет непосредственно охватывающий экземпляр анонимного класса по отношению к суперклассу. Этот экземпляр должен передаваться от анонимного класса его суперклассу, где он будет служить в качестве непосредственно охватывающего экземпляра. Поскольку суперкласс (его байт-код) может быть произведен компилятором, отличным от компилятора выражения создания экземпляра класса, необходимо передавать экземпляр стандартным путем, в качестве первого аргумента конструктора суперкласса. Обратите внимание, что сам анонимный класс с необходимостью производится тем же компилятором, что и выражение создания экземпляра класса, так что компилятор должен иметь возможность передать непосредственно охватывающий экземпляр по отношению к суперклассу анонимному классу так, как ему заблагорассудится, до того, как анонимный класс передаст экземпляр конструктору суперкласса. Однако для согласованности язык программирования Java полагает в §15.9.5.1, что в некоторых ситуациях конструктор анонимного класса неявно объявляет первый параметр как непосредственно охватывающий экземпляр по отношению к суперклассу.

Тот факт, что доступ к внутреннему классу-члену, не объявленному как `private`, может получить другой компилятор, отличный от скомпилировавшего его, в то время как доступ к локальному или анонимному классу получает только компилятор, скомпилировавший его, объясняет, почему бинарное имя внутреннего класса-члена, не являющегося `private`, является предсказуемым, в отличие от бинарного имени локального или анонимного класса (§13.1).

§8.8.2. Сигнатура конструктора

В случае объявления в классе двух конструкторов с эквивалентными в смысле перекрытия сигнатурами (§8.4.2) генерируется ошибка времени компиляции.

В случае объявления в классе двух конструкторов, сигнатуры которых имеют одно и то же затирание (§4.6), генерируется ошибка времени компиляции.

§8.8.3. Модификаторы конструкторов

ConstructorModifier: одно из
Annotation `public protected private`

Правила для модификаторов аннотаций в объявлениях конструкторов определены в §9.7.4 и §9.7.5.

Если одно и то же ключевое слово появляется в объявлении конструктора в качестве модификатора более одного раза, генерируется ошибка времени компиляции.

В обычном объявлении класса объявление конструктора без модификаторов доступа имеет доступ пакета.

Если в объявлении конструктора имеется два или более (различных) модификаторов конструктора, то, как правило (хотя и не обязательно) они находятся в порядке, согласующемся с показанным выше в продукции для *ConstructorModifier*.

В отличие от методов конструктор не может быть объявлен как `abstract`, `static`, `final`, `native`, `strictfp` или `synchronized`.

- Конструктор не наследуется, так что нет необходимости в объявлении его как `final`.
- Конструктор, объявленный как `abstract`, никогда не может быть реализован.
- Конструктор всегда вызывается по отношению к объекту, так что конструктор, объявленный как `static`, не имеет смысла.
- Нет никакого практического смысла в объявлении конструктора как `synchronized`, поскольку это блокировало бы объект при создании, но обычно объект и так недоступен другим потокам до тех пор, пока все конструкторы объекта не завершат свою работу.
- Отсутствие `native`-конструкторов представляет собой особенность дизайна языка программирования, которая призвана облегчить реализации виртуальной машины Java проверку того, что в процессе создания объекта были корректно вызваны конструкторы суперкласса.
- Невозможность объявить конструктор как `strictfp` (в отличие от метода (§8.4.3)) представляет собой преднамеренную особенность дизайна языка программирования; гарантируется, что конструктор является FP-строгим тогда и только тогда, когда FP-строгим является класс (§15.4).

§8.8.4. Обобщенные конструкторы

Конструктор является *обобщенным*, если он объявляет одну или несколько переменных типа (§4.4).

Эти переменные типа известны как *параметры типа* конструктора. Вид раздела параметров типа обобщенного конструктора идентичен виду раздела параметров типа обобщенного класса (§8.1.2).

Конструктор может быть объявлен как обобщенный, независимо от того, является ли обобщенным класс конструктора.

Объявление обобщенного конструктора определяет множество конструкторов, по одному для каждой возможной конкретизации раздела параметров аргументами типа. Аргументы типа могут не быть указаны явно при вызове обобщенного конструктора, так как зачастую они могут быть выведены (§18).

Область видимости и затенение параметров типа конструктора рассматриваются в §6.3 и §6.4.

§8.8.5. Конструкция `throws` у конструкторов

Конструкция `throws` конструктора идентична по структуре и поведению конструкции `throws` метода (§8.4.6).

§8.8.6. Тип конструктора

Тип конструктора состоит из его сигнатуры и типов исключений в его конструкции `throws`.

§8.8.7. Тело конструктора

Первой инструкцией тела конструктора может быть неявный вызов другого конструктора того же класса или непосредственного суперкласса (§8.8.7.1).

ConstructorBody:
{ *[ExplicitConstructorInvocation] [BlockStatements]* }

Если конструктор непосредственно или опосредованно вызывает сам себя с помощью ряда из одного или нескольких явных вызовов конструкторов, включающих `this`, генерируется ошибка времени компиляции.

Если тело конструктора не начинается с явного вызова конструктора, а объявленный конструктор не является частью изначального класса `Object`, то тело конструктора неявно начинается с вызова конструктора суперкласса `super();` — вызова конструктора непосредственного суперкласса без аргументов.

За исключением возможности явных вызовов конструкторов и запрета явного возврата значения (§14.17), тело конструктора аналогично телу метода (§8.4.7).

Инструкция `return` (§14.17) может быть использована в теле конструктора, если она не включает выражение.

Пример 8.8.7-1. Тела конструкторов

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, WHITE);
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

Здесь первый конструктор `ColoredPoint` вызывает второй, передавая ему дополнительный аргумент; второй конструктор `ColoredPoint` вызывает конструктор своего суперкласса `Point`, передавая ему координаты.

§8.8.7.1. Явные вызовы конструкторов

ExplicitConstructorInvocation:
[TypeArguments] `this` (*[ArgumentList]*) ;
[TypeArguments] `super` (*[ArgumentList]*) ;
ExpressionName . *[TypeArguments]* `super` (*[ArgumentList]*) ;
Primary . *[TypeArguments]* `super` (*[ArgumentList]*) ;

Для удобства ниже приведены продукции из §4.5.1 и §15.12.

TypeArguments:
    < *TypeArgumentList* >

ArgumentList:
    Expression {, *Expression*}

Инструкции явных вызовов конструкторов можно разделить на два вида.

- *Вызовы альтернативных конструкторов* начинаются с ключевого слова `this` (возможно, предваренного явными аргументами типа). Они используются для вызова других конструкторов того же самого класса.
- *Вызовы конструкторов суперкласса* начинаются с ключевого слова `super` (возможно, предваренного явными аргументами типа) или с выражения *Primary* или *ExpressionName*. Они используются для вызова конструктора непосредственного суперкласса. Вызовы конструкторов суперкласса также можно разделить на два вида.
  - *Вызовы конструкторов неквалифицированных суперклассов* начинаются с ключевого слова `super` (возможно, предваренного явными аргументами типа).
  - *Вызовы конструкторов квалифицированных суперклассов* начинаются с выражения *Primary* или *ExpressionName*. Они позволяют конструктору подкласса явно указать непосредственно охватывающий экземпляр вновь создаваемого объекта по отношению к непосредственному суперклассу (§8.1.3). Это может быть необходимо, когда суперкласс является внутренним классом.

Инструкция явного вызова конструктора в теле конструктора не может обращаться ни к каким переменным экземпляра, методам экземпляра или внутренним классам, объявленным в этом классе или любом суперклассе, или использовать ключевое слово `this` или `super` в любых выражениях; в противном случае генерируется ошибка времени компиляции.

> Этот запрет на использование текущего экземпляра поясняет, почему инструкция явного вызова конструктора считается осуществляющейся в статическом контексте (§8.1.3).

Типы исключений, которые может генерировать инструкция явного вызова конструктора, определены в §11.2.2.

Ели *TypeArguments* присутствует слева от `this` или `super`, то, если любой из аргументов типа представляет собой символ подстановки (§4.5.1), генерируется ошибка времени компиляции.

Пусть C — инстанцируемый класс и пусть S — непосредственный суперкласс класса C. Если инструкция вызова конструктора суперкласса неквалифицированная, то:

- если S является внутренним классом-членом, но S не является лексически охватывающим объявлением типа для C, то генерируется ошибка времени компиляции.

Если инструкция вызова конструктора суперкласса квалифицированная, то:

- если S не является внутренним классом или если объявление S находится в статическом контексте, генерируется ошибка времени компиляции;

- в противном случае пусть *p* представляет собой выражение *Primary* или *Expression Name*, непосредственно предшествующее "`.super`", и пусть *O* — наиболее глубоко лексически вложенный класс класса *S*. Если тип *p* не является ни *O*, ни подклассом *O* или если тип *p* не является доступным (§6.6), генерируется ошибка времени компиляции.

Типы исключений, которые может генерировать инструкция явного вызова конструктора, определены в §11.2.2.

При вычислении инструкции вызова альтернативного конструктора сначала вычисляются аргументы конструктора слева направо, как и при вызове обычного метода; затем выполняется вызов конструктора.

Вычисление инструкции вызова конструктора суперкласса — более сложный процесс.

1. Пусть *i* — создаваемый экземпляр. Непосредственно охватывающий *i* экземпляр по отношению к *S* (если таковой имеется) определяется следующим образом.
   - Если *S* не является внутренним классом или если объявление *S* находится в статическом контексте, не существует непосредственно охватывающего *i* экземпляра по отношению к *S*.
   - Если вызов конструктора суперкласса неквалифицированный, то *S* с необходимостью является локальным классом или внутренним классом-членом.

     Пусть *O* представляет собой непосредственно охватывающий *S* класс и пусть *n* — целое число, такое, что *O* представляет собой *n*-е лексически охватывающее *C* объявление типа.

     Непосредственно охватывающий *i* экземпляр по отношению к *S* является *n*-м лексически охватывающим `this` экземпляром.
   - Если вызов конструктора суперкласса квалифицированный, то вычисляется выражение *Primary* или *ExpressionName*, *p*, непосредственно предшествующее "`.super`".

     Если вычисление *p* дает `null`, генерируется исключение `NullPointerException`, и этим завершается вызов конструктора суперкласса.

     В противном случае результат этого вычисления представляет собой непосредственно охватывающий *i* экземпляр по отношению к *S*.
2. После определения непосредственно охватывающего *i* экземпляра по отношению к *S* (если таковой имеется) вычисление инструкции вызова конструктора суперкласса продолжается путем вычисления аргументов конструктора слева направо, как и в обычном вызове метода; затем вызывается сам конструктор.
3. Наконец, если инструкция вызова конструктора суперкласса завершается нормально, то выполняются все инициализаторы переменных экземпляра в *C* и все инициализаторы экземпляра *C*. Если инициализатор экземпляра или инициализатор переменной экземпляра *I* текстуально предшествует другому инициализатору экземпляра или инициализатору переменной экземпляра *J*, то *I* выполняется перед *J*.

   Инициализаторы переменных экземпляра и инициализаторы экземпляра выполняются вне зависимости от того, был ли вызов конструктора суперкласса осуществлен явной инструкцией вызова конструктора или предоставлен автоматически.

(Вызов альтернативного конструктора не осуществляет это дополнительное неявное выполнение.)

Пример 8.8.7.1-1. Ограничения инструкций явного вызова конструкторов

В первый конструктор `ColoredPoint` из примера к §8.8.7 внесены некоторые изменения.

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, color); // Замена на color вместо WHITE
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

Это приводит к генерации ошибки времени компиляции, поскольку переменная экземпляра `color` не может быть использована инструкцией явного вызова конструктора.

Пример 8.8.7.1-2. Вызов квалифицированного конструктора суперкласса

В приведенном ниже исходном тексте `ChildOfInner` не имеет лексически охватывающего объявления типа, так что экземпляр `ChildOfInner` не имеет охватывающего экземпляра. Однако суперкласс `ChildOfInner` (`Inner`) имеет лексически охватывающее объявление типа (`Outer`), и экземпляр `Inner` должен иметь охватывающий экземпляр `Outer`. Охватывающий экземпляр `Outer` определяется тогда, когда создается экземпляр `Inner`. Следовательно, когда мы создаем экземпляр `ChildOfInner`, который неявно является экземпляром `Inner`, мы должны предоставить и охватывающий экземпляр `Outer`, сделав это посредством инструкции вызова квалифицированного суперкласса в конструкторе `ChildOfInner`. Экземпляр `Outer` создает непосредственно охватывающий экземпляр `ChildOfInner` экземпляр `Inner`.

```
class Outer {
    class Inner {}
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner() { (new Outer()).super(); }
}
```

Возможно, это покажется удивительным, но один и тот же экземпляр `Outer` может служить непосредственно охватывающим экземпляром `ChildOfInner` по отношению к `Inner` *для нескольких экземпляров* `ChildOfInner`. Эти экземпляры

ChildOfInner неявно связаны с одним и тем же экземпляром Outer. Приведенная далее программа достигает этого с помощью передачи экземпляра Outer конструктору ChildOfInner, который использует этот экземпляр в инструкции вызова квалифицированного конструктора суперкласса. Правила для инструкции явного вызова конструктора не запрещают использование формальных параметров конструктора, который содержит эту инструкцию.

```
class Outer {
    int secret = 5;
    class Inner {
        int getSecret()        { return secret; }
        void setSecret(int s) { secret = s; }
    }
}

class ChildOfInner extends Outer.Inner {
    ChildOfInner(Outer x) { x.super(); }
}

public class Test {
    public static void main(String[] args) {
        Outer x = new Outer();
        ChildOfInner a = new ChildOfInner(x);
        ChildOfInner b = new ChildOfInner(x);
        System.out.println(b.getSecret());
        a.setSecret(6);
        System.out.println(b.getSecret());
    }
}
```

Вывод этой программы имеет вид

```
5
6
```

Результат заключается в том, что манипуляция переменными экземпляров в общем экземпляре Outer видима через ссылки на различные экземпляры ChildOf Inner, даже несмотря на то, что такие ссылки не являются псевдонимами в общепринятом смысле этого слова.

§8.8.8. Перегрузка конструкторов

Перегрузка конструкторов идентична по поведению перегрузке методов (§8.4.9). Перегрузка разрешается во время компиляции каждым выражением создания экземпляра класса (§15.9).

§8.8.9. Конструктор по умолчанию

Если класс не содержит объявлений конструкторов, то неявно объявляется конструктор по умолчанию. Конструктор по умолчанию для класса верхнего уровня, класса-члена или локального класса имеет следующий вид.

- Конструктор по умолчанию имеет ту же доступность, что и класс (§6.6).
- Конструктор по умолчанию не имеет формальных параметров, за исключением внутреннего класса-члена, не являющегося `private`, когда конструктор по умолчанию неявно объявляет один формальный параметр, представляющий охватывающий экземпляр класса (§8.8.1, §15.9.2, §15.9.3).
- Конструктор по умолчанию не имеет конструкции `throws`.
- Если объявленный класс является изначальным классом `Object`, конструктор по умолчанию имеет пустое тело. В противном случае конструктор по умолчанию просто вызывает конструктор суперкласса без аргументов.

Вид конструктора по умолчанию для анонимного класса определен в §15.9.5.1.

Если конструктор по умолчанию объявлен неявно, а у суперкласса нет доступного конструктора, который не получает аргументов и не имеет конструкции `throws`, генерируется ошибка времени компиляции.

Пример 8.8.9-1. Конструкторы по умолчанию

Объявление

```
public class Point {
    int x, y;
}
```

эквивалентно объявлению

```
public class Point {
    int x, y;
    public Point() { super(); }
}
```

в котором конструктор по умолчанию объявлен как `public`, поскольку как `public` объявлен сам класс `Point`.

Пример 8.8.9-2. Доступность конструкторов и классов

Правило, согласно которому конструктор по умолчанию некоторого класса имеет ту же доступность, что и сам класс, простое и интуитивно понятное. Заметим, однако, что отсюда не вытекает доступность конструктора в случае доступности класса. Рассмотрим следующий код.

```
package p1;
public class Outer {
    protected class Inner {}
}

package p2;
class SonOfOuter extends p1.Outer {
    void foo() {
        new Inner(); // Ошибка времени компиляции
    }
}
```

Конструктор класса `Inner` защищенный (`protected`). Однако конструктор является `protected` по отношению к `Inner`, в то время как `Inner` является `protected` по отношению к классу `Outer`. Так что `Inner` доступен в `SonOf Outer`, поскольку тот является подклассом `Outer`. Конструктор же `Inner` в `SonOfOuter` недоступен, поскольку класс `SonOfOuter` не является подклассом класса `Inner`! Следовательно, хотя класс `Inner` и доступен, его конструктор по умолчанию — нет.

§8.8.10. Предупреждение инстанцирования класса

Класс может быть спроектирован таким образом, чтобы не позволить коду вне объявления этого класса создавать его экземпляры. Для этого в классе должен быть объявлен хотя бы один конструктор, чтобы предупредить создание конструктора по умолчанию, и все конструкторы этого класса должны быть объявлены как `private`.

Класс, объявленный как `public`, может аналогично предупреждать создание своих экземпляров вне пакета, объявляя как минимум один конструктор (для предупреждения создания конструктора по умолчанию с открытым доступом) и не объявляя ни один конструктор как `public`.

Пример 8.8.10.1. Предупреждение инстанцирования с помощью доступа конструктора

```
class ClassOnly {
    private ClassOnly() { }
    static String just = "Очень одинокий";
}
```

Здесь класс `ClassOnly` не может быть инстанцирован, в то время как в приведенном ниже коде класс `PackageOnly` может быть инстанцирован только в пакете `just`, в котором он объявлен.

```
package just;
public class PackageOnly {
    PackageOnly() { }
    String[] justDesserts = { "cheesecake", "ice cream" };
}
```

§8.9. Перечисления

Объявление перечисления определяет новый *тип перечисления*.

EnumDeclaration:
{ClassModifier} `enum` *Identifier [Superinterfaces] EnumBody*

Объявление перечисления не может иметь модификатор `abstract` или `final`, иначе генерируется ошибка времени компиляции.

Объявление перечисления неявно объявляется как `final`, если только оно не содержит по крайней мере одну константу перечисления, имеющую тело класса (§8.9.1).

Вложенные типы перечислений неявно являются `static`. Разрешено также явное объявление вложенных типов перечислений как `static`.

> Отсюда вытекает невозможность объявления типа перечисления в теле внутреннего класса (§8.1.3), так как внутренний класс не может иметь статические члены, за исключением константных переменных.

Если одно и то же ключевое слово появляется в объявлении перечисления в качестве модификатора дважды, генерируется ошибка времени компиляции.

Непосредственным суперклассом типа перечисления *E* является `Enum<`*E*`>` (§8.1.4).

Тип перечисления не имеет экземпляров, кроме определенных его константами перечисления. В случае попытки явного инстанцирования типа перечисления (§15.9.1) генерируется ошибка времени компиляции.

> Метод `final clone` в `Enum` гарантирует, что константы перечисления никогда не могут быть клонированы, а особый режим механизма сериализации гарантирует, что дубликаты экземпляров не будут созданы в результате десериализации. Рефлективное инстанцирование типов перечисления запрещено. Все вместе это гарантирует отсутствие экземпляров типа перечисления, помимо определенных константами перечисления.

§8.9.1. Константы перечислений

Тело объявления перечисления может содержать *константы перечисления*. Константа перечисления определяет экземпляр типа перечисления.

EnumBody:
    { *[EnumConstantList] [,] [EnumBodyDeclarations]* }

EnumConstantList:
    EnumConstant {, EnumConstant}

EnumConstant:
    {EnumConstantModifier} Identifier [([ArgumentList])] [ClassBody]

EnumConstantModifier:
    Annotation

Для удобства ниже приведена продукция из §15.12.

ArgumentList:
    Expression {, Expression}

Правила для модификаторов аннотаций в объявлениях констант перечислений определены в §9.7.4 и §9.7.5.

Identifier в *EnumConstant* может использоваться в имени для обращения к константе перечисления.

Область видимости и затенение константы перечисления рассматриваются в §6.3 и §6.4.

За константой перечисления могут следовать аргументы, которые передаются конструктору перечисления при создании константы во время инициализации класса, как

описано ниже в данном разделе. Вызываемый конструктор выбирается с использованием обычных правил перегрузки (§15.12.2). Если аргументы опущены, считается, что передан пустой список аргументов.

Необязательное тело класса константы перечисления неявно определяет объявление анонимного класса (§15.9.5), который расширяет непосредственно охватывающий тип перечисления. Тело класса регулируется обычными правилами анонимных классов; в частности, оно не может содержать каких-либо конструкторов. Методы экземпляров, объявленные в этих телах классов, могут вызываться вне охватывающего типа перечисления, только если они перекрывают доступные методы охватывающего типа перечисления.

Если тело класса константы перечисления объявляет абстрактный метод, генерируется ошибка времени компиляции.

Поскольку имеется только по одному экземпляру каждой константы перечисления, при сравнении двух ссылок на объекты разрешается использовать оператор == вместо метода `equals`, если известно, что по крайней мере одна из них является ссылкой на константу перечисления.

> Метод `equals` в `Enum` представляет собой объявленный как `final` метод, который просто вызывает `super.equals` для своих аргументов и возвращает результат, тем самым выполняя сравнение на проверку тождественности.

§8.9.2. Объявления тел перечислений

В дополнение к константам перечислений тело объявления перечисления может содержать конструктор и объявления членов, а также инициализаторы экземпляра и статические инициализаторы.

EnumBodyDeclarations:
    ; *{ClassBodyDeclaration}*

Для удобства ниже приведены продукции из §8.1.6.

ClassBodyDeclaration:
    ClassMemberDeclaration
    InstanceInitializer
    StaticInitializer
    ConstructorDeclaration

ClassMemberDeclaration:
    FieldDeclaration
    MethodDeclaration
    ClassDeclaration
    InterfaceDeclaration
    ;

Любые объявления конструкторов или членов в теле объявления перечисления применимы к типу перечисления в точности так, как если бы они находились в теле класса обычного объявления класса, если явно не указано иное.

Если конструктор в объявлении перечисления объявлен как `public` или `protected`, генерируется ошибка времени компиляции.

Если объявление конструктора в объявлении перечисления содержит инструкцию вызова конструктора суперкласса (§8.8.7.1), генерируется ошибка времени компиляции.

При обращении к статическому полю типа перечисления из конструкторов, инициализаторов экземпляров или выражений инициализаторов переменных экземпляров типа перечисления, если только поле не представляет собой константную переменную (§4.12.4), генерируется ошибка времени компиляции.

В объявлении перечисления объявление конструктора без модификаторов доступа является `private`.

Если тип перечисления не имеет объявлений конструкторов, то неявно объявляется конструктор по умолчанию. Конструктор по умолчанию является `private`, без формальных параметров и конструкции `throws`.

> На практике компилятор, скорее всего, отобразит тип `Enum` с помощью объявления параметров `String` и `int` в конструкторе по умолчанию типа перечисления. Однако эти параметры не определены как "неявно объявленные", поскольку различные компиляторы не должны согласовывать вид конструктора по умолчанию. Только компилятор типа перечисления знает, как инстанцировать константы перечисления; другие компиляторы могут просто положиться на неявно объявленные как `public static` поля типа перечисления (§8.9.3) безотносительно того, как именно они инициализируются.

Если объявление перечисления E имеет абстрактный метод m в качестве члена, то если только E не имеет как минимум одной константы перечисления и все константы перечисления E не имеют тела класса, которые предоставляют конкретные реализации m, генерируется ошибка времени компиляции.

Если объявление перечисления объявляет финализатор (§12.6), генерируется ошибка времени компиляции. Экземпляр типа перечисления не может быть финализирован.

> **Пример 8.9.2-1. Объявления тела перечисления**
>
> ```
> enum Coin {
>     PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
>     Coin(int value) { this.value = value; }
>
>     private final int value;
>     public int value() { return value; }
> }
> ```
>
> Каждая константа перечисления имеет различное значение в поле `value`, переданное через конструктор. Поле представляет достоинство, в центах, американской монеты. Заметим, что нет никаких ограничений на тип или количество параметров, которые могут быть объявлены конструктором типа перечисления.

Пример 8.9.2-2. Ограничения на самообращения констант перечислений

Без этого правила, касающегося доступа к статическим полям, кажущийся разумным код будет сбоить во время выполнения из-за зацикливания инициализации, присущей типам перечислений. (Зацикливание имеется в любом классе с "самотипизированным" статическим полем.) Вот пример такого сбойного кода.

```
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;
    Color() { colorMap.put(toString(), this); }

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
}
```

Статическая инициализация этого типа перечисления приводила бы к генерации исключения `NullPointerException`, поскольку объявленная как `static` переменная `colorMap` инициализируется при выполнении конструкторов для констант перечисления. Рассмотренное выше ограничение гарантирует, что этот код не будет компилироваться. Заметим, что пример очень легко изменить так, чтобы он корректно работал.

```
import java.util.Map;
import java.util.HashMap;

enum Color {
    RED, GREEN, BLUE;

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
    static {
        for (Color c : Color.values())
            colorMap.put(c.toString(), c);
    }
}
```

Исправленная версия корректна, так как статическая инициализация выполняется сверху вниз.

§8.9.3. Члены перечислений

Членами типа перечисления *E* является следующее.

- Члены, объявленные в теле объявления *E*.
- Члены, унаследованные от `Enum<`*E*`>`.
- Для каждой константы перечисления *c*, объявленной в теле объявления *E*, *E* имеет неявно объявленное `public static final` поле типа *E* с тем же именем, что и у *c*. Поле имеет инициализатор переменной, состоящий из *c*, и аннотировано той же аннотацией, что и *c*.

Эти поля неявно объявлены в том же порядке, что и соответствующие константы перечисления, до любых статических полей, явно объявленных в теле объявления *E*.

Говорят, что константа перечисления *создана*, когда инициализировано соответствующее неявно объявленное поле.

- Следующий исходный текст неявно объявляет методы.

```
/**
* Возвращает массив, содержащий константы данного типа
* перечисления в порядке объявления. Этот метод может
* использоваться для итерации по всем константам
* следующим образом:
*
* for(E c : E.values())
* System.out.println(c);
*
* @Возвращает массив, содержащий константы данного типа
* перечисления в порядке объявления
*/
public static E[] values();

/**
* Возвращает константу перечисления данного типа с
* определенным именем.
* Строка должна точно соответствовать идентификатору,
* использованному в объявлении константы перечисления
* данного типа. (Лишние пробелы не разрешаются.)
*
* @Возвращает константу перечисления данного типа с
* определенным именем
* @throws IllegalArgumentException, если этот тип
* перечисления не имеет константы с указанным именем
*/
public static E valueOf(String name);
```

Отсюда следует, что объявление типа перечисления *E* не может ни содержать поля, которые конфликтуют с неявно объявленными полями, соответствующими константам перечисления *E*, ни содержать методы, которые конфликтуют с неявно объявленными методами или перекрывают финальные методы класса `Enum<`*E*`>`.

Пример 8.9.3-1. Итерация и константы перечисления в расширенном цикле `for`

```
public class Test {
    enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        for (Season s : Season.values())
            System.out.println(s);
    }
}
```

Вывод этой программы имеет следующий вид.

```
WINTER
SPRING
SUMMER
FALL
```

Пример 8.9.3-2. Выбор с использованием констант перечисления

Конструкция `switch` (§14.11) полезна при имитации добавления метода к типу перечисления извне типа. Приведенный ниже пример "добавляет" метод `color` к типу `Coin` из §8.9.2 и выводит таблицу монет, их достоинство и цвет.

```
class Test {
    enum CoinColor { COPPER, NICKEL, SILVER }

    static CoinColor color(Coin c) {
        switch (c) {
            case PENNY:
              return CoinColor.COPPER;
            case NICKEL:
              return CoinColor.NICKEL;
            case DIME: case QUARTER:
              return CoinColor.SILVER;
            default:
              throw new AssertionError("Unknown coin: "+c);
        }
    }

    public static void main(String[] args) {
        for (Coin c : Coin.values())
            System.out.println(c + "\t\t" +
                    c.value() + "\t" + color(c));
    }
}
```

Вывод данной программы имеет следующий вид.

```
PENNY      1      COPPER
NICKEL     5      NICKEL
DIME      10      SILVER
QUARTER   25      SILVER
```

Пример 8.9.3-3. Константы перечислений с телами классов

```
enum Operation {
    PLUS {
        double eval(double x, double y) { return x + y; }
    },
    MINUS {
        double eval(double x, double y) { return x - y; }
    },
    TIMES {
```

```
        double eval(double x, double y) { return x * y; }
    },
    DIVIDED_BY {
        double eval(double x, double y) { return x / y; }
    };

    // Каждая константа поддерживает арифметическую операцию
    abstract double eval(double x, double y);

    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y +
                               " = " + op.eval(x, y));
    }
}
```

Тела классов добавляют к константам перечисления определенное поведение. Вывод программы имеет следующий вид.

```
java Operation 2.0 4.0
2.0 PLUS 4.0 = 6.0
2.0 MINUS 4.0 = -2.0
2.0 TIMES 4.0 = 8.0
2.0 DIVIDED_BY 4.0 = 0.5
```

Приведенный шаблон гораздо безопаснее использования конструкции `switch` в базовом типе (`Operation`), поскольку шаблон исключает возможность забыть добавить поведение к новой константе (так как объявление перечисления будет приводить к генерации ошибки времени компиляции).

Пример 8.9.3-4. Множественные типы перечислений

В приведенной далее программе класс игральных карт построен поверх двух простых типов перечислений.

```
import java.util.List;
import java.util.ArrayList;
class Card implements Comparable<Card>,
                       java.io.Serializable {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN,
                       EIGHT, NINE, TEN,JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    public Rank rank() { return rank; }
    public Suit suit() { return suit; }

    private Card(Rank rank, Suit suit) {
```

```
        if (rank == null || suit == null)
            throw new NullPointerException(rank + ", " + suit);
        this.rank = rank;
        this.suit = suit;
    }

    public String toString() { return rank + " of " + suit; }

    // Сначала сортировка по масти, затем — по достоинству
    public int compareTo(Card c) {
        int suitCompare = suit.compareTo(c.suit);
        return (suitCompare != 0 ?
                suitCompare :
                rank.compareTo(c.rank));
    }

    private static final List<Card> prototypeDeck =
        new ArrayList<Card>(52);

    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                prototypeDeck.add(new Card(rank, suit));
    }

    // Возврат новой колоды
    public static List<Card> newDeck() {
        return new ArrayList<Card>(prototypeDeck);
    }
}
```

Приведенная далее программа использует класс `Card`. Она получает из командной строки два параметра, представляющие количество игроков и количество карт у каждого игрока.

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }

    /**
    * Возвращает новый ArrayList, состоящий из последних n
    * элементов колоды, которые из колоды удаляются.
```

```
     * Возвращаемый список отсортирован в соответствии с
     * естественным упорядочением элементов.
     */
     public static <E extends Comparable<E>>
     ArrayList<E> dealHand(List<E> deck, int n) {
         int deckSize = deck.size();
         List<E> handView = deck.subList(deckSize - n,
deckSize);
         ArrayList<E> hand = new ArrayList<E>(handView);
         handView.clear();
         Collections.sort(hand);
         return hand;
     }
}
```

Вывод данной программы имеет следующий вид.

```
java Deal 4 3
[DEUCE of CLUBS, SEVEN of CLUBS, QUEEN of DIAMONDS]
[NINE of HEARTS, FIVE of SPADES, ACE of SPADES]
[THREE of HEARTS, SIX of HEARTS, TEN of SPADES]
[TEN of CLUBS, NINE of DIAMONDS, THREE of SPADES]
```

ГЛАВА 9

Интерфейсы

Объявление интерфейса вводит новый ссылочный тип, членами которого являются классы, интерфейсы, константы и методы. Этот тип не имеет переменных экземпляров и обычно объявляет один или несколько абстрактных методов; в противном случае несвязанные классы могут реализовывать интерфейс путем предоставления реализаций его абстрактных методов. Интерфейсы не могут быть инстанцированы непосредственно.

Вложенным интерфейсом является любой интерфейс, объявление которого находится в теле другого класса или интерфейса.

Интерфейс верхнего уровня представляет собой интерфейс, не являющийся вложенным.

Различают два вида интерфейсов — обычные интерфейсы и типы аннотаций.

В этой главе рассматривается общая семантика всех интерфейсов — как обычных интерфейсов, верхнего уровня (§7.6) и вложенных (§8.5, §9.5), так и типов аннотаций (§9.6). Детали, специфичные для определенных типов интерфейсов, обсуждаются в разделах, посвященных этим конструкциям.

Программы могут использовать интерфейсы, чтобы сделать необязательным для связанных классов наличие общего абстрактного суперкласса или добавление методов к классу `Object`.

Интерфейс может быть объявлен как *непосредственное расширение* одного или нескольких других интерфейсов. Это означает, что он наследует все типы-члены, методы экземпляров и константы интерфейсов, которые он расширяет, за исключением тех типов-членов и констант, которые он может перекрывать или скрывать.

Класс может быть объявлен как *непосредственно реализующий* один или несколько интерфейсов. Это означает, что любой экземпляр класса реализует все абстрактные методы указанного интерфейса или интерфейсов. Класс обязательно реализует все интерфейсы, которые реализуют его прямые суперклассы и прямые суперинтерфейсы. Это (множественное) наследование интерфейсов позволяет объектам поддерживать несколько общих функциональностей без разделяемого суперкласса.

Переменная, объявленный тип которой является типом интерфейса, может иметь в качестве значения ссылку на любой экземпляр класса, который реализует указанный интерфейс. Недостаточно, чтобы класс реализовал все абстрактные методы интерфейса; класс или один из его суперклассов должен быть объявлен как реализующий интерфейс, иначе класс не рассматривается как реализующий интерфейс.

§9.1. Объявления интерфейсов

Объявление интерфейса определяет новый именованный ссылочный тип. Имеется два вида объявлений интерфейсов — *объявления обычных интерфейсов* и *объявления аннотированных типов* (§9.6).

InterfaceDeclaration:
NormalInterfaceDeclaration
AnnotationTypeDeclaration

NormalInterfaceDeclaration:
{InterfaceModifier} `interface` *Identifier [TypeParameters]*
[ExtendsInterfaces] InterfaceBody

Identifier в объявлении интерфейса определяет имя последнего.

Если интерфейс имеет то же самое простое имя, что и любой из его охватывающих классов или интерфейсов, генерируется ошибка времени компиляции.

Область видимости и затенение объявления интерфейса рассматриваются в §6.3 и §6.4.

§9.1.1. Модификаторы интерфейсов

Объявление интерфейса может включать *модификаторы интерфейсов*.

InterfaceModifier: одно из
Annotation `public protected private`
`abstract static strictfp`

Правила для модификаторов аннотаций в объявлениях интерфейсов определены в §9.7.4 и §9.7.5.

Модификатор доступа `public` (§6.6) относится к каждому виду объявления интерфейса.

Модификаторы доступа `protected` и `private` относятся только к интерфейсам-членам в непосредственно охватывающем объявлении класса или перечисления (§8.5.1).

Модификатор `static` относится только к интерфейсам-членам (§8.5.1, §9.5), но не к интерфейсам верхнего уровня (§7.6).

Если одно и то же ключевое слово появляется в объявлении интерфейса в качестве модификатора больше одного раза, генерируется ошибка времени компиляции.

Если в объявлении интерфейса имеется два или более (различных) модификаторов интерфейса, то принято, хотя это и не является обязательным, чтобы они находились в порядке, согласующемся с показанным выше в продукции для *InterfaceModifier*.

§9.1.1.1. Абстрактные интерфейсы

Каждый интерфейс неявно объявлен как `abstract`.

Этот модификатор устаревший и не должен использоваться в новых программах.

§9.1.1.2. Интерфейсы `strictfp`

Модификатор `strictfp` применяется для того, чтобы сделать все выражения `float` или `double` в объявлении интерфейса FP-строгими (§15.4) явным образом.

Это означает, что все вложенные типы, объявленные в интерфейсе, неявно являются объявленными как `strictfp`.

§9.1.2. Обобщенные интерфейсы и параметры типов

Интерфейс является *обобщенным*, если он объявляет одну или несколько переменных типа (§4.4).

Эти переменные типа известны как *параметры типа* интерфейса. Раздел параметров типа следует за именем интерфейса и заключен в угловые скобки.

Для удобства далее приведены продукции из §8.1.2 и §4.4.

TypeParameters:
    < *TypeParameterList* >

TypeParameterList:
    TypeParameter {, *TypeParameter*}

TypeParameter:
    {TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
    Annotation

TypeBound:
    `extends` *TypeVariable*
    `extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
    & *InterfaceType*

Правила для модификаторов аннотаций в объявлениях параметров типа определены в §9.7.4 и §9.7.5.

В разделе параметров типа интерфейса переменная типа T *непосредственно зависит* от переменной типа S, если S является границей T, в то время как T *зависит* от S, если либо T непосредственно зависит от S, либо T непосредственно зависит от переменной типа U, которая зависит от S (с рекурсивным использованием данного определения).

Если переменная типа в разделе параметров типа интерфейса зависит сама от себя, генерируется ошибка времени компиляции.

Область видимости параметра типа интерфейса определена в §6.3.

Обращение к параметру типа интерфейса I в любом месте в объявлении поля или типа-члена I ведет к генерации ошибки времени компиляции.

Объявление обобщенного интерфейса определяет множество параметризованных типов (§4.5), по одному для каждой возможной параметризации раздела параметров

аргументами типа. Все эти параметризованные типы совместно используют интерфейс во время выполнения.

§9.1.3. Суперинтерфейсы и подынтерфейсы

Если в объявлении имеется конструкция `extends`, то объявляемый интерфейс расширяет каждый из прочих именованных интерфейсов, а потому наследует типы-члены, методы и константы каждого из прочих именованных интерфейсов.

Эти именованные интерфейсы являются непосредственными суперинтерфейсами объявленного интерфейса.

Любой класс, реализующий (т.е. содержащий конструкцию `implements` в объявлении) объявленный интерфейс, считается также реализующим интерфейсы, которые расширяют данный интерфейс.

ExtendsInterfaces:
`extends` *InterfaceTypeList*

Приведенная далее продукция взята из §8.1.5 для облегчения понимания читателем.

InterfaceTypeList:
InterfaceType {`,` *InterfaceType*}

Каждый *InterfaceType* в конструкции `extends` объявления интерфейса должен именовать доступный тип интерфейса (§6.6), иначе генерируется ошибка времени компиляции.

Если *InterfaceType* имеет аргументы типов, он должен описывать корректно сформированный параметризованный тип (§4.5), и ни один из аргументов типа не может быть аргументом с символом подстановки, иначе генерируется ошибка времени компиляции.

Для данного (возможно, обобщенного) объявления интерфейса для $I<F_1,\ldots,F_n>$ ($n \geq 0$) *непосредственными суперинтерфейсами* типа интерфейса $I<F_1,\ldots,F_n>$ являются типы, указанные в конструкции `extends` объявления I, если таковая присутствует.

Пусть $I<F_1,\ldots,F_n>$ ($n \geq 0$) является объявлением обобщенного интерфейса. Непосредственными суперинтерфейсами типа параметризованного интерфейса $I<T_1,\ldots,T_n>$, где T_i ($1 \leq i \leq n$) представляет собой тип, являются все типы $J<U_1\ \theta,\ldots,U_k\ \theta>$, где $J<U_1,\ldots,U_k>$ является непосредственным суперинтерфейсом $I<F_1,\ldots,F_n>$, а θ представляет собой подстановку $[F_1:=T_1,\ldots,F_n:=T_n]$.

Отношение *суперинтерфейса* представляет собой транзитивное замыкание отношения непосредственного суперинтерфейса. Интерфейс K представляет собой суперинтерфейс интерфейса I, если верно любое из приведенных далее утверждений.

- K является непосредственным суперинтерфейсом I.
- Существует интерфейс J, такой, что K является суперинтерфейсом J, а J является суперинтерфейсом I, с рекурсивным применением данного определения.

Интерфейс I называется *подынтерфейсом* K, когда K является суперинтерфейсом I.

В то время как каждый класс является расширением класса `Object`, нет единого интерфейса, для которого все интерфейсы являются расширениями.

Интерфейс I *непосредственно зависит* от типа T, если T упоминается в конструкции `extends` I либо как суперинтерфейс, либо как квалификатор в имени суперинтерфейса.

Интерфейс `I` *зависит* от типа `T`, если выполняется любое из следующих условий.

- `I` непосредственно зависит от `T`.
- `I` непосредственно зависит от класса `C`, который зависит (§8.1.5) от `T`.
- `I` непосредственно зависит от интерфейса `J`, который зависит от `T` (с рекурсивным применением данного определения).

Если интерфейс зависит сам от себя, генерируется ошибка времени компиляции.

Если во время выполнения в процессе загрузки интерфейсов (§12.2) обнаруживаются циклически объявленные интерфейсы, генерируется исключение `ClassCircularityError`.

§9.1.4. Тело интерфейса и объявления членов

Тело интерфейса может объявлять члены интерфейса, т.е. поля (§9.3), методы (§9.4), классы (§9.5) и интерфейсы (§9.5).

InterfaceBody:
{ *{InterfaceMemberDeclaration}* }

InterfaceMemberDeclaration:
ConstantDeclaration
InterfaceMethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

Область видимости объявления члена `m`, объявленного в типе интерфейса `I` или унаследованного им, определена в §6.3.

§9.2. Члены интерфейса

Существуют следующие члены интерфейса.

- Члены, объявленные в теле интерфейса (§9.1.4).
- Члены, унаследованные от непосредственных суперинтерфейсов (§9.1.3).
- Если интерфейс не имеет непосредственных суперинтерфейсов, то этот интерфейс неявно объявляет `public abstract` метод-член `m` с сигнатурой `s`, возвращаемым типом `r` и конструкцией `throws t`, соответствующий каждому `public`-методу экземпляра `m` с сигнатурой `s`, возвращаемым типом `r` и конструкцией `throws t`, объявленному в `Object`, если только абстрактный метод с той же сигнатурой, тем же возвращаемым типом и совместимой конструкцией `throws` не объявлен интерфейсом явно.

  Если интерфейс явно объявляет такой метод `m`, то в случае, если `m` объявлен в `Object` как `final`, генерируется ошибка времени компиляции.

  Отсюда вытекает, что если интерфейс объявляет метод с сигнатурой, являющейся эквивалентной в смысле перекрытия (§8.4.2), методу `Object`, объявленному как

`public`, но имеющему другой возвращаемый тип или несовместимую конструкцию `throws`, генерируется ошибка времени компиляции.

Интерфейс наследует от интерфейсов, которые он расширяет, все члены этих интерфейсов, за исключением полей, классов и интерфейсов, которые он скрывает; абстрактных методов или методов по умолчанию, которые он перекрывает (§9.4.1); и статических методов.

Поля, методы и типы-члены типа интерфейса могут иметь одно и то же имя, поскольку они используются в разных контекстах и неоднозначности для них разрешаются разными процедурами поиска (§6.5). Однако это не приветствуется с точки зрения стиля.

§9.3. Объявления полей (констант)

ConstantDeclaration:
{ConstantModifier} UnannType VariableDeclaratorList ;

ConstantModifier: одно из
Annotation `public`
`static final`

Продукцию для *UnannType* вы найдете в §8.3. Далее для удобства приведены продукции из §4.3 и §8.3.

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]` *}*

VariableInitializer:
Expression
ArrayInitializer

Правила для модификаторов аннотаций в объявлениях полей интерфейсов определены в §9.7.4 и §9.7.5.

Каждое объявление поля в теле интерфейса неявно объявлено как `public`, `static` и `final`. Для таких полей разрешается избыточное указание любого из этих модификаторов (или всех).

> Если в объявлении поля находятся два или большее количество (различных) модификаторов полей, то обычно (хотя это и не требуется) они перечисляются в порядке, согласующемся с показанным выше в продукции для *ConstantModifier*.

Если одно и то же ключевое слово встречается в объявлении поля в качестве модификатора более одного раза, генерируется ошибка времени компиляции.

Если тело объявления интерфейса объявляет два поля с одним и тем же именем, генерируется ошибка времени компиляции.

Объявленный тип поля описывается нетерминалом *UnannType*, который находится в объявлении поля и за которым следует любая пара скобок, а за ней — *Identifier*.

Если интерфейс объявляет поле с определенным типом, то это объявление поля *скрывает* любые доступные объявления полей с тем же именем в суперинтерфейсах этого интерфейса.

Интерфейс может наследовать больше одного поля с одним и тем же именем. Сама по себе такая ситуация не приводит к ошибке времени компиляции. Однако при любой попытке в теле интерфейса обратиться к такому полю по простому имени в силу неоднозначности генерируется ошибка времени компиляции.

Могут быть разные пути, которыми одно и то же поле наследуется интерфейсом. В такой ситуации поле рассматривается как унаследованное только один раз, и обращаться к нему можно по простому имени без какой бы то ни было неоднозначности.

Пример 9.3-1. Неоднозначность унаследованных полей

Если два поля с одним и тем же именем унаследованы интерфейсом, например, потому что два из его непосредственных суперинтерфейсов объявляют поля с таким именем, то в результате получаем один неоднозначный член. Любое использование этого неоднозначного члена ведет к генерации ошибки времени компиляции.

```
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
interface PrintColors extends BaseColors {
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

В приведенной программе интерфейс `LotsOfColors` наследует два поля с именем `YELLOW`. Это не приводит к неприятностям, пока интерфейс не содержит обращений к полю `YELLOW` по его простому имени. (Такое обращение может иметь место в инициализаторе переменной для поля.)

Даже если бы интерфейс `PrintColors` определял значение `YELLOW` как `3`, а не как `8`, обращение к полю `YELLOW` в интерфейсе `LotsOfColors` все равно рассматривалось бы как неоднозначное.

Пример 9.3-2. Множественно унаследованные поля

Если одно поле наследуется несколько раз от одного и того же интерфейса, потому что, например, этот интерфейс и один из его непосредственных суперинтерфейсов расширяют интерфейс, объявляющий это поле, в результате получается только один член. Эта ситуация сама по себе не вызывает ошибку времени компиляции.

В предыдущем примере поля `RED`, `GREEN` и `BLUE` наследуются интерфейсом `LotsOfColors` более чем одним способом, через интерфейс `RainbowColors`, а также через интерфейс `PrintColors`, но обращение к полю `RED` в интерфейсе `LotsOfColors` не считается неоднозначным, поскольку имеется только одно фактическое объявление поля `RED`.

§9.3.1. Инициализация полей в интерфейсах

Каждый декларатор в объявлении поля интерфейса должен иметь инициализатор переменной, в противном случае генерируется ошибка времени компиляции.

Инициализатор не должен быть константным выражением (§15.28).

Если инициализатор поля интерфейса содержит ссылку на простое имя того же поля или другого поля, объявление которого располагается текстуально позже в том же интерфейсе, генерируется ошибка времени компиляции.

Если в выражении инициализации для поля интерфейса находится ключевое слово `this` (§15.8.3) или ключевое слово `super` (§15.11.2, §15.12), то, если только это не происходит в пределах тела анонимного класса (§15.9.5), генерируется ошибка времени компиляции.

Вычисление и присваивание инициализатора переменной выполняется только один раз, когда инициализируется интерфейс (§12.4.2).

Во время выполнения те поля интерфейсов, которые являются константными переменными (§4.12.4), инициализируются до прочих полей интерфейса. Это также относится и к полям `static final`, которые являются константными переменными в классах (§8.3.2). Эти поля являются “константами”, которые никогда не предоставляют свои начальные значения по умолчанию (§4.12.5) даже самым коварно написанным программам.

Пример 9.3.1-1. Предварительная ссылка на поле

```
interface Test {
    float f = j;
    int   j = 1;
    int   k = k + 1;
}
```

Эта программа приводит к двум ошибкам времени компиляции, поскольку обращение к `j` происходит в инициализации `f` до объявления `j` и поскольку инициализация `k` обращается к самому полю `k`.

§9.4. Объявления методов

InterfaceMethodDeclaration:
    {InterfaceMethodModifier} MethodHeader MethodBody

InterfaceMethodModifier: одно из
    Annotation `public`
    `abstract default static strictfp`

Далее для удобства приведены продукции из §8.4, §8.4.5 и §8.4.7.

MethodHeader:
    Result MethodDeclarator [Throws]
    TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
    UnannType
    `void`

MethodDeclarator:
    Identifier (*[FormalParameterList]*) *[Dims]*

MethodBody:
    Block
    ;

Правила для модификаторов аннотаций в объявлениях методов интерфейсов определены в §9.7.4 и §9.7.5.

Каждое объявление метода в теле интерфейса неявно объявлено как `public` (§6.6). Разрешено, но не приветствуется с точки зрения стиля, излишнее применение модификатора `public` для метода, объявленного в интерфейсе.

Метод по умолчанию представляет собой метод, который объявлен в интерфейсе с модификатором `default`; его тело всегда представлено блоком. Он предоставляет реализацию по умолчанию для любого класса, который реализует интерфейс без перекрытия метода. Методы по умолчанию отличны от конкретных методов (§8.4.3.1), которые объявляются в классах.

Интерфейс может объявлять статические методы, которые вызываются без ссылки на конкретный объект.

Использование имени параметра типа любого охватывающего объявления в заголовке или теле статического метода интерфейса приводит к генерации ошибки времени компиляции.

Влияние модификатора `strictfp` заключается в том, что все выражения `float` или `double` в теле метода по умолчанию или статического метода явно становятся FP-строгими (§15.4).

Метод интерфейса, у которого отсутствует модификатор `default` или `static`, неявно является абстрактным, так что его тело представлено точкой с запятой, а не блоком. Разрешено, но не рекомендуется с точки зрения стиля, избыточное применение модификатора `abstract` в объявлении такого метода.

Если одно и то же ключевое слово встречается в качестве модификатора у метода, объявленного в интерфейсе, более одного раза, генерируется ошибка времени компиляции.

Если метод, объявленный в интерфейсе, объявлен более чем с одним из модификаторов `abstract`, `default` или `static`, генерируется ошибка времени компиляции.

Если объявление абстрактного метода содержит ключевое слово `strictfp`, генерируется ошибка времени компиляции.

Если тело интерфейса объявляет, явно или неявно, два метода с эквивалентными в смысле перекрытия сигнатурами (§8.4.2), генерируется ошибка времени компиляции. Однако интерфейс может наследовать несколько абстрактных методов с такими сигнатурами (§9.4.1).

Метод в интерфейсе может быть обобщенным. Правила для параметров типа обобщенного метода в интерфейсе такие же, как и для обобщенного метода в классе (§8.4.4).

§9.4.1. Наследование и перекрытие

Интерфейс I *наследует* из своих непосредственных суперинтерфейсов все абстрактные методы и методы по умолчанию m, для которых выполняются все следующие условия.

- m является членом непосредственного суперинтерфейса J интерфейса I.
- Никакой метод, объявленный в I, не имеет сигнатуру, которая является подсигнатурой (§8.4.2) сигнатуры m.
- Не существует метода m', который является членом непосредственного суперинтерфейса J' интерфейса I (m отличен от m', а J отличен от J'), такого, что m' перекрывает из J' объявление метода m.

Обратите внимание, что методы перекрываются на основе анализа сигнатур. Если, например, интерфейс объявляет два `public`-метода с одним и тем же именем (§9.4.2) и подынтерфейс перекрывает один из них, то подынтерфейс продолжает наследовать другой метод.

Третий пункт выше препятствует повторному наследованию метода, который был перекрыт другим из суперинтерфейсов. Например, в приведенной далее программе `K` наследует `name()` от `I`, но `Child` наследует `name()` от `J`, а не от `K`. Дело в том, что `name()` из `J` перекрывает `I.name()`.

```
interface I {
    default String name() { return "unnamed"; }
}
interface J extends I {
    default String name() { return getClass().getName(); }
}
interface K extends I {}
interface Child extends J, K {}
```

Интерфейс не наследует статические методы своих суперинтерфейсов.

Если экземпляр I объявляет статический метод m, а сигнатура m представляет собой подсигнатуру метода экземпляра m' в суперинтерфейсе I, и m' был бы иначе доступен коду в I, то генерируется ошибка времени компиляции.

По сути, статический метод в интерфейсе не может “скрывать” метод экземпляра в суперинтерфейсе. Это похоже на правило из §8.4.8.2 в соответствии с которым статический метод класса не может скрывать метод экземпляра в суперклассе или суперинтерфейсе. Обратите внимание, что правило в §8.4.8.2 говорит о классе,

который "объявляет или наследует статический метод", в то время как приведенное выше правило говорит только об интерфейсе, который "объявляет статический метод", поскольку интерфейс не может наследовать статический метод. Обратите также внимание, что правило в §8.4.8.2 позволяет скрывать как методы экземпляра, так и статические методы суперклассов/суперинтерфейсов, в то время как правило, приведенное выше, рассматривает только методы экземпляра в суперинтерфейсах.

§9.4.1.1. Перекрытие (методом экземпляра)

Метод экземпляра m_1, объявленный в интерфейсе I, *перекрывает из* I другой метод экземпляра m_2, объявленный в интерфейсе J, тогда и только тогда, когда выполняются оба следующих условия:

- I является подынтерфейсом J;
- сигнатура m_1 является подсигнатурой (§8.4.2) сигнатуры m_2.

Наличие или отсутствие модификатора `strictfp` абсолютно не влияет на правила перекрытия методов. Например, метод, не являющийся FP-строгим, может перекрывать FP-строгий метод, как и FP-строгий метод может перекрывать метод, не являющийся FP-строгим.

Обратиться к перекрытому методу по умолчанию можно с помощью выражения вызова метода (§15.12), который содержит ключевое слово `super`, квалифицированное именем суперинтерфейса.

§9.4.1.2. Требования к перекрытию

Взаимоотношения между типом возвращаемого значения метода интерфейса и возвращаемыми типами любых перекрытых методов интерфейса определены в §8.4.8.3.

Взаимоотношения между конструкцией `throws` метода интерфейса и конструкциями `throws` любых перекрытых методов интерфейса определены в §8.4.8.3.

Взаимоотношения между сигнатурой метода интерфейса и сигнатурами любых перекрытых методов интерфейса определены в §8.4.8.3.

Если метод по умолчанию эквивалентен в смысле перекрытия методу класса `Object`, не объявленному как `private`, генерируется ошибка времени компиляции, поскольку любой класс, реализующий интерфейс, будет наследовать собственную реализацию этого метода.

Запрет объявлять один из методов `Object` в качестве метода по умолчанию может показаться удивительным. В конце концов, есть такие примеры, как `java.util.List`, в котором точно определено поведение `toString` и `equals`. Однако мотивация становится понятнее, если разобраться в некоторых более широких проектных решениях.

- Во-первых, методы, унаследованные от суперкласса, позволяют перекрывать методы, унаследованные от суперинтерфейсов (§8.4.8.1). Так, каждый реализующий класс будет автоматически перекрывать метод по умолчанию `toString` интерфейса. Это давнее поведение в языке программирования Java. Это не то, что мы хотели бы изменить в дизайне методов по умолчанию, поскольку такое решение будет конфликтовать с целью разрешить интерфейсам ненавязчивое

развитие, предоставляя поведение по умолчанию только тогда, когда класс не имеет его посредством иерархии классов.

- Во-вторых, интерфейсы *не* наследуют `Object`, но неявно объявляют многие методы, имеющиеся у `Object` (§9.2). Так, не имеется общего предка у `toString`, объявленного в `Object`, и `toString`, объявленного в интерфейсе. Если бы они оба были кандидатами на наследование классом, в лучшем случае это привело бы к конфликту. Решение этой проблемы потребовало бы неудобного смешения деревьев наследования класса и интерфейса.
- В-третьих, использование методов, объявленных в `Object`, в интерфейсах обычно предполагает линейную иерархию интерфейсов; эта возможность плохо обобщается на сценарии множественного наследования.
- В-четвертых, методы `Object` настолько фундаментальны, что представляется опасным позволить произвольному суперинтерфейсу молча добавлять методы по умолчанию, которые изменят их поведение.

Однако интерфейс может определять другой метод, который предоставляет поведение, полезное для классов, перекрывающих методы `Object`. Например, интерфейс `java.util.List` может объявить метод `elementString`, который генерирует строку, описываемую контрактом `toString`; реализации `toString` в классах после этого смогут использовать этот метод.

§9.4.1.3. Наследование методов с эквивалентными в смысле перекрытия сигнатурами

Возможно наследование интерфейсом нескольких методов с эквивалентными в смысле перекрытия сигнатурами (§8.4.2).

Если интерфейс *I* наследует метод по умолчанию, сигнатура которого эквивалентна в смысле перекрытия другому методу, унаследованному *I*, генерируется ошибка времени компиляции. (Это происходит независимо от того, объявлен ли другой метод как `abstract` или `default`.)

В противном случае все унаследованные методы являются `abstract`, и интерфейс рассматривается как унаследовавший все эти методы.

Однако один из унаследованных методов должен быть заменяемым по возвращаемому типу для каждого другого наследованного метода; в противном случае генерируется ошибка времени компиляции. (Конструкции `throws` в этом случае ошибок не вызывают.)

Могут иметься несколько путей, по которым интерфейс наследует одно и то же объявление метода. Этот факт не вызывает сложностей и никогда сам по себе не приводит к ошибкам времени компиляции.

> Естественно, когда два разных метода по умолчанию с совпадающими сигнатурами наследуются подынтерфейсом, возникает поведенческий конфликт. Этот конфликт активно обнаруживается, и разработчик уведомляется об ошибке, не ожидая проблем компиляции конкретного класса. Ошибки можно избежать путем объявления нового метода, который перекрывает все конфликтующие методы и, таким образом, предотвращает наследование.
>
> Аналогично сообщение об ошибке выводится при наследовании абстрактного метода и метода по умолчанию с одинаковыми сигнатурами. В этом случае можно

было бы повысить приоритет одного или другого метода — так, можно было бы предположить, что метод по умолчанию обеспечивает разумную реализацию абстрактного метода. Но это рискованно, так как, помимо случайного совпадения имени и сигнатуры, у нас нет оснований полагать, что метод по умолчанию ведет себя в соответствии с контрактом абстрактного метода, ведь метода по умолчанию могло даже не существовать при первоначальной разработке подынтерфейса. В этой ситуации безопаснее запросить пользователя, чтобы он подтвердил, что реализация по умолчанию в данном случае подходит (с помощью перекрывающего объявления).

В отличие от этого давно принятое поведение унаследованных конкретных методов в классах состоит в том, что они переопределяют абстрактные методы, объявленные в интерфейсах (см. §8.4.8). Здесь применимы те же рассуждения о возможном нарушении контракта, но в данном случае свою играет роль дисбаланс между классами и интерфейсами. Мы предпочитаем для сохранения независимого характера иерархии классов свести к минимуму конфликты между интерфейсами и классами, придавая повышенный приоритет конкретным методам.

§9.4.2. Перегрузка

Если два метода интерфейса (не имеет значения, объявлены ли они оба в одном интерфейсе или оба унаследованы интерфейсом, или один объявлен, а другой унаследован) имеют одно и то же имя, но разные сигнатуры, не являющиеся эквивалентными в смысле перекрытия (§8.4.2), то имя метода является *перегруженным* (overloaded).

Этот факт не вызывает сложностей и никогда сам по себе не приводит к ошибкам времени компиляции. Нет никаких обязательных взаимоотношений между возвращаемыми типами или между конструкциями `throws` двух методов с одним и тем же именем, но разными сигнатурами, не являющимися эквивалентными в смысле перекрытия.

Пример 9.4.2-1. Перегрузка объявления абстрактного метода

```
interface PointInterface {
    void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
    void move(float dx, float dy);
    void move(double dx, double dy);
}
```

Здесь метод с именем `move` перегружен в интерфейсе `RealPointInterface` с тремя различными сигнатурами, двумя объявленными и одной унаследованной. Любой не абстрактный класс, который реализует интерфейс `RealPoint Interface`, должен предоставить реализации всех трех сигнатур методов.

§9.4.3. Тело метода интерфейса

Метод по умолчанию имеет тело. Этот блок кода предоставляет реализацию метода в ситуации, когда класс реализует интерфейс, но не предоставляет собственную реализацию метода.

Статический метод также имеет тело, которое предоставляет реализацию метода.

Если объявление метода интерфейса явно или неявно является абстрактным и имеет блок тела, генерируется ошибка времени компиляции.

Если объявление метода интерфейса является `default` или `static` и имеет в качестве тела точку с запятой, генерируется ошибка времени компиляции.

Если в теле статического метода выполняется попытка обратиться к текущему объекту с помощью ключевого слова `this` или `super`, генерируется ошибка времени компиляции.

Правила для инструкции `return` в теле метода указаны в §14.17.

Если метод объявлен как имеющий возвращаемый тип (§8.4.5) и если тело этого метода может завершиться нормально (§14.1), генерируется ошибка времени компиляции.

§9.5. Объявления типов-членов

Интерфейсы могут содержать объявления типов-членов (§8.5).

Объявление типа-члена в интерфейсе неявно является `static` и `public`. Разрешается избыточное указание любого из этих модификаторов (или обоих).

Если объявление типа-члена в интерфейсе имеет модификатор `protected` или `private`, генерируется ошибка времени компиляции.

Если одно и то же ключевое слово появляется в качестве модификатора в объявлении типа-члена в интерфейсе более одного раза, генерируется ошибка времени компиляции.

Если интерфейс объявляет тип-член с определенным именем, объявление этого типа *скрывает* любые доступные объявления членов-типов с тем же именем в суперинтерфейсах интерфейса.

Интерфейс наследует от непосредственного суперинтерфейса все не-`private` типы-члены суперинтерфейсов, которые доступны коду в интерфейсе и не скрыты объявлениями в интерфейсе.

Интерфейс может наследовать два или более объявлений типа с одним и тем же именем. При попытке обратиться к неоднозначно наследованному классу или интерфейсу по его простому имени генерируется ошибка времени компиляции.

Если объявление типа наследуется от интерфейса несколькими путями, класс или интерфейс считается наследованным только один раз; к нему можно обращаться по его простому имени без какой бы то ни было неоднозначности.

§9.6. Типы аннотаций

Объявление типа аннотации — это *новый тип аннотации*, представляющий собой особый вид объявления интерфейса. Для того чтобы отличить объявление типа аннотации от обычного объявления интерфейса, перед ключевым словом `interface` ставится символ `@`.

AnnotationTypeDeclaration:
{InterfaceModifier} `@` `interface` *Identifier AnnotationTypeBody*

> Обратите внимание: символ `@` и ключевое слово `interface` представляют собой два различных токена. Технически они могут быть разделены пробельным символом, но это не поощряется с точки зрения стиля.

Правила для модификаторов аннотаций в объявлениях методов интерфейсов определены в §9.7.4 и §9.7.5.

Identifier в объявлении типа аннотации определяет имя типа аннотации.

Если тип аннотации имеет то же простое имя, что и любой из охватывающих его классов или интерфейсов, генерируется ошибка времени компиляции.

Непосредственным суперинтерфейсом каждого типа аннотации является `java.lang.annotation.Annotation`.

> В силу синтаксиса *AnnotationTypeDeclaration* объявление типа аннотации не может быть обобщенным, и в нем не разрешено наличие конструкции `extends`.
>
> Следствием того факта, что тип аннотации не может явно объявлять суперкласс или суперинтерфейс, является то, что подкласс или подынтерфейс типа аннотации сам по себе никогда не является типом аннотации. Аналогично `java.lang.annotation.Annotation` сам по себе типом аннотации не является.

Тип аннотации наследует ряд членов от `java.lang.annotation.Annotation`, включая неявно объявленные методы, соответствующие методам экземпляра в `Object`, однако эти методы не определяют элементы типа аннотации (§9.6.1).

> Поскольку эти методы не определяют элементы типа аннотации, их нельзя использовать в аннотациях данного типа. Без этого правила мы не могли бы гарантировать, что эти элементы имели бы типы, представимые в аннотациях, или что методы доступа для них были бы доступны.

Если в тексте явно не указано иное, все правила, применимые к объявлениям обычных интерфейсов, применимы к объявлениям типов аннотаций.

> Например, типы аннотаций разделяют то же пространство имен, что и типы обычных классов и интерфейсов; а объявления типов аннотаций допустимы везде, где допустимы объявления интерфейсов, и имеют те же область видимости и доступность.

§9.6.1. Элементы типа аннотации

Тело типа аннотации может содержать объявления методов, каждое из которых определяет *элемент* типа аннотации. Тип аннотации не имеет элементов, кроме определенных методами, которые он явно объявляет.

AnnotationTypeBody:
    { *{AnnotationTypeMemberDeclaration}* }

AnnotationTypeMemberDeclaration:
    AnnotationTypeElementDeclaration
    ConstantDeclaration
    ClassDeclaration

InterfaceDeclaration
;

AnnotationTypeElementDeclaration:
{AnnotationTypeElementModifier} *UnannType* *Identifier* ( )
[Dims] *[DefaultValue]* *;*

AnnotationTypeElementModifier: одно из
Annotation `public`
`abstract`

В силу синтаксиса *AnnotationTypeElementDeclaration* объявление метода в объявлении типа аннотации не может иметь формальных параметров, параметров типа или конструкцию `throws`. Далее для удобства приведена продукция из §4.3.

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]}`

В силу продукции *AnnotationTypeElementModifier*, объявление метода в объявлении типа аннотации не может быть `default` или `static`. Таким образом, тип аннотации не может объявлять такое же разнообразие методов, что и обычный тип интерфейса. Обратите внимание, что тип аннотации может наследовать метод по умолчанию из своего неявного суперинтерфейса, `java.lang.annotation.Annotation`, хотя такого метода по умолчанию в Java SE 8 не существует.

По соглашению *AbstractMethodModifiers* не должны присутствовать в элементе типа аннотации, за исключением аннотаций.

Возвращаемый тип объявленного в типе аннотации метода должен быть одним из приведенного списка; в противном случае генерируется ошибка времени компиляции.

- Примитивный тип.
- `String`.
- `Class` или конкретизация `Class` (§4.5).
- Тип перечисления.
- Тип аннотации.
- Тип массива, тип компонентов которого является одним из перечисленных типов (§10.1).

Это правило не включает элементы, типами которых являются вложенные массивы. Например, приведенное далее объявление типа аннотации некорректное.

```
@interface Verboten {
    String[][] value();
}
```

Объявление метода, возвращающего массив, может использовать пару квадратных скобок, которые описывают тип массива, после пустого списка формальных параметров. Этот синтаксис поддерживается для совместимости с ранними версиями языка программирования Java. Использовать его в новом коде крайне не рекомендуется.

Если любой метод, объявленный в типе аннотации, имеет сигнатуру, которая эквивалентна в смысле перекрытия сигнатуре любого `public`- или `protected`-метода, объяв-

ленного в классе `Object` или в интерфейсе `java.lang.annotation.Annotation`, генерируется ошибка времени компиляции.

Если объявление типа аннотации T прямо или косвенно содержит элемент типа T, генерируется ошибка времени компиляции.

Например, следующий код некорректен:

```
@interface SelfRef { SelfRef value(); }
```

Некорректен и код

```
@interface Ping { Pong value(); }
@interface Pong { Ping value(); }
```

Тип аннотации без элементов называется *типом аннотации заметки* или *типом аннотации-маркера*.

Тип аннотации с одним элементом называется *типом одноэлементной аннотации*.

По соглашению имя единственного элемента в типе одноэлементной аннотации — `value`. Языковая поддержка данного соглашения предоставляется конструкцией одноэлементной аннотации (§9.7.3).

Пример 9.6.1-1. Объявления типа аннотации

Приведенное далее объявление типа аннотации определяет тип аннотации с несколькими элементами.

```
/**
* Описывает "запрос на улучшение", ведущий к
* наличию аннотированного элемента API.
*/
@interface RequestForEnhancement {
    int id();           // Уникальный ID запроса
    String synopsis(); // Обзор запроса
    String engineer(); // Имя инженера, реализовавшего запрос
    String date();     // Дата реализации запроса
}
```

Пример 9.6.1-2. Объявления типа аннотации заметки

Приведенное далее объявление типа аннотации определяет тип аннотации заметки.

```
/**
* Аннотация этого типа указывает, что спецификация
* аннотированного элемента API предварительная и
* будет изменена.
*/
@interface Preliminary {}
```

Пример 9.6.1-3. Объявления типа одноэлементной аннотации

Приведенное выше соглашение о том, что тип одноэлементной аннотации определяет элемент с именем `value`, проиллюстрировано следующим объявлением типа аннотации.

```
/**
 * Заметка об авторских правах,
```

```
 * связанная с аннотируемым элементом API.
 */
@interface Copyright {
    String value();
}
```

Следующее объявление типа аннотации определяет тип одноэлементной аннотации, единственный элемент которой имеет тип массива.

```
/**
 * Связывает с классом список подтверждающих записей.
 */
@interface Endorsers {
    String[] value();
}
```

Следующее объявление типа аннотации демонстрирует аннотацию `Class`, значение которой ограничено символами подстановки.

```
interface Formatter {}

// Создает форматировщик для красивого
// вывода аннотированного класса
@interface PrettyPrinter {
    Class<? extends Formatter> value();
}
```

Следующее объявление типа аннотации содержит элемент, типом которого является тип аннотации.

```
/**
 * Указывает автора аннотированного элемента программы.
 */
@interface Author {
    Name value();
}
/**
 * Имя человека. Этот тип аннотации создан не для применения
 * непосредственно к элементам аннотации программы, а для
 * определения элементов других типов аннотации.
 */
@interface Name {
    String first();
    String last();
}
```

Заметим, что грамматика объявлений типов аннотаций допускает объявления других элементов, помимо объявлений методов. Например, можно объявить вложенное перечисление для использования совместно с типом аннотации.

```
@interface Quality {
    enum Level { BAD, INDIFFERENT, GOOD }
    Level value();
}
```

§9.6.2. Значения по умолчанию для элементов типа аннотации

Элемент типа аннотации может иметь *значение по умолчанию*, специфичное для данного элемента. Это делается с помощью следующего за (пустым) списком параметров ключевого слова `default` и значения элемента по умолчанию.

DefaultValue:
`default` *ElementValue*

Если тип элемента не сопоставим (commensurate) (§9.7) с указанным значением по умолчанию, генерируется ошибка времени компиляции.

Значения по умолчанию применяются динамически во время чтения аннотаций; значения по умолчанию не компилируются в аннотации. Таким образом, изменение значения по умолчанию влияет на аннотации даже в классах, которые были скомпилированы до внесения данного изменения (в предположении отсутствия указания явного значения для таких элементов).

Пример 9.6.2-1. Объявление типа аннотации со значениями по умолчанию

Вот как выглядит уточнение типа аннотации `RequestForEnhancement` из §9.6.1.

```
@interface RequestForEnhancementDefault {
    int id();              // Значения по умолчанию нет — должно
                           // быть указано в каждой аннотации
    String synopsis();     // Значения по умолчанию нет — должно
                           // быть указано в каждой аннотации
    String engineer()  default "[unassigned]";
    String date()      default "[unimplemented]";
}
```

§9.6.3. Повторяемые типы аннотаций

Тип аннотации *T* является *повторяемым* (repeatable), если его объявление (мета)аннотировано с помощью аннотации `@Repeatable` (§9.6.4.8), элемент `value` которой указывает *тип аннотации, содержащий T*.

Тип аннотации *TC* является *типом аннотации, содержащим T*, если истинны все следующие условия.

1. *TC* объявляет метод `value()`, возвращаемый тип которого — *T*`[]`.
2. Любые методы, объявленные *TC* и отличные от `value()`, имеют значение по умолчанию.
3. *TC* сохраняется (имеется в виду сохранение аннотации в байт-коде и доступность через рефлексию — см. §9.6.4.2) по меньшей мере столько же, сколько и *T*, где сохранение выражается явно или неявно с помощью аннотации `@Retention` (§9.6.4.2). В частности, справедливо следующее.
   - Если сохранение *TC* представляет собой `java.lang.annotation.RetentionPolicy.SOURCE`, то сохранение *T* представляет собой `java.lang.annotation.RetentionPolicy.SOURCE`.

- Если сохранение *TC* представляет собой `java.lang.annotation.RetentionPolicy.CLASS`, то сохранение *T* представляет собой либо `java.lang.annotation.RetentionPolicy.CLASS`, либо `java.lang.annotation.RetentionPolicy.SOURCE`.
- Если сохранение *TC* представляет собой `java.lang.annotation.RetentionPolicy.RUNTIME` то сохранение *T* представляет собой `java.lang.annotation.RetentionPolicy.SOURCE`, `java.lang.annotation.RetentionPolicy.CLASS` или `java.lang.annotation.RetentionPolicy.RUNTIME`.

4. *T* применим как минимум к тем же видам элемента программ, что и *TC* (§9.6.4.1). В частности, если виды элементов программы, к которым применим *T*, обозначить как множество m_1, а виды элементов программы, к которым применим *T*, обозначить как множество m_2, то каждый вид в m_2 должен встречаться в m_1, за исключением следующего.
   - Если вид в m_2 представляет собой `java.lang.annotation.ElementType.ANNOTATION_TYPE`, то как минимум один из элементов `java.lang.annotation.ElementType.ANNOTATION_TYPE`, `java.lang.annotation.ElementType.TYPE` или `java.lang.annotation.ElementType.TYPE_USE` должен находиться в m_1.
   - Если вид в m_2 представляет собой `java.lang.annotation.ElementType.TYPE`, то как минимум один из элементов `java.lang.annotation.ElementType.TYPE` или `java.lang.annotation.ElementType.TYPE_USE` должен находиться в m_1.
   - Если вид в m_2 представляет собой `java.lang.annotation.ElementType.TYPE_PARAMETER`, то как минимум один из элементов `java.lang.annotation.ElementType.TYPE_PARAMETER` или `java.lang.annotation.ElementType.TYPE_USE` должен находиться в m_1.

Это положение реализует стратегию, заключающуюся в том, что тип аннотации может быть *повторяемым* только для некоторых видов элементов программы там, где он *применим*.

5. Если объявление *T* имеет (мета)аннотацию, которая соответствует `java.lang.annotation.Documented`, то объявление *TC* должно иметь (мета)аннотацию, которая соответствует `java.lang.annotation.Documented`.

Обратите внимание, что *TC* может быть `@Documented`, в то время как *T* не является `@Documented`.

6. Если объявление *T* имеет (мета)аннотацию, которая соответствует `java.lang.annotation.Inherited`, то объявление *TC* должно иметь (мета)аннотацию, которая соответствует `java.lang.annotation.Inherited`.

Обратите внимание, что *TC* может быть `@Inherited`, в то время как *T* не является `@Inherited`.

Если тип аннотации *T* является (мета)аннотированным аннотацией `@Repeatable`, элемент `value` которой указывает тип, не содержащий тип аннотации *T*, генерируется ошибка времени компиляции.

Пример 9.6.3-1. Некорректный содержащий тип аннотации

Рассмотрим следующие объявления.

```
@Repeatable(FooContainer.class)
@interface Foo {}

@interface FooContainer { Object[] value(); }
```

Компиляция объявления `Foo` приводит к ошибке времени компиляции, поскольку `Foo` использует `@Repeatable` для попытки указать `FooContainer` как содержащий его тип аннотации, но `FooContainer` фактически не является содержащим типом аннотации для `Foo`. (`Foo[]` не является возвращаемым типом `FooContainer.value()`.)

Аннотация `@Repeatable` не может быть повторена, так что повторимый тип аннотации может определять только один содержащий тип аннотации.

Позволение указывать более одного содержащего типа аннотации ведет к нежелательному выбору во время компиляции, когда несколько аннотаций повторяемого типа аннотации логически заменяются контейнерной аннотацией (§9.7.5).

Тип аннотации может быть содержащим типом аннотации не более одного типа аннотации.

Это вытекает из требования, что если объявление типа аннотации *T* определяет содержащий тип аннотации *TC*, то метод `value()` в *TC* имеет возвращаемый тип, включающий *T*, а именно — `T[]`.

Тип аннотации не может определять сам себя как собственный содержащий тип аннотации.

Это вытекает из требования к методу `value()` содержащего типа аннотации. В частности, если тип аннотации *A* определяет сам себя (с помощью `@Repeatable`) как свой содержащий тип аннотации, то возвращаемый тип метода `value()` типа аннотации *A* должен быть `A[]`; но это привело бы к ошибке времени компиляции, так как тип аннотации не может ссылаться на себя в своих элементах (§9.6.1). В более общем случае два типа аннотации не могут указывать один другой в качестве своих содержащих типов аннотации, поскольку циклические объявления типов аннотации недопустимы.

Тип аннотации *TC* может быть содержащим типом аннотации некоторого типа аннотации *T* и иметь также собственный содержащий тип аннотации *TC'*, т.е. содержащий тип аннотации сам по себе может быть повторяемым типом аннотации.

Пример 9.6.3-2. Ограничения на повторяемость аннотаций

Аннотация, объявление типа которой указывает целью `java.lang.annotation.ElementType.TYPE`, может находиться как минимум в таком количестве место

положений, что и аннотация, объявление типа которой указывает целью `java.lang.annotation.ElementType.ANNOTATION_TYPE`. Например, пусть имеются следующие объявления повторяемого и содержащего аннотацию типов.

```
@Target(ElementType.TYPE)
@Repeatable(FooContainer.class)
@interface Foo {}

@Target(ElementType.ANNOTATION_TYPE)
@Interface FooContainer {
    Foo[] value();
}
```

`@Foo` может находиться в объявлении любого типа, в то время как `@FooContainer` может находиться только в объявлениях типов аннотаций. Значит, следующее объявление типа аннотации вполне корректно:

```
@Foo @Foo
@interface X {}
```

в то время как приведенное ниже объявление — нет:

```
@Foo @Foo
interface X {}
```

Говоря более широко, если `Foo` представляет собой повторяемый тип аннотации, а `FooContainer` является его содержащим типом аннотации, то справедливо следующее.

- Если `Foo` не имеет мета-аннотации `@Target` и `FooContainer` не имеет метааннотации `@Target`, то `@Foo` может быть повторяемым для любого элемента программы, поддерживающего аннотации.
- Если `Foo` не имеет мета-аннотации `@Target`, а `FooContainer` имеет метааннотацию `@Target`, то `@Foo` может быть повторяемым только для тех элементов программы, в которых может появляться `@FooContainer`.
- Если `Foo` имеет мета-аннотацию `@Target`, то по представлениям проектировщиков языка программирования Java `FooContainer` должен быть объявлен со знанием применимости `Foo`. В частности, виды программных элементов, в которых может появляться `FooContainer`, должны логически совпадать или быть подмножеством видов `Foo`.
- Например, если `Foo` применим к объявлениям полей и методов, то `FooContainer` может обоснованно служить в качестве содержащего типа аннотации для `Foo`, если `FooContainer` применим только к объявлениям полей (предупреждая `@Foo` от повторения в объявлениях методов). Но если `FooContainer` применим только к объявлениям формальных параметров, то `FooContainer` является плохим выбором в качестве содержащего типа аннотации для `Foo`, поскольку `@FooContainer` не может быть неявно объявленным для некоторых элементов программы, в которых повторяется `@Foo`.
- Аналогично, если `Foo` применим к объявлениям полей и методов, то `FooContainer` не может обоснованно служить в качестве содержащего типа аннотации для `Foo`, если `FooContainer` применим к объявлениям полей и параметров. Хотя можно было бы взять пересечение элементов программы и сделать

`Foo` повторимым только для объявлений полей, наличие дополнительных элементов программ для `FooContainer` указывает, что `FooContainer` не был разработан в качестве содержащего типа аннотации для `Foo`. Поэтому полагаться на него было бы для `Foo` опасно.

Пример 9.6.3-3. Повторяемый содержащий тип аннотации

Приведенные далее объявления корректны.

```
// Foo: Повторяемый тип аннотации
@Repeatable(FooContainer.class)
@interface Foo { int value(); }

// FooContainer: Содержащий тип аннотации Foo
// Также повторяемый тип аннотации сам по себе
@Repeatable(FooContainerContainer.class)
@interface FooContainer { Foo[] value(); }

// FooContainerContainer: Содержащий тип аннотации FooContainer
@interface FooContainerContainer { FooContainer[] value(); }
```

Таким образом, аннотация, тип которой является содержащим типом аннотации, сама по себе может быть повторяемой.

```
@FooContainer({@Foo(1)}) @FooContainer({@Foo(2)})
class A {}
```

Тип аннотации, который является и повторяемым, и содержащим, подчиняется правилам, которые являются смешением правил для аннотаций повторяемого типа и аннотаций содержащего типа (§9.7.5). Например, невозможно ни написать несколько аннотаций `@Foo` вместе с несколькими аннотациями `@FooContainer`, ни написать несколько аннотаций `@FooContainer` вместе с несколькими аннотациями `@FooContainerContainer`. Однако, если тип `FooContainerContainer` сам по себе повторяемый, можно записать несколько аннотаций `@Foo` вместе с несколькими аннотациями `@FooContainerContainer`.

§9.6.4. Предопределенные типы аннотаций

Некоторые типы аннотаций предопределены в библиотеках платформы Java SE. Некоторые из этих предопределенных аннотаций типов имеют специальную семантику. Эти семантики описаны в данном разделе. В разделе нет полной спецификации описываемых предопределенных аннотаций — для этого следует обратиться к соответствующим спецификациям API. Здесь указывается только семантика, которая требует специального поведения со стороны компилятора или реализации виртуальной машины Java.

§9.6.4.1 `@Target`

Тип аннотации `java.lang.annotation.Target` предназначен для использования в объявлении типа аннотации *T* для указания контекстов, в которых *T* является *применимым*. Для указания контекста `java.lang.annotation.Target` имеет один элемент `value` типа `java.lang.annotation.ElementType[]`.

Типы аннотаций могут быть применимы в *контекстах объявлений*, где аннотации применяются к объявлениям, или в *контекстах типов*, где аннотации применяются к типам, используемым в объявлениях и выражениях.

Имеется восемь контекстов объявлений, каждый из которых соответствует константе перечисления `java.lang.annotation.ElementType`.

1. Объявления пакетов (§7.4.1).

   Соответствует `java.lang.annotation.ElementType.PACKAGE`.

2. Объявления типов: объявления классов, интерфейсов, перечислений и типов аннотаций (§8.1.1, §9.1.1, §8.5, §9.5, §8.9, §9.6).

   Соответствует `java.lang.annotation.ElementType.TYPE`.

   Кроме того, объявления типов аннотаций соответствуют `java.lang.annotation.ElementType.ANNOTATION_TYPE`.

3. Объявления методов (включая элементы типов аннотаций) (§8.4.3, §9.4, §9.6.1).

   Соответствует `java.lang.annotation.ElementType.METHOD`.

4. Объявления конструкторов (§8.8.3).

   Соответствует `java.lang.annotation.ElementType.CONSTRUCTOR`.

5. Объявления параметров типов обобщенных классов, интерфейсов, методов и конструкторов (§8.1.2, §9.1.2, §8.4.4, §8.8.4).

   Соответствует `java.lang.annotation.ElementType.TYPE_PARAMETER`.

6. Объявления полей (включая константы перечислений) (§8.3.1, §9.3, §8.9.1).

   Соответствует `java.lang.annotation.ElementType.FIELD`.

7. Объявления формальных параметров и параметров исключений (§8.4.1, §9.4, §14.20).

   Соответствует `java.lang.annotation.ElementType.PARAMETER`.

8. Объявления локальных переменных (включая переменные циклов `for` и переменные ресурсов инструкций `try`-с-ресурсами) (§14.4, §14.14.1, §14.14.2, §14.20.3).

   Соответствует `java.lang.annotation.ElementType.LOCAL_VARIABLE`.

Имеется 16 контекстов типов (§4.11); все они представлены константой `TYPE_USE` типа `java.lang.annotation.ElementType`.

Если некоторая константа перечисления встречается более одного раза в элементе `value` аннотации типа `java.lang.annotation.Target`, генерируется ошибка времени компиляции.

Если аннотация типа `java.lang.annotation.Target` отсутствует в объявлении типа аннотации T, то T применим во всех контекстах объявлений, за исключением объявлений параметров типа исключений, и не применим ни в одном контексте типа.

> Эти контексты представляют собой синтаксические местоположения, в которых в Java SE 7 были применимы аннотации.

§9.6.4.2. `@Retention`

Аннотации могут присутствовать только в исходном коде или в бинарном виде класса или интерфейса. Аннотация, присутствующая в бинарном виде, может быть доступна (а может и не быть) через библиотеки отражения (рефлексии) платформы Java SE во время выполнения. Тип аннотации `java.lang.annotation.Retention` используется для выбора одной из этих возможностей.

Если аннотация *a* соответствует типу *T*, а *T* имеет (мета)аннотацию *m*, которая соответствует `java.lang.annotation.Retention`, то происходит следующее.

- Если *m* имеет элемент со значением `java.lang.annotation.RetentionPolicy.SOURCE`, то компилятор Java должен гарантировать, что *a* отсутствует в бинарном представлении класса или интерфейса, в котором находится *a*.
- Если *m* имеет элемент со значением `java.lang.annotation.RetentionPolicy.CLASS` или `java.lang.annotation.RetentionPolicy.RUNTIME`, то компилятор Java должен гарантировать, что *a* имеется в бинарном представлении класса или интерфейса, в котором находится *a*, если только *m* не аннотирует объявление локальной переменной.

  Аннотация объявления локальной переменной никогда не хранится в бинарном представлении.

  Кроме того, если *m* имеет элемент, значение которого — `java.lang.annotation.RetentionPolicy.RUNTIME`, библиотеки отражения платформы Java SE должны делать *a* доступной во время выполнения.

Если *T* не имеет (мета)аннотации *m*, которая соответствует `java.lang.annotation.Retention`, то компилятор Java должен рассматривать *T* так, как если бы он имел метааннотацию *m* с элементом, значение которого — `java.lang.annotation.RetentionPolicy.CLASS`.

§9.6.4.3. `@Inherited`

Тип аннотации `java.lang.annotation.Inherited` используется для указания того, что аннотация к классу *C* соответствует данному типу аннотации и наследуется подклассами *C*.

§9.6.4.4. `@Override`

Иногда программисты перегружают объявление метода, в то время как хотят перекрыть его, что приводит к тонким проблемам. Тип аннотации `@Override` поддерживает раннее выявление таких проблем.

> Классический пример — применение метода `equals`. Программист пишет в классе `Foo` код
>
> ```
> public boolean equals(Foo that) { ... }
> ```
>
> в то время как он хотел написать
>
> ```
> public boolean equals(Object that) { ... }
> ```

> Код совершенно корректен, но класс `Foo` наследует реализацию `equals` от `Object`, что может привести к некоторым очень тонким трудно обнаруживаемым ошибкам.

Если объявление метода аннотировано с помощью аннотации `@Override`, но метод не перекрывает и не реализует метод, объявленный в супертипе, или если он не является эквивалентным в смысле перекрытия открытому методу `Object`, генерируется ошибка времени компиляции.

> Это поведение отличается от поведения Java SE 5.0, где `@Override` приводит к ошибке времени компиляции, только если применяется к методу, который реализует метод из суперинтерфейса, который при этом не присутствует в суперклассе.
>
> Утверждение о перекрытии открытого метода мотивируется использованием `@Override` в интерфейсе. Рассмотрим следующие объявления типа.
>
> ```
> class     Foo { @Override public int hashCode() {..} }
> interface Bar { @Override int hashCode(); }
> ```
>
> Применение `@Override` в объявлении класса корректно в первом объявлении, поскольку `Foo.hashCode` перекрывает `Object.hashCode` (§8.4.8).
>
> В случае объявления интерфейса считается, что поскольку интерфейс не имеет `Object` в качестве супертипа, он имеет объявленные как `public abstract` члены, соответствующие `public`-членам `Object` (§9.2). Если интерфейс объявляет их явно (например, для объявления членов, эквивалентных в смысле перегрузки `public`-методам `Object`), то считается, что интерфейс их перекрывает (§8.4.8) и использование `@Override` разрешено.
>
> Однако рассмотрим интерфейс, который пытается использовать `@Override` в методе `clone` (в этом примере можно также использовать `finalize`):
>
> ```
> interface Quux { @Override Object clone(); }
> ```
>
> Поскольку `Object.clone` не является `public`, в `Quux` нет неявно объявленного члена `clone`. Следовательно, явное объявление `clone` в `Quux` не рассматривается как "реализующее" некоторый другой метод, и применение `@Override` ошибочное. (Тот факт, что `Quux.clone` объявлен как `public`, роли не играет.)
>
> Напротив, объявление класса, в котором объявлен `clone`, просто перекрывает `Object.clone`, так что здесь можно использовать `@Override`:
>
> ```
> class Beep { @Override protected Object clone() {..} }
> ```

§9.6.4.5. `@SuppressWarnings`

Компиляторы Java становятся все более склонны к полезным многословным предупреждениям. Чтобы это многословие приносило реальную пользу, следует иметь возможность каким-то образом отключать предупреждение в той части программы, в которой программист знает, что это предупреждение является неуместным.

Тип аннотации `SuppressWarnings` поддерживает контроль программиста над предупреждениями компилятора Java. Он содержит единственный элемент, который представляет собой массив `String`.

Если объявление в программе аннотировано с помощью аннотации `@SuppressWarnings(value={`S_1`, ..., `S_k`})`, то компилятор Java не должен выводить любое

предупреждение, определяемое одной из строк $S_1 \ldots S_k$, если это предупреждение генерируется аннотированным объявлением или любой его частью.

Предупреждения о непроверенных типах идентифицируются строкой `"unchecked"`.

> Разработчики компиляторов должны документировать для данного типа аннотации имена предупреждений, которые они поддерживают. Производители приглашаются к сотрудничеству с целью гарантии работы одних и тех же имен в множестве компиляторов.

§9.6.4.6. `@Deprecated`

Элемент программы, аннотированный как `@Deprecated`, является элементом, который программистам не рекомендуется использовать, обычно из-за его опасности или наличия лучшей альтернативы.

Компилятор Java должен вывести предупреждение о нежелательности использования в явно или неявно объявленной конструкции, когда применяется тип, метод, поле или конструктор, объявление которого аннотировано с помощью `@Deprecated` (например, при перекрытии, вызове или обращении по имени), кроме следующих случаев.

- Применение в сущности, которая сама по себе аннотирована как `@Deprecated`.
- Применение в сущности, в которой вывод предупреждений подавлен аннотацией `@SuppressWarnings("deprecation")`.
- Использование и объявление находятся в одном и том же наиболее внешнем классе.

Использование аннотации `@Deprecated` в объявлении локальной переменной или объявлении параметра не приводит ни к какому эффекту.

> Единственной неявно объявленной конструкцией, которая может приводить к предупреждению о нежелательности, является содержащая аннотация (§9.7.5). А именно, если *T* является повторимым типом аннотации, а *TC* — его содержащий тип аннотации, то повторение аннотации `@T` приведет к указанному предупреждению. Данное предупреждение вызывается неявной содержащей аннотацией `@TC`. Настоятельно не рекомендуется использовать `@Deprecate` для содержащей аннотации без применения этой аннотации к соответствующему повторимому типу.

§9.6.4.7. `@SafeVarargs`

Параметр с переменной арностью с типом элемента, недоступным во время выполнения (§4.7), может привести к замусориванию кучи (§4.12.2) и породить предупреждения о непроверяемом преобразовании во время компиляции (§5.1.9). Такие предупреждения неинформативны, если тело метода с переменным количеством аргументов является правильно работающим в отношении параметра с переменной арностью.

Тип аннотации `SafeVarargs` при использовании для аннотирования объявления метода или конструктора предохраняет компилятор Java от предупреждений о непроверенных типах при объявлении или вызове метода или конструктора с переменным количеством аргументов там, где без такой аннотации компилятор выводил бы такое предупреждение из-за того, что параметр с переменной арностью имеет тип элемента, недоступный во время выполнения.

> Аннотация `SafeVarargs` обладает нелокальным воздействием, поскольку подавляет предупреждения о непроверенных типах в выражениях вызова метода в дополнение к предупреждениям, относящимся к объявлению самого метода с переменным количеством аргументов (§8.4.1). В противоположность ей аннотация `@SuppressWarnings("unchecked")` имеет локальное действие, поскольку подавляет только предупреждения о непроверенных типах, относящихся к объявлению метода.
>
> Каноническим объектом для аннотации `SafeVarargs` является метод наподобие `java.util.Collections.addAll`, объявление которого начинается следующим образом.
>
> ```
> public static <T> boolean
>     addAll(Collection<? super T> c, T... elements)
> ```
>
> Параметр переменной арности имеет объявленный тип `T[]`, который недоступен во время выполнения. Однако метод просто выполняет чтение элементов из входного массива и добавляет их в коллекцию, и обе эти операции безопасны по отношению к массиву. Следовательно, любые предупреждения времени компиляции о непроверенных типах в выражениях вызова метода `java.util.Collections.addAll` являются ложными и неинформативными. Применение аннотации `SafeVarargs` к объявлению метода предотвращает генерацию этих предупреждений в выражениях вызова метода.

Если объявление метода или конструктора с фиксированным количеством аргументов аннотировано с помощью аннотации `SafeVarargs`, генерируется ошибка времени компиляции.

Если объявление метода с переменным количеством аргументов, который не является ни `static`, ни `final`, аннотировано с помощью аннотации `SafeVarargs`, генерируется ошибка времени компиляции.

> Поскольку аннотация `SafeVarargs` применима только к методам, объявленным как `static`, методам экземпляра, объявленным как `final`, и конструкторам, эта аннотация неприменима там, где происходит перекрытие метода. Наследование аннотаций работает только в классах (не для методов, интерфейсов или конструкторов), так что аннотация в стиле `SafeVarargs` не может быть передана через методы экземпляра классов или через интерфейсы.

§9.6.4.8. `@Repeatable`

Тип аннотации `java.lang.annotation.Repeatable` используется в объявлениях *повторимых типов аннотаций* для указания содержащего типа аннотации (§9.6.3).

> Обратите внимание, что метааннотации `@Repeatable` объявления *T*, указывающей *TC*, *недостаточно*, чтобы сделать *TC* содержащим типом аннотации для *T*. Имеется множество правил правильного формирования *TC* так, чтобы он рассматривался как содержащий тип аннотации *T*.

§9.6.4.9. `@FunctionalInterface`

Тип аннотации `FunctionalInterface` используется для указания того факта, что интерфейс является функциональным интерфейсом (§9.8). Это облегчает раннее обнару-

жение некорректных объявлений методов, находящихся в интерфейсе, который должен быть функциональным, или унаследованных интерфейсом.

Если объявление интерфейса аннотировано с помощью `@FunctionalInterface`, но фактически интерфейс функциональным не является, генерируется ошибка времени компиляции.

Поскольку некоторые интерфейсы являются функциональными случайно, не является необходимым или даже желательным, чтобы все объявления функциональных интерфейсов были аннотированы с помощью `@FunctionalInterface`.

§9.7. Аннотации

Аннотация представляет собой маркер, который связывает информацию с программной конструкцией, но при этом не влияет на выполнение программы. Аннотация обозначает конкретный вызов типа аннотации (§9.6) и обычно предоставляет значения элементов данного типа.

Имеется три разновидности аннотаций. Первая из них наиболее общая, в то время как остальные две по сути представляют собой сокращения первой.

Annotation:
    NormalAnnotation
    MarkerAnnotation
    SingleElementAnnotation

Обычные аннотации рассматриваются в §9.7.1, аннотации-маркеры — в §9.7.2, а одноэлементные аннотации — в §9.7.3. Аннотации могут находиться в программе в различных синтаксических местоположениях, как описано в §9.7.4. Количество аннотаций одного и того же типа, которые могут находиться в одном месте, определяется их типом, как описано в §9.7.5.

§9.7.1. Обычные аннотации

Обычная аннотация определяет имя типа аннотации и необязательный список разделенных запятыми *пар "элемент–значение"*. Каждая пара содержит *значение элемента*, связанное с элементом типа аннотации (§9.6.1).

NormalAnnotation:
    `@` *TypeName* `(` *[ElementValuePairList]* `)`

ElementValuePairList:
    ElementValuePair {`,` *ElementValuePair*}

ElementValuePair:
    Identifier `=` *ElementValue*

ElementValue:
    ConditionalExpression

ElementValueArrayInitializer
Annotation

ElementValueArrayInitializer:
{ *[ElementValueList]* *[,]* }

ElementValueList:
ElementValue {, *ElementValue*}

> Обратите внимание, что символ @ сам по себе является токеном (§3.11). Технически возможно поместить пробельный символ между ним и *TypeName*, но это не приветствуется с точки зрения стиля.

TypeName именует тип аннотации, соответствующий данной аннотации.

Если *TypeName* именует тип аннотации, недоступный (§6.6) в точке использования аннотации, генерируется ошибка времени компиляции.

Identifier в паре "элемент–значение" должен быть простым именем одного из элементов (например, методов) типа аннотации, идентифицированного с помощью *TypeName*; в противном случае генерируется ошибка времени компиляции.

Возвращаемый тип этого метода идентифицирует *тип элемента* пары "элемент–значение".

Если тип элемента представляет собой тип массива, то использовать фигурные скобки для указания значения элемента в паре не обязательно. При этом, если значение элемента не является *ElementValueArrayInitializer*, то с элементом связывается значение массива, единственный элемент которого представляет собой значение элемента. Если значение элемента представляет собой *ElementValueArrayInitializer*, то с элементом связывается значение массива, представленное *ElementValueArrayInitializer*.

Если тип элемента `T` не *сопоставим* (commensurate) со значением элемента, генерируется ошибка времени компиляции. Тип элемента `T` сопоставим со значением элемента `V` тогда и только тогда, когда выполняется одно из следующих условий.

- `T` представляет собой тип массива `E[]` и
  - либо `V` представляет собой *ConditionalExpression* или *Annotation*, то `V` сопоставим с `E`;
  - либо `V` представляет собой *ElementValueArrayInitializer*, то каждое значение элемента, которое содержит `V`, сопоставимо с `E`;

  > *ElementValueArrayInitializer* подобен инициализатору обычного массива (§10.6), с тем отличием, что *ElementValueArrayInitializer* может синтаксически содержать аннотации, как и выражения и вложенные инициализаторы. Однако вложенные инициализаторы не являются семантически законными в *ElementValueArrayInitializer*, поскольку они никогда не являются сопоставимыми с элементами типа массива в объявлении типа аннотации (вложенные типы массивов не разрешены).

- `T` не является типом массива, а тип `V` совместим по присваиванию (§5.2) с `T`, и кроме того, выполняется следующее.
  - Если `T` является примитивным типом или `String`, а `V` является константным выражением (§15.28).

- Если *T* представляет собой `Class`, или конкретизацию `Class`, а *V* представляет собой литерал класса (§15.8.2).
- Если *T* представляет собой тип перечисления, а *V* является константой перечисления.
- *V* — не `null`.

> Обратите внимание, что если тип элемента не является типом аннотации или типом массива, значение элемента должно представлять собой *ConditionalExpression* (§15.25). Применение *ConditionalExpression* вместо более общей продукции *Expression* представляет собой синтаксический трюк для предотвращения использования в качестве значений элементов выражений присваивания. Поскольку выражение присваивания не является константным выражением, оно не может быть сопоставимым значением для примитивного элемента или элемента типа `String`.
>
> Формально нельзя говорить об *ElementValue*, как о FP-строгом (§15.4), поскольку оно может быть аннотацией или литералом класса. Тем не менее неформально мы можем говорить об *ElementValue* как о FP-строгом, когда оно либо является константным выражением, массивом константных выражений или аннотацией, значения элементов которых (рекурсивно) определяются как константные выражения; в конце концов, каждое константное выражение является FP-строгим.

Обычная аннотация должна содержать пару "элемент–значение" для каждого элемента соответствующего типа аннотации, за исключением элементов со значениями по умолчанию, иначе генерируется ошибка времени компиляции.

Обычная аннотация может, но не обязана, содержать пары "элемент–значение" для элементов со значениями по умолчанию.

> Принято, хотя и не обязательно, чтобы пары "элемент–значение" в аннотациях были представлены в том же порядке, что и соответствующие элементы в объявлении типа аннотации.

Аннотация в объявлении типа аннотации известна как *метааннотация*.

Тип аннотации *T* может находиться в качестве метааннотации в объявлении самого типа *T*. В более общем случае допускается зацикливание в транзитивном замыкании отношения "аннотация".

> Например, возможно аннотировать объявление типа аннотации *S* метааннотацией типа *T*, а объявление типа *T* аннотировать метааннотацией типа *S*. Предопределенные типы аннотаций содержат несколько таких зацикливаний.

Пример 9.7.1-1. Обычные аннотации

Вот пример нормальной аннотации, использующей тип аннотации из §9.6.1.

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date     = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

А вот пример обычной аннотации, использующей преимущества значений по умолчанию, с использованием типа аннотации из §9.6.2.

```
@RequestForEnhancement(
    id       = 4561414,
    synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

§9.7.2. Аннотации-маркеры

Второй вид аннотаций, *маркеры*, представляет собой сокращение, предназначенное для использования с типами аннотаций-маркеров (§9.6.1).

MarkerAnnotation:
    @ *TypeName*

Это сокращение для обычной аннотации.

@*TypeName*()

Разрешено использовать форму маркера для типов аннотаций с элементами, лишь бы все элементы имели значения по умолчанию (§9.6.2).

Пример 9.7.2-1. Аннотации-маркеры

Вот пример использования маркера `Preliminary` из §9.6.1.

```
@Preliminary public class TimeTravel { ... }
```

§9.7.3. Одноэлементные аннотации

Третий вид аннотаций, *одноэлементные аннотации*, представляет собой сокращение, предназначенное для использования с типами одноэлементных аннотаций (§9.6.1).

SingleElementAnnotation:
    @ *TypeName* (*ElementValue*)

Это сокращение для обычной аннотации.

@*TypeName*(value = *ElementValue*)

Разрешается использовать одноэлементные аннотации вместо типов аннотаций с несколькими элементами, если один элемент имеет имя `value`, а все прочие элементы имеют значения по умолчанию (§9.6.2).

Пример 9.7.3-1. Одноэлементные аннотации

Все приведенные ниже аннотации используют одноэлементные типы аннотаций из §9.6.1.

Вот пример одноэлементной аннотации.

```
@Copyright("2002 Yoyodyne Propulsion Systems, Inc.")
public class OscillationOverthruster { ... }
```

Вот пример одноэлементной аннотации со значением-массивом.

```
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
```

Вот пример одноэлементной аннотации, элемент которой является массивом с одним элементом. Обратите внимание на опущенные фигурные скобки.

```
@Endorsers("Epicurus")
public class Pleasure { ... }
```

Вот пример одноэлементной аннотации с элементом `Class`, значение которого ограничено использованием ограниченного символа подстановки.

```
class GorgeousFormatter implements Formatter { ... }

@PrettyPrinter(GorgeousFormatter.class)
public class Petunia { ... }

// Неверно; String не является подтипом Formatter
@PrettyPrinter(String.class)
public class Begonia { ... }
```

Вот пример одноэлементной аннотации, содержащей обычную аннотацию.

```
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

Вот пример одноэлементной аннотации, использующей тип перечисления, определенный в типе аннотации.

```
@Quality(Quality.Level.GOOD)
public class Karma { ... }
```

§9.7.4. Где могут находиться аннотации

Аннотация объявления представляет собой аннотацию, которая применяется к объявлению и собственный тип которой применим в контексте объявления (§9.6.4.1), представленном этим объявлением.

Аннотация типа представляет собой аннотацию, которая применяется к типу (или любой части типа) и собственный тип которой применим в контексте типа (§4.11).

Например, в данном объявлении поля

```
@Foo int f;
```

`@Foo` представляют собой аннотацию объявления `f`, если `Foo` метааннотирована как `@Target(ElementType.FIELD)`, и тип аннотации `int`, если `Foo` метааннотирована как `@Target(ElementType.TYPE_USE)`. `@Foo` может быть одновременно аннотацией объявления и аннотацией типа.

Аннотации типа могут быть применены к типу массива или к любому типу его компонентов. Предположим, например, что `A`, `B` и `C` — типы аннотаций, метааннотированные с помощью `@Target(ElementType.TYPE_USE)`, и что у нас есть следующее объявление поля.

```
@C int @A [] @B [] f;
```

> `@A` применятся к типу массива `int[][]`, `@B` применятся к типу его компонента `int[]`, а `@C` — к типу элемента `int`.
>
> Важное свойство этого синтаксиса заключается в том, что в двух объявлениях, отличающихся только количеством уровней (размерностью) массивов, аннотации слева от типа ссылаются на один и тот же тип. Например, `@C` применяется к типу `int` во всех приведенных далее объявлениях.
>
> ```
> @C int f;
> @C int[] f;
> @C int[][] f;
> ```
>
> Обычная, хотя и не обязательная, практика — записывать аннотации объявлений до всех других модификаторов, а аннотации типов непосредственно перед типами, к которым они применяются.

Аннотация может находиться в синтаксическом местоположении в программе, где она может применяться к объявлению, типу или к ним обоим. Это может произойти в любом из пяти контекстов объявлений, в которых модификаторы непосредственно предшествуют типу объявляемой сущности.

- Объявление методов (включая элементы типов аннотаций).
- Объявления конструкторов.
- Объявления полей (включая константы перечислений).
- Объявления формальных параметров и параметров исключений.
- Объявления локальных переменных (включая переменные циклов `for` и переменных ресурсов в инструкциях `try`-с-ресурсами).

Грамматика языка программирования Java однозначно рассматривает аннотации в этих местах как модификаторы объявлений (§8.3), но это чисто синтаксический вопрос. Применяется ли аннотация к объявлению или к типу объявляемой сущности — и, таким образом, является ли аннотация *аннотацией объявления* или *аннотацией типа* — зависит от применимости типа аннотации.

- Если тип аннотации применим в контексте объявления, соответствующего объявлению, а не в контексте типа, то аннотация рассматривается как применимая только к объявлению.
- Если тип аннотации применим в контексте типа, а не в контексте объявления, соответствующего объявлению, то аннотация рассматривается как применимая только к типу, ближайшему к аннотации.
- Если тип аннотации применим в контексте объявления, соответствующего объявлению, *и* в контексте типа, то аннотация рассматривается как применимая как к объявлению, так *и* к типу, ближайшему к аннотации.

Во втором и третьем случаях тип, являющийся *ближайшим* к аннотации, определяется сначала типом объявленной сущности, а затем — первым токеном, который сам по себе описывает тип.

> Например, в объявлении поля `@Foo public static java.lang.String f;` типом, ближайшим к `@Foo`, является `String`. В объявлении обобщенного метода

`@Foo <T> int[] m() {...}` типом объявляемой сущности является `int[]`, первый токен которого, обозначающий тип, представляет собой `int`, так что `@Foo` применяется к типу `int`.

Объявления локальных переменных подобны объявлениям формальных параметров лямбда-выражений в том, что в исходных текстах и те, и другие допускают как аннотации объявлений, так и аннотации типов, но в `class`-файле могут храниться только аннотации типов.

Есть два особых случая, связанных с объявлениями методов/конструкторов.

- Если аннотация появляется перед объявлением конструктора и считается, что она применяется к типу, ближайшему к аннотации, то этот тип представляет собой тип вновь создаваемого объекта. Типом вновь создаваемого объекта является полностью квалифицированное имя типа, непосредственно охватывающего объявление конструктора. В этом полностью квалифицированном имени аннотация применяется к простому имени типа, указанному в объявлении конструктора.
- Если аннотация появляется перед объявлением `void`-метода и считается применимой только к типу, ближайшему к аннотации, генерируется ошибка времени компиляции.

Если аннотация типа *T* синтаксически является модификатором для любого элемента из приведенного ниже списка, генерируется ошибка времени компиляции.

- Объявление пакета, но *T* не применим к объявлениям пакетов.
- Объявление класса, интерфейса или перечисления, но *T* не применим к объявлениям типов или контекстам типов; или объявление типа аннотации, но *T* не применим к объявлениям типов аннотаций, объявлениям типов и контекстам типов.
- Объявление метода (включая элемент типа аннотации), но *T* не применим к объявлениям методов и контекстам типов.
- Объявление конструктора (включая элемент типа аннотации), но *T* не применим к объявлениям конструкторов и контекстам типов.
- Объявление параметра типа обобщенного класса, интерфейса, метода или конструктора, но *T* не применим к объявлениям параметров типа и контекстам типов.
- Объявление поля (включая константы перечислений), но *T* не применим к объявлениям полей и контекстам типов.
- Объявление формального параметра или параметра исключения, но *T* не применим ни к объявлениям формальных параметров или параметров исключений, ни к контекстам типов.
- Параметр-получатель, но *T* не применим к контекстам типов.
- Объявление локальной переменной (включая переменные цикла `for` или переменные ресурса в инструкции `try`-с-ресурсами), но *T* не применим к объявлениям локальных переменных и к контекстам типов.

Обратите внимание, что в большей части элементов списка упоминается “и к контекстам типов”, потому что, даже если аннотация не применяется к объявлению, она по-прежнему может применяться к типу объявляемой сущности.

Аннотация типа является *приемлемой* (admissible), если выполняются оба следующих условия.

- Простое имя, к которому аннотация расположена ближе всего, классифицируется как *TypeName*, а не *PackageName*.
- Если за простым именем, к которому аннотация расположена ближе всего, следует "." и другое *TypeName* — т.е. аннотация выглядит как `@Foo T.U`, — то `U` обозначает внутренний класс `T`.

Второе условие интуитивно понятно следующим образом: если выражение `Outer.this` корректно во вложенном классе, охватываемом `Outer`, то `Outer` может быть аннотирован, поскольку представляет тип некоторого объекта времени выполнения. С другой стороны, если `Outer.this` некорректно — поскольку класс, где оно появляется, не имеет охватывающего экземпляра `Outer` во время выполнения, — то `Outer` не может быть аннотирован, поскольку логически это просто имя сродни компонентам имени пакета в полностью квалифицированном имени типа.

Например, в теле `B` приведенной далее программы нельзя записать `A.this`, так как `B` не имеет лексически охватывающих экземпляров (8.5.1). Следовательно, применение `@Foo` к `A` в типе `A.B` невозможно, поскольку `A` логически является просто именем, но не типом.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    class A {
        static class B {}
    }

    @Foo A.B x; // Некорректно
}
```

С другой стороны, в теле `D` следующей программы можно записать `C.this`. Следовательно, применение `@Foo` к `C` в типе `C.D` возможно, поскольку `C` во время выполнения представляет тип некоторого объекта.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    static class C {
        class D {}
    }

    @Foo C.D x; // Корректно
}
```

Наконец заметим, что во втором случае `D` находится в квалифицированном типе, только на один уровень глубже. Дело в том, что статический класс может быть вложен только в класс верхнего уровня или в иной вложенный статический класс. Невозможно записать вложенную структуру наподобие следующей.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
    class E {
        class F {
            static class G {}
        }
    }

    @Foo E.F.G x;
}
```

Предположим на минуту, что эта вложенность корректна (доказательство от противного). В типе поля `x`, `E` и `F` логически являются именами, квалифицирующими `G`, так что запись `E.F.this` в теле `G` некорректна. Тогда `@Foo` не должна быть корректной после `E`. Однако технически `@Foo` должна быть приемлема после `E`, поскольку следующий по глубине член `F` обозначает внутренний класс; что противоречиво, так как, в первую очередь, некорректно исходное предположение о вложенности классов.

Если аннотация типа *T* применяется к самому внешнему уровню типа в контексте типа и *T* не применим в контексте типа или в контексте объявления (если таковой имеется), находящемся в том же синтаксическом местоположении, генерируется ошибка времени компиляции.

Если аннотация типа *T* применяется к части типа (т.е. не к самому внешнему уровню) в контексте типа и если *T* не применим в контексте типа, генерируется ошибка времени компиляции.

Если аннотация типа *T* применяется к типу (или к любой части типа) в контексте типа, *T* применим в контексте типа, а аннотация неприемлема, генерируется ошибка времени компиляции.

Предположим, что имеется тип аннотации `TA`, метааннотированный только лишь с помощью `@Target(ElementType.TYPE_USE)`. Записи `@TA java.lang.Object` и `java.@TA lang.Object` некорректны, поскольку простое имя, к которому `@TA` располагается ближе всего, классифицируется как имя пакета. С другой стороны, `java.lang.@TA Object` является некорректным.

Обратите внимание, что некорректные записи некорректны “везде”. Запрет на аннотированные имена пакетов применяется широко: как к местоположениям, являющимся исключительно контекстами типов, таким как `class ... extends @TA java.lang.Object {...}`, так и к местоположениям, являющимся как контекстами объявлений, так и контекстами типов, таким как `@TA java.lang.Object f;`. (Не существует местоположений, являющихся исключительно контекстами объявлений, там, где может быть аннотировано имя пакета, так как объявления классов, пакетов и параметров типов используют только простые имена.)

Если `TA` дополнительно метааннотирован с помощью `@Target(ElementType.FIELD)`, то запись `@TA java.lang.Object` корректна в местоположениях, которые являются как контекстами объявлений, так и контекстами типов, таких как

объявление поля `@TA java.lang.Object f;`. Здесь `@TA` считается примененной к объявлению `f` (а не к типу `java.lang.Object`), поскольку `TA` применим в контексте объявления поля.

§9.7.5. Многократные аннотации одного типа

Если несколько аннотаций одного и того же типа T находятся в контексте объявления или в контексте типа, то, если только T не является повторяемым (§9.6.3) и как T, так и содержащий тип аннотации T не являются применимыми в контексте объявления или в контексте типа (§9.6.4.1), то генерируется ошибка времени компиляции.

Обычно, хотя это и не обязательно, несколько аннотаций одного и того же типа располагаются подряд.

Если контекст объявления или контекст типа имеет многократные аннотации повторяемого типа аннотации T, то это выглядит так, как если бы контекст не имел явно объявленной аннотации типа T и одну неявно объявленную аннотацию содержащего типа аннотации для T.

Неявно объявленная аннотация называется *контейнерной аннотацией* (container annotation), а множественные аннотации типа T, находящиеся в этом контексте, называются *базовыми аннотациями* (base annotations). Для массива `value` контейнерной аннотации все его элементы являются базовыми аннотациями в порядке следования элементов слева направо.

Если в контексте объявления или типа имеется несколько аннотаций повторяемого типа аннотации T и любые аннотации содержащего типа аннотации T, генерируется ошибка времени компиляции.

Другими словами, невозможно повторять аннотации там, где встречаются аннотации того же типа, что и их контейнер. Тем самым запрещается глупый код наподобие

```
@Foo(0) @Foo(1) @FooContainer({@Foo(2)})
class A {}
```

Если бы этот код был допустимым, то потребовалось бы несколько уровней содержания: сначала аннотации типа `Foo` должны были бы содержаться в неявно объявленной контейнерной аннотации типа `FooContainer`, затем эта аннотация и явно объявленная аннотация типа `FooContainer` должны были бы содержаться в еще одной неявно объявленной аннотации. Такая сложность, по мнению разработчиков языка программирования Java, нежелательна. Другой подход, рассматривающий аннотации типа `Foo` так, как если бы они находились наряду с `@Foo(2)` в явной аннотации `@FooContainer`, нежелателен, поскольку может изменить интерпретацию аннотации `@FooContainer` рефлективными программами.

Если в контексте объявления или типа имеется одна аннотация повторяемого типа аннотации T и несколько аннотаций содержащего типа аннотации T, генерируется ошибка времени компиляции.

Это правило разработано для того, чтобы позволить следующий код.

```
@Foo(1) @FooContainer({@Foo(2)})
class A {}
```

В случае только одной аннотации повторяемого типа аннотации `Foo` контейнерная аннотация неявно не объявляется, даже если `FooContainer` представляет собой содержащий тип аннотации для `Foo`. Однако повторение аннотации типа `FooContainer`, как в приведенном исходном тексте

```
@Foo(1) @FooContainer({@Foo(2)}) @FooContainer({@Foo(3)})
class A {}
```

запрещено, даже если `FooContainer` повтор*и*м с содержащим типом аннотации для самого себя. Это глупо — повторять аннотации, которые сами являются контейнерами, когда в наличии имеется аннотация базового повторяемого типа.

§9.8. Функциональные интерфейсы

Функциональный интерфейс представляет собой интерфейс, который имеет только один абстрактный метод (отличный от методов `Object`), и, таким образом, предоставляет только один контракт функции. Этот "единственный" метод может принимать вид множества абстрактных методов с эквивалентными в смысле перекрытия сигнатурами, унаследованных от суперинтерфейсов; в таком случае унаследованные методы логически представляют один метод.

Пусть имеется интерфейс I и пусть M — множество абстрактных методов, являющихся членами I, сигнатуры которых не совпадают ни с одной из сигнатур открытых методов класса `Object`. Тогда, если в M имеется метод m, для которого выполняются оба приведенных ниже условия, то I является *функциональным интерфейсом*.

- Сигнатура m является подсигнатурой (§8.4.2) сигнатуры каждого из методов в M.
- m является заменяемым по возвращаемому типу (§8.4.5) для каждого метода из M.

В дополнение к обычному процессу создания экземпляра интерфейса с помощью объявления и инстанцирования класса (§15.9) экземпляры функциональных интерфейсов могут создаваться с помощью выражений ссылок на методы и лямбда-выражений (§15.13, §15.27).

Определение *функционального интерфейса* исключает методы интерфейса, которые одновременно являются открытыми методами в `Object`. Это сделано для того, чтобы позволить функциональную трактовку интерфейсов наподобие `java.util.Comparator<T>`, в которых объявлено несколько абстрактных методов, из которых только один в действительности является "новым" — `int compare(T,T)`. Другой метод — `boolean equals(Object)` — является явным объявлением абстрактного метода, который в противном случае был бы объявлен неявно и автоматически реализовывался бы каждым классом, реализующим указанный интерфейс.

Обратите внимание, что если в интерфейсе объявлены методы `Object`, не являющиеся `public` (такие, как `clone()`), они *не* реализуются автоматически каждым классом, реализующим интерфейс. Унаследованная от `Object` реализация является `protected`, в то время как метод интерфейса обязан быть `public`. Единственный способ реализации такого интерфейса классом состоит в перекрытии методов `Object`, не являющихся `public`, методами, объявленными как `public`.

Пример 9.8-1. Функциональные интерфейсы

Простым примером функционального интерфейса является

```
interface Runnable {
    void run();
}
```

Приведенный далее интерфейс не является функциональным, поскольку он не объявляет ничего, что бы уже не было членом `Object`.

```
interface NonFunc {
    boolean equals(Object obj);
}
```

Однако его подынтерфейс может быть функциональным благодаря объявлению абстрактного метода, который не является членом `Object`.

```
interface Func extends NonFunc {
    int compare(String o1, String o2);
}
```

Аналогично хорошо известный интерфейс `java.util.Comparator<T>` является функциональным, поскольку он имеет единственный абстрактный метод, которого нет у `Object`.

```
interface Comparator<T> {
    boolean equals(Object obj);
    int compare(T o1, T o2);
}
```

Приведенный далее интерфейс не является функциональным, поскольку, хотя он и объявляет только один абстрактный метод, не являющийся членом `Object`, он объявляет *два* абстрактных метода, не являющихся открытыми членами `Object`.

```
interface Foo {
    int m();
    Object clone();
}
```

Пример 9.8-2. Функциональные интерфейсы и затирания

В рассматриваемой далее иерархии интерфейсов `Z` представляет собой функциональный интерфейс, поскольку, хотя он наследует два абстрактных метода, которые не являются членами `Object`, они имеют одинаковые сигнатуры, так что унаследованные методы логически представляют один метод.

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {}
```

Аналогично `Z` в приведенной далее иерархии интерфейсов является функциональным интерфейсом, поскольку `Y.m` является подсигнатурой `X.m` и является заменяемым по возвращаемому типу для `X.m`.

```
interface X { Iterable m(Iterable<String> arg); }
interface Y { Iterable<String> m(Iterable arg); }
interface Z extends X, Y {}
```

Определение *функционального интерфейса* учитывает тот факт, что интерфейс не может иметь два члена, которые не являются подсигнатурами один другого, но имеют при этом одинаковое затирание (§9.4.1.2). Таким образом, в следующих трех иерархиях интерфейсов, где `Z` приводит к ошибке времени компиляции, `Z` не является функциональным интерфейсом (поскольку ни один из его абстрактных членов не является подсигнатурой для всех прочих абстрактных членов).

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {}

interface X { int m(Iterable<String> arg, Class c); }
interface Y { int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {}

interface X<T> { void m(T arg); }
interface Y<T> { void m(T arg); }
interface Z<A, B> extends X<A>, Y<B> {}
```

Аналогично определение "функционального интерфейса" учитывает тот факт, что интерфейс может иметь методы с сигнатурами, эквивалентными в смысле перекрытия, только если один из них является заменяемым по возвращаемому типу для всех прочих. Таким образом, в приведенной далее иерархии интерфейсов там, где `Z` приводит к ошибке времени компиляции, `Z` не является функциональным интерфейсом (поскольку ни один из его абстрактных членов не является заменяемым по возвращаемому типу для всех остальных абстрактных членов).

```
interface X { long m(); }
interface Y { int m(); }
interface Z extends X, Y {}
```

В следующем примере объявления `Foo<T,N>` и `Bar` корректны: в каждом из них методы с именем `m` не являются подсигнатурами один другого, но имеют различные затирания. Тем не менее тот факт, что методы в каждом из объявлений не являются подсигнатурами, означает, что `Foo<T,N>` и `Bar` не являются функциональными подынтерфейсами. Однако `Baz` представляет собой функциональный интерфейс, поскольку методы, которые он наследует от `Foo<Integer,Integer>`, имеют одну и ту же сигнатуру, так что логически они представляют собой один метод.

```
interface Foo<T, N extends Number> {
    void m(T arg);
    void m(N arg);
}
interface Bar extends Foo<String, Integer> {}
interface Baz extends Foo<Integer, Integer> {}
```

Наконец в следующих примерах продемонстрированы те же правила, что и выше, но с обобщенными методами.

```
interface Exec { <T> T execute(Action<T> a); }
    // Функциональный
```

```
interface X { <T> T execute(Action<T> a); }
interface Y { <S> S execute(Action<S> a); }
interface Exec extends X, Y {}
    // Функциональный: сигнатуры логически одинаковы

interface X { <T> T execute(Action<T> a); }
interface Y { <S,T> S execute(Action<S> a); }
interface Exec extends X, Y {}
    // Ошибка: разные сигнатуры, одни и те же затирания
```

Пример 9.8-3. Обобщенные функциональные интерфейсы

Функциональные интерфейсы могут быть обобщенными, такими как `java.util.function.Predicate<T>`. Такой функциональный интерфейс может быть параметризован таким образом, что дает различные абстрактные методы, т.е. несколько методов, которые не могут быть корректно перекрыты одним объявлением, например

```
interface I { Object m(Class c); }
interface J<S> { S m(Class<?> c); }
interface K<T> { T m(Class<?> c); }
interface Functional<S,T> extends I, J<S>, K<T> {}
```

`Functional<S,T>` представляет собой функциональный интерфейс — `I.m` является заменяемым по возвращаемому типу для `J.m` и `K.m`, — но тип функционального интерфейса `Functional<String,Integer>`, очевидно, не может быть реализован с единственным методом. Однако возможны другие параметризации `Functional<S,T>`, являющиеся типами функциональных интерфейсов.

Объявление функционального интерфейса позволяет использовать в программе *тип функционального интерфейса*. Имеются четыре разновидности типа функционального интерфейса.

- Тип необобщенного (§6.1) функционального интерфейса.
- Параметризованный тип, который представляет собой параметризацию (§4.5) обобщенного функционального интерфейса.
- Несформированный тип (§4.8) обобщенного функционального интерфейса.
- Тип пересечения (§4.9), который порождает отвлеченный функциональный интерфейс.

В особых случаях полезно рассматривать тип пересечения как тип функционального интерфейса. Как правило, это выглядит как пересечение типа функционального интерфейса с одним или несколькими типами интерфейса-маркера, как, например, `Runnable & java.io.Serializable`. Такое пересечение может использоваться в приведениях (§15.16), которые заставляют лямбда-выражение соответствовать определенному типу. Если один из типов интерфейсов в пересечении является `java.io.Serializable`, запускается специальная поддержка времени выполнения для сериализации (§15.27.4).

§9.9. Типы функций

Типом функции функционального интерфейса `I` является тип метода (§8.2), который может использоваться для перекрытия (§8.4.8) абстрактного метода (методов) `I`.

Пусть `M` представляет собой множество абстрактных методов, определенных для `I`. Тип функции `I` состоит из следующего.

- Параметры типов, формальные параметры и возвращаемый тип.

  Пусть `m` представляет собой метод в `M` с

  1. сигнатурой, которая является подсигнатурой сигнатуры каждого из методов в `M`; и
  2. возвращаемым типом, который представляет собой подтип возвращаемых типов каждого из методов в `M` (после адаптации для любых параметров типов (§8.4.4)).

  Если такой метод не существует, то пусть `m` представляет собой метод в `M`, который

  1. имеет сигнатуру, которая является подсигнатурой сигнатуры каждого метода в `M`; и
  2. является заменяемым по возвращаемому типу (§8.4.5) для каждого метода в `M`.

  Параметры типа, типы формальных параметров и возвращаемый тип у типа функции задаются методом `m`.

- Конструкция `throws`.

  Конструкция `throws` типа функции является производной от конструкций `throws` методов в `M`. Если тип функции является обобщенным, эти конструкции сначала адаптируются к параметрам типов типа функции (§8.4.4). Если тип функции не является обобщенным, но как минимум один метод в `M` обобщенный, эти конструкции сначала затираются. Затем конструкция `throws` типа функции включает каждый тип `E`, который удовлетворяет следующим ограничениям:

  ✦ `E` упоминается в одной из конструкций `throws`;

  ✦ для каждой конструкции `throws` `E` является подтипом некоторого типа, указанного в этой конструкции.

  Если одни возвращаемые типы в `M` несформированные, а другие таковыми не являются, определение типа функции пытается выбрать наиболее конкретный тип, насколько это возможно. Например, если возвращаемыми типами являются `LinkedList` и `LinkedList<String>`, то в качестве возвращаемого типа для типа функции выбирается второй из них. Если наиболее конкретного типа не существует, определение уточняется с помощью поиска наиболее подставляемого возвращаемого типа. Например, если имеется третий возвращаемый тип, `List<?>`, то в этом случае один из возвращаемых типов не является подтипом всех других (несформированный `LinkedList` не является подтипом `List<?>`); вместо этого в качестве возвращаемого типа для типа функции выбирается `LinkedList<String>`, поскольку он является заменяемым по возвращаемому типу как для `LinkedList`, так и для `List<?>`.

  Цель, которой руководствуются при определении типов исключений типа функции состоит в поддержке инварианта, заключающегося в том, что метод с результирующей

> конструкцией `throws` может перекрывать каждый абстрактный метод функционального интерфейса. Согласно §8.4.6 это означает, что тип функции не может генерировать "больше" исключений, чем любой отдельный метод в множестве *M*, так что мы ищем как можно больше типов исключений, которые "покрываются" конструкциями `throws` каждого метода.

Тип функции типа функционального интерфейса определяется следующим образом.

- Тип функции типа необобщенного функционального интерфейса I представляет собой просто тип функции функционального интерфейса I, как определено выше.
- Тип функции типа параметризированного функционального интерфейса $I<A_1...A_n>$, где $A_1...A_n$ — типы, а соответствующими параметрами типов I являются $P_1...P_n$, получается применением подстановки $[P_1:=A_1, ..., P_n:=A_n]$ к типу функции обобщенного функционального интерфейса $I<P_1...P_n>$.
- Типом функции типа параметризированного функционального интерфейса $I<A_1...A_n>$, где один или несколько из аргументов $A_1...A_n$ представляют собой символ подстановки, является тип функции *параметризации без символов подстановки* (nonwildcard parameterization) типа I, $I<T_1...T_n>$. Параметризация без символов подстановки определяется следующим образом.

  Пусть $P_1...P_n$ являются параметрами типов I с соответствующими границами $B_1...B_n$. Для всех i $(1 \le i \le n)$ T_i получается в соответствии с видом A_i.

  - Если A_i является типом, то $T_i = A_i$.
  - Если A_i является символом подстановки, а соответствующая граница параметра типа, B_i, упоминает один из параметров $P_1...P_n$, то T_i не определено и типа функции не существует.
  - В противном случае имеем следующее.
    - Если A_i представляет собой неограниченный символ подстановки `?`, то $T_i = B_i$.
    - Если A_i представляет собой ограниченный сверху символ подстановки `? extends` U_i, то $T_i = \text{glb}(U_i, B_i)$ (§5.1.10).
    - Если A_i представляет собой ограниченный снизу символ подстановки `? super` L_i, то $T_i = L_i$.
- Тип функции несформированного типа обобщенного функционального интерфейса $I<...>$ является затиранием типа функции обобщенного функционального интерфейса $I<...>$.
- Тип функции типа пересечения, которое порождает отвлеченный функциональный интерфейс, представляет собой тип функции отвлеченного функционального интерфейса.

> **Пример 9.9-1. Типы функций**
>
> Пусть даны следующие интерфейсы:
>
> ```
> interface X { void m() throws IOException; }
> interface Y { void m() throws EOFException; }
> interface Z { void m() throws ClassNotFoundException; }
> ```

Тогда типом функции для

```
interface XY extends X, Y {}
```

является

```
()->void throws EOFException
```

в то время как типом функции для

```
interface XYZ extends X, Y, Z {}
```

является

```
()->void (throws nothing)
```

Если даны интерфейсы

```
interface A {
    List<String> foo(List<String> arg)
        throws IOException, SQLTransientException;
}
interface B {
    List foo(List<String> arg)
        throws EOFException, SQLException, TimeoutException;
}
interface C {
    List foo(List arg) throws Exception;
}
```

то типом функции для

```
interface D extends A, B {}
```

является

```
(List<String>)->List<String>
    throws EOFException, SQLTransientException
```

в то время как типом функции для

```
interface E extends A, B, C {}
```

является

```
(List)->List throws EOFException, SQLTransientException
```

Тип функции функционального интерфейса определяется недетерминистически: в то время как сигнатуры в *M* являются "одинаковыми", синтаксически они могут различаться (например, `HashMap.Entry` и `Map.Entry`); возвращаемый тип может быть подтипом каждого из прочих возвращаемых типов, но могут быть и другие возвращаемые типы, которые *также* являются подтипами (например, `List<?>` и `List<? extends Object>`); порядок типов исключений не определен. Это тонкие различия, но иногда они могут оказаться важными. Однако типы функций в языке программирования Java не используются таким образом, при котором недетерминизм мог бы иметь значение. Обратите внимание, что возвращаемый тип и конструкция `throws` "наиболее конкретного метода" при наличии нескольких абстрактных методов также определяются недетерминистически (§15.12.2.5).

Когда обобщенный функциональный интерфейс параметризован символами подстановки, имеется множество различных инстанцирований, которые могут удовлетворять символам подстановки и генерировать различные типы функций. Например,

типы `Predicate<Integer>` (тип функции `Integer -> boolean`), `Predicate <Number>` (тип функции `Number -> boolean`) и `Predicate<Object>` (тип функции `Object -> boolean`) являются `Predicate<? super Integer>`. Иногда оказывается возможным узнать из контекста, такого как типы параметров лямбда-выражения, какой тип функции намечался (§15.27.3). В других случаях необходимо его выбрать; при этом используются границы. (Эта простая стратегия не может гарантировать, что результирующий тип будет удовлетворять некоторым сложным границам, так что поддерживаются не все сложные случаи.)

Пример 9.9-2. Обобщенные типы функций

Тип функции может быть обобщенным, если обобщенным может быть абстрактный метод функционального интерфейса. Например, в приведенной иерархии интерфейсов

```
interface G1 {
    <E extends Exception> Object m() throws E;
}
interface G2 {
    <F extends Exception> String m() throws Exception;
}
interface G extends G1, G2 {}
```

типом функции `G` является

```
<F extends Exception> ()->String throws F
```

Обобщенный тип функции для функционального интерфейса может быть реализован с помощью выражения ссылки на метод (§15.13), но не с помощью лямбда-выражения (§15.27), так как синтаксис для обобщенных лямбда-выражений отсутствует.

ГЛАВА 10

Массивы

В языке программирования Java *массивы* являются объектами (§4.3.1), создаются динамически и могут присваиваться переменным типа `Object` (§4.3.2). Все методы класса `Object` можно вызывать для массива.

Объект массива содержит ряд переменных. Количество переменных может быть равно нулю (в этом случае массив считается пустым). Переменные, содержащиеся в массиве, не имеют имен. Вместо этого обращение к ним осуществляется с помощью выражений доступа к массиву, использующих значения индекса, которые представляют собой неотрицательные целые числа. Эти переменные называются *компонентами массива*. Если массив имеет *n* компонентов, мы говорим, что *n* представляет собой длину массива; обращение к компонентам массива осуществляется с помощью целочисленных индексов от 0 до $n-1$ включительно.

Все компоненты массива имеют один и тот же тип — *тип компонента* массива. Если типом компонента массива является *T*, то тип самого массива записывается как `T[]`.

Значение компонента массива типа `float` всегда является элементом набора значений float (§4.2.3); аналогично значение компонента массива типа `double` всегда является элементом набора значений double. Значениям компонентов массива типа `float` не разрешено быть элементами набора значений float с расширенным показателем степени, не являющимися одновременно элементами набора значений float. Аналогично значениям компонентов массива типа `double` не разрешено быть элементами набора значений double с расширенным показателем степени, не являющимися одновременно элементами набора значений double.

Тип компонента массива сам может быть типом массива. Компоненты такого массива могут содержать ссылки на подмассивы. Если, начиная с любого типа массива, рассмотреть тип его компонентов, а затем (если он также представляет собой тип массива) тип компонентов этого типа и так далее, то в конце концов будет достигнут тип компонента, который не является типом массива. Он называется *типом элемента* исходного массива, а компоненты на этом уровне структуры данных называются *элементами* исходного массива.

Существует несколько ситуаций, в которых элемент массива может быть массивом: если тип элемента представляет собой `Object`, `Cloneable` или `java.io.Serializable`, то некоторые или все элементы могут быть массивами, поскольку любой объект массива может быть присвоен любой переменной этих типов.

§10.1. Типы массивов

Типы массивов используются в объявлениях и выражениях приведения (§15.16).

Тип массива записывается как имя типа элементов, за которым следует некоторое количество пустых пар квадратных скобок `[]`. Количество пар скобок указывает глубину вложенности массива.

Длина массива частью типа не является.

Тип элемента массива может быть любым типом, примитивным или ссылочным. В частности, справедливо следующее.

- Разрешены массивы с типом интерфейса в качестве типа элемента.

  Элемент такого массива может иметь в качестве значения пустую ссылку или экземпляр любого типа, который реализует данный интерфейс.
- Разрешены массивы с типом абстрактного класса в качестве типа элементов.

  Элемент такого массива может иметь в качестве значения пустую ссылку или экземпляр любого не являющегося абстрактным подкласса этого абстрактного класса.

Супертипы типа массива описаны в §4.10.3.

> Отношение супертипа для типов массивов не совпадает с отношением суперкласса. Непосредственным супертипом `Integer[]` в соответствии с §4.10.3 является `Number[]`, но непосредственным суперклассом `Integer[]` является `Object` в соответствии с объектом `Class` для `Integer[]` (§10.8). На практике это не имеет значения, так как `Object` является супертипом для всех типов массивов.

§10.2. Переменные массивов

Переменная типа массива хранит ссылку на объект. Объявление переменной типа массива не создает объект массива и не выделяет память для его компонентов. Оно создает только саму переменную, которая может содержать ссылку на массив.

Однако инициализирующая часть объявления (§8.3, §9.3, §14.4.1) может создавать массив, ссылка на который становится исходным значением переменной.

Пример 10.2-1. Объявления переменных массивов

```
int[]      ai;         // Массив int
short[][]  as;         // Массив массивов short
short      s,          // Переменная типа short
           aas[][];    // Массив массивов short
Object[]   ao,         // Массив Object
           otherAo;    // Массив Object
Collection<?>[] ca;    // Массив Collection неизвестного типа
```

Приведенные выше объявления не создают объектов массивов. Объявления переменных массивов, которые создают объекты массивов, показаны ниже.

```
Exception ae[]  = new Exception[3];
Object aao[][]  = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
```

```
char ac[]       = { 'n', 'o', 't', ' ', 'a', ' ',
                    'S', 't', 'r', 'i', 'n', 'g' };
String[] aas    = { "array", "of", "String", };
```

Квадратные скобки `[]` могут встречаться в качестве части типа в начале объявления либо быть частью объявления определенной переменной, либо находиться в обоих местах одновременно.

Например:

```
byte[] rowvector, colvector, matrix[];
```

Это объявление эквивалентно следующему.

```
byte rowvector[], colvector[], matrix[][];
```

В объявлении переменной (§8.3, §8.4.1, §9.3, 9.4, §14.4.1, §14.14.2, §15.27.1), за исключением параметра переменной арности, тип массива переменной обозначается типом массива, находящимся в начале объявления, за которым следуют пары квадратных скобок, а затем *Identifier* переменной.

Например, объявление локальных переменных

```
int a, b[], c[][];
```

эквивалентно ряду объявлений

```
int a;
int[] b;
int[][] c;
```

Скобки в объявлениях разрешены как дань традиции C и C++. Однако общие правила объявлений переменных разрешают скобкам находиться как в типе, так и у идентификатора переменной, так что объявление локальных переменных

```
float[][] f[][], g[][][], h[];
```

эквивалентно ряду объявлений

```
float[][][][] f;
float[][][][][] g;
float[][][] h;
```

Мы не рекомендуем такую смешанную запись, когда пары скобок имеются в обеих частях объявления.

После создания объекта массива его длина никогда не меняется. Чтобы сделать переменную массива ссылающейся на массив иной длины, переменной следует присвоить ссылку на другой массив.

Одна переменная типа массива может содержать ссылки на массивы разной длины, потому что длина массива не является частью его типа.

Если переменная массива `v` имеет тип `A[]`, где `A` является ссылочным типом, то `v` может содержать ссылку на экземпляр любого типа массива `B[]`, что обеспечивает возможность того, что `B` может быть присвоен `A` (§5.2). Это может привести к исключению во время выполнения при позднем присваивании; этот вопрос рассматривается в §10.5.

§10.3. Создание массива

Массив создается с помощью выражения создания массива (§15.10.1) или инициализатора массива (§10.6).

Выражение создания массива указывает тип элемента, количество уровней вложенности массивов и длину массива как минимум для одного из уровней вложенности. Длина массива доступна как переменная экземпляра `length`, объявленная как `final`.

Инициализатор массива создает массив и предоставляет исходные значения его компонентов.

§10.4. Доступ к массивам

Обращение к компоненту массива осуществляется с помощью выражения доступа к массиву (§15.10.3), которое состоит из выражения, значение которого представляет собой ссылку на массив, за которой следует индексное выражение в квадратных скобках, как в записи `A[i]`.

Нумерация элементов любых массивов начинается с нуля. Массив длиной n может быть индексирован целыми числами от 0 до $n-1$.

Пример 10.4-1. Обращение к массиву

```
class Gauss {
    public static void main(String[] args) {
        int[] ia = new int[101];
        for (int i = 0; i < ia.length; i++) ia[i] = i;
        int sum = 0;
        for (int e : ia) sum += e;
        System.out.println(sum);
    }
}
```

Вывод программы имеет вид

```
5050
```

Программа объявляет переменную `ia`, которая имеет тип массива элементов типа `int`, т.е. `int[]`. Переменная `ia` инициализируется как указывающая на новый объект, созданный с помощью выражения создания массива (§15.10.1). Выражение создания массива указывает, что в массиве должен быть `101` компонент. Как показано в исходном тексте, длина массива доступна посредством поля `length`. Программа заполняет массив целыми числами от `0` до `100`, суммирует их и выводит результат.

Массивы должны индексироваться значениями типа `int`; значения типа `short`, `byte` или `char` также могут использоваться в качестве значений индексов, так как над ними может быть выполнено унарное числовое повышение (§5.6.1) и они станут значениями типа `int`.

Попытки обращения к компонентам массива с помощью индекса типа `long` приводят к ошибке времени компиляции.

Все обращения к массивам проверяются во время выполнения; попытка использовать индекс меньше нуля или больший или равный длине массива приводит к генерации исключения `ArrayIndexOutOfBoundsException` (§15.10.4).

§10.5. Исключение `ArrayStoreException`

Для массива типа `A[]`, где `A` является ссылочным типом, во время выполнения выполняется проверка присваиваний компоненту массива, чтобы гарантировать, что конкретное значение действительно может быть присвоено компоненту массива.

Если тип присваиваемого значения не совместим по присваиванию (§5.2) с типом компонента, генерируется исключение `ArrayStoreException`.

> Если тип компонента массива недоступен во время выполнения (§4.7), виртуальная машина Java не может выполнить проверку, описанную в предыдущем абзаце. Поэтому выражение создания массива с недоступным во время выполнения типом элементов запрещено (§15.10.1). Можно объявить переменную типа массива, тип элемента которого недоступен во время выполнения, но присваивание результата выражения создания массива переменной обязательно вызовет предупреждение о непроверенном типе (§5.1.9).
>
> **Пример 10.5-1. `ArrayStoreException`**
>
> ```
> class Point { int x, y; }
> class ColoredPoint extends Point { int color; }
> class Test {
>     public static void main(String[] args) {
>         ColoredPoint[] cpa = new ColoredPoint[10];
>         Point[] pa = cpa;
>         System.out.println(pa[1] == null);
>         try {
>             pa[0] = new Point();
>         } catch (ArrayStoreException e) {
>             System.out.println(e);
>         }
>     }
> }
> ```
>
> Вывод этой программы имеет вид
>
> ```
> true
> java.lang.ArrayStoreException: Point
> ```
>
> Переменная `pa` имеет тип `Point[]`, а переменная `cpa` имеет в качестве значения ссылку на объект типа `ColoredPoint[]`. Объект типа `ColoredPoint` может быть присвоен объекту типа `Point`; следовательно, значение `cpa` может быть присвоено переменной `pa`.
>
> Обращение к этому массиву `pa`, например, при проверке `pa[1]` на равенство `null` не приводит к ошибке времени выполнения. Это связано с тем, что элемент массива типа `ColoredPoint[]` представляет собой `ColoredPoint`, а каждый

> `ColoredPoint` может заменять собой `Point`, поскольку `Point` является суперклассом для `ColoredPoint`.
>
> С другой стороны, присваивание массиву `pa` может привести к ошибке времени выполнения. Во время компиляции присваивание элементу `pa` проверяется, чтобы гарантировать, что присваиваемое значение представляет собой `Point`. Но поскольку `pa` хранит ссылку на массив `ColoredPoint`, присваивание корректно, только если тип присваиваемого значения во время выполнения представляет собой именно `ColoredPoint`.
>
> Виртуальная машина Java выполняет проверку таких ситуаций во время выполнения, чтобы гарантировать корректность присваивания; в случае некорректного присваивания генерируется исключение `ArrayStoreException`.

§10.6. Инициализаторы массивов

Инициализатор массива может быть указан в объявлении поля (§8.3, §9.3) или локальной переменной (§14.4) или как часть выражения создания массива (§15.10.1) для создания массива и представления некоторых его исходных значений.

ArrayInitializer:
{ *[VariableInitializerList] [,]* }

VariableInitializerList:
VariableInitializer {, VariableInitializer}

Далее для ясности следует повторение фрагмента из §8.3.

VariableInitializer:
Expression
ArrayInitializer

Инициализатор массива записывается как список выражений, разделенных запятыми, заключенный в фигурные скобки { }.

После последнего выражения списка может располагаться завершающая запятая, которая игнорируется.

Каждый инициализатор переменной должен быть совместим по присваиванию (§5.2) с типом компонентов массива, иначе генерируется ошибка времени компиляции.

Если тип компонента инициализируемого массива недоступен во время выполнения (§4.7), генерируется ошибка времени компиляции.

Длина создаваемого массива должна быть равна количеству инициализаторов переменных, непосредственно заключенных в фигурные скобки инициализатора массива. Память выделяется для нового массива именно этой длины. Если для массива не хватает памяти, вычисление инициализатора массива прерывается генерацией исключения `OutOfMemoryError`. В противном случае создается одномерный массив указанной длины, а каждый компонент массива инициализируется его значением по умолчанию (§4.12.5).

Затем сразу же выполняются заключенные в фигурные скобки инициализаторы переменных. Выполнение происходит слева направо в порядке их появления в исходном

тексте. *n*-й инициализатор переменной определяет значение $n-1$-го компонента массива. В случае прерывания выполнения инициализатора переменной по той же причине завершается выполнение инициализатора массива. Если все выражения инициализаторов переменных завершаются нормально, нормально завершается и инициализатор массива, давая значение вновь инициализированного массива.

Если тип компонента представляет собой тип массива, то инициализатор переменной, определяющий компонент, сам может быть инициализатором массива; таким образом, инициализаторы массивов могут быть вложенными. В этом случае выполнение вложенного инициализатора массива строит и инициализирует объект с помощью рекурсивного применения описанного выше алгоритма и присваивает его компоненту.

Пример 10.6-1. Инициализаторы массивов

```
class Test {
    public static void main(String[] args) {
        int ia[][] = { {1, 2}, null };
        for (int[] ea : ia) {
            for (int e: ea) {
                System.out.println(e);
            }
        }
    }
}
```

Вывод этой программы имеет следующий вид.

```
12
```

После этого происходит генерация исключения `NullPointerException` при попытке индексации второго компонента массива `ia`, который представляет собой ссылку `null`.

§10.7. Члены массивов

Члены типа массива перечислены ниже.

- Поле `length`, объявленное как `public final`, которое содержит количество компонентов массива. Значение `length` может быть положительным или нулевым.
- Метод `clone`, объявленный как `public`, который перекрывает метод с тем же именем класса `Object` и не генерирует проверяемых исключений. Типом возвращаемого значения метода `clone` типа массива *`T`*`[]` является *`T`*`[]`.

  Клон многомерного массива строится на основании так называемого поверхностного копирования, т.е. создается только один новый массив. Подмассивы же являются общими.
- Все эти члены унаследованы от класса `Object`; единственный метод `Object`, который не является унаследованным, — метод `clone`.

Смотрите в §9.6.4.4 другую ситуацию, в которой различия между методами `Object`, объявленными как `public` и не как `public`, требуют особого внимания.

Таким образом, массив имеет те же `public`-поля и методы, что и следующий класс.

```
class A<T> implements Cloneable, java.io.Serializable {
    public final int length = X ;
    public T[] clone() {
        try {
            return (T[])super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.getMessage());
        }
    }
}
```

Заметим, что приведение к `T[]` в рассмотренном выше примере генерировало бы предупреждение о непроверенном типе (§5.1.9), если бы массивы действительно были реализованы таким образом.

Пример 10.7-1. Массивы клонируемы

```
class Test1 {
    public static void main(String[] args) {
        int ia1[] = { 1, 2 };
        int ia2[] = ia1.clone();
        System.out.print((ia1 == ia2) + " ");
        ia1[1]++;
        System.out.println(ia2[1]);
    }
}
```

Вывод этой программы имеет следующий вид.

```
false 2
```

Он демонстрирует, что компоненты массивов, на которые ссылаются переменные `ia1` и `ia2`, являются разными переменными.

Пример 10.7-2. Общие подмассивы после клонирования

Тот факт, что подмассивы после клонирования многомерного массива являются общими, иллюстрируется следующей программой.

```
class Test2 {
    public static void main(String[] args) throws Throwable {
        int ia[][] = { {1,2}, null };
        int ja[][] = ia.clone();
        System.out.print((ia == ja) + " ");
        System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
    }
}
```

Вывод этой программы имеет следующий вид.

```
false true
```

Он демонстрирует, что массив `int[]`, который представляет собой `ia[0]`, и массив `int[]`, который представляет собой `ja[0]`, являются одним и тем же массивом.

§10.8. Объекты `Class` массивов

Каждый массив имеет связанный с ним объект типа `Class`, разделяемый со всеми другими массивами с тем же типом компонентов.

Хотя тип массива не является классом, объект `Class` каждого массива действует так, как если бы выполнялось следующее:

- непосредственным суперклассом каждого типа массива был бы `Object`;
- каждый тип массива реализовывал бы интерфейсы `Cloneable` и `java.io.Serializable`.

Пример 10.8-1. Объект `Class` массива

```
class Test1 {
    public static void main(String[] args) {
        int[] ia = new int[3];
        System.out.println(ia.getClass());
        System.out.println(ia.getClass().getSuperclass());
        for (Class<?> c : ia.getClass().getInterfaces())
            System.out.println("Superinterface: " + c);
    }
}
```

Вывод этой программы имеет следующий вид.

```
class [I
class java.lang.Object
Superinterface: interface java.lang.Cloneable
Superinterface: interface java.io.Serializable
```

Здесь строка `"[I"` представляет собой сигнатуру типа времени выполнения для объекта `Class` "массив с типом компонентов `int`".

Пример 10.8-2. Разделяемые объекты `Class` массива

```
class Test2 {
    public static void main(String[] args) {
        int[] ia = new int[3];
        int[] ib = new int[6];
        System.out.println(ia == ib);
        System.out.println(ia.getClass() == ib.getClass());
    }
}
```

Вывод этой программы имеет следующий вид.

```
false
true
```

В то время как `ia` и `ib` ссылаются на различные массивы, результат сравнения объектов `Class` демонстрирует, что все массивы, компоненты которых имеют тип `int`, являются экземплярами одного и того же типа массива (а именно — `int[]`).

§10.9. Массив символов не является строкой

В языке программирования Java, в отличие от C, ни массив элементов типа `char` не является строкой `String`, ни строка `String` не является массивом элементов типа `char` с завершающим символом `'\u0000'` (символ NUL).

Объект `String` является неизменяемым, т.е. его содержимое никогда не изменяется, в то время как массив элементов типа `char` имеет изменяемые элементы.

Метод `toCharArray` в классе `String` возвращает массив символов, содержащий ту же последовательность символов, что и `String`. Класс `StringBuffer` реализует полезные методы для изменяемых массивов символов.

ГЛАВА 11

Исключения

Когда программа нарушает семантические ограничения языка программирования Java, виртуальная машина Java сигнализирует об этой ошибке программе с помощью *исключения*.

Примером такого нарушения является попытка обращения к элементу массива, находящемуся за его границами. Одни языки программирования и их реализации реагируют на такие ошибки аварийным завершением программы; другие языки позволяют реализации реагировать произвольным или непредсказуемым образом. Ни один из этих подходов не совместим с целями разработки платформы Java SE: обеспечить переносимость и надежность.

Язык программирования Java вместо этого определяет, что в случае нарушения семантических ограничений будет *сгенерировано* (throw) исключение, которое приведет к нелокальной передаче управления из точки, в которой исключение сгенерировано, в точку, которая может быть указана программистом (где оно *перехватывается* (catch)).

Программы также могут явно генерировать исключения с помощью инструкции `throw` (§14.18).

Явное применение инструкций `throw` предоставляет альтернативу старому стилю обработки ошибок путем возврата некоторых специальных значений, таких как значение `-1`, там, где отрицательное значение обычно не ожидается. Опыт показывает, что такие значения слишком часто игнорируются или не проверяются вызывающим кодом, что приводит к ненадежности программ или их нежелательному поведению.

Каждое исключение представлено экземпляром класса `Throwable` или одним из его подклассов (§11.1). Такой объект может быть использован для переноса информации из точки, в которой сгенерировано исключение, в точку, в которой оно перехватывается обработчиком. Обработчики указываются конструкциями `catch` инструкций `try` (§14.20).

В процессе генерации исключения виртуальная машина Java немедленно поочередно завершает все выражения, инструкции, вызовы методов и конструкторов, инициализаторы и выражения инициализации полей, которые были начаты, но не завершены в текущем потоке. Этот процесс продолжается до тех пор, пока не будет обнаружен обработчик, который указывает, что он обрабатывает это конкретное исключение, путем именования класса исключения или его суперкласса (§11.2). Если такой обработчик не найден, то исключение может быть обработано одним из элементов иерархии обработчиков неперехваченных сообщений (§11.3). Таким образом, делается все возможное, чтобы не допустить возможности существования неперехваченного исключения.

Механизм исключений платформы Java SE интегрирован с его моделью синхронизации (§17.1), так что немедленно освобождаются мониторы, заблокированные инструкциями `synchronized` (§14.19) и вызовами методов, объявленных как `synchronized` (§8.4.3.6, §15.12).

§11.1. Виды и причины исключений

§11.1.1. Виды исключений

Исключение представлено экземпляром класса `Throwable` (непосредственным подклассом `Object`) или одним из его подклассов.

`Throwable` и все его подклассы коллективно представляют собой *классы исключений.*

Классы `Exception` и `Error` являются непосредственными подклассами класса `Throwable`.

- `Exception` является суперклассом всех исключений, при которых обычные программы могут пожелать восстановиться.

  Класс `RuntimeException` является непосредственным подклассом `Exception`. `RuntimeException` представляет собой суперкласс всех исключений, которые могут быть сгенерированы по разным причинам в процессе вычисления выражения, но восстановление после которых все еще возможно.

  `RuntimeException` и все его подклассы коллективно представляют собой *классы исключений времени выполнения.*

- `Error` является суперклассом всех исключений, при которых восстановление обычных программ нереально.

  `Error` и все его подклассы коллективно представляют собой *классы ошибок.*

Классы непроверяемых исключений представляют собой классы исключений времени выполнения и классы ошибок.

К *классам проверяемых исключений* относятся все классы исключений, отличные от непроверяемых исключений. То есть классы проверяемых исключений представляют собой все подклассы `Throwable`, отличные от `RuntimeException` и его подклассов, а также `Error` и его подклассов.

Программы в инструкциях `throw` могут как использовать предопределенные классы исключений API платформы Java SE, так и определять дополнительные классы исключений как подклассы `Throwable` или любого подходящего из его подклассов. Чтобы воспользоваться преимуществами проверки обработчиков исключений времени компиляции (§11.2), обычно определяются новые классы исключений, являющиеся проверяемыми исключениями, т.е. как подклассы `Exception`, не являющиеся подклассами `RuntimeException`.

Класс `Error` представляет собой отдельный подкласс `Throwable`, отличный от `Exception` в иерархии классов, что позволяет программе использовать идиому "`} catch(Exception e) {`" (§11.2.3) для перехвата всех исключений, восстанов

ление после которых возможно, без перехвата ошибок, восстановление после которых обычно невозможно.

Обратите внимание, что подкласс `Throwable` не может быть обобщенным (§8.1.2).

§11.1.2. Причины исключений

Исключения генерируются по одной из трех причин.

- Выполнение инструкции `throw` (§14.18).
- Синхронное обнаружение виртуальной машиной Java условий анормального выполнения, а именно:
  - ✦ вычисление выражения нарушает обычную семантику языка программирования Java (§15.6), например, при делении на нуль;
  - ✦ ошибка при загрузке, связывании или инициализации части программы (§12.2, §12.3, §12.4); в этом случае генерируется экземпляр подкласса `LinkageError`;
  - ✦ внутренняя ошибка или ограничение ресурсов, не позволяющее виртуальной машине Java реализовать семантику языка программирования Java; в этом случае генерируется экземпляр подкласса `VirtualMachineError`.

  Эти исключения генерируются не в произвольной точке программы, а в точке, где они указываются как возможный результат вычисления выражения или выполнения инструкций.
- Произошло асинхронное исключение (§11.1.3).

§11.1.3. Асинхронные исключения

Большинство исключений генерируются синхронно, как результат действий потока, в котором они генерируются, и в точке программы, в которой возможным результатом указано такое исключение. *Асинхронное исключение*, напротив, является исключением, которое потенциально может произойти в любой точке выполняемой программы.

Асинхронные исключения могут быть сгенерированы только в результате следующих событий.

- Вызов (устаревшего) метода `stop` класса `Thread` или `ThreadGroup`.

  (Устаревшие) методы `stop` могут быть вызваны одним потоком для воздействия на другой поток или на все потоки в определенной группе потоков. Они являются асинхронными, поскольку могут осуществиться в любой точке выполнения другого потока или потоков.
- Внутренняя ошибка ограничения ресурсов виртуальной машины Java, которая не позволяет ей реализовать семантику языка программирования Java. В этом случае генерируемое асинхронное исключение представляет собой экземпляр подкласса `VirtualMachineError`.

Обратите внимание, что `StackOverflowError`, подкласс `VirtualMachineError`, может генерироваться как синхронно вызовом метода (§15.12.4.5), так и

асинхронно из-за выполнения метода, объявленного как `native`, или ограничения ресурсов виртуальной машины Java. Аналогично `OutOfMemoryError`, другой подкласс `VirtualMachineError`, может быть сгенерирован как синхронно в процессе создания объекта (§15.10.1, §10.6), инициализации класса (§12.4.2) и преобразования упаковки (§5.1.7), так и асинхронно.

Платформа Java SE допускает, что перед генерацией асинхронного исключения в программе будет выполнено определенное количество кода.

Асинхронные исключения редки, но если вы хотите создавать высококачественный машинный код, без правильного понимания их семантики не обойтись.

Упомянутая выше задержка разрешена для того, чтобы позволить оптимизированному коду обнаруживать и генерировать эти исключения в точках, в которых при подчинении семантике языка программирования Java их обработка более практична. Простая реализация может спрашивать о наличии асинхронных исключений в точке каждой инструкции передачи управления. Поскольку программа имеет конечный размер, общая задержка при обнаружении асинхронного исключения ограничена. Поскольку между передачами управления асинхронные исключения не будут генерироваться, генератор кода имеет возможность более гибко переупорядочивать вычисления для достижения более высокой производительности. Более подробную информацию можно почерпнуть из статьи *Polling Efficiently on Stock Hardware* Марка Фили (Marc Feeley), в *Proc. 1993 Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, pp. 179–187.

§11.2. Проверка исключений времени компиляции

Язык программирования Java требует, чтобы программа содержала обработчики для *проверяемых исключений*, которые могут быть сгенерированы в результате выполнения метода или конструктора (§8.4.6, §8.8.5). Эта проверка времени компиляции на наличие обработчиков исключений призвана уменьшить количество исключений, которые не могут быть корректно обработаны. Для каждого проверяемого исключения, которое может быть сгенерировано, в конструкции `throws` метода или конструктора должен упоминаться соответствующий класс исключения или один из суперклассов класса этого исключения (§11.2.3).

Классы проверяемых исключений (§11.1.1), перечисленные в конструкции `throws`, являются частью контракта между создателем и пользователем метода или конструктора. Конструкция `throws` перекрытого метода может не указывать, что этот метод может закончиться генерацией некоторого проверяемого исключения, которое не разрешено в перекрытом методе согласно его конструкции `throws` (§8.4.8.3). В случае интерфейсов одно перекрывающее объявление может перекрыть более чем одно объявление метода. В таком случае перекрывающее объявление должно иметь конструкцию `throws`, которая совместима со всеми перекрытыми объявлениями (§9.4.1).

Классы непроверяемых исключений (§11.1.1) освобождаются от проверки времени компиляции.

Среди классов непроверяемых исключений классы ошибок освобождаются от проверок потому, что они могут возникать во многих точках в программе, и восстановление после них трудно или невозможно. Программа, объявляющая такие исключения, будет беспорядочной и неэффективной. Очень сложные программы могут захотеть перехватить некоторые из таких ситуаций и восстановиться после них.

Среди классов непроверяемых исключений классы исключений времени выполнения освобождаются от проверок, поскольку, по мнению разработчиков языка программирования Java, объявление таких исключений не может существенно помочь в установлении корректности программ. Многие операции и конструкции языка программирования Java могут приводить к генерации исключений времени выполнения. Информации, доступной для компилятора Java, и уровня анализа, выполняемого компилятором, обычно недостаточно для того, чтобы установить, что такие исключения времени выполнения не могут быть сгенерированы, несмотря на то что это может быть очевидно программисту. Требование объявления таких классов исключений будет просто раздражать программиста.

Например, некоторый код может реализовывать циклическую структуру данных, в которой в соответствии с ее построением никогда не будет нулевой ссылки; программист может быть уверен, что исключение `NullPointerException` невозможно, но для компилятора Java доказательство этого факта — слишком сложная задача. Методика доказательства теорем, необходимая для установления таких глобальных свойств структур данных, выходит за рамки данной спецификации.

Мы говорим, что инструкция или выражение *может генерировать* класс проверяемых исключений *E*, если согласно правилам из §11.2.1 и §11.2.2 выполнение инструкции или выражения может привести к генерации исключения класса *E*.

Мы говорим, что конструкция `catch` *может перехватывать* свои перехватываемые классы исключений.

- *Класс перехватываемых исключений* конструкции с одним `catch` представляет собой объявленный тип ее параметра исключения (§14.20).
- *Классы перехватываемых исключений* конструкции с несколькими `catch` представляют собой элементы объединения типов, описывающего типы их параметров исключений (§14.20).

§11.2.1. Анализ исключений выражений

Выражение создания экземпляра класса (§15.9) может генерировать класс исключения *E* тогда и только тогда, когда:

- выражение является выражением создания экземпляра квалифицированного класса и квалифицирующее выражение может генерировать E;
- некоторое выражение в списке аргументов может генерировать E;
- E определено как класс исключения в конструкции throws вызываемого конструктора (§15.12.2.6);

- выражение создания экземпляра класса включает ClassBody, и некоторый блок инициализатора экземпляра или выражения инициализатора переменной экземпляра в *ClassBody* может генерировать *E*.

Выражение вызова метода (§15.12) может генерировать класс исключения *E* тогда и только тогда, когда:

- вызываемый метод имеет вид *Primary . [TypeArguments] Identifier* и выражение *Primary* может генерировать *E*;
- некоторое выражение в списке аргументов может генерировать *E*;
- *E* определено как класс исключения в конструкции `throws` вызываемого метода (§15.12.2.6).

Лямбда-выражение (§15.7) не может генерировать классы исключений.

Для всех прочих видов выражений выражение может генерировать класс исключения *E* тогда и только тогда, когда одно из его непосредственных подвыражений может генерировать *E*.

> Обратите внимание, что выражение ссылки на метод (§15.13) вида *Primary : : [Type Arguments] Identifier* может генерировать класс исключения, если генерировать класс исключения может подвыражение *Primary*. Напротив, лямбда-выражение не может генерировать ничего, и не имеет непосредственных подвыражений, над которыми выполняется анализ исключений. *Тело* лямбда-выражения содержит выражения и инструкции, которые могут генерировать классы исключений.

§11.2.2. Анализ исключений инструкций

Инструкция `throw` (§14.18), генерируемое выражение которой имеет статический тип *E* и не является `final`-параметром, может генерировать *E* или любой иной класс исключения, которое может быть сгенерировано выражением инструкции `throw`.

> Например, инструкция `throw new java.io.FileNotFoundException();` может генерировать только `java.io.FileNotFoundException`. Формально это не та ситуация, когда выражение "может генерировать" подкласс или суперкласс `java.io.FileNotFoundException`.

Инструкция `throw`, выражение которой является `final`-параметром конструкции `catch` *C*, может генерировать класс исключения *E* тогда и только тогда, когда:

- *E* представляет собой класс исключения, которое может генерировать блок `try` инструкции `try`, объявляющей *C*, и
- *E* совместимо по присваиванию с некоторым из перехватываемых классов исключений *C*, и
- *E* не совместимо по присваиванию с любым из перехватываемых классов исключений конструкций `catch`, объявленных слева от *C* в той же инструкции `try`.

Инструкция `try` (§14.20) может генерировать класс исключения *E* тогда и только тогда, когда:

- блок `try` может генерировать *E* или выражение, использованное для инициализации ресурса (в инструкции `try`-с-ресурсами), может генерировать *E*, или автоматический вызов метода `close()` ресурса (в инструкции `try`-с-ресурсами) может генерировать *E*, и *E* не совместимо по присваиванию ни с каким перехватываемым классом исключения любой конструкции `catch` инструкции `try`, и либо блок `finally` отсутствует, либо блок `finally` может завершиться нормально; или
- некоторый блок `catch` инструкции `try` может генерировать *E* и либо блок `finally` отсутствует, либо блок `finally` может завершиться нормально; или
- имеется блок `finally`, который может генерировать *E*.

Инструкция явного вызова конструктора (§8.8.7.1) может генерировать класс исключения *E* тогда и только тогда, когда:

- некоторое выражение списка параметров вызова конструктора может генерировать *E* или
- *E* определен как класс исключения конструкции `throws` вызываемого конструктора (§15.12.2.6).

Любая иная инструкция *S* может генерировать класс исключения *E* тогда и только тогда, когда выражение (или инструкция), непосредственно содержащееся в *S*, может генерировать *E*.

§11.2.3. Проверка исключений

Если тело метода или конструктора *может генерировать* некоторый класс исключения *E*, где *E* представляет собой класс проверяемого исключения и *E* не является подклассом некоторого класса, объявленного в конструкции `throws` метода или конструктора, генерируется ошибка времени компиляции.

Если тело лямбда-выражения *может генерировать* некоторый класс исключения *E*, в котором *E* представляет собой класс проверяемого исключения и *E* не является подклассом некоторого класса, объявленного в конструкции `throws` типа функции, являющегося целевым для лямбда-выражения, генерируется ошибка времени компиляции.

Если инициализатор переменной класса (§8.3.2) или статический инициализатор (§8.7) именованного класса или интерфейса *может генерировать* класс проверяемого исключения, генерируется ошибка времени компиляции.

Если инициализатор переменной экземпляра или инициализатор экземпляра (§8.3.2), или инициализатор экземпляра (§8.6) именованного класса *может генерировать* класс проверяемого исключения, то, если только этот класс исключения или один из его суперклассов не является явно объявленным в конструкции `throws` каждого конструктора класса и если класс не имеет как минимум одного явно объявленного конструктора, генерируется ошибка времени компиляции.

> Обратите внимание, что ошибки времени компиляции нет, если инициализатор переменной экземпляра или инициализатор экземпляра анонимного класса (§15.9.5) может генерировать класс исключения. В именованном классе ответственность за распространение информации о том, какие классы исключений могут быть сгенерированы инициализаторами, возлагается на программиста. Он может сделать это

с помощью объявления соответствующей конструкции `throws` в любом явном объявлении конструктора. Эта взаимосвязь между классами проверяемых исключений, сгенерированными инициализаторами классов, и классами проверяемых исключений, объявленными конструкторами классов, для объявления анонимного класса обеспечивается неявно, поскольку явное объявление конструктора невозможно, и компилятор Java всегда генерирует для объявления этого анонимного класса конструктор с подходящей конструкцией `throws`, основанной на тех классах проверяемых исключений, которые может генерировать инициализатор.

Если конструкция `catch` *может перехватывать* класс проверяемого исключения E_1, а соответствующий блок *не может генерировать* класс проверяемых исключений, являющийся подклассом или суперклассом E_1, то генерируется ошибка времени компиляции, если только E_1 не представляет собой `Exception` или суперкласс `Exception`.

Если конструкция `catch` *может перехватывать* класс проверенного исключения E_1, а предыдущая конструкция `catch` непосредственно охватывающей инструкции `try` *может перехватывать* E_1 или суперкласс E_1, генерируется ошибка времени компиляции.

Компилятору Java рекомендуется выдавать предупреждение, если конструкция `catch` *может генерировать* (§11.2) класс проверяемых исключений E_1, а блок `try`, соответствующий конструкции `catch`, *может перехватывать* класс проверяемых исключений E_2, где $E_2 <: E_1$, а предшествующая конструкция `catch` непосредственно охватывающей инструкции `try` *может перехватывать* класс проверяемых исключений E_3, где $E_2 <: E_3 <: E_1$.

Пример 11.2.3-1. Перехват проверяемых исключений

```
import java.io.*;

class StaticallyThrownExceptionsIncludeSubtypes {
    public static void main(String[] args) {
        try {
            throw new FileNotFoundException();
        } catch (IOException ioe) {
            // "catch IOException" перехватывает IOException
            // и любой из его подтипов.
        }

        try {
            throw new FileNotFoundException();
                // Инструкция "может генерировать"
                // FileNotFoundException.
                // Она не может генерировать подтип или
                // супертип FileNotFoundException.
        } catch (FileNotFoundException fnfe) {
            // ... Обработка исключения ...
        } catch (IOException ioe) {
            // Разрешено, но компилятору рекомендовано
            // выдавать предупреждение, как в Java SE 7,
            // поскольку подтипы IOException, которые блок
```

```
        // try может сгенерировать, уже перехвачены.
    }

    try {
        m();
            // Объявление метода m гласит
            // "throws IOException", так что
            // m "может генерировать" IOException. m не
            // "может генерировать" подтип или супертип
            // IOException, например Exception.
        } catch (FileNotFoundException fnfe) {
            // Разрешено, поскольку динамический тип
            // исключения может быть FileNotFoundException.
        } catch (IOException ioe) {
            // Разрешено, поскольку динамический тип
            // исключения может быть подтипом IOException.
        } catch (Throwable t) {
            // Всегда может генерировать Throwable.
        }
    }

    static void m() throws IOException {
        throw new FileNotFoundException();
    }
}
```

Согласно приведенным выше правилам каждая альтернатива в множественной конструкции `catch` (§14.20) должна быть способна перехватывать некоторый класс исключения, сгенерированный блоком `try` и не перехваченный предыдущими конструкциями `catch`. Например, вторая конструкция `catch` из показанных ниже приводит к ошибке времени компиляции, поскольку анализ исключений определяет, что `SubclassOfFoo` будет перехвачен первой конструкцией `catch`.

```
try { ... }
catch (Foo f) { ... }
catch (Bar | SubclassOfFoo e) { ... }
```

§11.3. Обработка исключений времени выполнения

При генерации исключения (§14.18) управление передается из кода, вызвавшего исключение, ближайшей могущей обработать это исключение динамически охватывающей конструкции `catch` (если таковая имеется) инструкции `try` (§14.20).

Инструкция (или выражение) является *динамически охваченной* конструкцией `catch`, если она находится в блоке `try` инструкции `try`, частью которой является данная конструкция `catch`, или если вызывающий код инструкции или выражения динамически охватывается этой конструкцией `catch`.

Вызывающий код инструкции (или выражения) зависит от того, где она находится.

- Если она находится в методе, то вызывающим кодом является выражение вызова метода (§15.12), которое было выполнено для того, чтобы метод был вызван.
- Если она находится в конструкторе или инициализаторе переменной экземпляра, то вызывающий код представляет собой выражение создания экземпляра класса (§15.9) или вызов метода `newInstance`, который используется для создания объекта.
- Если она находится в статическом инициализаторе или инициализаторе переменной, объявленной как `static`, то вызывающий код представляет собой выражение, использованное классом или интерфейсом для его инициализации (§12.4).

В состоянии ли некоторая конструкция `catch` *обработать* исключение, определяется путем сравнения класса сгенерированного объекта с перехватываемыми классами конструкции `catch`. Конструкция `catch` может обработать исключение, если один из ее перехватываемых классов является классом исключения или суперклассом класса исключения.

> Что то же самое, конструкция `catch` будет перехватывать все объекты исключений, которые представляют собой `instanceof` (§15.20.2) для одного из ее перехватываемых классов исключений.

Передача управления при генерации исключения приводит к преждевременному завершению выражений (§15.6) и инструкций (§14.1), пока не будет встречена конструкция `catch`, которая может обработать это исключение; после этого выполняется блок этой конструкции `catch`. Выполнение кода, вызвавшего исключение, не продолжается.

Все исключения (синхронные и асинхронные) являются *точными*: при передаче управления все результаты инструкций и выражений, выполненных до генерации исключения, должны иметь место. Никакое выполнение выражений, инструкций или их частей после точки генерации исключения не может иметь места.

> Если оптимизированный код умозрительно выполнил некоторые из выражений или инструкций, которые следуют за точкой генерации исключения, такой код должен быть готов скрыть это умозрительное исполнение от видимого пользователю состояния программы.

Если конструкция `catch`, способная обработать сгенерированное исключение, не найдена, то текущий поток выполнения (поток, в котором сгенерировано исключение) завершается. Перед завершением выполняются все конструкции `finally`, а неперехваченное исключение обрабатывается в соответствии со следующими правилами.

- Если текущий поток имеет установленный обработчик неперехваченных исключений, вызывается этот обработчик.
- В противном случае для группы `ThreadGroup`, которая представляет собой родителя текущего потока, вызывается метод `uncaughtException`. Если `ThreadGroup` и ее родительские `ThreadGroup` не перекрывают `uncaughtException`, то вызывается метод `uncaughtException` обработчика по умолчанию.

> В ситуации, когда желательно гарантировать выполнение одного блока после другого, даже если этот другой блок кода завершается преждевременно, можно использовать инструкцию `try` с конструкцией `finally` (§14.20.2).

Если блок `try` или `catch` в инструкции `try-finally` или `try-catch-finally` завершается преждевременно, то конструкция `finally` выполняется в процессе распространения исключения, даже если в конечном итоге соответствующая конструкция `catch` так и не будет найдена.

Если конструкция `finally` выполняется из-за преждевременного завершения блока `try` и сама конструкция `finally` завершается преждевременно, то причина преждевременного завершения блока `try` отбрасывается, и начиная с этой точки происходит распространение новой причины преждевременного завершения.

Точные правила преждевременного завершения и перехвата исключений детально описаны в спецификации каждой инструкции в §14, а для выражений — в §15 (в частности, в §15.6).

Пример 11.3-1. Генерация и перехват исключений

Приведенная далее программа объявляет класс исключения `TestException`. Метод `main` класса `Test` вызывает метод `thrower` четыре раза, что в трех из четырех случаев приводит к генерации исключения. Инструкция `try` в методе `main` перехватывает каждое сгенерированное исключение. Независимо от того, завершается ли вызов `thrower` нормально или преждевременно, выводится сообщение, описывающее случившееся.

```
class TestException extends Exception {
    TestException()          { super(); }
    TestException(String s) { super(s); }
}

class Test {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                thrower(arg);
                System.out.println("Test \"" + arg +
                                   "\" didn't throw an exception");
            } catch (Exception e) {
                System.out.println("Test \"" + arg +
                                   "\" threw a " + e.getClass() +
                                   "\n with message: " +
                                   e.getMessage());
            }
        }
    }
    static int thrower(String s) throws TestException {
        try {
                if (s.equals("divide")) {
                int i = 0;
                return i/i;
            }
            if (s.equals("null")) {
```

```
                s = null;
                return s.length();
            }
                if (s.equals("test")) {
                throw new TestException("Test message");
            }
            return 0;
        } finally {
            System.out.println("[thrower(\"" + s + "\") done]");
            }
    }
}
```

При выполнении программы с передачей ей аргументов

```
divide null not test
```

мы получим следующий вывод.

```
[thrower("divide") done]
Test "divide" threw a class java.lang.ArithmeticException
    with message: / by zero
[thrower("null") done]
Test "null" threw a class java.lang.NullPointerException
    with message: null
[thrower("not") done]
Test "not" didn't throw an exception
[thrower("test") done]
Test "test" threw a class TestException
    with message: Test message
```

Объявление метода `thrower` должно иметь конструкцию `throws`, поскольку он может генерировать экземпляры класса `TestException`, который представляет собой класс проверенного исключения (§11.1.1). Если конструкцию `throws` опустить, сгенерируется ошибка времени компиляции.

Обратите внимание, кто конструкция `finally` выполняется при каждом вызове `thrower` независимо от того, происходит ли генерация исключения, что видно из вывода "`[thrower(...) done]`" при каждом вызове метода.

ГЛАВА 12

Выполнение

В этой главе описано, что происходит при выполнении программы. Она сосредоточена на жизненном цикле виртуальной машины Java, а также классов, интерфейсов и объектов, образующих программу.

Виртуальная машина Java начинает работу с загрузки определенного класса и выполнения его метода `main`. В разделе §12.1 в качестве введения в концепции данной главы описываются этапы загрузки, связывания и инициализации, вовлеченные в выполнение `main`. В последующих разделах описаны подробности загрузки (§12.2), связывания (§12.3) и инициализации (§12.4).

Затем в главе рассматриваются процедуры создания новых экземпляров классов (§12.5) и финализация этих экземпляров (§12.6). Завершается глава описанием выгрузки классов (§12.7) и процедурой завершения программы (§12.8).

§12.1. Запуск виртуальной машины Java

Виртуальная машина Java начинает работу с выполнения метода `main` некоторого определенного класса, передавая ему единственный аргумент, который представляет собой массив строк. В примерах в этой книге этот первый класс обычно имеет название `Test`.

Точная семантика запуска виртуальной машины Java приведена в главе 5 книги *The Java Virtual Machine Specification, Java SE 8 Edition*. Здесь будет представлен обзор этого процесса с точки зрения языка программирования Java.

Порядок, в котором начальный класс определяется виртуальной машиной Java, выходит за рамки данной книги, но обычно в средах, в которых используются командные строки, полностью квалифицированное имя класса указывается в качестве аргумента командной строки, а последующие аргументы командной строки используются как строки, передаваемые в качестве аргумента методу `main`.

> Например, в реализации UNIX командная строка
>
> ```
> java Test reboot Bob Dot Enzo
> ```
>
> запустит виртуальную машину Java с выполнением метода `main` класса `Test` (класса в безымянном пакете), передавая ему массив из четырех строк — `"reboot"`, `"Bob"`, `"Dot"` и `"Enzo"`.

Теперь вкратце рассмотрим шаги, которые виртуальная машина Java может предпринять для выполнения `Test`, в качестве примера процессов загрузки, связывания и инициализации, которые будут описаны в последующих разделах.

§12.1.1. Загрузка класса `Test`

При первоначальной попытке выполнить метод `main` класса `Test` обнаруживается, что класс `Test` не загружен, т.е. что виртуальная машина Java в настоящее время не содержит бинарное представление данного класса. Тогда виртуальная машина Java использует загрузчик классов, пытаясь найти требуемое бинарное представление. Если этот процесс завершился неудачно, то генерируется ошибка. Этот процесс загрузки описан далее, в §12.2.

§12.1.2. Связывание `Test`: проверка, подготовка (необязательное) разрешение

После того как `Test` будет загружен и до того как `main` будет вызван, класс должен быть инициализирован. Класс `Test`, как и все типы классов или интерфейсов, перед инициализацией должен быть связан. Связывание включает проверку, подготовку и (необязательное) разрешение. Связывание описывается далее, в §12.3.

Проверка (верификация) убеждается, что загруженное представление `Test` корректно сформировано и с правильной таблицей символов. Проверяется также, подчиняется ли код, реализующий `Test`, семантическим требованиям языка программирования Java и виртуальной машины Java. Если в процессе проверки обнаруживается проблема, генерируется ошибка. Подробнее проверка описана в §12.3.1.

Подготовка включает выделение статической памяти и всех структур данных, которые внутренне используются виртуальной машиной Java, таких как таблицы методов. Подробнее подготовка описана в §12.3.2.

Разрешение представляет собой процесс проверки символьных ссылок из `Test` в другие классы и интерфейсы, выполняемый путем загрузки этих других упоминающихся классов и интерфейсов, и проверки корректности ссылок.

Шаг разрешения является необязательным в момент первоначального связывания. Реализация может разрешать символьные ссылки из класса или интерфейса с ранним связыванием вплоть до рекурсивного разрешения всех символьных ссылок из классов и интерфейсов, на которые имеются ссылки. (Такое разрешение может привести к ошибкам при дальнейших шагах загрузки и связывания.) Этот выбор реализации представляет одну крайность и подобен разновидности “статического” связывания, которое много лет выполнялось в простых реализациях языка программирования C. (В этих реализациях скомпилированная программа обычно представляет собой файл `"a.out"`, который содержит полностью связанную версию программы, включая полностью разрешенные ссылки на библиотечные подпрограммы, используемые данной программой. Копии этих библиотечных подпрограмм включены в файл `"a.out"`.)

Вместо этого реализация может выбрать разрешение символьных ссылок только при их использовании. Последовательное применение этой стратегии для всех символьных ссылок представляет собой “наиленивейшую” разновидность разрешения. В этом случае,

если `Test` имеет несколько символьных ссылок на другой класс, ссылки могут быть разрешены по одной по ходу их использования, но, возможно, не все, поскольку некоторые ссылки могут так и не быть использованы в процессе выполнения программы.

Единственным требованием ко времени выполнения разрешения является то, что любые ошибки, обнаруженные во время разрешения, должны быть сгенерированы в той точке программы, в которой некоторое предпринятое программой действие может прямо или косвенно потребовать связь с классом или интерфейсом, связанным с этой ошибкой. При использовании упомянутого выше "статического" разрешения ошибки загрузки и связывания могут быть обнаружены до выполнения программы, если они касаются класса или интерфейса, упомянутого в классе `Test` или в любом рекурсивно упоминаемом классе или интерфейсе. В системе, реализующей "ленивое" (отложенное) разрешение, эти ошибки обнаруживаются только тогда, когда неверная символьная ссылка используется программой.

Процесс разрешения описан в §12.3.3.

§12.1.3. Выполнение инициализаторов

Продолжая выполнение примера, виртуальная машина Java пытается выполнить метод `main` класса `Test`. Это разрешено, только если класс был инициализирован (§12.4.1).

Инициализация состоит из выполнения всех инициализаторов переменной класса и статических инициализаторов класса `Test` в текстуальном порядке. Но до того, как `Test` может быть инициализирован, должен быть инициализирован его непосредственный суперкласс, как и непосредственный суперкласс его непосредственного суперкласса, и (рекурсивно) т.д. В простейшем случае `Test` имеет в качестве неявного непосредственного суперкласса `Object`; если класс `Object` еще не был инициализирован, то он должен быть инициализирован раньше класса `Test`. Класс `Object` не имеет суперкласса, так что на нем рекурсия завершается.

Если класс `Test` имеет другой класс `Super` в качестве суперкласса, то `Super` должен быть инициализирован до `Test`. Это требует загрузки, проверки и подготовки `Super`, если это еще не было сделано, и в зависимости от реализации может включать разрешение символьных ссылок из `Super` и, рекурсивно, т.д.

Таким образом, инициализация может привести к ошибкам загрузки, связывания и инициализации, включая эти ошибки для других типов.

Процесс инициализации описан в §12.4.

§12.1.4. Вызов `Test.main`

По завершении инициализации класса `Test` (в процессе которой могут выполняться другие косвенные загрузки, связывания и инициализации) вызывается метод `main` класса `Test`.

Метод `main` должен быть объявлен как `public`, `static` и `void`. Он должен иметь формальный параметр (§8.4.1), объявленный тип которого — массив `String`. Следовательно, приемлемым является любое из объявлений

```
public static void main(String[] args)
public static void main(String... args)
```

§12.2. Загрузка классов и интерфейсов

Загрузка представляет собой процесс поиска бинарного представления типа класса или интерфейса с определенным именем (возможно, вычисляемым "на лету", но обычно получаемым путем выборки бинарного представления, ранее созданного компилятором Java из исходного текста) и построение на его основе объекта `Class` для представления класса или интерфейса.

Точная семантика загрузки описана в главе 5 книги *The Java Virtual Machine Specification, Java SE 8 Edition*. Здесь будет представлен обзор этого процесса с точки зрения языка программирования Java.

Бинарный формат класса или интерфейса обычно представляет собой формат файла `class`, описанный в упомянутой выше книге *The Java Virtual Machine Specification, Java SE 8 Edition*, но возможно применение и других форматов, обеспечивающих соответствие требованиям, указанным в §13.1. Метод `defineClass` класса `ClassLoader` может использоваться для построения объектов `Class` из бинарных представлений в формате файлов `class`.

Правильный загрузчик классов поддерживает следующие свойства.

- При получении одного и того же имени класса загрузчик должен возвращать один и тот же объект класса.
- Если загрузчик классов L_1 делегирует загрузку класса C другому загрузчику L_2, то для любого типа T, который является непосредственным суперклассом или суперинтерфейсом C, или типом поля в C, или типом формального параметра метода или конструктора C, или возвращаемым типом метода в C, L_1 и L_2 должны возвращать один и тот же объект `Class`.

Некорректный загрузчик классов может нарушать указанные правила. Однако он не может подорвать безопасность системы типов, поскольку виртуальная машина Java следит за этим.

> Дополнительную информацию по данным вопросам можно найти в книге *The Java Virtual Machine Specification, Java SE 8 Edition* и статье *Dynamic Class Loading in the Java Virtual Machine* Шен Лианга (Sheng Liang) и Джилада Брача (Gilad Bracha), в *Proceedings of OOPSLA '98*, опубликованной как *ACM SIGPLAN Notices*, Volume 33, Number 10, October 1998, pages 36–44. Базовые принципы дизайна языка программирования Java заключаются в том, что система типов времени выполнения не может быть разрушена ни кодом, написанным на Java, ни даже реализацией таких важных системных классов, как `ClassLoader` и `SecurityManager`.

§12.2.1. Процесс загрузки

Процесс загрузки реализован классом `ClassLoader` и его подклассами.

Различные подклассы `ClassLoader` могут реализовывать различные стратегии загрузки. В частности, загрузчик классов может кешировать бинарные представления классов и интерфейсов, выполнять их предварительную выборку на основе ожидаемого применения или загружать группу связанных между собой классов. Эти действия могут

не быть совершенно прозрачными для работающего приложения, если, например, вновь скомпилированная версия класса не найдена из-за того, что старая версия кеширована загрузчиком. Однако загрузчик классов отвечает за отражение ошибок загрузки только в тех точках программы, где они могут возникнуть без предварительной выборки или групповой загрузки.

Если ошибка возникает в процессе загрузки класса, то в любой точке программы, которая (прямо или косвенно) использует этот тип, генерируется исключение, представляющее собой экземпляр одного из следующих подклассов класса `LinkageError`.

- `ClassCircularityError`: класс или интерфейс не может быть загружен в силу того, что он является собственным суперклассом или суперинтерфейсом (§8.1.4, §9.1.3, §13.4.4).
- `ClassFormatError`: бинарные данные требуемого скомпилированного класса или интерфейса имеют неверный формат.
- `NoClassDefFoundError`: соответствующим загрузчиком классов не может быть найдено определение требуемого класса или интерфейса.

Поскольку загрузка включает выделение памяти для новых структур данных, возможна генерация исключения `OutOfMemoryError`.

§12.3. Связывание классов и интерфейсов

Связывание представляет собой процесс получения бинарного представления типа класса или интерфейса и объединение его в состояние виртуальной машины Java времени выполнения, так что его можно выполнить. Тип класса или интерфейса всегда загружается до связывания.

Связывание включает три различных действия: проверку, подготовку и разрешение символьных ссылок.

Точная семантика связывания описана в главе 5 книги *The Java Virtual Machine Specification, Java SE 8 Edition.* Здесь будет представлен обзор этого процесса с точки зрения языка программирования Java.

Данная спецификация допускает гибкость реализации при связывании (и в силу рекурсии при загрузке) при условиях, что соблюдена семантика языка программирования Java, что класс или интерфейс полностью проверен и приготовлен до инициализации и что обнаруженные в процессе связывания ошибки генерируются в точке программы, где программой предпринято некоторое действие, которое может потребовать связывания с классом или интерфейсом, имеющим отношение к ошибке.

Например, реализация может разрешать каждую символьную ссылку класса или интерфейса индивидуально, только при ее применении (“ленивое”, или позднее, разрешение), или разрешать их все одновременно при проверке класса (статическое разрешение). Это означает, что процесс разрешения в некоторых реализациях может продолжаться после инициализации класса или интерфейса.

Поскольку связывание включает выделение памяти для новых структур данных, возможна генерация исключения `OutOfMemoryError`.

§12.3.1. Проверка бинарного представления

Проверка (верификация) убеждается, что бинарное представление класса или интерфейса является структурно корректным. Например, проверяется, что каждая инструкция имеет корректный код операции; что каждая инструкция ветвления ведет к началу некоторой другой инструкции, а не к ее средине; что каждый метод снабжен структурно корректной сигнатурой; и что каждая инструкция подчиняется дисциплине типов языка виртуальной машины Java.

Если в процессе проверки обнаруживается ошибка, то в точке программы, вызвавшей проверку класса, генерируется исключение, представляющее собой экземпляр следующего подкласса класса `LinkageError`.

- `VerifyError`: бинарное определение класса или интерфейса не проходит требуемый набор проверок, которые должны убедиться в том, что оно подчиняется семантике языка виртуальной машины Java и что оно не в состоянии нарушить целостность виртуальной машины Java. (Смотрите некоторые примеры в §13.4.2, §13.4.4, §13.4.9 и §13.4.17.)

§12.3.2. Подготовка типа класса или интерфейса

Подготовка включает создание `static`-полей (переменных и констант класса) класса или интерфейса и инициализацию таких полей значениями по умолчанию (§4.12.5). Это не требует выполнения никакого исходного кода; явные инициализаторы для статических полей выполняются как часть инициализации (§12.4), а не подготовки.

> Реализации виртуальной машины Java могут предвычислять дополнительные структуры данных во время подготовки для того, чтобы сделать более поздние операции над классом или интерфейсом более эффективными. Одной особенно полезной структурой данных является "таблица методов" или иная структура данных, которая позволяет любому методу быть вызванным для экземпляров классов без поиска суперклассов во время вызова.

§12.3.3. Разрешение символьных ссылок

Бинарное представление класса или интерфейса ссылается на другие классы и интерфейсы и их поля, методы и конструкторы символьно, с использованием бинарных имен (§13.1) этих других классов и интерфейсов (§13.1). В случае полей и методов эти символьные ссылки включают имя типа класса или интерфейса, для которого это поле или метод является членом, а также имя самого поля или метода вместе с соответствующей информацией о типах.

До того как символьные ссылки могут быть использованы, они должны пройти процесс разрешения, в котором для символьной ссылки проверяется ее корректность и, как правило, выполняется замена на прямую ссылку, которая может быть более эффективной при многократном использовании.

Если в процессе разрешения происходит ошибка, генерируется исключение. Чаще всего это экземпляр одного из перечисленных далее подклассов класса `Incompatible`

`ClassChangeError`, но это может быть и экземпляр некоторого иного подкласса `IncompatibleClassChangeError` или даже экземпляр самого класса `IncompatibleClassChangeError`. Эта ошибка может быть сгенерирована в любой точке программы, которая прямо или косвенно использует символьную ссылку на тип.

- `IllegalAccessError`: встретилась символьная ссылка, которая определяет использование или присваивание поля, или вызов метода, или создание экземпляра класса, к которой код, содержащий эту ссылку, не имеет права доступа, поскольку поле или метод объявлен как `private`, `protected` или с доступом пакета (не `public`) или поскольку класс не объявлен как `public`.

  Это может случиться, например, если изначально объявленное как `public` поле изменяется на `private` после компиляции другого класса, ссылающегося на это поле (§13.4.7).

- `InstantiationError`: встретилась символьная ссылка, которая используется в выражении создания экземпляра класса, но экземпляр не может быть создан, поскольку выясняется, что ссылка ссылается на интерфейс или абстрактный класс.

  Это может случиться, например, если класс, который исходно не является абстрактным, изменен таким образом, что становится объявленным как `abstract` после того, как другой обращающийся к нему класс уже скомпилирован (§13.4.1).

- `NoSuchFieldError`: встретилась символьная ссылка, которая обращается к определенному полю определенного класса или интерфейса, но этот класс или интерфейс не содержат поле с таким именем.

  Это может случиться, например, если объявление поля было удалено из класса после того, как другой обращающийся к этому полю класс уже был скомпилирован (§13.4.8).

- `NoSuchMethodError`: встретилась символьная ссылка, которая обращается к определенному методу определенного класса или интерфейса, но этот класс или интерфейс не содержат метод с такой сигнатурой.

  Это может случиться, например, если объявление метода было удалено из класса после того, как другой класс, обращающийся к этому методу, уже был скомпилирован (§13.4.12).

Кроме того, если класс объявляет `native`-метод, реализация которого не найдена, может быть сгенерировано исключение, представляющее собой экземпляр класса `UnsatisfiedLinkError`, являющегося подклассом `LinkageError`. Это исключение генерируется при использовании метода или ранее в зависимости от того, какая стратегия разрешения используется реализацией виртуальной машины Java (§12.3).

§12.4. Инициализация классов и интерфейсов

Инициализация класса состоит из выполнения его статических инициализаторов и инициализаторов полей, объявленных в классе как `static` (переменных класса).

Инициализация интерфейса состоит из выполнения инициализаторов полей (констант), объявленных в интерфейсе.

Перед классом должен быть инициализирован его непосредственный суперкласс, но интерфейсы, реализованные классом, могут быть не инициализированы. Аналогично суперинтерфейсы интерфейса не инициализируются до инициализации интерфейса.

§12.4.1. Когда осуществляется инициализация

Тип класса или интерфейса `T` будет инициализироваться непосредственно перед первым осуществлением одного из следующих событий.

- `T` представляет собой класс и создается экземпляр `T`.
- `T` представляет собой класс и вызывается `static`-метод, объявленный классом `T`.
- Выполняется присваивание `static` полю, объявленному типом `T`.
- Используется `static`-поле, объявленное типом `T`, и это поле не является константной переменной (§4.12.4).
- `T` является классом верхнего уровня (§7.6) и выполняется инструкция `assert` (§14.10), лексически вложенная в `T` (§8.1.3).

Ссылка на `static`-поле (§8.3.1.1) приводит к инициализации только того класса или интерфейса, который его объявляет, несмотря на то что обращение может осуществляться через имя подкласса, подынтерфейса или класса, реализующего интерфейс.

Вызов некоторых рефлективных методов в классе `Class` и в пакете `java.lang.reflect` также приводит к инициализации класса или интерфейса.

При любых других условиях класс или интерфейс не будет инициализироваться.

Цель состоит в том, что тип класса или интерфейса имеет набор инициализаторов, которые переводят его в согласованное состояние, и что это состояние является первым состоянием, которое наблюдается другими классами. Статические инициализаторы и инициализаторы переменных класса выполняются в порядке появления в исходном тексте и не могут обращаться к переменным класса, объявленным в классе, объявления которого текстуально располагаются после этого использования, даже несмотря на то, что эти переменные класса находятся в области видимости (§8.3.3). Это ограничение предназначено для обнаружения во время компиляции большинства циклических или некорректных по иным причинам инициализаций.

Тот факт, что код инициализации является неограниченным, позволяет построить примеры, в которых значение переменной класса можно наблюдать, когда оно все еще равно начальному значению по умолчанию, до выполнения его выражения инициализации, но такие примеры на практике редки. (Такие примеры могут быть построены и для инициализации переменных экземпляров (§12.5).) В этих инициализаторах доступна вся мощь языка программирования Java; программистам следует проявлять в ее отношении определенную осторожность. Эта мощь налагает дополнительное бремя на генератор кода, но это бремя возникнет в любом случае из-за параллельного характера языка программирования Java (§12.4.2).

Пример 12.4.1-1. Суперклассы инициализируются до подклассов

```
class Super {
    static { System.out.print("Super "); }
}
class One {
    static { System.out.print("One "); }
}
class Two extends Super {
    static { System.out.print("Two "); }
}
class Test {
    public static void main(String[] args) {
        One o = null;
        Two t = new Two();
        System.out.println((Object)o == (Object)t);
    }
}
```

Вывод этой программы имеет следующий вид.

```
Super Two false
```

Класс `One` не инициализируется, так как не используется, а значит, с ним никто не связывается. Класс `Two` инициализируется только после того, как инициализируется его суперкласс `Super`.

Пример 12.4.1-2. Инициализируется только класс, объявляющий статическое поле

```
class Super {
    static int taxi = 1729;
}
class Sub extends Super {
    static { System.out.print("Sub "); }
}
class Test {
    public static void main(String[] args) {
        System.out.println(Sub.taxi);
    }
}
```

Эта программа выводит только

```
1729
```

поскольку класс `Sub` никогда не инициализируется; ссылка на `Sub.taxi` является ссылкой на поле, в действительности объявленное в классе `Super`, и не вызывает инициализации класса `Sub`.

Пример 12.4.1-3. Инициализация интерфейса не инициализирует суперинтерфейс

```
interface I {
    int i = 1, ii = Test.out("ii", 2);
```

```
}
interface J extends I {
    int j = Test.out("j", 3), jj = Test.out("jj", 4);
}
interface K extends J {
    int k = Test.out("k", 5);
}
class Test {
    public static void main(String[] args) {
        System.out.println(J.i);
        System.out.println(K.j);
    }
    static int out(String s, int i) {
        System.out.println(s + "=" + i);
        return i;
    }
}
```

Вывод этой программы имеет вид

```
1
j=3
jj=4
3
```

Ссылка на `J.i` является обращением к полю, которое представляет собой константную переменную (§4.12.4); следовательно, она не приводит к инициализации `I` (§13.4.9).

Ссылка на `K.j` является обращением к полю, которое в действительности объявлено в интерфейсе `J` и не является константной переменной; это приводит к инициализации полей интерфейса `J`, но не полей его суперинтерфейса `I` или полей интерфейса `K`.

Несмотря на тот факт, что имя `K` использовано для обращения к полю `j` интерфейса `J`, интерфейс `K` не инициализируется.

§12.4.2. Детальная процедура инициализации

Поскольку язык программирования Java многопоточный, инициализация класса или интерфейса требует аккуратной синхронизации, так как некоторый другой поток может попытаться инициализировать тот же класс или интерфейс в тот же момент времени. Имеется также возможность, что инициализация класса или интерфейса может быть затребована рекурсивно как часть инициализации этого класса или интерфейса; например, инициализатор переменной в классе *A* может вызвать метод несвязанного класса *B*, который, в свою очередь, может вызвать метод класса *A*. Реализация виртуальной машины Java отвечает за меры по обеспечению синхронизации и рекурсивной инициализации с помощью следующей процедуры.

В процедуре предполагается, что объект `Class` уже проверен и подготовлен и что объект `Class` содержит состояние, которое указывает одно из четырех состояний.

- Этот объект `Class` проверен и подготовлен, но не инициализирован.
- Этот объект `Class` инициализируется некоторым конкретным потоком *T*.
- Этот объект `Class` полностью инициализирован и готов к употреблению.
- Этот объект `Class` находится в состоянии ошибки, вероятно, из-за неудачной попытки инициализации.

Для каждого класса или интерфейса *C* имеется уникальная блокировка инициализации *LC*. Отображение *C* на *LC* оставляется на усмотрение реализации виртуальной машины Java. Тогда процедура инициализации *C* имеет следующий вид.

1. Синхронизация блокировки инициализации *LC* для *C*. Это включает ожидание захвата *LC* текущим потоком.
2. Если объект `Class` для *C* указывает, что инициализация *C* в процессе выполнения некоторым иным потоком, блокировка *LC* освобождается, а текущий поток блокируется до тех пор, пока не будет проинформирован о полном завершении выполняющейся инициализации, после чего этот шаг повторяется.
3. Если объект `Class` для *C* указывает, что инициализация *C* находится в процессе выполнения текущим потоком, то это означает рекурсивный запрос инициализации. Освобождаем *LC* и завершаем процедуру нормально.
4. Если объект `Class` для *C* указывает, что инициализация *C* уже выполнена, то дальнейшие действия не выполняются. Освобождаем *LC* и завершаем процедуру нормально.
5. Если объект `Class` для *C* находится в ошибочном состоянии, то инициализация невозможна. Освобождаем *LC* и генерируем исключение `NoClassDefFoundError`.
6. В противном случае записываем тот факт, что инициализация объекта `Class` для *C* находится в процессе выполнения текущим потоком, и освобождаем *LC*.

   Затем инициализируем статические поля *C*, которые являются константными переменными (§4.12.4, §8.3.2, §9.3.1).
7. Далее, если *C* представляет собой класс, а не интерфейс, и его суперкласс *SC* еще не был инициализирован, то рекурсивно выполняем всю процедуру для *SC*. Если необходимо, сначала проверяем и подготавливаем *SC*. Если инициализация *SC* завершается преждевременно из-за генерации исключения, то захватываем *LC*, помечаем объект `Class` для *C* как ошибочный, уведомляем все ждущие потоки, освобождаем *LC* и завершаем процедуру преждевременно, генерируя то же исключение, которое получили в результате инициализации *SC*.
8. Затем определяем, разрешены ли утверждения (§14.10) для *C* путем запроса к определяющему загрузчику класса.
9. Затем выполняются либо инициализаторы переменных класса и статические инициализаторы класса, либо инициализаторы полей интерфейса в порядке их появления в исходном тексте, как если бы они были единым блоком.
10. Если выполнение инициализаторов завершается нормально, выполняется захват *LC*, объект `Class` для *C* помечается как полностью инициализированный,

уведомляются все ждущие потоки, освобождается *LC*, и процедура завершается нормально.

11. В противном случае инициализаторы должны завершиться преждевременно путем генерации некоторого исключения *E*. Если класс исключения *E* не является `Error` или одним из его подклассов, то создается новый экземпляр класса `ExceptionInInitializerError`, с *E* в качестве аргумента, и этот объект используется вместо *E* на следующем шаге. Но если новый экземпляр `ExceptionInInitializerError` не может быть создан из-за ошибки `OutOfMemoryError`, то вместо *E* на следующем шаге используется объект `OutOfMemoryError`.

12. Захватываем *LC*, помечаем объект `Class` для *C* как ошибочный, уведомляем все ждущие потоки, освобождаем *LC* и завершаем данную процедуру преждевременно генерацией определенного на предыдущем шаге исключения *E* или заменяющего его.

Реализация может оптимизировать данную процедуру, отменяя получение блокировки на шаге 1 (и освобождение на шаге 4/5), если она может определить, что инициализация класса уже завершена, при условии, что, в терминах модели памяти, все то, что происходило бы до упорядочения, которое было бы, если бы блокировка была захвачена, все еще происходит при оптимизации.

Генераторы кода должны сохранять точки возможной инициализации класса или интерфейса, вставляя вызов только что описанной процедуры инициализации. Если эта процедура инициализации завершается нормально и объект `Class` полностью инициализирован и готов к использованию, то вызов процедуры инициализации более не является необходимым и может быть удален из кода, например, вставкой заглушки или повторной генерацией кода.

Анализ времени компиляции может в некоторых случаях быть способен удалить из генерируемого кода множество проверок того, что тип инициализирован, если может быть определен порядок инициализации для группы или связанных типов. Однако такой анализ должен полностью учитывать параллельные вычисления и тот факт, что код инициализации ничем не ограничен.

§12.5. Создание новых экземпляров классов

Новый экземпляр класса явно создается, когда вычисление выражения создания экземпляра класса (§15.9) приводит к инстанцированию класса.

Новый экземпляр класса может быть создан неявно в следующих ситуациях.

- Загрузка класса или интерфейса, который содержит литерал `String` (§3.10.5), может приводить к созданию нового объекта `String` для представления этого литерала. (Этого может и не случиться, если та же строка `String` была интернирована ранее (§3.10.5).)

- Выполнение операции, вызывающей преобразование упаковки (§5.1.7). Преобразование упаковки может создавать новый объект класса-оболочки, связанного с одним из примитивных типов.
- Выполнение оператора конкатенации строк + (§15.18.1), который является частью константного выражения (§15.28), иногда создает новый объект `String` для представления результата. Операторы конкатенации строк могут также создавать временные объекты-оболочки для значений примитивных типов.
- Вычисление выражения ссылки на метод (§15.13.3) или лямбда-выражения (§15.27.4) может потребовать, чтобы был создан новый экземпляр класса, который реализует тип функционального интерфейса.

Каждая из этих ситуаций определяет конкретный конструктор (§8.8), вызываемый с определенными аргументами (возможно, без таковых) как часть процесса создания экземпляра класса.

Всякий раз, когда создается новый экземпляр класса, выделяется память для него со всеми переменными экземпляра, объявленными в этом типе класса, и для всех переменных экземпляров, объявленных в каждом суперклассе данного типа класса, включая все переменные экземпляров, которые могут быть скрыты (§8.3).

Если имеется недостаточно памяти для объекта, создание экземпляра класса завершается преждевременно генерацией исключения `OutOfMemoryError`. В противном случае все переменные экземпляра нового объекта, включая объявленные в суперклассах, инициализируются их значениями по умолчанию (§4.12.5).

Непосредственно перед обращением ко вновь созданному объекту, возвращенному в качестве результата, указанный конструктор инициализирует новый объект с использованием следующей процедуры.

1. Присваивает аргументы конструктора вновь созданным переменным параметров для данного вызова конструктора.
2. Если этот конструктор начинается с явного вызова конструктора (§8.8.7.1) некоторого другого конструктора того же самого класса (с применением ключевого слова `this`), то вычисление аргументов и процесс рекурсивного вызова этого конструктора проходят через те же пять шагов. Если происходит преждевременное завершение вызова конструктора с генерацией исключения, то данная процедура завершается преждевременно по той же причине; в противном случае процедура переходит к шагу 5.
3. Данный конструктор не начинается с явного вызова другого конструктора того же класса (с применением ключевого слова `this`). Если это конструктор класса, отличающегося от класса `Object`, то он начинается с явного или неявного вызова конструктора суперкласса (с применением ключевого слова `super`). Вычисление аргументов и процесс рекурсивного вызова этого конструктора проходят через те же пять шагов. Если происходит преждевременное завершение вызова этого конструктора с генерацией исключения, то данная процедура завершается преждевременно по той же причине. В противном случае процедура переходит к шагу 4.

4. Выполняются инициализаторы экземпляров и инициализаторы переменных экземпляра для данного класса и присваивание значений инициализаторов переменных экземпляра соответствующим переменным экземпляра в порядке слева направо, в котором они текстуально появляются в исходном коде класса. Если выполнение любого из этих инициализаторов приводит к генерации исключения, более никакие инициализаторы не выполняются, а процедура завершается преждевременно с тем же самым исключением. В противном случае процедура переходит к шагу 5.
5. Выполняется оставшаяся часть этого конструктора. Если выполнение завершается преждевременно, данная процедура завершается преждевременно по той же самой причине. В противном случае процедура завершается нормально.

В отличие от C++, язык программирования Java не изменяет правила вызова методов в процессе создания нового экземпляра класса. Если вызываемые методы перекрыты в подклассах инициализируемого объекта, то используются эти перекрытые методы, даже до того, как новый объект будет полностью инициализирован.

Пример 12.5-1. Создание экземпляра

```
class Point {
    int x, y;
    Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        System.out.println(cp.color);
    }
}
```

Здесь создается новый экземпляр `ColoredPoint`. Сначала выделяется память для нового `ColoredPoint`, для хранения полей `x`, `y` и `color`. Затем все эти поля инициализируются их значениями по умолчанию (в данном случае `0` для каждого поля). Затем сначала вызывается конструктор `ColoredPoint` без аргументов. Поскольку `ColoredPoint` не объявляет конструкторов, неявно объявляется конструктор по умолчанию следующего вида.

```
ColoredPoint() { super(); }
```

Затем этот конструктор вызывает конструктор `Point` без аргументов. Конструктор `Point` не начинает с вызова конструктора, так что компилятор Java обеспечивает неявный вызов конструктора суперкласса без аргументов, как если бы исходный текст имел вид

```
Point() { super(); x = 1; y = 1; }
```

Таким образом, вызывается конструктор `Object` без аргументов.

Класс `Object` не имеет суперкласса, так что рекурсия на этом завершается. Затем вызываются инициализаторы экземпляров и переменных экземпляра `Object`. После этого выполняется тело конструктора `Object` без аргументов. Такой конструктор в `Object` не объявлен, так что компилятор Java создает конструктор по умолчанию, который в данном частном случае имеет вид

```
Object() { }
```

Этот конструктор не выполняет никаких действий.

Затем приходит очередь инициализаторов переменных экземпляра класса `Point`. Так как объявления `x` и `y` не предоставляют никаких выражений инициализации, этот шаг в данном примере не выполняет никаких действий. Затем выполняется тело конструктора `Point`, которое устанавливает значения `x` и `y` равными `1`.

После этого выполняются инициализаторы переменных экземпляра класса `ColoredPoint`. На этом шаге выполняется присваивание значения `0xFF00FF` переменной экземпляра `color`. Наконец выполняется остальная часть тела конструктора `ColoredPoint` (часть после вызова `super`); так как здесь нет никаких инструкций, никаких дополнительных действий не требуется, и инициализация завершается.

Пример 12.5-2. Динамическая диспетчеризация в процессе создания экземпляра

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
    int three = (int)Math.PI; // T.e. 3
    void printThree() { System.out.println(three); }
    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

Вывод этой программы имеет следующий вид.

```
0
3
```

Это демонстрирует, что вызов `printThree` в конструкторе класса `Super` не затрагивает определение `printThree` в классе `Super`, а вызывает перекрытое определение `printThree` в классе `Test`. Таким образом, данный метод работает до выполнения инициализаторов полей в классе `Test`, что объясняет первое выведенное значение `0`, значение по умолчанию, которым инициализируется поле `three` класса `Test`. Более поздний вызов `printThree` в методе `main` вызывает то же самое определение `printThree`, но после выполнения инициализатора переменной экземпляра `three`, так что в этот раз выводится значение `3`.

§12.6. Финализация экземпляров классов

Класс `Object` имеет метод `finalize`, объявленный как `protected`; этот метод может быть перекрыт другими классами. Конкретное определение метода `finalize`, который может вызываться для объекта, называется *финализатором* (finalizer) этого объекта. Перед тем как объект будет уничтожен сборщиком мусора, виртуальная машина Java вызовет финализатор этого объекта.

Финализаторы предоставляют шанс освободить ресурсы, которые не могут быть освобождены автоматически менеджером памяти. В таких ситуациях простое освобождение памяти, использованной объектом, не гарантирует освобождение захваченных объектом ресурсов.

Язык программирования Java не определяет, когда будет вызван финализатор; все, что мы знаем, — что это произойдет до того, как будет повторно использована память, выделенная объекту.

Язык программирования Java не определяет, какой поток вызовет финализатор некоторого данного объекта.

> Важно отметить, что могут быть активны многие потоки финализаторов (иногда это необходимо в многопроцессорных системах с большой разделяемой памятью) и что, если большая связанная структура данных становится мусором, все методы `finalize` всех объектов этой структуры данных могут быть вызваны одновременно — каждый вызов финализатора в своем потоке выполнения.

Язык программирования Java не накладывает какое-либо упорядочение на вызовы методов `finalize`. Финализаторы могут вызываться в любом порядке и даже параллельно.

> В качестве примера можно привести ситуацию, когда недоступной (или доступной для финализаторов) становится циклически связанная группа нефинализированных объектов. Тогда все объекты могут стать финализируемыми вместе. В конечном итоге финализаторы этих объектов могут быть вызваны в любом порядке или даже параллельно разными потоками. Если позже автоматический менеджер памяти обнаружит, что эти объекты недоступны, выделенная им память может быть освобождена.
>
> Достаточно просто реализовать класс, у которого для множества объектов будет вызываться набор финализатороподобных методов в определенном порядке, когда все эти объекты станут недоступными. Определение такого класса оставляем читателю в качестве упражнения.

Гарантируется, что поток, который вызывает финализатор, не будет использовать какие-либо видимые пользователю блокировки при вызове финализатора.

Если в процессе финализации сгенерировано неперехваченное исключение, это исключение игнорируется и финализация этого объекта завершается.

Завершение конструктора объекта связано отношением предшествования (§17.4.5) с выполнением его метода `finalize` (в формальном смысле отношения предшествования).

Метод `finalize`, объявленный в классе `Object`, не выполняет никаких действий. Тот факт, что класс `Object` объявляет метод `finalize`, означает, что метод `finalize` любого класса всегда может вызвать метод `finalize` своего суперкласса. Это всегда

следует делать, если только программист не имеет намерения отменить действия финализатора суперкласса. (В отличие от конструкторов финализаторы не вызывают автоматически финализатор суперкласса; такой вызов должен быть закодирован явно.)

> Для большей эффективности реализация может отслеживать классы, которые не перекрывают метод `finalize` класса `Object` или перекрывают его тривиальным образом.
>
> Например:
>
> ```
> protected void finalize() throws Throwable {
>     super.finalize();
> }
> ```
>
> Мы поощряем рассмотрение реализациями таких объектов, как имеющих неперекрытый финализатор, и их более эффективное финализирование, как описано в §12.6.1.

Финализатор может быть вызван явно, так же, как и любой другой метод.

Пакет `java.lang.ref` описывает слабые ссылки, которые взаимодействуют со сборкой мусора и финализацией. Как и в случае любого API, которое имеет специальные средства взаимодействия с языком программирования Java, разработчики должны быть осведомлены о требованиях, накладываемых API `java.lang.ref`. В данной книге слабые ссылки вообще не рассматриваются; заинтересованные читатели могут найти детальную информацию по данному вопросу в документации, посвященной этому API.

§12.6.1. Реализация финализации

Каждый объект можно охарактеризовать двумя атрибутами: он может быть *достижим* (reachable), *достижим для финализации* (finalizer-reachable) или *недостижим* (unreachable), а также может быть *нефинализуем* (unfinalized), *финализуем* (finalizable) или *финализирован* (finalized).

Достижимый объект представляет собой любой объект, который может быть доступен для любого потенциального продолжения вычислений из любого активного потока.

Достижимый для финализации объект может быть достижим из некоторого финализуемого объекта посредством некоторой цепочки ссылок, но не из любого активного потока.

Недостижимый объект не может быть достигнут никакими средствами.

Финализатор *нефинализуемого* объекта никогда не вызывается автоматически.

У *финализированного* объекта финализатор уже был автоматически вызван.

У *финализируемого* объекта финализатор не был вызван автоматически, но виртуальная машина Java в конечном итоге может это сделать.

Объект `o` не является финализуемым до тех пор, пока его конструктор не вызовет конструктор `Object` для `o` и этот вызов не завершится успешно (т.е. без генерации исключения). Любая предфинализационная запись поля объекта должна быть видима финализации этого объекта. Кроме того, никакие предфинализационные чтения полей этого объекта не в состоянии видеть записи, происшедшие после инициации финализации объекта.

Может быть разработано оптимизирующее преобразование программы, которое снижает количество достижимых объектов, и оно становится меньше количества объектов,

наивно рассматриваемых как достижимые. Например, компилятор Java или генератор кода может установить более неиспользуемую переменную или параметр равным `null`, так что память такого объекта потенциально становится доступной для освобождения.

Другой пример — значения полей объекта хранятся в регистрах. Программа при этом может обратиться к регистрам вместо объекта и никогда не обращаться к самому объекту. Это приводит к тому, что объект становится мусором. Заметим, что такой вид оптимизации позволен, только когда ссылки находятся в стеке, а не хранятся в куче.

Рассмотрим, например, шаблон *Finalizer Guardian*.

```
class Foo {
    private final Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
            /* Финализация внешнего объекта Foo */
        }
    }
}
```

Этот шаблон обеспечивает вызов `super.finalize`, если подкласс перекрывает `finalize` и явно не вызывает `super.finalize`.

Если описанная оптимизация разрешена для ссылок, которые хранятся в куче, то компилятор Java может обнаружить, что поле `finalizerGuardian` никогда не будет прочтено, обнулить его и вызвать финализатор заранее. Это идет вразрез с намерением программиста, который, вероятно, хотел бы вызова финализатора `Foo` тогда, когда экземпляр `Foo` станет недоступным. Следовательно, такой тип преобразований некорректен: объект внутреннего класса должен быть доступен, если доступен объект внешнего класса.

Преобразование данного вида может привести к вызовам метода `finalize` ранее, чем этого можно было бы ожидать в противном случае. Чтобы позволить пользователю это предотвратить, мы обеспечиваем возможность сохранить объект с помощью синхронизации. *Если финализатор объекта может привести к синхронизации на этом объекте, то этот объект должен быть активен и рассматриваться как достижимый при наличии его блокировки.*

Обратите внимание, что это не препятствует удалению синхронизации: синхронизация сохраняет объект активным, только если финализатор синхронизуется на нем. Поскольку финализатор выполняется в другом потоке, во многих случаях синхронизация не может быть удалена.

§12.6.2. Взаимодействие с моделью памяти

Модель памяти (§17.4) должна иметь возможность решать, когда могут выполняться действия, происходящие в финализаторе. В этом разделе описывается взаимодействие финализации с моделью памяти.

Каждое исполнение имеет несколько *точек принятия решения о достижимости* (reachability decision points), помеченных как *di*. Каждое действие либо *происходит до di*, либо *происходит после di*. Отличные от явно указанных упорядочения, описанные в этом разделе, не связаны с другими упорядочениями модели памяти.

Если *r* представляет собой чтение, которое видит запись *w,* и *r* происходит до *di*, то *w* должно происходить до *di*.

Если *x* и *y* представляют собой действия синхронизации по отношению к одной и той же переменной или монитору, так, что *so(x,y)* (§17.4.4) и *y* происходит до *di*, то *x* должно происходить до *di*.

В каждой точке принятия решения о достижимости некоторый набор объектов помечается как недостижимый, а некоторое их подмножество помечается как финализуемое. Эти точки принятия решения о достижимости являются также точками, в которых ссылки проверяются, становятся в очередь и очищаются в соответствии с правилами, описанными в документации API пакета `java.lang.ref`.

Единственными объектами, рассматриваемыми как определенно достижимые в точке *di*, являются те, достижимость которых может быть показана с помощью следующих правил.

- Объект *B* определенно достижим в *di* из `static`-поля, если существует запись *w1* статического поля *v* класса *C*, такая, что значение, записываемое *w1*, является ссылкой на *B*, класс *C* загружается достижимым загрузчиком класса и не существует записи *w2* поля *v*, такой, что *hb(w2,w1)* ложно, и как *w1*, так и *w2* происходят до *di*.
- Объект *B* определенно достижим из объекта *A* в *di*, если существует запись *w1* элемента *v* в *A*, такая, что значение, записываемое *w1*, представляет собой ссылку на *B* и не существует записи *w2* элемента *v*, такой, что *hb(w2,w1)* ложно, и как *w1*, так и *w2* происходят до *di*.
- Если объект *C* определенно достижим из объекта *B* и объект *B* определенно достижим из объекта *A*, то объект *C* определенно достижим из объекта *A*.

Если объект *X* помечен как недостижимый в *di*, то:

- *X* должен не быть определенно достижим в *di* из поля `static`; и
- все *активные использования X* в потоке *t*, которые происходят после *di*, должны происходить в вызове финализатора *X* или как результат выполнения потоком *t* чтения, происходящего после *di*, ссылки на *X*; и
- все чтения, происходящие после *di*, которые видят ссылку на *X*, должны видеть записи в элементы объектов, которые были недостижимы в *di*, или видеть записи, происходящие после *di*.

Действие *a* является активным использованием *X* тогда и только тогда, когда выполняется как минимум одно из перечисленных условий.

- *a* читает или записывает элемент *X*.
- *a* блокирует или разблокирует *X* и имеется действие по блокировке *X*, которое в смысле отношения предшествования выполняется после вызова финализатора *X*.
- *a* записывает ссылку на *X*.
- *a* является активным использованием объекта *Y*, а *X* определенно достижим из *Y*.

Если объект *X* помечен в *di* как финализируемый, то:

- *X* должен быть помечен в *di* как недостижимый; и
- *di* должен быть единственным местом, где *X* помечен как финализируемый; и

- действия, которые в смысле отношения предшествования выполняются после вызова финализатора, должны происходить после *di*.

§12.7. Выгрузка классов и интерфейсов

Реализация языка программирования Java может *выгружать* классы.

Класс или интерфейс может быть выгружен тогда и только тогда, когда определенный им загрузчик класса может быть утилизирован сборщиком мусора, как обсуждалось в §12.6.

Классы и интерфейсы, загруженные встроенным загрузчиком, не могут быть выгружены.

Выгрузка класса является оптимизацией, помогающей уменьшить количество требующейся для работы программы памяти. Очевидно, что семантика программы не должна зависеть от того, будет ли выполняться такая оптимизация, как выгрузка классов (и каким именно образом она будет выполняться); в противном случае встает вопрос о переносимости такой программы. Следовательно, принятие решения о выгрузке должно быть прозрачным для программы.

Однако, если класс или интерфейс *C* был выгружен, в то время как его определяющий (определенный им) загрузчик был потенциально достижим, то *C* может быть перезагружен. Никогда нельзя гарантировать, что это не произойдет. Даже если на класс не ссылаются никакие другие загруженные классы, на него может ссылаться некоторый класс или интерфейс *D*, который еще не загружен. Когда *D* загружается определяющим загрузчиком *C*, его выполнение может привести к перезагрузке *C*.

Перезагрузка может не быть прозрачной, если, к примеру, класс содержит статические переменные (состояние которых будет утеряно), статические инициализаторы (которые могут иметь побочные эффекты) или методы, объявленные как `native` (которые могут хранить статическое состояние). Кроме того, хеш-значение объекта `Class` зависит от его идентичности. Поэтому в общем случае невозможно перезагрузить класс или интерфейс полностью прозрачным образом.

Из того, что мы никогда не можем гарантировать, что выгрузка класса или интерфейса, загрузчик которого потенциально доступен, не вызовет перегрузки, а при том, что перезагрузка не является прозрачной, выгрузка должна быть таковой, следует, что нельзя выгружать класс или интерфейс в то время, когда его загрузчик потенциально достижим. Аналогичные рассуждения могут использоваться для вывода о том, что классы и интерфейсы, загруженные встроенным загрузчиком, никогда не могут быть выгружены.

Можно также доказать, что выгрузка класса *C* безопасна, если его определяющий загрузчик классов может быть утилизирован. Если определяющий загрузчик классов может быть утилизирован, то на него не может быть никакой активной ссылки (включая ссылки, которые не являются активными, но могут стать таковыми из-за финализаторов). Это, в свою очередь, может быть истинно только тогда, когда нет ни одной активной ссылки ни на один из классов, определенных этим загрузчиком, включая *C*, как из их экземпляров, так и из кода.

Выгрузка классов представляет собой оптимизацию, играющую роль только в приложениях, загружающих большое число классов, и в приложениях, в которых через некоторое время большинство из этих классов перестает использоваться. В первую очередь, примером такого приложения является веб-браузер, но есть и другие. Особенностью таких приложений является то, что они управляют классами путем явного использования загрузчиков классов. В результате для них хорошо работает описанная выше стратегия.

Строго говоря, обсуждение этого вопроса в данной книге не столь важно, поскольку выгрузка класса — это не более чем оптимизация. Однако это очень тонкий вопрос, и поэтому он упоминается и разъясняется так подробно.

§12.8. Выход из программы

Программа завершает всю свою деятельность и осуществляет *выход*, когда происходит одно из двух:

- все потоки, не являющиеся потоками демонов, завершаются;
- некоторый поток вызывает метод `exit` класса `Runtime` или класса `System`, и операция `exit` не запрещена менеджером безопасности.

ГЛАВА 13

Бинарная совместимость

Средства разработки для языка программирования Java при доступности исходных текстов должны поддерживать автоматическую перекомпиляцию в случае необходимости. Конкретные реализации могут также хранить в базе данных управления версиями исходные тексты и бинарные представления типов и реализовывать `ClassLoader`, который использует механизмы целостности базы данных для предотвращения ошибок связывания, предоставляя клиентам бинарно совместимые версии типов.

Разработчики широко распространяемых пакетов и классов сталкиваются с другим набором проблем. В Интернете, который является нашим любимым примером широко распределенной системы, часто нецелесообразно или невозможно автоматически перекомпилировать существующие двоичные файлы, которые прямо или косвенно зависят от типа, который должен быть изменен. Вместо этого спецификация Java определяет набор изменений, которые разработчикам разрешается внести в пакет или в тип класса или интерфейса при сохранении (не нарушая) совместимости с уже существующими бинарными файлами.

В рамках *Release-to-Release Binary Compatibility in SOM* (Форман (Forman), Коннер (Conner), Данфорт (Danforth) и Рэпер (Raper), *Proceedings of OOPSLA '95*) бинарные файлы языка программирования Java бинарно совместимы при всех соответствующих преобразованиях, идентифицируемых авторами (с некоторыми оговорками в отношении добавления переменных экземпляров). При использовании этой схемы список некоторых важных бинарно совместимых изменений, поддерживаемых языком программирования Java, имеет следующий вид.

- Повторная реализация существующих методов, конструкторов и инициализаторов для повышения производительности.
- Изменение методов или конструкторов для возврата значений для входных данных, при которых ранее генерировались исключения, которых обычно не должно быть, выполнялся бесконечный цикл или осуществлялась взаимоблокировка.
- Добавление новых полей, методов или конструкторов к существующему классу или интерфейсу.
- Удаление полей, методов или конструкторов класса, объявленных как `private`.
- Удаление при обновлении целого пакета полей, методов или конструкторов классов или интерфейсов пакета с доступом пакета.
- Переупорядочение полей, методов и конструкторов в существующих объявлениях типов.

- Перемещение метода вверх по иерархии классов.
- Переупорядочение списка непосредственных суперинтерфейсов класса или интерфейса.
- Вставка новых типов классов или интерфейсов в иерархию типов.

В этой главе определены минимальные стандарты для бинарной совместимости, гарантируемой всеми реализациями. Язык программирования Java гарантирует совместимость, когда смешиваются бинарные представления классов и интерфейсов, о которых не известно, происходят ли они из совместимых исходных текстов, но исходные тексты которых были изменены совместимыми способами, описанными выше. Обратите внимание, что мы обсуждаем совместимость между разными выпусками одного приложения. Обсуждение совместимости между выпусками платформы Java SE выходит за рамки настоящей главы.

Мы поощряем наличие в системах разработки возможностей оповещения разработчиков о влиянии вносимых ими изменений на уже существующие бинарные представления, которые не могут быть перекомпилированы.

Эта глава сначала определяет некоторые свойства, которыми должен обладать любой бинарный формат языка программирования Java (§13.1). Далее определяется бинарная совместимость, поясняется, что это такое и чем она не является (§13.2). Наконец перечисляется большой набор возможных изменений в пакетах (§13.3), классах (§13.4) и интерфейсах (§13.5) с указанием, какие из этих изменений будут гарантированно сохранять бинарную совместимости, а какие — нет.

Местами в данной главе для указания концепций из *The Java Virtual Machine Specification, Java SE 8 Edition* используются ссылки вида (JVMS §x.y).

§13.1. Форма бинарного представления

Программы должны быть скомпилированы либо в формат файла `class`, определенный в *The Java Virtual Machine Specification, Java SE 8 Edition*, либо в представление, которое может быть отображено в этот формат загрузчиком классов, написанным на языке программирования Java.

Получающийся в результате `class`-файл должен обладать определенными свойствами. Ряд этих свойств призван обеспечить поддержку преобразований исходных текстов, сохраняющих бинарную совместимость. Этими свойствами являются следующие.

1. Класс или интерфейс должен именоваться с помощью его *бинарного имени*, которое должно удовлетворять следующим ограничениям.
   - Бинарное имя типа верхнего уровня (§7.6) является его каноническим именем (§6.7).
   - Бинарное имя типа-члена (§8.5, §9.5) состоит из бинарного имени непосредственно охватывающего типа, за которым следует `$`, а за ним — простое имя члена.
   - Бинарное имя локального класса (§14.3) состоит из бинарного имени непосредственно охватывающего типа, за которым следует `$`, а за ним — непустая последовательность цифр, за которой следует простое имя локального класса.

- Бинарное имя анонимного класса (§15.9.5) состоит из бинарного имени непосредственно охватывающего типа, за которым следует `$`, а за ним — непустая последовательность цифр.
- Бинарное имя переменной типа, объявленной обобщенным классом или интерфейсом (§8.1.2, §9.1.2), состоит из бинарного имени непосредственно охватывающего типа, за которым следует `$`, а за ним — простое имя переменной типа.
- Бинарное имя переменной типа, объявленной обобщенным методом (§8.4.4), состоит из бинарного имени типа, объявляющего метод, за которым следует `$`, а за ним — дескриптор метода (JVMS §4.3.3); после дескриптора идут `$` и простое имя переменной типа.
- Бинарное имя переменной типа, объявленной обобщенным конструктором (§8.8.4), состоит из бинарного имени типа, объявляющего конструктор, за которым следует `$`, а за ним — дескриптор конструктора (JVMS §4.3.3); после дескриптора идут `$` и простое имя переменной типа.

2. Ссылка на другой тип класса или интерфейса должна быть символьной, с использованием бинарного имени типа.

3. Ссылки на поле, которое представляет собой константную переменную (§4.12.4), должны разрешаться во время компиляции в значение *V*, обозначаемое инициализатором константной переменной.

Если такое поле является статическим, то в коде в бинарном файле не должно быть ссылок на это поле, в том числе в классе или интерфейсе, который объявляет данное поле. Такое поле всегда должно появляться инициализированным (§12.4.2); исходное значение по умолчанию для этого поля (если оно отлично от *V*) не должно быть видимым.

Если это поле не статическое, то в коде в бинарном файле не должно быть ссылок на такое поле, за исключением содержащего это поле класса. (Это должен быть именно класс, а не интерфейс, поскольку интерфейс имеет только статические поля.) Класс должен содержать код для установки значения поля равным *V* в процессе создания экземпляра (§12.5).

4. Для заданного корректного выражения, описывающего доступ к полю в классе *C* и ссылающегося на поле с именем `f`, не являющееся константной переменной, и объявленное в (возможно, ином) классе или интерфейсе *D*, мы определяем *квалифицирующий тип ссылки на поле* следующим образом.

- Если на выражение ссылаются с помощью простого имени, и если `f` представляет собой член текущего класса или интерфейса *C*, то пусть *T* является *C*. В противном случае пусть *T* представляет собой наиболее глубоко вложенный лексически охватывающий класс, членом которого является `f`. В любом случае *T* является квалифицирующим типом ссылки.
- Если выражение имеет вид *TypeName* . `f`, где *TypeName* обозначает класс или интерфейс, то класс или интерфейс, обозначаемый *TypeName*, представляет собой квалифицирующий тип ссылки.
- Если выражение представляет собой форму *ExpressionName* . `f` или *Primary* . `f`, то:

— если тип времени компиляции *ExpressionName* или *Primary* представляет собой тип пересечения (§4.9) V_1 & ... & V_n, то квалифицирующим типом ссылки является V_1;

— в противном случае квалифицирующий тип ссылки представляет собой тип времени компиляции *ExpressionName* или *Primary*.

- Если выражение имеет вид `super.`*f*, то квалифицирующим типом ссылки является суперкласс *C*.
- Если выражение имеет вид *TypeName*`.super.`*f*, квалифицирующим типом ссылки является суперкласс класса, обозначаемого с помощью *TypeName*.

Ссылка на *f* должна быть скомпилирована в символьную ссылку на затирание (§4.6) квалифицирующего типа ссылки плюс простое имя поля, *f*. Ссылка должна также включать символьную ссылку на затирание объявленного типа поля, так что в процессе проверки можно убедиться, что тип соответствует ожидаемому.

5. Для заданного выражения вызова метода или выражения ссылки на метод в классе или интерфейсе *C*, ссылающегося на метод с именем *m*, объявленный (или неявно объявленный (§9.2)) в (возможно, ином) классе или интерфейсе *D*, мы определяем *квалифицирующий тип вызова метода* следующим образом.
   - Если *D* представляет собой `Object`, то квалифицирующим типом выражения является `Object`.
   - В противном случае выполняется следующее.
     — Если обращение к методу выполняется по простому имени, то если *m* является членом текущего класса или интерфейса *C*, пусть *T* представляет собой *C*. В противном случае пусть *T* представляет собой наиболее глубоко вложенный лексически охватывающий класс, членом которого является *m*. В любом случае *T* является квалифицирующим типом вызова метода.
     — Если выражение имеет вид *TypeName*`.`*m* или *ReferenceType*`::`*m*, то тип, обозначаемый *TypeName* или *ReferenceType*, является квалифицирующим типом вызова метода.
     — Если выражение имеет вид *ExpressionName*`.`*m*, *Primary*`.`*m*, *ExpressionName*`::`*m* или *Primary*`::`*m*, то:
       » если тип времени компиляции *ExpressionName* или *Primary* представляет собой тип пересечения V_1 & ... & V_n (§4.9), то квалифицирующим типом вызова метода является V_1;
       » в противном случае квалифицирующим типом вызова метода является тип времени компиляции *ExpressionName* или *Primary*.
     — Если выражение имеет вид `super.`*m* или `super::`*m*, то квалифицирующим типом вызова метода является суперкласс *C*.
     — Если выражение имеет вид *TypeName*`.super.`*m* или *TypeName*`.super::`*m* и если *TypeName* обозначает класс *X*, то квалифицирующим типом вызова метода является суперкласс *X*; если *TypeName* обозначает интерфейс *X*, то квалифицирующим типом вызова метода является *X*.

Ссылка на метод должна быть разрешена во время компиляции в символьную ссылку на затирание (§4.6) квалифицирующего типа вызова, плюс затирание сигнатуры (§8.4.2) метода. Сигнатура метода должна включать все элементы приведенного далее списка, как определено в §15.12.3.

- Простое имя метода.
- Количество параметров метода.
- Символьная ссылка на тип каждого параметра.

Ссылка на метод должна также включать либо символьную ссылку на затирание возвращаемого типа описываемого метода, либо указание, что метод объявлен как `void` и не возвращает значение.

6. Для заданного выражения создания экземпляра (§15.9), инструкции явного вызова конструктора (§8.8.7.1) или выражения ссылки на метод вида *ClassType*`::new` (§15.13) в классе или интерфейсе *C*, ссылающегося на конструктор *m*, объявленный в (возможно, ином) классе или интерфейсе *D*, мы определяем квалифицирующий тип вызова конструктора следующим образом.
   - Если выражение имеет вид `new` *D*`(...)`, *ExpressionName*`.new` *D*`(...)`, *Primary*`.new` *D*`(...)` или *D*`::new`, то квалифицирующим типом вызова является *D*.
   - Если выражение имеет вид `new` *D*`(...){...}`, *ExpressionName*`.new` *D*`(...){...}` или *Primary*`.new` *D*`(...){...}`, то квалифицирующим типом выражения является тип выражения времени компиляции.
   - Если выражение имеет вид `super(...)`, *ExpressionName*`.super(...)` или *Primary*`.super(...)`, то квалифицирующим типом выражения является непосредственный суперкласс *C*.
   - Если выражение имеет вид `this(...)`, то квалифицирующим типом выражения является *C*.

   Ссылка на конструктор должна быть разрешена во время компиляции в символьную ссылку на затирание (§4.6) квалифицирующего типа вызова плюс сигнатура конструктора (§8.8.2). Сигнатура конструктора должна включать
   - количество параметров конструктора и
   - символьную ссылку на тип каждого формального параметра.

Бинарное представление класса или интерфейса должно также содержать все пункты приведенного далее списка.

1. Если это класс, но не `Object`, то символьную ссылку на затирание (§4.6) непосредственного суперкласса данного класса.
2. Символьную ссылку на затирание каждого непосредственного суперинтерфейса, если таковой имеется.
3. Спецификацию каждого поля, объявленного в классе или интерфейсе, заданную как простое имя поля и символьная ссылка на затирание типа поля.
4. Если это класс, то затертую сигнатуру каждого конструктора, как описано выше.

5. Для каждого метода, объявленного в классе или интерфейсе (исключая в случае интерфейса его неявно объявленные методы (§9.2)), его затертую сигнатуру и возвращаемый тип, как описано выше.
6. Код, необходимый для реализации класса или интерфейса:
   - для интерфейса — код инициализаторов полей и реализации каждого метода по умолчанию;
   - для класса — код инициализаторов полей, статических инициализаторов и инициализаторов экземпляров, а также реализации каждого метода и конструктора.
7. Каждый тип должен содержать достаточную информацию для восстановления его канонического имени (§6.7).
8. Каждый тип-член должен содержать достаточную информацию для восстановления его исходного модификатора доступа.
9. Каждый вложенный класс и вложенный интерфейс должен содержать символьные ссылки на непосредственно охватывающий его класс (§8.1.3).
10. Каждый класс должен содержать символьные ссылки на все свои типы-члены (§8.5) и на все локальные и анонимные классы, которые встречаются в его методах, конструкторах, статических инициализаторах, инициализаторах экземпляров и инициализаторах полей.

    Каждый интерфейс должен содержать символьные ссылки на все свои типы-члены (§9.5) и на все локальные и анонимные классы, которые встречаются в его методах по умолчанию и инициализаторах полей.
11. Конструкция, генерируемая компилятором Java, должна быть помечена как *синтетическая*, если она не соответствует конструкции, явно или неявно объявленной в исходном тексте, если только генерируемая конструкция не является методом инициализации класса (JVMS §2.9).
12. Конструкция, генерируемая компилятором Java, должна быть помечена как обязательная, если она соответствует формальному параметру, неявно объявленному в исходном тексте (§8.8.1, §8.8.9, §8.9.3, §15.9.5.1).

    В исходном тексте неявно объявляются следующие формальные параметры:
    - первый формальный параметр конструктора внутреннего класса-члена, не объявленного как `private` (§8.8.1, §8.8.9);
    - первый формальный параметр анонимного конструктора анонимного класса, суперкласс которого является внутренним или локальным (не в статическом контексте) (§15.9.5.1);
    - формальный параметр `name` метода `valueOf`, который неявно объявлен в типе перечисления (§8.9.3).

    В случае ссылок в исходном тексте неявно объявляются перечисленные ниже конструкции, но они не маркируются как обязательные, так как в `class`-файле так могут помечаться только формальные параметры (JVMS §4.7.22).
    - Конструкторы по умолчанию типов классов и перечислений (§8.8.9, §8.9.2).
    - Анонимные конструкторы (§15.9.5.1).

- Методы `values` и `valueOf` типов перечислений (§8.9.3).
- Некоторые `public`-поля типов перечислений (§8.9.3).
- Некоторые `public`-методы интерфейсов (§9.2).
- Содержащие аннотации (§9.7.5).

В следующих разделах рассматриваются изменения, которые могут быть внесены в объявления типов классов и интерфейсов без нарушения совместимости с уже существующими бинарными файлами. При соответствии приведенным выше требованиям транслляции виртуальная машина Java и формат `class`-файла поддерживают эти изменения. Любой иной допустимый бинарный формат, например сжатое или зашифрованное представление, отображаемое обратно в `class`-файлы загрузчиком классов согласно указанным выше требованиям, также обязательно будет поддерживать эти изменения.

§13.2. Что такое бинарная совместимость и чем она не является

Изменение типа *бинарно совместимо с* (или, что то же самое, не *нарушает бинарную совместимость с*) существующими бинарными файлами, если существующие бинарные файлы, которые ранее связывались без ошибок, продолжают связываться без ошибок.

Бинарные файлы компилируются с учетом доступных членов и конструкторов других классов и интерфейсов. Для сохранения бинарной совместимости класс или интерфейс должен рассматривать свои доступные члены и конструкторы, их существование и поведение как *контракт* с пользователями.

Язык программирования Java разработан так, чтобы предотвращать добавления к контрактам и случайные коллизии имен, нарушающие бинарную совместимость. В частности, добавление большего количества методов, перегружающих некоторый определенный метод, не нарушают бинарную совместимость с существующими бинарными файлами. Сигнатура метода, используемая существующим бинарным файлом для поиска метода, выбирается алгоритмом разрешения перегрузки во время компиляции (§15.12.2).

> Если бы язык программирования Java был разработан так, чтобы выбор конкретного выполняемого метода проводился во время выполнения программы, то такая неоднозначность могла бы быть обнаружена во время выполнения. Это правило привело бы к тому, что добавление дополнительного перегруженного метода могло бы нарушить совместимость с неизвестным количеством уже существующих бинарных файлов. Смотрите более полное обсуждение этого вопроса в §13.4.23.

Бинарная совместимость отличается от совместимости исходных текстов. В частности, пример в §13.4.6 показывает, что множество совместимых бинарных файлов может быть получено из исходных текстов, которые не будут компилироваться все вместе. Это типичный пример: добавление нового объявления, изменяющего смысл имени в неизменной части исходного кода, в то время как существующие бинарные файлы этой неизменной части исходного кода сохраняют полностью квалифицированное предыдущее значение имени. Получение совместимого множества исходного кода требует предоставления квалифицированных имен или выражений доступа к полям, соответствующих предыдущему смыслу.

§13.3. Эволюция пакетов

Новый класс или интерфейс верхнего уровня может быть добавлен в пакет без нарушения бинарной совместимости с существующими бинарными файлами при условии, что новый тип не использует имя, данное ранее не связанному с ним типу.

Если новый тип повторно использует имя, данное ранее не связанному с ним типу, в результате может создаться конфликтная ситуация, поскольку бинарные файлы для обоих типов не могут быть загружены одним и тем же загрузчиком классов.

Изменения в типах классов или интерфейсов верхнего уровня, не являющихся `public` и не являющихся соответственно суперклассами или суперинтерфейсами `public`-типа, влияют только на те типы в пакете, в которых они объявлены. Такие типы могут быть удалены или изменены иным образом, даже при описанных здесь несовместимостях, при условии, что бинарные файлы пакета, на которые влияет это изменение, будут обновлены все вместе.

§13.4. Эволюция классов

В этом разделе описывается влияние изменений в объявлении класса и его членов и конструкторов на существующие бинарные файлы.

§13.4.1. Абстрактные классы

Если класс, который не был объявлен как `abstract`, изменяется и становится абстрактным, то существующие бинарные файлы, которые пытаются создавать новые экземпляры этого класса, будут генерировать либо исключение `InstantiationError` во время связывания, либо (при использовании рефлективного метода) исключение `InstantiationException` времени выполнения; следовательно, такое изменение не рекомендуется для широко распространенных классов.

Изменение класса, объявленного как `abstract`, заключающееся в том, что он перестает быть абстрактным, не нарушает совместимость с существующими бинарными файлами.

§13.4.2. Классы `final`

Если класс, который не был объявлен как `final`, изменяется и становится таковым, то при загрузке бинарного представления существующего подкласса данного класса генерируется исключение `VerifyError`, поскольку классы, объявленные как `final`, не могут иметь подклассов; такое изменение не рекомендуется для широко распространенных классов.

Изменение класса, объявленного как `final`, заключающееся в том, что он перестает быть `final`, не нарушает совместимость с существующими бинарными файлами.

§13.4.3. Классы `public`

Изменение класса, который не был объявлен как `public`, заключающееся в том, что он становится `public`, не нарушает совместимость с существующими бинарными файлами.

Если класс, который был объявлен как `public`, изменяется и перестает быть таковым, то при загрузке существующего бинарного представления, которому требуется доступ к данному типу класса, но которое его больше не имеет, генерируется исключение `IllegalAccessError`; такое изменение не рекомендуется для широко распространенных классов.

§13.4.4. Суперклассы и суперинтерфейсы

Если класс является собственным суперклассом, во время загрузки генерируется исключение `ClassCircularityError`. Изменения в иерархии классов, которые могут привести к такой цикличности, когда вновь скомпилированные бинарные представления загружаются вместе с существующими, не рекомендуются для широко распространенных классов.

Изменение непосредственного суперкласса или множества непосредственных суперинтерфейсов типа класса не будет нарушать совместимость с имеющимися бинарными файлами при условии, что в целом множество соответственно суперклассов или суперинтерфейсов этого типа класса не теряет члены.

Если изменение в непосредственном суперклассе или множестве непосредственных суперинтерфейсов приводит к тому, что некоторый класс или интерфейс перестает быть соответственно суперклассом или суперинтерфейсом, то в результате загрузки существующих бинарных файлов с бинарным представлением модифицированного класса могут происходить ошибки связывания. Такие изменения не рекомендуются для широко распространенных классов.

Пример 13.4.4-1. Изменение суперкласса

Предположим, что имеется тестовая программа

```
class Hyper { char h = 'h'; }
class Super extends Hyper { char s = 's'; }
class Test extends Super {
    public static void printH(Hyper h) {
        System.out.println(h.h);
    }
    public static void main(String[] args) {
        printH(new Super());
    }
}
```

которая компилируется и при выполнении выводит

```
h
```

Предположим, что затем компилируется новая версия класса `Super`:

```
class Super { char s = 's'; }
```

Эта версия класса `Super` не является подклассом `Hyper`. Если мы после этого запустим на выполнение существующие бинарные файлы `Hyper` и `Test` с новой версией `Super`, то во время связывания будет сгенерировано исключение `VerifyError`. Дело в том, что результат `new Super()` не может быть передан в

качестве аргумента вместо формального параметра типа `Hyper`, поскольку `Super` не является подклассом `Hyper`.

Поучительно рассмотреть, что бы произошло, не будь обязательного этапа проверки: программа при этом выполнялась бы и давала вывод

```
s
```

Это демонстрирует, что без проверки система типов Java может быть разрушена путем связывания несовместимых бинарных файлов, несмотря на то что каждый из них получен в результате корректной работы компилятора Java.

Урок заключается в том, что реализация, в которой отсутствует или некорректно выполняется проверка, не обеспечивает безопасность типов, а значит, не является корректной реализацией.

Требование, чтобы альтернативы в конструкции мульти-`catch` (§14.20) не были подклассами или суперклассами друг друга, является всего лишь ограничением источника. Предположим, что приведенный далее клиентский код корректен.

```
try {
    throwAorB();
} catch(ExceptionA | ExceptionB e) {
    ...
}
```

Здесь `ExceptionA` и `ExceptionB` не связаны отношением "подкласс/суперкласс" при компиляции клиентского кода и бинарно совместимы с клиентом, у которого `ExceptionA` и `ExceptionB` связаны таким соотношением при выполнении.

Это аналог иной ситуации, когда преобразование класса, бинарно совместимое с клиентом, может оказаться не совместимым в смысле исходного текста с тем же самым клиентом.

§13.4.5. Параметры типа класса

Добавление или удаление параметра типа класса само по себе не влияет на бинарную совместимость.

Если такой параметр типа используется в типе поля или метода, это может иметь обычные последствия изменения упомянутого типа.

Переименование параметра типа класса не влияет на существующие бинарные файлы.

Изменение первой границы параметра типа класса может изменить затирание (§4.6) любого члена, который использует этот параметр типа в собственном типе, и это может влиять на бинарную совместимость. Изменение такой границы аналогично изменению первой границы параметра типа метода или конструктора (§13.4.13).

Изменение любой иной границы не влияет на бинарную совместимость.

§13.4.6. Объявления тела и члена класса

Никакая несовместимость с существующими бинарными файлами не вызывается ни добавлением члена экземпляра (соответственно, `static`) с теми же именем и доступом (для полей) или с теми же именем, доступом, сигнатурой и возвращаемым типом (для методов), ни члена экземпляра (соответственно, `static`) суперкласса или подкласса.

Никакой ошибки не происходит, даже если множество связываемых классов вызывает ошибку времени компиляции.

Удаление члена или конструктора класса, который не объявлен как `private`, может вызвать ошибку связывания, если этот член или конструктор используется существующими бинарными файлами.

Пример 13.4.6-1. Изменение тела класса

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
class Test {
    public static void main(String[] args) {
        new Super().hello();
    }
}
```

Вывод этой программы имеет вид

```
hello from Super
```

Предположим, что имеется новая версия класса `Super`:

```
class Super extends Hyper {}
```

Тогда, перекомпилируя `Super` и выполняя новые бинарные файлы вместе с исходными бинарными файлами `Test` и `Hyper`, мы получим ожидаемый вывод

```
hello from Hyper
```

Ключевое слово `super` может быть использовано для доступа к методу, объявленному в суперклассе, минуя любые методы, объявленные в текущем классе. Выражение `super`.*Identifier* разрешается во время компиляции в метод *m* суперкласса *S*. Если метод *m* является методом экземпляра, то метод, вызываемый во время выполнения, представляет собой метод с той же сигнатурой, что и у *m*, который является членом непосредственного суперкласса класса, содержащего выражение, включающее `super`.

Пример 13.4.6-2. Изменение суперкласса

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        new Test().hello();
    }
    void hello() {
        super.hello();
    }
}
```

Вывод этой программы имеет вид

```
hello from Hyper
```

Предположим, что имеется новая версия класса `Super`:

```
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
```

Тогда, если перекомпилированы `Super` и `Hyper`, но не `Test`, выполнение новых бинарных файлов с существующими бинарными файлами `Test` имеет ожидаемый вид

```
hello from Super
```

§13.4.7. Доступ к членам и конструкторам

Изменение объявленного доступа к члену или конструктору, допускающее меньший уровень доступа, может нарушить совместимость с существующими бинарными файлами, приводя к генерации ошибки связывания при разрешении этих бинарных файлов. Меньший доступ разрешается, если модификатор доступа изменяется с доступа уровня пакета на `private`; с `protected` до доступа уровня пакета или `private`; или с `public` до `protected`, доступа уровня пакета или `private`. Изменение члена или конструктора, приводящее к уменьшению доступа, таким образом, не рекомендуется для широко распространенных классов.

Возможно, это покажется удивительным, но бинарный формат определен так, чтобы изменения члена или конструктора на более доступные не приводили к ошибке связывания, когда подкласс (уже) определил метод с меньшим доступом.

Пример 13.4.7-1. Изменение доступности

Если пакет `points` определяет класс `Point`

```
package points;
public class Point {
    public int x, y;
    protected void print() {
        System.out.println("(" + x + "," + y + ")");
    }
}
```

используемый в программе

```
class Test extends points.Point {
    public static void main(String[] args) {
        Test t = new Test();
        t.print();
    }
    protected void print() {
        System.out.println("Test");
    }
}
```

то эти классы успешно компилируются, и выполнение Test приводит к выводу

```
Test
```

Если метод `print` в классе `Point` изменяется и становится `public` и если после этого перекомпилируется только класс `Point`, а затем выполняется с существующими бинарными файлами `Test`, то никаких ошибок связывания не возникает. Это происходит несмотря даже на некорректное во время компиляции перекрытие `public`-метода методом `protected` (в чем можно убедиться, пытаясь скомпилировать класс `Test`, что не будет получаться с новым классом `Point`, пока вы не измените `print` на `Test`, сделав его `public`).

Позволение суперклассам изменять `protected`-методы на `public` без нарушения бинарной совместимости с существующими подклассами делает бинарные представления менее хрупкими. Альтернативный подход, при котором такое изменение приводит к ошибке связывания, создавал бы дополнительные бинарные несовместимости.

§13.4.8. Объявления полей

Широко распространенные программы не должны предоставлять какие-либо поля своим клиентам. Помимо рассматриваемых ниже вопросов бинарной совместимости, это в общем случае является правилом хорошего тона в программной инженерии. Добавление поля в класс может нарушить бинарную совместимость с существующими бинарными файлами, которые не были перекомпилированы.

Предположим, что имеется ссылка на поле *f* с квалифицирующим типом *T*. Предположим далее, что *f* фактически является полем экземпляра (соответственно, `static`), объявленным в суперклассе *S* класса *T*, и что типом *f* является *X*.

Если в подкласс класса *S*, который является суперклассом *T*, или в сам *T* добавляется новое поле типа *X* с тем же именем, что и *f*, то может произойти ошибка связывания. Такая ошибка связывания случится, только если в дополнение к сказанному выше выполнится одно из следующих условий:

- новое поле менее доступно, чем старое;
- новое поле является `static`-полем (соответственно, полем экземпляра).

В частности, никаких ошибок связывания не будет, если класс не может быть более перекомпилирован из-за доступа к полю, ранее представлявшему собой доступ к полю суперкласса с несовместимым типом. Ранее скомпилированный класс с такой ссылкой будет продолжать обращаться к полю, объявленному в суперклассе.

Пример 13.4.8-1. Добавление объявления поля

```
class Hyper { String h = "hyper"; }
class Super extends Hyper { String s = "super"; }
class Test {
    public static void main(String[] args) {
        System.out.println(new Super().h);
    }
}
```

Вывод программы будет имеет вид

```
hyper
```

Предположим, что имеется новая версия класса `Super`:

```
class Super extends Hyper {
    String s = "super";
    int h = 0;
}
```

Тогда, перекомпилируя `Hyper` и `Super` и выполняя полученные новые бинарные файлы со старым бинарным файлом `Test`, мы получим вывод

```
hyper
```

Исходным бинарным файлом `Test` выводится поле `h` класса `Hyper`. При том что, на первый взгляд, это может показаться удивительным, это правило служит снижению количества несовместимостей, которые могут обнаруживаться во время выполнения. (В идеальном мире при внесении изменения в один исходный текст перекомпилируются все файлы, нуждающиеся в этом, тем самым устраняя любые поводы к удивлению. Но такая массовая перекомпиляция часто оказывается непрактичной или невозможной, в особенности в Интернете. И, как отмечалось ранее, такая перекомпиляция может потребовать дальнейших изменений в исходном тексте.)

Рассмотрим другой пример. Если программу

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

скомпилировать и выполнить, ее вывод будет имеет вид

```
Hyper
```

Предположим, что теперь компилируется новая версия класса `Super`:

```
class Super extends Hyper { char h = 'h'; }
```

Если использовать получающийся в результате бинарный файл вместе с существующими бинарными файлами `Test` и `Hyper`, вывод программы останется неизменным:

```
Hyper
```

несмотря на то, что компиляция исходного текста для

```
class Hyper { String h = "Hyper"; }
class Super extends Hyper { char h = 'h'; }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

должна привести к ошибке времени компиляции, поскольку `h` в исходном тексте `main` будет теперь рассматриваться как поле `char`, объявленное в `Super`, а значение `char` не может быть присвоено переменной типа `String`.

Удаление поля из класса нарушает бинарную совместимость со всеми существующими бинарными файлами, которые обращаются к этому полю, и когда происходит связывание такого обращения в существующем бинарном представлении, генерируется исключение `NoSuchFieldError`. Из широко распространенных классов могут быть безопасно удалены только поля, объявленные как `private`.

С целью бинарной совместимости добавление или удаление поля *f*, тип которого включает переменные типа (§4.4) или параметризованные типы (§4.5), эквивалентно добавлению (соответственно, удалению) поля с тем же именем, тип которого является затиранием (§4.6) типа *f*.

§13.4.9. Поля `final` и статические константные переменные

Если поле, которое не было объявлено как `final`, изменяется так, что теперь оно становится объявленным как `final`, то это может нарушить бинарную совместимость с существующими бинарными файлами, пытающимися присвоить новые значения этому полю.

Пример 13.4.9-1. Изменение переменной на `final`

```
class Super { char s; }
class Test extends Super {
    public static void main(String[] args) {
        Super x = new Super();
        x.s = 'a';
        System.out.println(x.s);
    }
}
```

Вывод этой программы имеет вид

```
a
```

Предположим, что имеется новая версия класса `Super`:

```
class Super { final char s = 'b'; }
```

Если перекомпилировать `Super`, но не `Test`, выполнение нового бинарного файла вместе с имеющимся бинарным файлом `Test` приведет к ошибке `IllegalAccessError`.

Удаление ключевого слова `final` или изменение значения, которым инициализируется поле, не нарушает совместимость с существующими бинарными файлами.

Если поле представляет собой константную переменную (§4.12.4), а кроме того, является статическим, то удаление ключевого слова `final` или изменение ее значения не нарушит совместимость с существующими бинарными файлами, что привело бы к невозможности выполнения программы, но новые значения не будут восприняты программой до ее перекомпиляции. Этот результат является побочным эффектом решения о поддержке условной компиляции, рассматриваемого в конце §14.21. (Можно предположить, что

новое значение не видимо, если использование осуществляется в константном выражении (§15.28), но видимо в противном случае. Это не так; уже имеющиеся бинарные представления не увидят новое значение вообще.)

> Еще одной причиной для требования встраивания значений статических константных переменных являются инструкции `switch`. Они являются единственным видом инструкций, основанных на константных значениях, а именно — каждая метка `case` инструкции `switch` должна представлять собой константное выражение, значение которого отлично от значений всех прочих меток `case`. Метки `case` часто используют статические константные переменные, так что может не быть очевидно, что у всех меток — различные значения. Если во время компиляции доказано, что дубликатов меток нет, то встраивание этих значений в `class`-файл гарантирует, что дубликатов не будет и во время выполнения, что является очень желательным свойством.

Пример 13.4.9-2. Условная компиляция

Если скомпилировать и выполнить пример

```
class Flags { static final boolean debug = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

то вывод программы будет имеет вид

```
debug is true
```

Предположим, что имеется новая версия класса `Flags`.

```
class Flags { static final boolean debug = false; }
```

Если перекомпилировать `Flags`, но не `Test`, то выполнение нового бинарного представления вместе с существующим бинарным файлом `Test` приведет к выводу

```
debug is true
```

поскольку значение `debug` было константным выражением и могло использоваться при компиляции `Test` без обращения к классу `Flags`.

Такое поведение не изменится, если изменить `Flags`, сделав его интерфейсом, как в приведенном далее примере.

```
interface Flags { boolean debug = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

Более подробно условная компиляция обсуждается в конце §14.21.

Наилучшим способом избежать проблем с "неконстантными константами" в широко распространенном коде — использовать статические константные переменные только для значений, которые действительно никогда не будут изменяться. Помимо истинно математических констант, мы рекомендуем использовать статические константные переменные класса как можно реже.

> Если требуется поведение "только для чтения" переменной `final`, лучше объявить ее как `private static` и создать соответствующий метод доступа для получения ее значения.
>
> Таком образом, вместо
>
> ```
> public static final int N = ...;
> ```
>
> мы рекомендуем использовать
>
> ```
> private static int N;
> public static int getN() { return N; }
> ```
>
> Не будет никаких проблем с
>
> ```
> public static int N = ...;
> ```
>
> если переменная `N` не должна предназначаться только для чтения.

Мы также рекомендуем в качестве общего правила то, чтобы полям интерфейсов присваивались только константные выражения.

Отметим, не рекомендуя, что если поле примитивного типа в интерфейсе может изменяться, то его значение можно идиоматически выразить так, как в приведенном исходном тексте:

```
interface Flags {
    boolean debug = new Boolean(true).booleanValue();
}
```

гарантируя, что данное значение не является константой. Аналогичные идиомы имеются и для других примитивных типов.

Следует также отметить, что статические константные переменные никогда не должны появляться со значениями по умолчанию для своего типа (§4.12.5). Это означает, что все такие поля будут впервые инициализироваться в процессе инициализации класса (§8.3.2, §9.3.1, §12.4.2).

§13.4.10. Поля `static`

Если поле, не объявленное как `private`, не было объявлено как `static` и изменяется, становясь `static` (или наоборот), то в результате его использования существующими бинарными файлами, ожидающими поле иного вида, генерируется ошибка связывания, а именно — `IncompatibleClassChangeError`. Такие изменения не рекомендуются для широко распространенного кода.

§13.4.11. Поля `transient`

Добавление (или удаление) модификатора `transient` к полю не нарушает бинарную совместимость с существующими бинарными файлами.

§13.4.12. Объявления методов и конструкторов

Добавление объявления метода или конструктора в класс не нарушает бинарной совместимости с ранее существовавшими бинарными файлами, даже если тип более не может быть перекомпилирован из-за имевшихся ранее вызовов методов или конструкторов суперкласса с несовместимым типом. Ранее скомпилированный класс с такими ссылками будет продолжать обращаться к методу или конструктору, объявленному в суперклассе.

Предположим, у нас имеется обращение к методу *m* с квалифицирующим типом *T*. Предположим также, что *m* фактически является методом экземпляра (соответственно, `static`), объявленным в суперклассе *S* класса *T*.

Если новый метод типа *X* с той же сигнатурой и возвращаемым типом, что и у *m*, добавляется в подкласс класса *S*, который является суперклассом *T* или самим *T*, то может произойти ошибка связывания. Такая ошибка связывания произойдет только тогда, когда в дополнение к сказанному выше справедливо одно из следующих условий:

- новый метод менее доступен, чем старый;
- новый метод объявлен как `static` (соответственно, является методом экземпляра).

Удаление метода или конструктора из класса может нарушать бинарную совместимость с существующими бинарными файлами, обращающимися к этому методу или конструктору; при связывании такого обращения может генерироваться исключение `NoSuchMethodError`. Такая ошибка происходит только тогда, когда нет метода с соответствующей сигнатурой и возвращаемым типом, объявленного в суперклассе.

Если исходный текст класса, не являющегося внутренним, не содержит объявленных конструкторов, неявно объявляется конструктор по умолчанию без параметров (§8.8.9). Добавление одного или нескольких объявлений конструкторов в исходный текст такого класса предотвращает неявное объявление конструктора по умолчанию и удаляет его, если только один из новых конструкторов не является конструктором без параметров, который замещает конструктор по умолчанию. Конструктор по умолчанию без параметров имеет тот же модификатор доступа, что и класс, в котором он объявлен, так что любой замещающий конструктор для сохранности бинарной совместимости с существующими бинарными файлами должен иметь такой же или больший доступ.

§13.4.13. Параметры типа методов и конструкторов

Добавление или удаление параметра типа метода или конструктора само по себе на бинарную совместимость влияния не оказывает.

Если такой параметр типа используется в типе метода или конструктора, это может иметь обычные последствия в виде изменений упомянутого типа.

Переименование параметра типа метода или конструктора не влияет на взаимоотношения с существующими бинарными файлами.

Изменение первой границы параметра типа метода или конструктора может изменить затирание (§4.6) любого члена, который использует этот параметр типа в собственном типе, и это может повлиять на бинарную совместимость. В частности,

- если параметр типа используется в качестве типа поля, результат оказывается таким же, как если бы поле было удалено и добавлено поле с тем же именем, тип которого представляет собой новое затирание переменной типа;
- если параметр типа используется в качестве типа любого формального параметра метода, но не в качестве возвращаемого типа, результат оказывается таким же, как если бы метод был удален и заменен новым методом, идентичным во всем, за исключением типов упомянутых формальных параметров, которые теперь в качестве своих типов имеют новое замыкание параметра типа;
- если параметр типа используется в качестве возвращаемого типа метода, но не в качестве типа любого формального параметра метода, результат оказывается таким же, как если бы метод был удален и заменен новым методом, идентичным во всем, за исключением возвращаемого типа, который теперь представляет собой новое затирание параметра типа;
- если параметр типа используется в качестве возвращаемого типа метода и в качестве типа одного или нескольких формальных параметров метода, результат оказывается таким же, как если бы метод был удален и заменен новым методом, идентичным во всем, за исключением возвращаемого типа, который теперь представляет собой новое затирание параметра типа, и за исключением типов упомянутых формальных параметров, которые теперь в качестве своих типов имеют новое замыкание параметра типа.

Изменение любой другой границы не влияет на бинарную совместимость.

§13.4.14. Формальные параметры методов и конструкторов

Изменение имени формального параметра метода или конструктора не влияет на бинарную совместимость.

Изменение имени метода, типа формального параметра метода или конструктора, добавление параметра или удаление параметра из объявления метода или конструктора создает метод или конструктор с новой сигнатурой и обладает комбинированным эффектом удаления метода или конструктора со старой сигнатурой и эффектом добавления метода или конструктора с новой сигнатурой (§13.4.12).

Изменение типа последнего формального параметра метода с T`[]` на параметр переменной арности (§8.4.1) типа T (т.е. на T`...`) и обратно не влияет на существующие бинарные файлы.

С точки зрения бинарной совместимости добавление или удаление метода или конструктора m, сигнатура которого включает переменные типа (§4.4) или параметризованные типы (§4.5), эквивалентно добавлению (соответственно, удалению) эквивалентного метода, сигнатура которого представляет собой затирание (§4.6) сигнатуры m.

§13.4.15. Возвращаемый тип метода

Изменение возвращаемого типа метода, замена возвращаемого типа на `void` или замена `void` возвращаемым типом обладает комбинированным эффектом удаления старо-

го метода и добавления нового метода с новым возвращаемым типом или новым результатом `void` (§13.4.12).

С точки зрения бинарной совместимости добавление или удаление метода или конструктора *m*, возвращаемый тип которого включает переменные типа (§4.4) или параметризованные типы (§4.5), эквивалентно добавлению (соответственно, удалению) эквивалентного метода, возвращаемый тип которого представляет собой затирание (§4.6) возвращаемого типа *m*.[1]

§13.4.16. Методы `abstract`

Изменение метода, объявленного как `abstract`, заключающееся в том, что он более не является объявленным как `abstract`, не нарушает совместимости с существующими бинарными файлами.

Изменение метода, не объявленного как `abstract`, заключающееся в том, что он становится объявленным как `abstract`, нарушает совместимость с существующими бинарными файлами, которые ранее вызывали этот метод, генерируя исключение `AbstractMethodError`.

Пример 13.4.16-1. Превращение метода в абстрактный

```
class Super { void out() { System.out.println("Out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println("Way ");
        t.out();
    }
}
```

Вывод программы имеет вид

```
Way
Out
```

Предположим, что имеется новая версия класса `Super`.

```
abstract class Super {
    abstract void out();
}
```

Если класс `Super` перекомпилирован, а класс `Test` — нет, то запуск нового бинарного файла с существующим бинарным файлом `Test` приводит к генерации исключения `AbstractMethodError`, поскольку класс `Test` не имеет реализации метода `out`, а следовательно, является (или должен являться) `abstract`.

---

[1] Сказанное о возвращаемых типах не имеет отношения к конструкторам, которые не имеют возвращаемого типа. — *Примеч. пер.*

§13.4.17. Методы `final`

Изменение метода, объявленного как `final`, заключающееся в том, что он более не является объявленным как `final`, не нарушает совместимости с существующими бинарными файлами.

Изменение метода, не объявленного как `final`, заключающееся в том, что он становится объявленным как `final`, может нарушить совместимость с существующими бинарными файлами, которые зависят от возможности перекрытия этого метода.

Пример 13.4.17-1. Изменение метода на `final`

```
class Super { void out() { System.out.println("out"); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        t.out();
    }
    void out() { super.out(); }
}
```

Эта программа генерирует вывод

```
out
```

Предположим, что имеется новая версия класса `Super`.

```
class Super { final void out() { System.out.println("!"); } }
```

Если класс `Super` перекомпилирован, а класс `Test` — нет, то запуск нового бинарного файла с существующим бинарным файлом `Test` приводит к ошибке `VerifyError`, поскольку класс `Test` пытается перекрыть метод экземпляра `out`, что является некорректным.

Изменение метода класса (`static`), который не объявлен как `final`, заключающееся в его объявлении как `final`, не нарушает совместимость с существующими бинарными файлами, поскольку этот метод не может быть переопределен.

§13.4.18. Методы `native`

Добавление или удаление модификатора метода `native` не нарушает совместимость с существующими бинарными файлами.

Влияние изменений на типы существующих методов `native`, которые не были перекомпилированы, выходит за рамки данной спецификации и должно предоставляться с описанием конкретной реализации. Реализациям рекомендуется (но не требуется) реализовывать методы `native` так, чтобы ограничивать такое влияние.

§13.4.19. Методы `static`

Если метод, не объявленный как `private`, объявлен как `static` (т.е. является методом класса) и изменен таким образом, что перестает быть `static` (т.е. превращается в метод экземпляра) или наоборот, то совместимость с существующими бинарными файла-

ми может быть нарушена, так что в случае, когда эти методы используются существующими бинарными файлами, происходит ошибка связывания, а именно — `IncompatibleClassChangeError`. Такие изменения не рекомендуется вносить в широко распространенный код.

§13.4.20. Методы `synchronized`

Добавление или удаление модификатора `synchronized` к методу не нарушает бинарной совместимости с существующими бинарными файлами.

§13.4.21. Конструкция `throws` методов и конструкторов

Изменения в конструкции `throws` в методах и конструкторах не нарушает бинарной совместимости с существующими бинарными файлами; эта конструкция проверяется только во время компиляции.

§13.4.22. Тела методов и конструкторов

Изменения в теле метода или конструктора не нарушают бинарной совместимости с существующими бинарными файлами.

Ключевое слово `final` у метода не означает, что он может быть безопасно встроен в код; оно означает только, что метод не может быть перекрыт. Тем не менее новая версия метода во время связывания вполне возможна. Кроме того, для целей рефлексии следует сохранять структуру исходной программы.

Таким образом, заметим, что компилятор Java не может встраивать метод во время компиляции. В общем случае предполагается, что реализации используют позднюю (времени выполнения) генерацию и оптимизацию кода.

§13.4.23. Перегрузка методов и конструкторов

Добавление новых методов и конструкторов, перегружающих существующие методы и конструкторы, не нарушает совместимость с существующими бинарными файлами. Сигнатура, используемая при каждом вызове, определялась при компиляции существующих бинарных файлов; следовательно, новые методы или конструкторы просто не будут использоваться, даже если их сигнатуры применимы и более точны по сравнению с исходно выбранными.

В то время как добавление нового перегруженного метода или конструктора может привести к ошибке времени компиляции при очередной компиляции класса или интерфейса из-за отсутствия наиболее подходящего метода или конструктора (§15.12.2.5), при выполнении программы такая ошибка невозможна, так как во время выполнения разрешение перегрузки не выполняется.

Пример 13.4.23-1. Добавление перегруженного метода

```
class Super {
    static void out(float f) {
        System.out.println("float");
    }
}
class Test {
    public static void main(String[] args) {
        Super.out(2);
    }
}
```

Вывод программы имеет вид

```
float
```

Предположим, что имеется новая версия класса `Super`.

```
class Super {
    static void out(float f) { System.out.println("float"); }
    static void out(int i) { System.out.println("int"); }
}
```

Если класс `Super` перекомпилирован, а `Test` — нет, то запуск нового бинарного файла с существующим бинарным файлом `Test` будет продолжать давать тот же вывод

```
float
```

Однако при перекомпиляции класса `Test` с использованием нового класса `Super` вывод изменится и станет

```
int
```

т.е. таким, который можно было наивно ожидать в предыдущем случае.

§13.4.24. Перекрытие метода

Если метод экземпляра добавляется в подкласс и перекрывает метод в суперклассе, то вызовы метода в существующих бинарных файлах будут находить метод подкласса, и никакого влияния на эти бинарные файлы оказано не будет. Если в класс добавляется метод класса, то этот метод не будет найден, если только квалифицирующий тип ссылки не является типом подкласса.

§13.4.25. Статические инициализаторы

Добавление, удаление или изменение статического инициализатора (§8.7) класса не влияет на существующие бинарные файлы.

§13.4.26. Эволюция перечислений

Добавление или переупорядочение констант в перечислении не нарушает совместимость с существующими бинарными файлами.

Если существующий бинарный файл пытается обратиться к более не существующей константе перечисления, произойдет ошибка времени выполнения `NoSuchFieldError`. Поэтому такое изменение не рекомендуется для широко распространенных перечислений.

Во всех прочих отношениях правила бинарной совместимости для перечислений идентичны таковым для классов.

§13.5. Эволюция интерфейсов

В этом разделе описано влияние изменений в объявлении интерфейса и его членов на существующие бинарные файлы.

§13.5.1. `public`-интерфейсы

Изменение интерфейса, который не был объявлен как `public`, а после изменения стал `public`, не нарушает совместимость с существующими бинарными файлами.

Если интерфейс, объявленный как `public`, изменен и перестал быть `public`, то при связывании существующего бинарного файла, которому требуется доступ к типу интерфейса и которого он больше не может получить, генерируется ошибка `IllegalAccessError`. Поэтому такое изменение не рекомендуется для широко распространенных интерфейсов.

§13.5.2. Суперинтерфейсы

Изменение иерархии интерфейсов приводит к ошибкам точно таким же образом, как и изменение иерархии классов, описанное в §13.4.4. В частности, изменения, которые заключаются в том, что ранее бывший суперинтерфейсом класса интерфейс перестает быть таковым, могут привести к нарушению совместимости с существовавшими бинарными файлами и вызвать генерацию исключения `VerifyError`.

§13.5.3. Члены интерфейса

Добавление абстрактного метода в интерфейс не нарушает совместимость с существующими бинарными файлами.

Поле, добавленное в суперинтерфейс класса *C*, может скрывать поле, наследованное от суперкласса класса *C*. Если исходная ссылка обращалась к полю экземпляра, то в результате мы получим ошибку `IncompatibleClassChangeError`. Если исходное обращение было присваиванием, мы получим ошибку `IllegalAccessError`.

Удаление члена из интерфейса может привести к ошибкам связывания в существующих бинарных файлах.

Пример 13.5.3-1. Удаление члена интерфейса

```
interface I { void hello(); }
class Test implements I {
    public static void main(String[] args) {
```

```
        I anI = new Test();
        anI.hello();
    }
    public void hello() { System.out.println("hello"); }
}
```

Вывод этой программы имеет вид

```
hello
```

Предположим, что скомпилирована новая версия интерфейса `I`.

```
interface I {}
```

Если перекомпилирован только `I`, но не `Test`, то выполнение нового бинарного файла с существующим бинарным файлом `Test` приведет к ошибке `NoSuchMethodError`.

§13.5.4. Параметры типа интерфейса

Влияние изменений параметров типа интерфейса такое же, как и аналогичных изменений параметров типа класса.

§13.5.5. Объявления полей

Рассуждения в случае изменения объявлений полей в интерфейсах такие же, как и для объявленных как `static final` полей классов, рассмотренные в §13.4.8 и §13.4.9.

§13.5.6. Объявления методов интерфейсов

Рассуждения в случае изменения объявлений `abstract`-методов в интерфейсах такие же, как и в случае `abstract`-методов в классах, описанных в §13.4.14, §13.4.15, §13.4.21 и §13.4.23.

Добавление метода по умолчанию или изменение метода с абстрактного на метод по умолчанию не нарушает совместимость с существующими бинарными файлами, но может привести к ошибке `IncompatibleClassChangeError`, если существующий бинарный файл попытается вызвать метод. Эта ошибка возникает, если квалифицирующий тип T является подтипом двух интерфейсов, I и J, где как I, так и J объявляют `default`-метод с одними и теми же сигнатурой и результатом, и ни I, ни J не являются подынтерфейсами один другого.

Другими словами, добавление метода по умолчанию является бинарно совместимым изменением, поскольку оно не вносит ошибки при компоновке, даже если приводит к ошибке времени компиляции или во время вызова. На практике риск случайного конфликта из-за введения метода по умолчанию подобен риску, связанному с добавлением нового метода в класс, не являющийся финальным. В случае конфликта добавление метода к классу вряд ли приведет к ошибке `LinkageError`, но случайное перекрытие метода в дочернем классе может привести к непредсказуемому поведению метода. Оба варианта изменений могут приводить к ошибкам времени компиляции.

Пример 13.5.6-1. Добавление метода по умолчанию

```
interface Painter {
    default void draw() {
        System.out.println("Here's a picture...");
    }
}

interface Cowboy {}

public class CowboyArtist implements Cowboy, Painter {
    public static void main(String... args) {
        new CowboyArtist().draw();
    }
}
```

Вывод данной программы имеет вид

```
Here's a picture...
```

Предположим, что в `Cowboy` **добавлен метод по умолчанию.**

```
interface Cowboy {
    default void draw() {
        System.out.println("Bang!");
    }
}
```

Если перекомпилировать `Cowboy`**, но не** `CowboyArtist`**, то выполнение нового бинарного файла с существующим бинарным файлом** `CowboyArtist` **приведет к компоновке без ошибок, но вызовет ошибку** `IncompatibleClassChangeError`**, когда** `main` **попытается вызвать** `draw()`.

§13.5.7. Эволюция типов аннотаций

Типы аннотаций ведут себя в точности так же, как и любые другие интерфейсы. Добавление элемента в тип аннотации или удаление из него аналогично добавлению или удалению метода. Имеются важные соображения, касающиеся других изменений типов аннотаций, например такие, как сделать тип аннотации повторяемым (§9.6.3), но они не влияют на связывание бинарных файлов виртуальной машиной Java. Скорее, такие изменения влияют на поведение рефлективных API, работающих с аннотациями. Это поведение при различных изменениях в базовых типах аннотаций описывается в документации этих API.

Добавление или удаление аннотаций не влияет на корректность связывания бинарных представлений программ на языке программирования Java.

ГЛАВА 14

Блоки и инструкции

Последовательность выполнения программы управляется *инструкциями* (statements), которые выполняются для получения требуемого результата и не имеют значений.

Некоторые инструкции *содержат* другие инструкции как часть своей структуры; такие инструкции являются подынструкциями инструкции. Мы говорим, что инструкция *S* *непосредственно содержит* инструкцию *U*, если нет такой инструкции *T*, отличной от *S* и *U*, что *S* содержит *T*, а *T* содержит *U*. Аналогично некоторые инструкции содержат как часть своей структуры выражения (§15).

В первом разделе этой главы рассматривается различие между нормальным и преждевременным (abrupt) завершением инструкции (§14.1). Большинство из оставшихся разделов рассматривают различные виды инструкций, подробно описывая как их нормальное поведение, так и преждевременное завершение.

Первыми рассматриваются блоки (§14.2), затем — объявления локальных классов (§14.3) и инструкции объявлений локальных переменных (§14.4).

Далее поясняется грамматический маневр, позволяющий обойти знакомую проблему "висящего else" (§14.5).

В последнем разделе (§14.21) этой главы рассматривается требование *достижимости* каждой инструкции в определенном техническом смысле.

§14.1. Нормальное и преждевременное завершение инструкций

Каждая инструкция имеет нормальный режим выполнения, в котором выполняются некоторые вычислительные шаги. В следующих разделах описывается нормальный режим выполнения инструкций каждого вида.

Если все шаги выполняются, как описано, без указания на преждевременное завершение, говорится, что инструкция *завершилась нормально*. Однако ряд событий может помешать нормальному завершению инструкции.

- Инструкции `break` (§14.15), `continue` (§14.16) и `return` (§14.17) вызывают передачу управления, которая может препятствовать нормальному завершению инструкций, содержащих данные.
- Вычисление некоторых выражений может привести к генерации исключений виртуальной машиной Java (§15.6). Явная инструкция `throw` (§14.18) также вызывает ге-

нерацию исключения. Исключение приводит к передаче управления, которое может препятствовать нормальному завершению инструкций.

Если произошло такое событие, то выполнение одной или нескольких инструкций может быть прекращено до того, как будут выполнены все шаги, выполняемые в нормальном режиме; такие инструкции называются *завершенными преждевременно* (complete abruptly).

Такое невыполнение всегда имеет *причину*, которой является одна из следующих:

- инструкция `break` без метки;
- инструкция `break` с заданной меткой;
- инструкция `continue` без метки;
- инструкция `continue` с заданной меткой;
- инструкция `return` без значения;
- инструкция `return` с заданным значением;
- инструкция `throw` с заданным значением, включая исключения, сгенерированные виртуальной машиной Java.

Термины "завершена нормально" и "завершена преждевременно" применимы также к вычислению выражений (§15.6). Единственная причина, по которой выражение может быть завершено преждевременно, — генерация исключения, или из-за инструкции `throw` с заданным значением (§14.18), или из-за исключения или ошибки времени выполнения (§11, §15.6).

Если инструкция вычисляет выражение, преждевременное завершение выражения всегда приводит к немедленному завершению инструкции по той же причине. Все последующие шаги нормального режима не выполняются.

Если в данной главе явно не указано иное, преждевременное завершение подынструкций приводит к немедленному завершению самой инструкции по той же причине, и все последующие шаги нормального режима не выполняются.

Если явно не указано иное, инструкция завершается нормально, если все выражения, которые она вычисляет, и все подынструкции, которые она выполняет, завершаются нормально.

§14.2. Блоки

Блок представляет собой последовательность инструкций, объявлений локальных классов и локальных переменных в фигурных скобках.

Block:
    { *[BlockStatements]* }

BlockStatements:
    BlockStatement {BlockStatement}

BlockStatement:
    LocalVariableDeclarationStatement

ClassDeclaration
Statement

Выполнение блока осуществляется путем выполнения каждой инструкции объявления локальной переменной и других инструкций в порядке от первой до последней (слева направо). Если все эти инструкции блока завершаются нормально, то блок завершается нормально. Если любая из инструкций блока по любой причине завершается преждевременно, то весь блок завершается преждевременно по той же причине.

§14.3. Объявления локальных классов

Локальным классом является вложенный класс (§8), не являющийся членом никакого класса и имеющий имя (§6.2, §6.7).

Все локальные классы являются внутренними классами (§8.1.3).

Каждая инструкция объявления локального класса непосредственно содержит блок (§14.2). Инструкции объявлений локальных классов могут свободно чередоваться с инструкциями другого вида в пределах блока.

Если объявление локального класса содержит любой из модификаторов `public`, `protected` или `private` (§6.6) или модификатор `static` (§8.1.1), генерируется ошибка времени компиляции.

Область видимости и затенение объявления локального класса определены в §6.3 и §6.4.

Пример 14.3-1. Объявления локальных классов

Вот пример, иллюстрирующий некоторые аспекты приведенных выше правил.

```
class Global {
    class Cyclic {}
    void foo() {
        new Cyclic();                       // Создает Global.Cyclic
        class Cyclic extends Cyclic {} // Цикличное определение
        {
            class Local {}
            {
                class Local {} // Ошибка времени компиляции
            }
            class Local {}     // Ошибка времени компиляции
            class AnotherLocal {
                void bar() {
                    class Local {}     // ok
                }
            }
        }
        class Local {}          // ok, не в области видимости
    }                           //  предыдущего Local
}
```

Первая инструкция метода `foo` создает экземпляр класса-члена `Global.Cyclic`, а не экземпляр локального класса `Cyclic`, поскольку инструкция находится до области видимости объявления локального класса.

Тот факт, что область видимости локального класса охватывает все его объявление (а не только тело), означает, что определение локального класса `Cyclic` в действительности является циклическим, так как класс расширяет сам себя, а не `Global.Cyclic`. Следовательно, объявление локального класса `Cyclic` будет отвергнуто на этапе компиляции.

Поскольку имена локальных классов не могут быть объявлены заново в том же самом методе (или конструкторе, или инициализаторе), второе и третье объявления `Local` приводят к ошибкам времени компиляции. Однако `Local` может быть объявлен заново в контексте другого, более глубоко вложенного класса, такого как `AnotherLocal`.

Четвертое, и последнее, объявление `Local` корректно, так как находится вне области видимости любого более раннего объявления `Local`.

§14.4. Инструкции объявления локальных переменных

Инструкция объявления локальной переменной объявляет одно или несколько имен локальных переменных.

LocalVariableDeclarationStatement:
    LocalVariableDeclaration ;

LocalVariableDeclaration:
    {VariableModifier} UnannType VariableDeclaratorList

UnannType описывается в §8.3. Ниже для большей ясности повторены продукции из §4.3, §8.4.1 и §8.3.

VariableModifier: одно из
    Annotation `final`

VariableDeclaratorList:
    VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
    VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
    Identifier [Dims]

Dims:
    {Annotation} [ ] *{{Annotation}* [ ] *}*

VariableInitializer:

Expression
ArrayInitializer

Каждая инструкция объявления локальной переменной непосредственно содержится в блоке. Инструкции объявления локальных переменных могут свободно чередоваться с другими видами инструкций в блоке.

Помимо инструкций объявления локальных переменных, объявление локальной переменной может находиться в заголовке инструкции `for` (§14.14) или инструкции `try`-с-ресурсами (§14.20.3). В этих случаях оно выполняется так же, как если бы являлось частью инструкции объявления локальной переменной.

Правила для модификаторов аннотаций объявления локальной переменной указаны в §9.7.4 и §9.7.5.

Если `final` встречается в объявлении локальной переменной более одного раза, генерируется ошибка времени компиляции.

§14.4.1. Декларатоpы и типы локальных переменных

Каждый *декларатор* в объявлении локальной переменной объявляет одну локальную переменную, имя которой представляет собой *Identifier*, расположенный в декларатоpе.

Если в начале декларатора находится необязательное ключевое слово `final`, объявляемая переменная является финальной (§4.12.4).

Объявленный тип локальной переменной описывается нетерминалом *UnannType*, находящимся в объявлении локальной переменной, за которым следует любое количество пар квадратных скобок, за которыми в декларатоpе идет *Identifier*.

Локальная переменная типа `float` всегда содержит значение, которое представляет собой элемент набора значений float (§4.2.3); аналогично локальная переменная типа `double` всегда содержит значение, которое представляет собой элемент набора значений double. Локальной переменной типа `float` не разрешается содержать элемент расширенного набора значений float, который не является одновременно элементом набора значений float; локальной переменной типа `double` не разрешается содержать элемент расширенного набора значений double, который не является одновременно элементом набора значений double.

Область видимости и затенение локальной переменной определены в §6.3 и §6.4.

§14.4.2. Выполнение объявлений локальных переменных

Инструкция объявления локальной переменной представляет собой выполнимую инструкцию. Всякий раз при ее выполнении декларатоpы обрабатываются в порядке слева направо. Если декларатор имеет инициализирующее выражение, это выражение вычисляется, а его значение присваивается переменной.

> Если декларатор не имеет выражения инициализации, то каждому обращению к переменной должно предшествовать выполнение присваивания переменной, иначе согласно правилам §16 генерируется ошибка времени компиляции.

Каждая инициализация (за исключением первой) выполняется, только если вычисление предшествующей инициализации завершилось нормально.

Выполнение объявления локальной переменной завершается нормально, только если завершилось нормально вычисление последнего инициализирующего выражения.

Если объявление локальной переменной не содержит инициализирующих выражений, то его выполнение всегда завершается нормально.

§14.5. Инструкции

В языке программирования Java имеется много разных видов инструкций. Большинство из них соответствуют инструкциям языков программирования C и C++, но некоторые присущи только языку программирования Java.

Как и в C и C++, инструкция `if` в языке программирования Java страдает от так называемой "проблемы висящего `else`", иллюстрируемой приведенным неверно отформатированным фрагментом.

```
if (door.isOpen())
    if (resident.isVisible())
        resident.greet("Hello!");
else door.bell.ring(); // A "dangling else"
```

Проблема заключается в том, что как внешняя инструкция `if`, так и внутренняя инструкция `if` предположительно могут владеть конструкцией `else`. В данном примере можно предположить, что программист предусматривал принадлежность `else` внешней инструкции `if`.

Язык программирования Java, подобно C и C++ и многим языкам программирования до них, полагает, что конструкция `else` принадлежит наиболее глубоко вложенному `if`, которому она может принадлежать. Это правило описывается следующей грамматикой.

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
Block
EmptyStatement

ExpressionStatement
AssertStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

Ниже для большей ясности повторены продукции из §14.9.

IfThenStatement:
`if` (*Expression*) *Statement*

IfThenElseStatement:
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

Таким образом, инструкции грамматически делятся на две категории: те, которые могут завершаться инструкцией `if` без конструкции `else` ("краткая инструкция `if`"), и те, которые определенно не могут быть краткими.

Только те инструкции, которые определенно не могут завершаться краткой инструкцией `if`, могут рассматриваться как подынструкции, непосредственно предшествующие ключевому слову `else` в инструкции `if`, имеющей конструкцию `else`.

Это простое правило предотвращает появление "проблемы висящего `else`". Поведение выполнения инструкции с ограничением "нет кратких `if`" идентично поведению выполнения инструкций того же вида без указанного ограничения; различие делается только для устранения синтаксических сложностей.

§14.6. Пустая инструкция

Пустая инструкция не делает ничего.

EmptyStatement:
;

Выполнение пустой инструкции всегда завершается нормально.

§14.7. Помеченные инструкции

Инструкции могут иметь *метки*.

LabeledStatement:
Identifier : *Statement*

LabeledStatementNoShortIf:
Identifier : *StatementNoShortIf*

Identifier представляет собой метку инструкции *Statement*.

В отличие от C и C++ язык программирования Java не имеет инструкции `goto`; метки инструкций используются с инструкциями `break` и `continue` (§14.15, §14.16), которые могут находиться в любом месте помеченной инструкции.

Область видимости метки помеченной инструкции представляет собой эту инструкцию *Statement*.

Если имя метки помеченной инструкции используется в области видимости метки в качестве метки иной помеченной инструкции, генерируется ошибка времени компиляции.

Нет никаких ограничений на использование одного и того же идентификатора как метки и как имени пакета, класса, интерфейса, метода, поля, параметра или локальной переменной. Применение идентификатора в качестве метки инструкции не затемняет (§6.4.2) пакет, класс, интерфейс, метод, поле, параметр или локальную переменную с тем же именем. Применение идентификатора в качестве класса, интерфейса, метода, поля, локальной переменной или параметра обработчика исключения (§14.20) не затемняет метку инструкции с тем же именем.

Помеченная инструкция выполняется посредством выполнения непосредственно содержащейся инструкции *Statement*.

Если инструкция, помеченная с помощью *Identifier* и содержащая *Statement*, завершается преждевременно из-за `break` с тем же самым *Identifier*, то помеченная инструкция завершается нормально. Во всех прочих случаях преждевременного завершения *Statement* помеченная инструкция завершается преждевременно по той же причине.

Пример 14.7-1. Метки и идентификаторы

Приведенный далее код взят из версии класса `String` и его метода `indexOf`, метка в котором изначально называлась `test`. Замена имени метки тем же именем, что и у локальной переменной `i`, не приводит к затемнению метки в области видимости объявления переменной `i`. Таким образом, приведенный код вполне корректен.

```
class Test {
    char[] value;
    int offset, count;
    int indexOf(TestString str, int fromIndex) {
        char[] v1 = value, v2 = str.value;
        int max = offset + (count - str.count);
        int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
    i:
        for (int i = start; i <= max; i++) {
            int n = str.count, j = i, k = str.offset;
            while (n-- != 0) {
                if (v1[j++] != v2[k++])
                    continue i;
            }
            return i - offset;
```

```
        }
        return -1;
    }
}
```

Идентификатор `max` также может использоваться в качестве метки инструкции; метка не скроет локальную переменную `max` в пределах помеченной инструкции.

§14.8. Инструкции выражений

Ряд видов выражений может использоваться в качестве инструкций с помощью завершающей точки с запятой.

ExpressionStatement:
    StatementExpression ;

StatementExpression:
    Assignment
    PreIncrementExpression
    PreDecrementExpression
    PostIncrementExpression
    PostDecrementExpression
    MethodInvocation
    ClassInstanceCreationExpression

Инструкция выражения выполняется путем вычисления выражения; если выражение имеет значение, оно игнорируется.

Выполнение инструкции выражения завершается нормально тогда и только тогда, когда вычисление выражения завершается нормально.

> В отличие от C и C++ язык программирования Java разрешает использовать в качестве инструкций выражений только выражения определенного вида. Заметим, что язык программирования Java не допускает "приведение к `void`" — `void` не является типом, — так что традиционный для C трюк с записью инструкции выражения как
>
> ```
> (void)... ; // Неверно!
> ```
>
> не работает. С другой стороны, язык программирования Java допускает в инструкциях выражений все наиболее полезные виды выражений и не требует для вызова `void`-метода использования вызова метода в качестве инструкции выражения, так что такой трюк почти никогда не требуется. Если же он необходим, вместо него можно прибегнуть либо к инструкции присваивания (§15.26), либо к инструкции объявления локальной переменной (§14.4).

§14.9. Инструкция `if`

Инструкция `if` позволяет условное выполнение инструкции или условный выбор из двух инструкций, выполняя только одну из них.

IfThenStatement:
`if` (*Expression*) *Statement*

IfThenElseStatement:
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

Expression должно иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

§14.9.1. Инструкция `if-then`

Инструкция `if-then` выполняется, начиная с вычисления *Expression*. Если результат имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

Если вычисление *Expression* или последующее преобразование распаковки (если таковое выполняется) завершается преждевременно по той или иной причине, инструкция `if-then` завершается преждевременно по той же причине.

В противном случае выполнение продолжается и на основе вычисленного выражения выполняется выбор.

- Если вычисленное значение представляет собой `true`, то выполняется содержащаяся в инструкции подынструкция *Statement*; инструкция `if-then` завершается нормально тогда и только тогда, когда выполнение *Statement* завершается нормально.
- Если вычисленное значение представляет собой `false`, никакие дальнейшие действия не предпринимаются, и инструкция `if-then` завершается нормально.

§14.9.2. Инструкция `if-then-else`

Инструкция `if-then-else` выполняется, начиная с вычисления *Expression*. Если результат имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

Если вычисление *Expression* или последующее преобразование распаковки (если таковое выполняется) завершается преждевременно по той или иной причине, инструкция `if-then-else` завершается преждевременно по той же причине.

В противном случае выполнение продолжается и на основе вычисленного выражения выполняется выбор.

- Если вычисленное значение представляет собой `true`, то выполняется первая содержащаяся в инструкции подынструкция *Statement* (находящаяся перед ключевым словом `else`); инструкция `if-then-else` завершается нормально тогда и только тогда, когда выполнение этой подынструкции завершается нормально.
- Если вычисленное значение представляет собой `false`, то выполняется вторая содержащаяся в инструкции подынструкция *Statement* (находящаяся после ключевого слова `else`); инструкция `if-then-else` завершается нормально тогда и только тогда, когда выполнение этой подынструкции завершается нормально.

§14.10. Инструкция `assert`

Утверждение (assertion) представляет собой инструкцию `assert`, содержащую логическое выражение. Утверждение либо *включено* (enabled), либо *отключено* (disabled). Если оно включено, его выполнение приводит к вычислению логического выражения и выводу сообщения об ошибке, если вычисление дает значение `false`. Если утверждение отключено, его выполнение не делает ничего.

AssertStatement:
`assert` *Expression* `;`
`assert` *Expression* `:` *Expression* `;`

Для простоты первое *Expression* в обоих видах инструкции `assert` будем именовать *Expression1*, а второе *Expression* во второй инструкции — как *Expression2*.

Если выражение *Expression1* имеет тип, отличный от `boolean` или `Boolean`, генерируется ошибка времени компиляции.

Во второй форме инструкции `assert`, в случае, если *Expression2* имеет тип `void` (§15.1), генерируется ошибка времени компиляции.

Инструкция `assert`, которая выполняется после того, как ее класс завершил инициализацию, включена тогда и только тогда, когда базовая система определила, что класс верхнего уровня, который лексически содержит инструкцию `assert`, разрешает утверждения.

Разрешает ли утверждения класс верхнего уровня, определяется не позже самой ранней инициализации класса верхнего уровня и инициализации любого класса, вложенного в класс верхнего уровня, и это свойство не может быть изменено после его определения.

Инструкция `assert`, которая выполняется до того, как ее класс завершит инициализацию, включена.

> Это правило продиктовано ситуацией, которая требует специального рассмотрения. Напомним, что состояние утверждения класса устанавливается не позднее момента его инициализации. Возможно, хотя в общем случае нежелательно, выполнение методов или конструкторов до инициализации. Это может произойти, когда иерархия классов содержит цикличность в статической инициализации, как показано в следующем примере.
>
> ```
> public class Foo {
>     public static void main(String[] args) {
>         Baz.testAsserts();
>         // Будет выполнено после инициализации Baz.
>     }
> }
> class Bar {
>     static {
>         Baz.testAsserts();
>         // Будет выполнено до инициализации Baz!
>     }
> }
> class Baz extends Bar {
>     static void testAsserts() {
> ```

```
        boolean enabled = false;
        assert enabled = true;
        System.out.println("Asserts " +
            (enabled ? "enabled" : "disabled"));
    }
}
```

Вызов `Baz.testAsserts()` приводит к инициализации `Baz`. Перед тем как это произойдет, должен быть инициализирован `Bar`. Статический инициализатор `Bar` вновь вызывает `Baz.testAsserts()`. Поскольку инициализация `Baz` уже в процессе осуществления текущим потоком, второй вызов выполняется немедленно, хотя `Baz` и не инициализирован (§12.4.2).

В силу приведенного выше правила, если программа выполняется с отключенными утверждениями, она должна вывести

```
Asserts enabled
Asserts disabled
```

Отключенная инструкция `assert` не делает ничего. В частности, ни *Expression1*, ни *Expression2* (при наличии такового) не вычисляется. Выполнение отключенной инструкции `assert` всегда завершается нормально.

Выполнение включенной инструкции `assert` начинается с вычисления *Expression1*. Если результат имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

Если вычисление *Expression1* или последующее преобразование распаковки (если таковое выполняется) завершается преждевременно по той или иной причине, инструкция `assert` завершается преждевременно по той же причине.

В противном случае выполнение продолжается и на основе вычисленного выражения *Expression1* выполняется выбор.

- Если вычисленное значение равно `true`, никакие дальнейшие действия не выполняются, и инструкция `assert` завершается нормально.
- Если вычисленное значение равно `false`, поведение выполнения зависит от наличия выражения *Expression2*.
  - ✦ Если выражение *Expression2* в наличии, оно вычисляется.
    - — Если вычисление завершается преждевременно по некоторой причине, инструкция `assert` завершается преждевременно по той же причине.
    - — Если вычисление завершается нормально, создается экземпляр `AssertionError`, "детальное сообщение" которого представляет собой результирующее значение *Expression2*.
      - » Если создание экземпляра завершается преждевременно по некоторой причине, инструкция `assert` завершается преждевременно по той же причине.
      - » Если создание экземпляра завершается нормально, инструкция утверждения завершается преждевременно путем генерации вновь созданного объекта `AssertionError`.
  - ✦ Если выражение *Expression2* отсутствует, создается экземпляр `AssertionError` без "детального сообщения".

— Если создание экземпляра завершается преждевременно по некоторой причине, инструкция `assert` завершается преждевременно по той же причине.

— Если создание экземпляра завершается нормально, инструкция утверждения завершается преждевременно путем генерации вновь созданного объекта `AssertionError`.

Как правило, проверка утверждений включена во время разработки и тестирования программы и отключается для повышения производительности при ее выпуске.

Поскольку проверка утверждений может быть отключена, программы не должны полагаться на то, что будут вычисляться выражения, содержащиеся в утверждениях. Таким образом, в общем случае логические выражения не должны иметь побочных эффектов. Вычисление такого логического выражения не должно затрагивать никакое состояние, видимое после завершения вычисления. Наличие побочных действий при вычислении логического выражения в утверждении не является ошибкой, но обычно неуместно, так как может вызвать изменение поведения программы в зависимости от того, включена ли проверка утверждений.

Аналогично утверждения не должны использоваться для проверки аргументов в методах, объявленных как `public`. Проверка аргументов обычно является частью контракта метода, и этот контракт должен соблюдаться вне зависимости от того, включена ли проверка утверждений.

Еще одной проблемой с использованием утверждений для проверки аргументов является то, что ошибочные аргументы должны привести к соответствующим исключениям времени выполнения (таким, как `IllegalArgumentException`, `ArrayIndexOutOfBoundsException` или `NullPointerException`). Ошибка проверки утверждения не вызовет соответствующее исключение. И вновь, хотя применение утверждений к аргументам методов, объявленных как `public`, не является ошибкой, в общем случае это неуместно. Предполагается, что исключение `AssertionError` никогда не будет перехвачено, но это можно сделать; таким образом, правила для инструкций `try` должны рассматривать утверждения, содержащиеся в блоке `try`, аналогично текущему рассмотрению инструкций `throw`.

§14.11. Инструкция `switch`

Инструкция `switch` передает управление одной из нескольких инструкций в зависимости от значения выражения.

SwitchStatement:
    `switch` `(` *Expression* `)` *SwitchBlock*

SwitchBlock:
    `{` *{SwitchBlockStatementGroup} {SwitchLabel}* `}`

SwitchBlockStatementGroup:
    SwitchLabels BlockStatements

SwitchLabels:

SwitchLabel {SwitchLabel}

SwitchLabel:
`case` *ConstantExpression* `:`
`case` *EnumConstantName* `:`
`default :`

EnumConstantName:
Identifier

Тип *Expression* должен быть `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, `String` или типом перечисления (§8.9); в противном случае генерируется ошибка времени компиляции.

Тело инструкции `switch` известно как *блок switch*. Любая инструкция, непосредственно содержащаяся в блоке switch, может быть помечена одной или несколькими *метками switch*, которые представляют собой метки `case` или `default`. Каждая метка `case` имеет соответствующую константу, которая представляет собой либо константное выражение, либо имя константы перечисления. Метки `switch`, как и значения констант `case`, называются *связанными* с инструкцией `switch`.

Для заданной инструкции `switch` должны быть выполнены все перечисленные ниже условия, иначе генерируется ошибка времени компиляции.

- Каждая константа `case`, связанная с инструкцией `switch`, должна быть присваиваема типу *Expression* инструкции `switch` (§5.2).
- Если тип *Expression* инструкции `switch` является типом перечисления, то каждая константа `case`, связанная с инструкцией `switch`, должна быть константой соответствующего типа.
- Никакие две константы `case`, связанные с инструкцией `switch`, не могут иметь одно и то же значение.
- Ни одна константа `case`, связанная с инструкцией `switch`, не может иметь значение `null`.
- С одной инструкцией `switch` может быть связано не более одной метки `default`.

> Запрет на использование `null` в качестве константы `case` предупреждает написание кода, который никогда не может быть выполнен. Если *Expression* инструкции `switch` имеет тип ссылки, т.е. тип `String`, упакованный примитивный тип или тип перечисления, то, если результатом вычисления *Expression* является `null`, генерируется ошибка времени выполнения. По мнению разработчиков языка программирования Java, это лучший выход, чем тихий пропуск всей инструкции `switch` или выбор для выполнения инструкции (если таковые имеются) после метки `default` (при наличии таковой).
>
> Компилятору Java рекомендуется (хотя и не требуется в обязательном порядке) выводить предупреждение в случае, если в `switch` с выражениями значений перечисления отсутствует метка `default` и нет меток `case` для одного или нескольких констант типа перечисления. Такая инструкция может молча ничего не делать в случае, если вычисление выражения приводит к одной из отсутствующих констант.

В C и C++ тело инструкции `switch` может быть инструкцией, и инструкции с метками `case` не обязаны непосредственно содержаться в этой инструкции. Рассмотрим простой цикл

```
for (i = 0; i < n; ++i) foo();
```

в котором известно, что `n` представляет собой положительное число. В C и C++ для разворачивания цикла можно применить трюк под названием *Duff's device*, но этот код не является корректным в языке программирования Java.

```
int q = (n+7)/8;
switch (n%8) {
    case 0: do { foo(); // Отличный трюк,
    case 7:      foo(); // но не для Java.
    case 6:      foo();
    case 5:      foo();
    case 4:      foo();
    case 3:      foo();
    case 2:      foo();
    case 1:      foo();
            } while (--q > 0);
}
```

К счастью, этот трюк, похоже, не слишком известен и не используется. Да он и не нужен в настоящее время; такого рода преобразование кода вполне по силам современным оптимизирующим компиляторам.

При выполнении инструкции `switch` сначала вычисляется значение *Expression*. Если значением *Expression* является `null`, генерируется исключение `NullPointerException`, и вся инструкция `switch` преждевременно завершается по этой причине. В противном случае, если результат имеет ссылочный тип, выполняется преобразование распаковки (§5.1.8).

Если вычисление *Expression* или последующее преобразование распаковки (если таковое имеет место) по некоторой причине завершается преждевременно, инструкция `switch` завершается преждевременно по той же причине.

В противном случае выполнение продолжается путем сравнения значения *Expression* с каждой из констант `case`, в процессе чего делается выбор.

- Если одна из констант `case` равна значению выражения, то `case` *соответствует* вычисленному значению, и последовательно выполняются все инструкции после метки такого соответствующего `case` в блоке `switch`, если таковые имеются.

  Если все эти инструкции завершаются нормально или если после соответствующей метки `case` нет инструкций, вся инструкция `switch` завершается нормально.

- Если соответствующая метка `case` отсутствует, но имеется метка `default`, то последовательно выполняются все инструкции после метки `default` в блоке `switch`, если таковые имеются.

  Если все эти инструкции завершаются нормально или если после метки `default` нет инструкций, вся инструкция `switch` завершается нормально.

- Если нет ни соответствующей метки `case`, ни метки `default`, то никакие дальнейшие действия не предпринимаются и инструкция `switch` завершается нормально.

Если любая инструкция, непосредственно содержащаяся в теле *Block* инструкции `switch`, завершается преждевременно, она обрабатывается следующим образом.

- Если выполнение *Statement* завершается преждевременно из-за наличия `break` без метки, никакие дальнейшие действия не предпринимаются и инструкция `switch` завершается нормально.
- Если выполнение *Statement* завершается преждевременно по любой иной причине, инструкция `switch` завершается преждевременно по той же самой причине.

Случай преждевременного завершения по причине `break` с меткой обрабатывается общим правилом для помеченных инструкций (§14.7).

Пример 14.11-1. "Проваливание" в инструкции `switch`

Как и в C или C++, выполнение инструкций в блоке `switch` "проваливается" через метки. Например, программа

```
class TooMany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.print("one ");
            case 2: System.out.print("too ");
            case 3: System.out.println("many");
        }
    }
    public static void main(String[] args) {
        howMany(3);
        howMany(2);
        howMany(1);
    }
}
```

содержит блок `switch`, в котором код для каждого `case` "проваливается" в следующий `case`. В результате вывод программы имеет вид

```
many
too many
one too many
```

Чтобы избежать этого эффекта, следует использовать инструкции `break`, как в приведенном ниже примере.

```
class TwoMany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.println("one");
                    break; // Выход из switch
            case 2: System.out.println("two");
                    break; // Выход из switch
            case 3: System.out.println("many");
```

```
                    break; // Формально не требуется,
                           // просто правило хорошего стиля
            }
        }
        public static void main(String[] args) {
            howMany(1);
            howMany(2);
            howMany(3);
        }
}
```

Вывод этой программы имеет вид

```
one
two
many
```

§14.12. Инструкция `while`

Инструкция `while` многократно выполняет *Expression* и *Statement* до тех пор, пока значение *Expression* не станет равным `false`.

WhileStatement:
`while` (*Expression*) *Statement*

WhileStatementNoShortIf:
`while` (*Expression*) *StatementNoShortIf*

Выражение *Expression* должно иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

При выполнении инструкции `while` сначала вычисляется выражение *Expression*. Если результат имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

Если вычисление *Expression* или последующее преобразование распаковки (если таковое имело место) завершается преждевременно, инструкция `while` завершается преждевременно по той же самой причине.

В противном случае выполнение продолжается, при этом на основании вычисленного значения делается выбор.

- Если вычисленное значение представляет собой `true`, то выполняется содержащаяся в инструкции `while` подынструкция *Statement*. При этом возможны следующие варианты.
  - ✦ Если выполнение *Statement* завершается нормально, то вся инструкция `while` выполняется заново, начиная с повторного вычисления *Expression*.
  - ✦ Если выполнение *Statement* завершается преждевременно, смотрите §14.12.1.
- Если (возможно, нераспакованное) значение *Expression* равно `false`, никакие дальнейшие действия не предпринимаются, а инструкция `while` нормально завершается.

> Если (возможно, нераспакованное) значение *Expression* равно `false` при первом вычислении, то *Statement* не выполняется.

§14.12.1. Преждевременное завершение инструкции `while`

Преждевременное завершение содержащейся в инструкции `while` подынструкции *Statement* обрабатывается следующим образом.

- Если выполнение *Statement* завершается преждевременно из-за инструкции `break` без метки, дальнейшие действия не предпринимаются и инструкция `while` завершается нормально.
- Если выполнение *Statement* завершается преждевременно из-за инструкции `continue` без метки, вся инструкция `while` выполняется заново.
- Если выполнение *Statement* завершается преждевременно из-за инструкции `continue` с меткой *L*, выполняется следующий выбор дальнейших действий.
  - ✦ Если инструкция `while` имеет метку *L*, вся инструкция `while` выполняется заново.
  - ✦ Если инструкция `while` не имеет метку *L*, инструкция `while` завершается преждевременно из-за инструкции `continue` с меткой *L*.
- Если выполнение *Statement* завершается преждевременно по любой иной причине, инструкция `while` завершается преждевременно по той же причине.

Случай преждевременного завершения из-за инструкции `break` с меткой обрабатывается общим правилом для помеченных инструкций (§14.7).

§14.13. Инструкция do

Инструкция `do` многократно выполняет *Statement* и *Expression* до тех пор, пока значение *Expression* не станет равным `false`.

DoStatement:
`do` *Statement* `while (` *Expression* `) ;`

Выражение *Expression* должно иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

Выполнение инструкции `do` начинается с выполнения *Statement*. Затем возможны следующие варианты.

- Если выполнение *Statement* завершается нормально, вычисляется *Expression*. Если результат вычисления имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

  Если вычисление *Expression* или последующее преобразование распаковки (если таковое имеет место) завершается преждевременно по некоторой причине, то инструкция `do` завершается преждевременно по той же самой причине.

  В противном случае выполняется следующий выбор дальнейших действий.
  - ✦ Если значение равно `true`, вся инструкция `do` выполняется заново.
  - ✦ Если значение равно `false`, дальнейшие действия не предпринимаются и инструкция `do` завершается нормально.
- Если выполнение *Statement* завершается преждевременно, смотрите §14.13.1.

Выполнение инструкции `do` всегда приводит к выполнению содержащейся подынструкции *Statement* как минимум один раз.

§14.13.1. Преждевременное завершение инструкции do

Преждевременное завершение содержащейся в инструкции `do` подынструкции *Statement* обрабатывается следующим образом.

- Если выполнение *Statement* завершается преждевременно из-за инструкции `break` без метки, то никакие дальнейшие действия не предпринимаются и инструкция `do` завершается нормально.
- Если выполнение *Statement* завершается преждевременно из-за инструкции `continue` без метки, то вычисляется выражение *Expression*. Затем на основании вычисленного значения выполняются следующие действия.
  - ✦ Если вычисленное значение равно `true`, то вся инструкция `do` выполняется заново.
  - ✦ Если вычисленное значение равно `false`, никакие дальнейшие действия не предпринимаются и инструкция `do` завершается нормально.
- Если выполнение *Statement* завершается преждевременно из-за инструкции `continue` с меткой *L*, то выполняются следующие действия.
  - ✦ Если инструкция `do` имеет метку *L*, то вычисляется *Expression*; после этого
    - — если значение *Expression* равно `true`, то вся инструкция `do` выполняется заново;
    - — если значение *Expression* равно `false`, никакие дальнейшие действия не предпринимаются и инструкция `do` завершается нормально.
  - ✦ Если инструкция `do` не имеет метки *L*, инструкция `do` завершается преждевременно из-за инструкции `continue` с меткой *L*.
- Если выполнение *Statement* завершается преждевременно по некоторой иной причине, инструкция `do` завершается преждевременно по той же причине.

Случай преждевременного завершения из-за инструкции `break` с меткой обрабатывается общим правилом для помеченных инструкций (§14.7).

Пример 14.13-1. Инструкция do

Приведенный далее исходный текст представляет собой одну из возможных реализаций метода `toHexString` класса `Integer`.

```
public static String toHexString(int i) {
    StringBuffer buf = new StringBuffer(8);
    do {
        buf.append(Character.forDigit(i & 0xF, 16));
        i >>>= 4;
    } while (i != 0);
    return buf.reverse().toString();
}
```

Поскольку требуется сгенерировать по крайней мере одну цифру, инструкция `do` оказывается вполне подходящей структурой управления.

§14.14. Инструкция `for`

Инструкция `for` имеет две разновидности — базовую и расширенную.

ForStatement:
BasicForStatement
EnhancedForStatement

ForStatementNoShortIf:
BasicForStatementNoShortIf
EnhancedForStatementNoShortIf

§14.14.1. Базовая инструкция `for`

Базовая инструкция `for` выполняет некоторый инициализирующий код, затем многократно выполняет *Expression*, *Statement* и некоторый код обновления до тех пор, пока вычисленное значение *Expression* не станет равным `false`.

BasicForStatement:
`for` (*[ForInit]* ; *[Expression]* ; *[ForUpdate]*) *Statement*

BasicForStatementNoShortIf:
`for` (*[ForInit]* ; *[Expression]* ; *[ForUpdate]*) *StatementNoShortIf*

ForInit:
StatementExpressionList
LocalVariableDeclaration

ForUpdate:
StatementExpressionList

StatementExpressionList:
StatementExpression {, StatementExpression}

Значение *Expression* должно иметь тип type `boolean` или `Boolean`, в противном случае генерируется ошибка времени компиляции.

Область видимости и затенение локальной переменной, объявленной в части *ForInit* базовой инструкции `for`, определены в §6.3 и §6.4.

§14.14.1.1. Инициализация инструкции `for`

Выполнение инструкции `for` начинается с выполнения кода *ForInit*.

- Если код *ForInit* представляет собой список инструкций выражений (§14.8), то выражения вычисляются последовательно слева направо; их значения, если таковые имеются, игнорируются.

  Если вычисление любого выражения завершается преждевременно по некоторой причине, инструкция `for` завершается преждевременно по той же самой причине; любые инструкции *ForInit* справа от преждевременно завершившейся не вычисляются.

- Если исходный текст *ForInit* представляет собой объявление локальной переменной (§14.4), оно выполняется так, как если бы это была инструкция объявления локальной переменной, находящаяся в блоке.

  Если выполнение объявления локальной переменной завершается преждевременно по некоторой причине, инструкция `for` завершается преждевременно по той же причине.

- Если часть *ForInit* отсутствует, никакие дальнейшие действия не предпринимаются.

§14.14.1.2. Итерации инструкции `for`

Затем выполняется шаг итерации инструкции `for`, как описано ниже.

- Если в инструкции имеется выражение *Expression*, оно вычисляется. Если результат вычисления имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

  Если вычисление *Expression* или последующее преобразование распаковки (если таковое имело место) завершается преждевременно, инструкция `for` завершается преждевременно по той же самой причине.

> В противном случае имеется выбор, основанный на наличии или отсутствии выражения *Expression* и результирующего значения при наличии *Expression*; смотрите следующий абзац.

- Если *Expression* отсутствует или если присутствует, но результат его вычисления (включая возможную распаковку) равен `true`, выполняется подынструкция *Statement*. Затем возможны следующие варианты.

  ✦ Если выполнение *Statement* завершается нормально, то последовательно выполняются следующие два шага.

  1. Сначала, если присутствует часть *ForUpdate*, выражения вычисляются в порядке слева направо; их значения, если таковые имеются, игнорируются. Если вычисление любого выражения завершается преждевременно по некоторой причине, инструкция `for` завершается преждевременно по той же самой причине; ни одна инструкция *ForUpdate* справа от завершившейся преждевременно не вычисляется.

     Если часть *ForUpdate* отсутствует, никакие действия не предпринимаются.

  2. Затем выполняется очередная итерация `for`.

  ✦ Если выполнение инструкции *Statement* завершается преждевременно, смотрите §14.14.1.3.

- Если присутствует выражение *Expression* и его вычисление (включая возможную распаковку) дает значение `false`, никакие дальнейшие действия не предпринимаются и инструкция `for` завершается нормально.

> Если (возможно, распакованное) значение *Expression* равно `false` при первом вычислении, то инструкция *Statement* не выполняется.

Если выражение *Expression* отсутствует, единственный путь для нормального завершения инструкции `for` заключается в применении инструкции `break`.

§14.14.1.3. Преждевременное завершение инструкции `for`

Преждевременное завершение содержащейся в инструкции `for` подынструкции *Statement* обрабатывается следующим образом.

- Если выполнение *Statement* завершается преждевременно из-за инструкции `break` без метки, никакие дальнейшие действия не предпринимаются и инструкция `for` завершается нормально.
- Если выполнение *Statement* завершается преждевременно из-за инструкции `continue` без метки, последовательно выполняются следующие два шага.

  1. Сначала, если присутствует часть *ForUpdate*, выражения вычисляются в порядке слева направо; их значения, если таковые имеются, игнорируются.

     Если часть *ForUpdate* отсутствует, никакие действия не предпринимаются.

  2. Затем выполняется очередная итерация `for`.

- Если вычисление инструкции *Statement* завершается преждевременно из-за инструкции `continue` с меткой *L*, то выполняются следующие действия.

  ✦ Если инструкция `for` имеет метку *L*, то последовательно выполняются следующие шаги.

  1. Сначала, если присутствует часть *ForUpdate*, выражения вычисляются в порядке слева направо; их значения, если таковые имеются, игнорируются.

     Если часть *ForUpdate* отсутствует, никакие действия не предпринимаются.

  2. Затем выполняется очередная итерация `for`.

  ✦ Если инструкция `for` не имеет метки *L*, инструкция `for` завершается преждевременно из-за инструкции `continue` с меткой *L*.

- Если выполнение инструкции *Statement* завершается преждевременно по некоторой иной причине, инструкция `for` завершается преждевременно по той же самой причине.

> Случай преждевременного завершения из-за инструкции `break` с меткой обрабатывается общим правилом для помеченных инструкций (§14.7).

§14.14.2. Расширенная инструкция `for`

Расширенная инструкция `for` имеет вид

EnhancedForStatement:
`for` `(` *{VariableModifier} UnannType VariableDeclaratorId*
`:` *Expression* `)`
Statement

EnhancedForStatementNoShortIf:
`for` `(` *{VariableModifier} UnannType VariableDeclaratorId*
`:` *Expression* `)`
StatementNoShortIf

UnannType описан в §8.3. Ниже для большей ясности повторены продукции из 4.3, §8.4.1 и §8.3.

VariableModifier: одно из
    Annotation `final`

VariableDeclaratorId:
    Identifier [Dims]

Dims:
    {Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

Выражение *Expression* должно иметь тип `Iterable` или тип массива (§10.1), иначе генерируется ошибка времени компиляции.

Область видимости и затенение локальной переменной, объявленной в заголовке расширенной инструкции `for`, определены в §6.3 и §6.4.

Смысл расширенной инструкции `for` получается путем трансляции в базовую инструкцию `for` следующим образом.

- Если тип *Expression* представляет собой подтип `Iterable`, трансляция имеет следующий вид.

  Если тип *Expression* представляет собой подтип `Iterable<`*X*`>` для некоторого аргумента типа *X*, то обозначим через *I* тип `java.util.Iterator<`*X*`>`; в противном случае *I* означает несформированный тип `java.util.Iterator`.

  Расширенная инструкция `for` эквивалентна базовой инструкции `for` вида

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
    {VariableModifiers} TargetType Identifier =
        (TargetType) #i.next();
    Statement
}
```

`#i` представляет собой автоматически сгенерированный идентификатор, отличный от всех прочих идентификаторов (как автоматически сгенерированных, так и остальных), находящихся в области видимости (§6.3) в точке, где находится расширенная инструкция `for`.

Если *UnannType* локальной переменной в заголовке расширенной инструкции `for` представляет собой ссылочный тип, то *TargetType* равен *UnannType*; в противном случае *TargetType* представляет собой верхнюю границу преобразования при фиксации (§5.1.10) аргумента типа *I*, или `Object`, если *I* представляет собой несформированный тип.

Например, исходный текст

```
List<? extends Integer> l = ...
for (float i : l) ...
```

транслируется в

```
for (Iterator<Integer> #i = l.iterator(); #i.hasNext(); ) {
    float #i0 = (Integer)#i.next();
    ...
```

- В противном случае *Expression* с необходимостью имеет тип массива `T[]`.

  Пусть $L_1 \ldots L_m$ представляет собой (возможно, пустую) последовательность меток, непосредственно предшествующую расширенной инструкции `for`.

  Расширенная инструкция `for` эквивалентна базовой инструкции `for` вида

```
T[] #a = Expression;
L1: L2: ... Lm:
for (int #i = 0; #i < #a.length; #i++) {
    {VariableModifiers} TargetType Identifier = #a[#i];
    Statement
}
```

`#a` и `#i` представляют собой автоматически сгенерированные идентификаторы, отличные от всех прочих идентификаторов (как автоматически сгенерированных, так и остальных), находящихся в области видимости (§6.3) в точке, где находится расширенная инструкция `for`.

TargetType представляет собой тип переменной цикла, описываемый с помощью *UnannType* в заголовке расширенной инструкции `for`, за которым следует любое количество пар квадратных скобок, находящихся после *Identifier* из *FormalParameter* (§10.2).

Пример 14.14-1. Расширенная инструкция `for` и массивы

Приведенная далее программа, вычисляющая сумму целочисленного массива, демонстрирует работу расширенной инструкции `for`.

```
int sum(int[] a) {
    int sum = 0;
    for (int i : a) sum += i;
    return sum;
}
```

Пример 14.14-2. Расширенная инструкция `for` и преобразование распаковки

Приведенная далее программа комбинирует расширенную инструкцию `for` с автоматической распаковкой для преобразования гистограммы в таблицу частот.

```
Map<String, Integer> histogram = ...;
double total = 0;
for (int i : histogram.values())
    total += i;
for (Map.Entry<String, Integer> e : histogram.entrySet())
    System.out.println(e.getKey() + " " + e.getValue() /
total);
}
```

§14.15. Инструкция `break`

Инструкция `break` передает управление вовне охватывающей инструкции.

BreakStatement:
`break` *[Identifier]* `;`

Инструкция `break` без метки пытается передать управление наиболее глубоко вложенной охватывающей инструкции `switch`, `while`, `do` или `for` непосредственно охватывающего блока или инициализатора; затем эта инструкция, которая называется *целевой*, немедленно завершается нормально.

Строго говоря, инструкция `break` без метки всегда завершается преждевременно по той причине, что она представляет собой инструкцию `break` без метки.

Если в непосредственно охватывающем методе, конструкторе или инициализаторе нет инструкции `switch`, `while`, `do` или `for`, содержащей данную инструкцию `break`, генерируется ошибка времени компиляции.

Инструкция `break` с меткой *Identifier* пытается передать управление охватывающей помеченной инструкции (§14.7), которая имеет тот же *Identifier* в качестве метки; эта *целевая* инструкция немедленно завершается нормально. В данном случае целевая инструкция не обязана быть инструкцией `switch`, `while`, `do` или `for`.

Строго говоря, инструкция `break` с меткой *Identifier* всегда завершается преждевременно по той причине, что она представляет собой инструкцию `break` с меткой *Identifier*.

Инструкция `break` должна ссылаться на метку в непосредственно охватывающем методе, конструкторе, инициализаторе или теле лямбда-выражения. Нелокальные переходы в Java отсутствуют. Если помеченной инструкции с *Identifier* в качестве метки в непосредственно охватывающем рассматриваемую инструкцию `break` методе, конструкторе, инициализаторе или теле лямбда-выражения нет, генерируется ошибка времени компиляции.

> Таким образом, можно видеть, что инструкция `break` всегда завершается преждевременно.
>
> В приведенном описании говорится “пытается передать управление”, а не “передает управление”, поскольку если в целевой инструкции имеются инструкции `try` (§14.20), `try`- или `catch`-блоки которых содержат инструкцию `break`, то перед тем как управление будет передано целевой инструкции, выполняются любые конструкции `finally` этих инструкций `try` в порядке от внутренних к внешним. Преждевременное завершение конструкции `finally` может нарушить передачу управления, инициированную инструкцией `break`.

Пример 14.15-1. Инструкция `break`

В приведенном далее примере математический граф представлен массивом массивов. Граф состоит из множества узлов и множества ребер; каждое ребро представляет собой стрелку, которая указывает из некоторого узла в некоторый другой узел или из узла в этот же узел. В данном примере предполагается, что избыточных ребер нет, так что для любых двух узлов P и Q, где Q может совпадать с P, имеется не более одного ребра от P к Q.

Узлы представлены целыми числами, и для каждых `i` и `j`, для которых обращение к элементу `edges[i][j]` не генерирует исключения `ArrayIndexOutOfBoundsException`, имеется ребро, идущее из узла `i` в узел `edges[i][j]`.

Задача метода `loseEdges` заключается в построении для заданных `i` и `j` нового графа путем копирования заданного, но без ребра от узла `i` к узлу `j` и ребра от узла `j` к узлу `i`, если таковые имеются.

```
class Graph {
    int edges[][];
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
        for (int k = 0; k < n; ++k) {
edgelist:
{
            int z;
search:
{
            if (k == i) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == j) break search;
                }
            } else if (k == j) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == i) break search;
                }
            }
            // Нет ребер, которые надо удалять;
            // используем этот список.
            newedges[k] = edges[k];
            break edgelist;
} //search
            // Копируем список, пропуская ребро в позиции z.
            int m = edges[k].length - 1;
            int ne[] = new int[m];
            System.arraycopy(edges[k], 0, ne, 0, z);
            System.arraycopy(edges[k], z+1, ne, z, m-z);
            newedges[k] = ne;
} //edgelist
        }
        return new Graph(newedges);
    }
}
```

Обратите внимание на использование двух меток инструкций, `edgelist` и `search`, и инструкции `break`. Это позволяет коду, который копирует список, пропуская одно ребро, совместить две проверки, на существование ребра из узла `i` в узел `j` и ребра из узла `j` в узел `i`.

§14.16. Инструкция `continue`

Инструкция `continue` может находиться только в инструкциях `while`, `do` или `for`; инструкции этих трех видов называются *итеративными*. Управление передается в точку продолжения цикла инструкции итерации.

ContinueStatement:
`continue` *[Identifier]* `;`

Инструкция `continue` без метки пытается передать управление наиболее глубоко вложенной охватывающей инструкции `while`, `do` или `for` непосредственно охватывающего метода, конструктора или инициализатора; затем эта инструкция, которая называется *целевой для continue*, немедленно завершает текущую итерацию и начинает новую.

Строго говоря, такая инструкция `continue` всегда завершается преждевременно по той причине, что она представляет собой инструкцию `continue` без метки.

Если не имеется инструкции `while`, `do` или `for` непосредственно охватывающего метода, конструктора или инициализатора, которая содержит эту инструкцию `continue`, генерируется ошибка времени компиляции.

Инструкция `continue` с меткой *Identifier* пытается передать управление охватывающей помеченной инструкции (§14.7), которая в качестве метки имеет тот же *Identifier*; затем эта *целевая* инструкция немедленно завершает текущую итерацию и начинает новую.

Строго говоря, такая инструкция `continue` с меткой *Identifier* всегда завершается преждевременно по той причине, что она представляет собой инструкцию `continue` с меткой *Identifier*.

Целевая инструкция должна представлять собой инструкцию `while`, `do` или `for`, иначе генерируется ошибка времени компиляции.

Инструкция `continue` должна ссылаться на метку в непосредственно охватывающем методе, конструкторе, инициализаторе или теле лямбда-выражения. Нелокальные переходы в Java отсутствуют. Если помеченной инструкции с *Identifier* в качестве метки в непосредственно охватывающем рассматриваемую инструкцию `continue` методе, конструкторе, инициализаторе или теле лямбда-выражения нет, генерируется ошибка времени компиляции.

> Таким образом, можно видеть, что инструкция `continue` всегда завершается преждевременно.
>
> Смотрите описание обработки преждевременного завершения из-за инструкции `continue` в описаниях инструкций `while` (§14.12), `do` (§14.13) и `for` (§14.14).
>
> В приведенном описании говорится “пытается передать управление”, а не “передает управление”, поскольку если в целевой инструкции имеются инструкции `try` (§14.20), `try`- или `catch`-блоки которых содержат инструкцию `continue`, то перед тем как управление будет передано целевой инструкции, выполняются любые конструкции `finally` этих инструкций `try` в порядке от внутренних к внешним. Преждевременное завершение конструкции `finally` может нарушить передачу управления, инициированную инструкцией `continue`.

Пример 14.16-1. Инструкция `continue`

В классе `Graph` в §14.15 одна из инструкций `break` используется для завершения выполнения всего тела внешнего цикла `for`. Эта инструкция `break` может быть заменена инструкцией `continue`, если цикл `for` сам является помеченным.

```
class Graph {
    int edges[][];
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
edgelists:
    for (int k = 0; k < n; ++k) {
        int z;
search:
{
        if (k == i) {
            for (z = 0; z < edges[k].length; ++z) {
                if (edges[k][z] == j) break search;
            }
        } else if (k == j) {
            for (z = 0; z < edges[k].length; ++z) {
                if (edges[k][z] == i) break search;
            }
        }
        // Нет ребер, которые надо удалять;
        // используем этот список.
        newedges[k] = edges[k];
        continue edgelists;
} //search
            // Копируем список, пропуская ребро в позиции z.
            int m = edges[k].length - 1;
            int ne[] = new int[m];
            System.arraycopy(edges[k], 0, ne, 0, z);
            System.arraycopy(edges[k], z+1, ne, z, m-z);
            newedges[k] = ne;
        } //edgelists
        return new Graph(newedges);
    }
}
```

Какой именно способ использовать, зависит в большей степени от стиля программирования, так как функционально они эквивалентны.

§14.17. Инструкция `return`

Инструкция `return` возвращает управление коду, вызвавшему метод (§8.4, §15.12) или конструктор (§8.8, §15.9).

ReturnStatement:
`return` *[Expression]* `;`

Инструкция `return` *содержится* в наиболее глубоко вложенном конструкторе, методе, инициализаторе или лямбда-выражении, тело которого охватывает инструкцию `return`.

Если инструкция `return` содержится в инициализаторе экземпляра или статическом инициализаторе (§8.6, §8.7), генерируется ошибка времени компиляции.

Инструкция `return` без выражения *Expression* должна содержаться в одном из пунктов приведенного далее списка, иначе генерируется ошибка времени компиляции.

- В методе, который с помощью ключевого слова `void` объявлен как не возвращающий никакого значения (§8.4.5).
- В конструкторе (§8.8.7).
- В лямбда-выражении (§15.27).

Инструкция `return` без выражения *Expression* пытается передать управление коду, вызвавшему метод, конструктор или лямбда-выражение, содержащее эту инструкцию. Строго говоря, инструкция `return` без выражения *Expression* всегда завершается преждевременно по той причине, что она представляет собой инструкцию `return` без выражения *Expression*.

Инструкция `return` с выражением *Expression* должна содержаться в одном из пунктов приведенного далее списка, иначе генерируется ошибка времени компиляции.

- В методе, который объявлен как возвращающий значение (§8.4).
- В лямбда-выражении.

Выражение *Expression* должно означать переменную или значение, иначе генерируется ошибка времени компиляции.

Когда инструкция `return` с *Expression* находится в объявлении метода, *Expression* должно быть присваиваемым (§5.2) объявленному типу результата метода, иначе генерируется ошибка времени компиляции.

Инструкция `return` с выражением *Expression* пытается передать управление коду, вызвавшему метод или лямбда-выражение, содержащее эту инструкцию; значение *Expression* становится значением вызова метода. Говоря более строго, выполнение такой инструкции `return` сначала вычисляет *Expression*. Если вычисление *Expression* завершается преждевременно по некоторой причине, инструкция `return` завершается преждевременно по той же самой причине. Если вычисление *Expression* завершается нормально, давая значение *V*, то инструкция `return` завершается преждевременно по той причине, что она является возвратом значения *V*.

Если выражение имеет тип `float` и не является FP-строгим (§15.4), то это значение может быть элементом либо набора значений float, либо расширенного набора значений float (§4.2.3). Если выражение имеет тип `double` и не является FP-строгим, то это значение может быть элементом либо набора значений double, либо расширенного набора значений double.

Как можно видеть, инструкция `return` всегда завершается преждевременно.

В приведенном описании говорится "пытается передать управление", а не "передает управление", поскольку если в целевой инструкции имеются инструкции `try` (§14.20), `try`- или `catch`-блоки которых содержат инструкцию `return`, то перед тем как управление будет передано целевой инструкции, выполняются любые конструкции `finally` этих инструкций `try` в порядке от внутренних к внешним. Преждевременное завершение конструкции `finally` может нарушить передачу управления, инициированную инструкцией `return`.

§14.18. Инструкция `throw`

Инструкция `throw` приводит к генерации исключения (§11). Результатом является немедленная передача управления (§11.3), при которой может осуществиться выход из нескольких инструкций и нескольких конструкторов, инициализаторов экземпляров, вычислений статических инициализаторов и инициализаторов экземпляров, а также вызовов методов, пока не будет обнаружена инструкция `try` (§14.20), которая перехватывает сгенерированное значение. Если такая инструкция `try` не обнаружена, выполнение потока (§17), который выполнил инструкцию `throw`, прекращается (§11.3) после вызова метода `uncaughtException` для группы потоков, которой принадлежит данный поток.

ThrowStatement:
`throw` *Expression* `;`

Выражение *Expression* в инструкции `throw` должно означать либо переменную, либо значение ссылочного типа, который является присваиваемым (§5.2) типу `Throwable`, либо нулевой ссылкой, иначе генерируется ошибка времени компиляции.

Ссылочный тип *Expression* всегда представляет собой тип класса (поскольку никакие типы интерфейсов не присваиваемы типу `Throwable`), который не является параметризованным (поскольку подкласс `Throwable` не может быть обобщенным (§8.1.2)).

Должно выполняться как минимум одно из трех приведенных условий, иначе генерируется ошибка времени компиляции.

- Тип *Expression* представляет собой класс непроверяемого исключения (§11.1.1) или тип null (§4.1).
- Инструкция `throw` содержится в блоке `try` инструкции `try` (§14.20) и эта инструкция `try` не может перехватывать исключение типа *Expression*. (В этом случае мы говорим, что сгенерированное значение *перехватывается* инструкцией `try`.)
- Инструкция `throw` содержится в объявлении метода или конструктора и тип *Expression* является присваиваемым (§5.2) как минимум одному типу, перечисленному в конструкции `throws` (§8.4.6, §8.8.5) объявления.

Типы исключений, которые могут генерироваться инструкцией `throw`, указаны в §11.2.2.

Инструкция `throw` сначала вычисляет *Expression*. Затем выполняются следующие действия.

- Если вычисление *Expression* завершается преждевременно по некоторой причине, то инструкция `throw` завершается преждевременно по той же самой причине.
- Если вычисление *Expression* завершается нормально, выдавая значение *V*, не равное `null`, то инструкция `throw` завершается преждевременно по той причине, что она является инструкцией `throw` со значением *V*.
- Если вычисление *Expression* завершается нормально, выдавая значение `null`, то создается экземпляр *V'* класса `NullPointerException`, который и является исключением вместо `null`. Инструкция `throw` при этом завершается преждевременно по той причине, что она представляет собой инструкцию `throw` со значением *V'*.

Как можно видеть, инструкция `throw` всегда завершается преждевременно.

Если есть какие-либо охватывающие инструкции `try` (§14.20), блоки `try` которых содержат инструкцию `throw`, то любые конструкции `finally` этих инструкций `try` выполняются в процессе передачи управления наружу до тех пор, пока исключение не будет перехвачено. Заметим, что преждевременное завершение конструкции `finally` может нарушить передачу управления, инициированную инструкцией `throw`.

Если инструкция `throw` содержится в объявлении метода или лямбда-выражении, но ее значение не перехватывается некоторой инструкцией `try`, которая ее содержит, то вызов метода завершается преждевременно из-за инструкции `throw`.

Если инструкция `throw` содержится в объявлении конструктора, но ее значение не перехватывается некоторой инструкцией `try`, которая ее содержит, то выражение создания экземпляра класса, которое вызывает конструктор, завершается преждевременно из-за инструкции `throw` (§15.9.4).

Если инструкция `throw` содержится в статическом инициализаторе (§8.7), то проверка времени компиляции (§11.2.3) гарантирует, что либо ее значение всегда является непроверяемым исключением, либо ее значение всегда перехватывается некоторой инструкцией `try`, которая ее содержит. Если во время выполнения, несмотря на эту проверку, значение не перехватывается некоторой инструкцией `try`, которая содержит эту инструкцию `throw`, то значение регенерируется, если оно представляет собой экземпляр класса `Error` или одного из его подклассов; в противном случае оно оборачивается в объект исключения `ExceptionInInitializerError`, которое затем генерируется (§12.4.2).

Если инструкция `throw` содержится в инициализаторе экземпляра (§8.6), то проверка времени компиляции (§11.2.3) гарантирует, что либо ее значение всегда является непроверяемым исключением, либо ее значение всегда перехватывается некоторой инструкцией `try`, которая ее содержит, либо тип генерируемого исключения (или одного из его суперклассов) имеется в конструкции `throws` каждого конструктора класса.

По соглашению пользовательские генерируемые типы обычно объявляются как подклассы класса `Exception`, который, в свою очередь, является подклассом класса `Throwable` (§11.1.1).

§14.19. Инструкция `synchronized`

Инструкция `synchronized` захватывает взаимоисключающую блокировку (§17.1) от имени выполняющегося потока, выполняет блок, а затем освобождает блокировку. Во время выполнения потока, владеющего блокировкой, никакой другой поток не может ее захватить.

SynchronizedStatement:
`synchronized` (*Expression*) *Block*

Expression должен иметь ссылочный тип, иначе генерируется ошибка времени компиляции.

Инструкция `synchronized` начинает выполнение с вычисления *Expression*. Затем выполняется следующее.

- Если вычисление *Expression* завершается преждевременно по некоторой причине, инструкция `synchronized` завершается преждевременно по той же самой причине.
- В противном случае, если значение *Expression* равно `null`, генерируется исключение `NullPointerException`.
- В противном случае пусть не-`null`-значение *Expression* равно *V*. Выполняющийся поток блокирует монитор, связанный с *V*. Затем выполняется *Block*, после чего осуществляется выбор.
  - ✦ Если выполнение *Block* завершается нормально, то монитор разблокируется, и инструкция `synchronized` завершается нормально.
  - ✦ Если выполнение *Block* завершается преждевременно по некоторой причине, то монитор разблокируется, и инструкция `synchronized` завершается преждевременно по той же самой причине.

Блокировки, захваченные инструкциями `synchronized`, те же самые, что и блокировки, неявно захватываемые `synchronized`-методами (§8.4.3.6). Один поток может захватывать блокировку неоднократно.

Захват блокировки, связанной с объектом, сам по себе не предотвращает доступ других потоков к полям объекта или вызов ими не-`synchronized`-методов объекта. Другие потоки также могут использовать `synchronized`-методы или инструкции `synchronized` обычным способом для достижения взаимного исключения.

Пример 14.19-1. Инструкция `synchronized`

```
class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
    }
}
```

Вывод этой программы имеет вид

```
made it!
```

Обратите внимание, что эта программа оказалась бы в состоянии взаимоблокировки, если бы один поток не мог многократно блокировать монитор.

§14.20. Инструкция `try`

Инструкция `try` выполняет блок. Если генерируется исключение с некоторым значением и инструкция `try` имеет одну или несколько конструкций `catch`, которые могут его перехватить, то управление передается первой такой конструкции `catch`. Если инструкция `try` имеет конструкцию `finally`, то выполняется другой блок кода, независимо от того, завершился ли блок `try` нормально или

TryStatement:
`try` *Block Catches*
`try` *Block [Catches] Finally*
TryWithResourcesStatement

Catches:
CatchClause {CatchClause}

CatchClause:
`catch` (*CatchFormalParameter*) *Block*

CatchFormalParameter:
{VariableModifier} CatchType VariableDeclaratorId

CatchType:
UnannClassType {| *ClassType}*

Finally:
`finally` *Block*

UnannClassType описывается в §8.3. Ниже для большей ясности повторены продукции из §4.3, §8.3 и §8.4.1.

VariableModifier: одно из
Annotation `final`

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

Block непосредственно после ключевого слова `try` называется `try`-*блоком* инструкции `try`.

Block непосредственно после ключевого слова `finally` называется `finally`-*блоком* инструкции `try`.

Инструкция `try` может иметь конструкции `catch`, именуемые также *обработчиками исключений*.

Конструкция `catch` объявляет ровно один параметр, который называется *параметром исключения*.

Область видимости и затенение параметра исключения рассматриваются в §6.3 и §6.4.

Параметр исключения может указывать свой тип либо как один тип класса, либо как объединение двух или более типов класса (именуемых *альтернативами*). Альтернативы объединения синтаксически разделяются символом `|`.

Конструкция `catch`, параметр исключения которой представляет собой единственный тип класса, называется *конструкцией uni-*`catch`.

Конструкция `catch`, параметр исключения которой представляет собой объединение типов, называется *конструкцией multi-*`catch`.

Каждый тип класса, использованный в обозначении типа параметра исключения, должен быть классом `Throwable` или подклассом класса `Throwable`, иначе генерируется ошибка времени компиляции.

Если в обозначении типа параметра исключения используется переменная типа, генерируется ошибка времени компиляции.

Если объединение типов содержит две альтернативы, D_i и D_j ($i \neq j$), где D_i представляет собой подтип D_j (§4.10.2), генерируется ошибка времени компиляции.

Объявленный тип параметра исключения, который обозначает свой тип с помощью одного типа класса, является этим типом класса.

Объявленный тип параметра исключения, который обозначает свой тип как объединение с альтернативами $D_1 \mid D_2 \mid \ldots \mid D_n$, представляет собой $lub(D_1, D_2, \ldots, D_n)$.

Если параметр исключения конструкции multi-`catch` не объявлен явно как `final`, то он объявлен как `final` неявно.

Если параметру исключения, неявно или явно объявленному как `final`, выполняется присваивание в теле конструкции `catch`, генерируется ошибка времени компиляции.

Параметр исключения конструкции uni-`catch` никогда не объявляется как `final` неявно, но может либо быть объявлен как `final` явно, либо быть по сути финальным (§4.12.4).

> Неявно финальный параметр исключения является `final` в силу его объявления, в то время как по сути финальный параметр является (как бы) `final` в силу особенностей его применения. Параметр исключения конструкции multi-`catch` неявно финальный, так что он никогда не будет левым операндом оператора присваивания, но он *не* рассматривается как по сути финальный.
>
> Если параметр исключения является по сути финальным (в конструкции uni-`catch`) или неявно финальным (в конструкции multi-`catch`), то добавление явного модификатора `final` к объявлению не приведет к новым ошибкам времени компиляции. С другой стороны, если параметр исключения конструкции uni-`catch` явно объявлен как `final`, то удаление модификатора `final` может внести ошибки времени компиляции. Это связано с тем, что, например, на по сути финальный параметр исключения не могут ссылаться локальные классы. С другой стороны, если

ошибки времени компиляции отсутствуют, можно изменить программу так, что параметр исключения окажется переприсвоенным, и фактически перестанет быть по сути финальным.

Типы исключений, которые может генерировать инструкция `try`, определены в §11.2.2.

Отношения между исключениями, сгенерированными блоком `try` инструкции `try` и перехваченными конструкциями `catch` (если таковые имеют место) инструкции `try` определены в §11.2.3.

Обработчики исключений рассматриваются в порядке слева направо: исключение перехватывает наиболее ранняя способная к этому конструкция `catch`, получая в качестве аргумента сгенерированный объект исключения, как указано в §11.3.

Конструкцию multi-`catch` можно рассматривать как последовательность конструкций uni-`catch`. То есть конструкция `catch`, параметр исключения которой обозначает объединение $D_1 | D_2 | ... | D_n$, представляет собой эквивалент последовательности из *n* конструкций `catch`, параметры исключений которых имеют типы классов D_1, D_2, ..., D_n соответственно. В *Block* каждой из *n* конструкций `catch` объявленным типом параметра исключения является $lub(D_1, D_2, ..., D_n)$. Например, исходный текст

```
try {
    ... throws ReflectiveOperationException ...
}
catch (ClassNotFoundException | IllegalAccessException ex) {
    ... тело ...
}
```

семантически эквивалентен следующему.

```
try {
    ... throws ReflectiveOperationException ...
}
catch (final ClassNotFoundException ex1) {
    final ReflectiveOperationException ex = ex1;
    ... тело ...
}
catch (final IllegalAccessException ex2) {
    final ReflectiveOperationException ex = ex2;
    ... тело ...
}
```

Здесь конструкция multi-`catch` с двумя альтернативами преобразована в две разные конструкции `catch`, по одной для каждой альтернативы. От компилятора Java не требуется и не рекомендуется компилировать конструкцию multi-`catch` с помощью указанного дублирования кода, поскольку возможно представление конструкции multi-`catch` в `class`-файле без дублирования.

Конструкция `finally` гарантирует, что блок `finally` выполняется после блока `try` и любого блока `catch`, который может быть выполнен; при этом не имеет значения, как управление покидает блок `try` или `catch`. Обработка блока `finally` достаточно

сложная, так что инструкции `try` с блоком `finally` и без него описываются раздельно (§14.20.1, §14.20.2).

Инструкция `try` может обходиться без конструкций `catch` и `finally`, если это инструкция `try`-*с-ресурсами* (§14.20.3).

§14.20.1. Выполнение `try-catch`

Инструкция `try` без блока `finally` начинает выполнение с блока `try`.

- Если выполнение блока `try` завершается нормально, то никакие дальнейшие действия не предпринимаются и инструкция `try` завершается нормально.
- Если выполнение блока `try` завершается преждевременно из-за генерации исключения со значением *V*, то возможны следующие варианты.
  - ✦ Если тип времени выполнения *V* совместим по присваиванию (§5.2) с классом перехватываемого исключения некоторой конструкции `catch` инструкции `try`, то выбирается первая (крайняя слева) такая конструкция `catch`. Значение *V* присваивается параметру выбранной конструкции `catch`, и выполняется *Block* этой конструкции `catch`, при этом:
    - — если этот блок завершается нормально, то инструкция `try` завершается нормально;
    - — если этот блок завершается преждевременно по некоторой причине, инструкция `try` завершается преждевременно по той же самой причине.
  - ✦ Если тип времени выполнения *V* не совместим по присваиванию ни с одним классом перехватываемого исключения конструкций `catch` инструкции `try`, инструкция `try` завершается преждевременно из-за генерации исключения со значением *V*.
- Если выполнение `try`-блока завершается преждевременно по некоторой причине, инструкция `try` завершается преждевременно по той же самой причине.

Пример 14.20.1-1. Перехват исключения

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
    static void blowUp() throws BlewIt { throw new BlewIt(); }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (RuntimeException r) {
            System.out.println("Caught RuntimeException");
        } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
        }
    }
}
```

Здесь методом `blowup` сгенерировано исключение `BlewIt`. Инструкция `try-catch` в теле `main` имеет две конструкции `catch`. Типом времени выполнения исключения является `BlewIt`, который является не присваиваемым переменной типа `RuntimeException`, а присваиваемым переменной типа `BlewIt`, так что вывод данного примера имеет вид

```
Caught BlewIt
```

§14.20.2. Выполнение try-finally и try-catch-finally

Инструкция `try` с блоком `finally` выполняется, начиная с выполнения блока `try`.

- Если выполнение блока `try` завершается нормально, выполняется блок `finally`, при этом возможны следующие варианты.
  - ✦ Если блок `finally` завершается нормально, инструкция `try` завершается нормально.
  - ✦ Если блок `finally` завершается преждевременно по причине *S*, инструкция `try` завершается преждевременно по причине *S*.
- Если выполнение блока `try` завершается преждевременно по причине генерации исключения со значением *V*, то возможны следующие варианты.
  - ✦ Если тип времени выполнения *V* совместим по присваиванию с классом перехватываемого исключения некоторой конструкции `catch` инструкции `try`, то выбирается первая (крайняя слева) такая конструкция `catch`. Значение *V* присваивается параметру выбранной конструкции `catch`, и выполняется *Block* этой конструкции `catch`. Далее возможны следующие варианты.
    - — Если блок `catch` завершается нормально, то выполняется блок `finally`. Возможны варианты:
      - » если блок `finally` завершается нормально, инструкция `try` завершается нормально;
      - » если блок `finally` завершается преждевременно по некоторой причине, инструкция `try` завершается преждевременно по той же самой причине.
    - — Если блок `catch` завершается преждевременно по причине *R*, выполняется блок `finally`. Возможны варианты:
      - » если блок `finally` завершается нормально, инструкция `try` завершается преждевременно по причине *R*;
      - » если блок `finally` завершается преждевременно по причине *S*, инструкция `try` завершается преждевременно по причине *S* (причина *R* игнорируется).
  - ✦ Если тип времени выполнения *V* не совместим по присваиванию ни с одним классом перехватываемого исключения конструкций `catch` инструкции `try`, то выполняется блок `finally`. Далее возможны следующие варианты.
    - — Если блок `finally` завершается нормально, то инструкция `try` завершается преждевременно из-за генерации исключения *V*.

— Если блок `finally` завершается преждевременно по причине *S*, то инструкция `try` завершается преждевременно по причине *S* (сгенерированное исключение *V* игнорируется и забывается).

- Если выполнение блока `try` завершается преждевременно по некоторой иной причине *R*, выполняется блок `finally`. Далее возможны следующие варианты.
  - ✦ Если блок `finally` завершается нормально, то инструкция `try` завершается преждевременно по причине *R*.
  - ✦ Если блок `finally` завершается преждевременно по причине *S*, инструкция `try` завершается преждевременно по причине *S* (причина *R* игнорируется).

Пример 14.20.2-1. Обработка не перехваченного исключения с `finally`

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
    static void blowUp() throws BlewIt {
        throw new NullPointerException();
    }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
        } finally {
            System.out.println("Uncaught Exception");
        }
    }
}
```

Вывод этой программы имеет вид

```
Uncaught Exception
Exception in thread "main" java.lang.NullPointerException
        at Test.blowUp(Test.java:7)
        at Test.main(Test.java:11)
```

Исключение `NullPointerException` **(являющееся разновидностью** `RuntimeException`**), сгенерированное методом** `blowup`**, не перехватывается инструкцией** `try` **в** `main`**, поскольку** `NullPointerException` **не присваиваемо переменной типа** `BlewIt`**. Это приводит к выполнению конструкции** `finally`**, после чего поток, выполняющий** `main`**, который является единственным потоком тестовой программы, завершается из-за неперехваченного исключения, что обычно приводит к выводу названия исключения и простой трассировки. Спецификация языка не требует вывода трассировки.**

Проблема в случае обязательной трассировки заключается в том, что значение исключения может быть создано в одной точке программы, но само исключение оказывается сгенерированным позднее. Хранение информации трассировки в

> исключении, если оно генерируется на самом деле, оказывается достаточно дорогостоящим (в этом случае трассировку можно выполнить в процессе разворачивания стека). Поэтому требование трассировки в каждом исключении в спецификации отсутствует.

§14.20.3. `try`-с-ресурсами

Инструкция `try`-с-ресурсами параметризована переменными (известными как *ресурсы*), которые инициализируются перед выполнением блока `try` и закрываются автоматически в порядке, обратном порядку инициализации, после выполнения блока `try`. Конструкции `catch` и `finally` зачастую необходимы при автоматическом закрытии ресурсов.

TryWithResourcesStatement:
    `try` *ResourceSpecification Block [Catches] [Finally]*

ResourceSpecification:
    `(` *ResourceList [;]* `)`

ResourceList:
    Resource {; Resource}

Resource:
    {VariableModifier} UnannType VariableDeclaratorId = Expression

UnannType описывается в §8.3. Ниже для большей ясности повторены продукции из §4.3, §8.3 и §8.4.1.

VariableModifier: одно из
    Annotation `final`

VariableDeclaratorId:
    Identifier [Dims]

Dims:
    {Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

Спецификация ресурса объявляет одну или несколько локальных переменных с выражениями инициализаторов для работы в качестве *ресурсов* для инструкции `try`.

Если спецификация ресурса объявляет две переменные с одним и тем же именем, генерируется ошибка времени компиляции.

Если в качестве модификатора переменной, объявленной в спецификации ресурса, `final` встречается более одного раза, генерируется ошибка времени компиляции.

Ресурс, объявленный в спецификации ресурса, неявно объявлен как `final` (§4.12.4), если он не объявлен таковым явно.

Тип переменной, объявленной в спецификации ресурса, должен быть подтипом `AutoCloseable`, иначе генерируется ошибка времени компиляции.

Область видимости и затенение переменной, объявленной в спецификации ресурса, рассматриваются в §6.3 и §6.4.

Ресурсы инициализируются в порядке слева направо. Если происходит сбой при инициализации ресурса (т.е. если выражение инициализатора генерирует исключение), то все ресурсы, инициализированные к этому моменту инструкцией `try`-с-ресурсами, будут закрыты. Если все ресурсы инициализированы успешно, `try`-блок выполняется штатно, после чего все ненулевые ресурсы инструкции `try`-с-ресурсами закрываются.

Ресурсы закрываются в порядке, обратном порядку их инициализации. Ресурс закрывается, только если его инициализация дает ненулевое значение. Исключение при закрытии одного ресурса не препятствует закрытию других ресурсов. Такие исключения *подавляются*, если исключение было сгенерировано ранее инициализатором, `try`-блоком или закрытием ресурса.

Инструкция `try`-с-ресурсами, спецификация ресурсов которой объявляет несколько ресурсов, рассматривается так, как если бы это было несколько инструкций `try`-с-ресурсами, каждая из которых имеет конструкцию спецификации ресурса, объявляющую единственный ресурс. При трансляции инструкции `try`-с-ресурсами с n ресурсами ($n>1$) результат представляет собой инструкцию `try`-с-ресурсами с $n-1$ ресурсами. После n таких трансляций имеется n вложенных `try-catch-finally` инструкций, и вся трансляция завершается.

§14.20.3.1. Базовая инструкция `try`-с-ресурсами

Инструкция `try`-с-ресурсами без конструкций `catch` и `finally` называется *базовой* инструкцией `try`-с-ресурсами.

Смысл базовой инструкции `try`-с-ресурсами

```
try ({VariableModifiers} R Identifier = Expression ...)
    Block
```

понятен из следующей трансляции в объявление локальной переменной и инструкцию `try-catch-finally`.

```
{
    final {VariableModifierNoFinal} R Identifier = Expression;
    Throwable #primaryExc = null;
    try ResourceSpecification_tail
        Block
    catch (Throwable #t) {
        #primaryExc = #t;
        throw #t;
    } finally {
        if (Identifier != null) {
            if (#primaryExc != null) {
                try {
                    Identifier.close();
                } catch (Throwable #suppressedExc) {
                    #primaryExc.addSuppressed(#suppressedExc);
                }
            } else {
```

```
                    Identifier.close();
                }
            }
        }
    }
```

{VariableModifierNoFinal} определяется как *{VariableModifiers}* без `final`, если таковой нетерминал присутствует.

`#t`, `#primaryExc` и `#suppressedExc` являются автоматически генерируемыми идентификаторами, отличными от всех прочих идентификаторов (независимо от того, автоматически ли они сгенерированы), находящихся в области видимости в точке, где встречается инструкция `try`-с-ресурсами.

Если спецификация ресурсов объявляет один ресурс, то *ResourceSpecification_tail* является пустым (а инструкция `try-catch-finally` сама по себе не является инструкцией `try`-с-ресурсами).

Если спецификация ресурсов объявляет $n > 1$ ресурсов, то *ResourceSpecification_tail* состоит из второго, третьего, ..., n-го ресурсов, объявленных в спецификации ресурсов, в том же самом порядке (а инструкция `try-catch-finally` представляет собой инструкцию `try`-с-ресурсами).

Правила доступности и определенного присваивания для базовой инструкции `try`-с-ресурсами неявно определены описанной выше трансляцией.

В базовой инструкции `try`-с-ресурсами, которая управляет одним ресурсом, происходит следующее.

- Если инициализация ресурса завершается преждевременно из-за генерации исключения со значением *V*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*.
- Если инициализация ресурса завершается нормально, а блок `try` завершается преждевременно из-за генерации исключения со значением *V*, то:
  - если автоматическое закрытие ресурса завершается нормально, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*;
  - если автоматическое закрытие ресурса завершается преждевременно из-за генерации исключения со значением *V2*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V* с *V2*, добавленным в список подавленных исключений *V*.
- Если инициализация ресурса завершается нормально и блок `try` завершается нормально, а автоматическое закрытие ресурса завершается преждевременно из-за генерации исключения со значением *V*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*.

В случае базовой инструкции `try`-с-ресурсами, которая управляет несколькими ресурсами, происходит следующее.

- Если инициализация ресурса завершается преждевременно из-за генерации исключения со значением *V*, то:

- ✦ если автоматическое закрытие всех успешно инициализированных ресурсов (возможно, нуль) завершается нормально, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*;
- ✦ если автоматическое закрытие всех успешно инициализированных ресурсов (возможно, нуль) завершается преждевременно из-за генерации исключений со значениями *V1*...*Vn*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*, а все остальные значения *V1*...*Vn* добавляются в список подавленных исключений *V*.

- Если инициализация всех ресурсов завершается нормально, а блок `try` завершается преждевременно из-за генерации исключения со значением *V*, то:
  - ✦ если автоматическое закрытие всех успешно инициализированных ресурсов завершается нормально, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*;
  - ✦ если автоматическое закрытие одного или нескольких инициализированных ресурсов завершается преждевременно из-за генерации исключений со значениями *V1*...*Vn*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*, а все остальные значения *V1*...*Vn* добавляются в список подавленных исключений *V*.
- Если инициализация всех ресурсов завершается нормально и блок `try` завершается нормально, то:
  - ✦ если одно автоматическое закрытие инициализированного ресурса завершается преждевременно из-за генерации исключения со значением *V*, а все прочие автоматические закрытия инициализированных ресурсов завершаются нормально, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V*;
  - ✦ если завершается преждевременно более одного инициализированного ресурса из-за генерации исключений со значениями *V1*...*Vn*, то инструкция `try`-с-ресурсами завершается преждевременно по причине генерации исключения со значением *V1*, а все остальные значения *V2*...*Vn* добавляются в список подавленных исключений *V1* (где *V1* представляет собой исключение крайнего справа ресурса со сбоем закрытия, а *Vn* — исключение крайнего слева ресурса со сбоем закрытия).

§14.20.3.2. Расширенная инструкция `try`-с-ресурсами

Инструкция `try`-с-ресурсами как минимум с одной конструкцией `catch` и/или `finally` называется *расширенной* инструкцией `try`-с-ресурсами.

Значение расширенной инструкции `try`-с-ресурсами

```
try ResourceSpecification
    Block
[Catches]
[Finally]
```

понятно из следующей трансляции в базовую инструкцию `try`-с-ресурсами, вложенную в инструкцию `try-catch` (или `try-finally`, или `try-catch-finally`):

```
try {
    try ResourceSpecification
        Block
}
[Catches]
[Finally]
```

Результатом трансляции является размещение спецификации ресурсов "внутри" инструкции `try`. Это позволяет конструкции `catch` расширенной инструкции `try`-с-ресурсами перехватывать исключения, генерируемые при автоматической инициализации или закрытии ресурса.

Кроме того, все ресурсы будут закрыты (или будут выполнены попытки их закрытия) во время выполнения блока `finally` в соответствии с предназначением ключевого слова `finally`.

§14.21. Недостижимые инструкции

Если инструкция не может быть выполнена из-за *недостижимости*, генерируется ошибка времени компиляции.

> Этот раздел посвящен точному пояснению слова "достижим" (reachable). Идея заключается в том, что должен существовать некоторый путь выполнения от начала конструктора, метода, инициализатора экземпляра или статического инициализатора, который содержит инструкцию к рассматриваемой инструкции (к самому себе). Анализ принимает во внимание структуру инструкций. За исключением специального рассмотрения инструкций `while`, `do` и `for`, выражения условий которых имеют константное значение `true`, значения выражений при анализе потока во внимание не принимаются.
>
> Например, компилятор Java принимает код
>
> ```
> {
>     int n = 5;
>     while (n > 7) k = 2;
> }
> ```
>
> несмотря на то, что значение `n` известно во время компиляции и в принципе во время компиляции понятно, что присваивание значения переменной `k` никогда не будет выполнено.

Правила в этом разделе определяют два технических термина:

- является ли инструкция *достижимой*,
- *может* ли инструкция *завершиться нормально.*

Согласно определениям инструкция может завершиться нормально, только если она достижима.

Достижимая инструкция `break` выполняет *выход из инструкции*, если в инструкции, целевой для инструкции `break`, либо не имеется инструкций `try`, блоки `try` которых содержат инструкцию `break`, либо имеются инструкции `try`, блоки `try` которых со-

держат инструкцию `break`, и все конструкции `finally` этих инструкций `try` могут завершиться нормально.

> Это определение основано на логике вокруг "попытки передать управление" из §14.15.

Инструкция `continue` *продолжает выполнение инструкции* `do`, если в инструкции `do` либо не имеется инструкций `try`, блоки `try` которых содержат инструкцию `continue`, либо имеются инструкции `try`, блоки `try` которых содержат инструкцию `continue`, и все конструкции `finally` этих инструкций `try` могут завершиться нормально.

Правила имеют следующий вид.

- Блок, который является телом конструктора, метода, инициализатором экземпляра или статического инициализатора, достижим.
- Пустой блок, не являющийся блоком `switch`, может завершиться нормально тогда и только тогда, когда он достижим.

  Непустой блок, который не является блоком `switch`, может завершиться нормально тогда и только тогда, когда его последняя инструкция может завершиться нормально.

  Первая инструкция в непустом блоке, который не является блоком `switch`, достижима тогда и только тогда, когда блок достижим.

  Любая иная инструкция *S* в непустом блоке, который не является блоком `switch`, достижима тогда и только тогда, когда инструкция, предшествующая *S*, может завершиться нормально.
- Инструкция объявления локального класса может завершиться нормально тогда и только тогда, когда она достижима.
- Инструкция объявления локальной переменной может завершиться нормально тогда и только тогда, когда она достижима.
- Пустая инструкция может завершиться нормально тогда и только тогда, когда она достижима.
- Помеченная инструкция может завершиться нормально, если выполнено по крайней мере одно из следующих условий:
  - ✦ содержащаяся инструкция может завершиться нормально;
  - ✦ имеется достижимая инструкция `break`, которая осуществляет выход из помеченной инструкции.

  Содержащаяся в ней инструкция достижима тогда и только тогда, когда достижима помеченная инструкция.
- Инструкция выражения может завершиться нормально тогда и только тогда, когда она достижима.
- Инструкция `if-then` может завершиться нормально тогда и только тогда, когда она достижима.

  Инструкция `then` достижима тогда и только тогда, когда достижима инструкция `if-then`.

Инструкция `if-then-else` может завершиться нормально тогда и только тогда, когда может нормально завершиться инструкция `then` или когда может нормально завершиться инструкция `else`.

Инструкция `then` достижима тогда и только тогда, когда достижима инструкция `if-then-else`.

Инструкция `else` достижима тогда и только тогда, когда достижима инструкция `if-then-else`.

> Такая обработка инструкции `if` независимо от наличия части `else`, не совсем обычна. По этой причине она рассматривается отдельно в конце данного раздела.

- Инструкция `assert` может завершиться нормально тогда и только тогда, когда она достижима.
- Инструкция `switch` может завершиться нормально тогда и только тогда, когда выполняется как минимум одно из условий:
  - ✦ блок `switch` пуст или содержит только метки `switch`;
  - ✦ последняя инструкция в блоке `switch` может завершиться нормально;
  - ✦ имеется как минимум одна метка `switch` после последней группы инструкции блока `switch`;
  - ✦ блок `switch` не содержит метку `default`;
  - ✦ имеется достижимая инструкция `break`, которая осуществляет выход из инструкции `switch`.
- Блок `switch` достижим тогда и только тогда, когда инструкция `switch` достижима.
- Инструкция в блоке `switch` достижима тогда и только тогда, когда ее инструкция `switch` достижима и выполняется как минимум одно из условий:
  - ✦ она имеет метку `case` или `default`;
  - ✦ имеется предшествующая ей в блоке `switch` инструкция, и эта предшествующая инструкция может завершиться нормально.
- Инструкция `while` может завершиться нормально тогда и только тогда, когда выполняется как минимум одно из условий:
  - ✦ инструкция `while` достижима и выражение условия не является константным выражением (§15.28) со значением `true`;
  - ✦ имеется достижимая инструкция `break`, которая осуществляет выход из инструкции `while`.

  Содержащаяся внутри инструкция достижима тогда и только тогда, когда достижима инструкция `while` и выражение условия не является константным выражением со значением `false`.
- Инструкция `do` может завершиться нормально тогда и только тогда, когда выполняется как минимум одно из условий:
  - ✦ содержащаяся инструкция может завершиться нормально и выражение условия не является константным выражением (§15.28) со значением `true`;

- инструкция `do` содержит достижимую инструкцию `continue` без метки и инструкция `do` представляет собой наиболее глубоко вложенную инструкцию `while`, `do` или `for`, которая содержит инструкцию `continue`, и инструкция `continue` продолжает выполнение этой инструкции `do`, и выражение условия не является константным выражением со значением `true`;
- инструкция `do` содержит достижимую инструкцию `continue` с меткой *L* и инструкция `do` имеет метку *L*, и инструкция `continue` продолжает выполнение этой инструкции `do`, и выражение условия не является константным выражением со значением `true`;
- имеется достижимая инструкция `break`, которая осуществляет выход из инструкции `do`.

Содержащаяся инструкция достижима тогда и только тогда, когда достижима инструкция `do`.

- Базовая инструкция `for` может завершиться нормально тогда и только тогда, когда выполняется как минимум одно из следующих условий:
  - инструкция `for` достижима, имеет выражение условия, и это выражение условия не является константным выражением (§15.28) со значением `true`;
  - имеется достижимая инструкция `break`, которая осуществляет выход из инструкции `for`.

  Содержащаяся в инструкции `for` инструкция достижима тогда и только тогда, когда достижима инструкция `for` и выражение условия не является константным выражением со значением `false`.
- Расширенная инструкция `for` может завершиться нормально тогда и только тогда, когда она достижима.
- Инструкции `break`, `continue`, `return` и `throw` не могут завершиться нормально.
- Инструкция `synchronized` может завершиться нормально тогда и только тогда, когда содержащаяся инструкция может завершиться нормально.

  Содержащаяся инструкция достижима тогда и только тогда, когда достижима инструкция `synchronized`.
- Инструкция `try` может завершиться нормально тогда и только тогда, когда выполняются одновременно два условия:
  - блок `try` может завершиться нормально или некоторый блок `catch` может завершиться нормально;
  - если инструкция `try` имеет блок `finally`, то блок `finally` может завершиться нормально.
- Блок `try` достижим тогда и только тогда, когда достижима инструкция `try`.
- Блок `catch` *C* достижим тогда и только тогда, когда выполняются одновременно два следующих условия.
  - Либо тип параметра *C* представляет собой тип непроверяемого исключения, `Exception` или суперкласс `Exception`; либо некоторое выражение или ин-

струкция `throw` в блоке `try` является достижимой и может генерировать проверяемое исключение, тип которого присваиваем типу параметру *C*. (Выражение достижимо тогда и только тогда, когда достижима наиболее глубоко вложенная инструкция, содержащая его.)

Нормальное и преждевременное завершения выражений описаны в §15.6.

✦ В инструкции `try` не имеется более раннего блока `catch` *A*, такого, что тип параметра *C* совпадает с (или является) подклассом типа параметра *A*.

- *Block* блока `catch` достижим тогда и только тогда, когда достижим этот блок `catch`.
- Если имеется блок `finally`, он достижим тогда и только тогда, когда инструкция `try` достижима.

Можно ожидать, что инструкция `if` должна обрабатываться следующим образом.

- Инструкция `if-then` может завершиться нормально тогда и только тогда, когда выполняется как минимум одно из следующих условий.
  - — Инструкция `if-then` достижима, а ее выражение условия не является константным выражением со значением `true`.
  - — Инструкция `then` может завершиться нормально.

  Инструкция `then` достижима тогда и только тогда, когда инструкция `if-then` достижима и выражение условия не является константным выражением со значением `false`.
- Инструкция `if-then-else` может завершиться нормально тогда и только тогда, когда инструкция `then` может завершиться нормально или инструкция `else` может завершиться нормально.

  Инструкция `then` достижима тогда и только тогда, когда инструкция `if-then-else` достижима и выражение условия не является константным выражением со значением `false`.

  Инструкция `else` достижима тогда и только тогда, когда инструкция `if-then-else` достижима и выражение условия не является константным выражением со значением `true`.

Этот подход представляется согласующимся с рассмотрением других управляющих структур. Однако для того, чтобы позволить использовать данные инструкции для "условной компиляции", используются несколько отличающиеся правила.

В качестве примера инструкция

```
while (false) { x=3; }
```

приводит к ошибке времени компиляции, поскольку инструкция `x = 3;` недостижима; однако практически идентичный исходный текст

```
if (false) { x=3; }
```

не приводит к ошибке времени компиляции. Оптимизирующий компилятор может обнаружить, что инструкция `x=3;` никогда не будет выполняться, и ее код можно удалить из генерируемого `class`-файла, но инструкция `x=3;` не рассматривается как "недостижимая" в рассматриваемом здесь техническом смысле.

Основание для такой отличной от других ситуаций обработки заключается в том, чтобы позволить программисту определять “переменные-флаги”, как в рассмотренном ниже случае.

```
static final boolean DEBUG = false;
```

После этого можно записать код наподобие

```
if (DEBUG) { x=3; }
```

Идея заключается в том, что должна быть возможность изменить значение `DEBUG` с `false` на `true` или с `true` на `false`, и, если не было внесено иных изменений, исходный текст должен продолжать корректно компилироваться.

Эта возможность “условной компиляции” не связана с бинарной совместимостью (§13). Если множество классов, которые используют такую “переменную-флаг”, компилируются и условный код опускается, распространения только новой версии класса или интерфейса, который содержит определение флага, недостаточно.

Классы, которые используют флаг, не увидят его новое значение, так что их поведение может оказаться удивительным, но ошибки `LinkageError` при этом не будет. Таким образом, изменение значения флага бинарно совместимо с уже существующими бинарными файлами, но не совместимо функционально.

ГЛАВА 15

Выражения

Основная работа в программе выполняется путем вычисления выражений либо для осуществления побочных действий, таких как присваивание переменной, либо для получения значений, которые могут использоваться в качестве аргументов или операндов бо́льших выражений, либо для воздействия на последовательность выполнения в инструкциях, а также для любых сочетаний перечисленных целей.

В этой главе рассматриваются смысл выражений и правила их вычислений.

§15.1. Вычисления, обозначения и результаты

Когда *вычисляется* (*выполняется*) выражение программы, результат означает одно из трех:

- переменную (§4.12) (в языке программирования C это было бы названо *lvalue*);
- значение (§4.2, §4.3);
- ничего (выражение называется пустым (void)).

Если выражение обозначает переменную, а значение требуется для использования в дальнейших вычислениях, то используется значение этой переменной. В данном контексте, если выражение обозначает переменную или значение, мы говорим просто о *значении* выражения.

Преобразование набора значений (§5.1.13) применимо к результату любого выражения, производящего значение, включая ситуации использования значения переменной типа `float` или `double`.

Выражение не обозначает ничего тогда и только тогда, когда оно представляет собой вызов метода (§15.12), в котором вызываемый метод не возвращает значения, т.е. метода, объявленного как `void` (§8.4). Такое выражение может быть использовано только как инструкция выражения (§14.8), поскольку любой другой контекст, в котором может появиться выражение, требует, чтобы выражение что-то описывало. Инструкция выражения, которая представляет собой вызов метода, может вызывать и метод, который возвращает значение; в этом случае значение, возвращаемое методом, просто удаляется.

Вычисление выражения может давать побочные эффекты, потому что выражения могут содержать присваивания, операторы инкремента и декремента и вызовы методов.

Каждое выражение находится:

- либо в объявлении некоторого типа (класса или интерфейса): в инициализаторе поля, статическом инициализаторе, объявлении конструктора, объявлении метода или в аннотации;
- либо в аннотации объявления пакета или объявлении типа верхнего уровня.

§15.2. Виды выражений

Выражения можно разделить на несколько синтаксических разновидностей.

- Имена выражений (§6.5.6).
- Первичные выражения (§15.8–§15.13).
- Выражения с унарным оператором (§15.14–§15.16).
- Выражения с бинарным оператором (§15.17–§15.24 и §15.26).
- Выражения с тернарным оператором (§15.25).
- Лямбда-выражения (§15.27).

Приоритет операторов обеспечивается иерархией продукций грамматики. Оператор с наименьшим приоритетом — стрелка (->) лямбда-выражения, за которой следует оператор присваивания. Таким образом, все выражения синтаксически включены в нетерминалы *LambdaExpression* и *AssignmentExpression*.

Expression:
LambdaExpression
AssignmentExpression

Когда некоторые выражения находятся в определенных контекстах, они рассматриваются как *поливыражения* (poly expressions). Поливыражениями могут быть следующие виды выражений.

- Выражения в скобках (§15.8.5).
- Выражения создания экземпляров классов (§15.9).
- Выражения вызовов методов (§15.12).
- Выражения ссылок на методы (§15.13).
- Условные выражения (§15.25).
- Лямбда-выражения (§15.27).

Правила, определяющие, является ли выражение одного из перечисленных видов поливыражением, приведены в отдельных разделах, которые рассматривают эти виды выражений.

Выражения, которые не являются поливыражениями, называются *автономными выражениями* (standalone expressions). Автономные выражения представляют собой выражения приведенных выше видов, когда выяснено, что они не являются поливыражениями, а также все выражения всех иных видов. Считается, что выражения всех других видов имеют *автономный вид*.

Некоторые выражения имеют значение, которое может быть определено во время компиляции. Это — *константные выражения* (§15.28).

§15.3. Тип выражения

Если выражение обозначает переменную или значение, то выражение имеет известный во время компиляции тип. Тип автономного выражения можно определить из одного лишь содержания выражения; на тип же поливыражения может влиять целевой тип выражения (§5). Правила определения типа выражения поясняются ниже отдельно для каждого вида выражений.

Значение выражения совместимо по присваиванию (§5.2) с типом выражения, если только не произошло загрязнение кучи (§4.12.2).

Аналогично значение, хранящееся в переменной, всегда совместимо с типом переменной, если только не произошло загрязнение кучи.

Другими словами, значение выражения, тип которого — *T*, всегда годится для присваивания переменной типа *T*.

Заметим, что выражение, типом которого является тип класса *F*, объявленный как `final`, гарантированно имеет значение, представляющее собой либо нулевую ссылку, либо объект, классом которого является *F*, поскольку объявленные как `final` типы не имеют подклассов.

§15.4. FP-строгие выражения

Если выражение имеет тип `float` или `double`, то встает вопрос о том, к какому набору значений (§4.2.3) принадлежит значение выражения. Это регулируется правилами преобразования наборов значений (§5.1.13); эти правила, в свою очередь, зависят от того, является ли выражение *FP-строгим*.

Каждое константное выражение (§15.28) является FP-строгим.

Если выражение не является константным, то рассмотрим все объявления классов, объявления интерфейсов и объявления методов, которые содержат это выражение. Если *некоторое* такое объявление имеет модификатор `strictfp` (§8.1.1.3, §8.4.3.5, §9.1.1.2), то выражение является FP-строгим.

Если классе, интерфейс или метод *X* объявлен как `strictfp`, то *X* и любой класс, интерфейс, метод, конструктор, инициализатор экземпляра, статический инициализатор или инициализатор переменной в *X* является *FP-строгим*.

> Заметим, что значение элемента аннотации (§9.7) всегда FP-строгое, поскольку оно всегда является константным выражением.

Отсюда следует, что выражение не является FP-строгим тогда и только тогда, когда оно не является константным времени компиляции выражением *и* не находится в некотором объявлении, которое имеет модификатор `strictfp`.

В FP-строгом выражении все промежуточные значения должны быть элементами набора значений float или набора значений double, откуда вытекает, что результаты всех FP-строгих выражений должны быть предсказанными арифметикой IEEE 754 над операндами, представленными с использованием одинарного и двойного форматов.

В выражении, которое не является FP-строгим, реализации предоставляется некоторая свобода использования расширенного набора для представления промежуточных

результатов; чистый эффект, грубо говоря, заключается в том, что расчет может дать "правильный ответ" в ситуациях, когда исключительное использование набора значений float или double может привести к переполнению или потере точности.

§15.5. Выражения и проверки времени выполнения

Если тип выражения является примитивным типом, то значение выражения имеет тот же примитивный тип.

Если тип выражения является ссылочным типом, то класс объекта, на который указывает ссылка, не обязательно известен во время компиляции. Более того, не обязательно известно даже, является ли значение ссылкой на объект, а не `null`. Есть несколько мест в языке программирования Java, где фактический класс объекта влияет на выполнение программы таким образом, который нельзя вывести из типа выражения. Вот эти места.

- Вызов метода (§15.12). Конкретный метод, использованный в вызове `o.m(...)`, выбирается на основе методов, которые являются частью класса или интерфейса, который является типом `o`. В случае методов экземпляров класс объекта, на который ссылается значение времени выполнения `o`, участвует в операции потому, что подкласс может перекрывать конкретный метод, уже объявленный в родительском классе, так что вызывается именно этот перекрывающий метод. (Перекрывающий метод может как выбрать, так и не выбрать дальнейший вызов исходного перекрытого метода `m`.)
- Оператор `instanceof` (§15.20.2). Выражение, тип которого представляет собой ссылочный тип, может быть проверено с использованием `instanceof`, что позволяет выяснить, является ли объект, на который ссылается значение выражения времени выполнения, совместимым по присваиванию (§5.2) с некоторым другим ссылочным типом.
- Приведение (§5.5, §15.16). Класс объекта, на который ссылается значение времени выполнения операнда выражения, может быть не совместимым с типом, указанным в приведении. В случае ссылочных типов может потребоваться проверка времени выполнения, генерирующая исключение, если во время выполнения обнаруживается, что класс объекта, на который указывает ссылка, не совместим по присваиванию (§5.2) с целевым типом.
- Присваивание компоненту массива ссылочного типа (§10.5, §15.13, §15.26.1). Правила проверки типов позволяют рассматривать тип массива `S[]` как подтип `T[]`, если `S` является подтипом `T`, но это требует проверки времени выполнения для присваивания компоненту массива, аналогичной проверке для приведения.
- Обработка исключений (§14.20). Исключение перехватывается конструкцией `catch`, только если класс объекта сгенерированного исключения представляет собой `instanceof` типа формального параметра конструкции `catch`.

Ситуации, когда класс объекта не является статически известным, могут приводить к ошибкам времени выполнения.

Кроме того, имеются ситуации, когда статически известный тип может оказаться неточным во время выполнения. Такие ситуации могут возникать в программе, которая

порождает предупреждения времени компиляции о непроверенных типах. Такие предупреждения генерируются в ответ на операции, безопасность которых нельзя гарантировать статически и которые не могут быть немедленно подвергнуты динамической проверке, поскольку включают недоступные во время выполнения типы (§4.7). В результате более поздняя динамическая проверка в ходе выполнения программы может обнаружить несоответствия и привести к ошибкам типа времени выполнения.

Ошибки типов времени выполнения могут возникать только в следующих ситуациях.

- В приведении, когда фактический класс объекта, на который ссылается значение операнда выражения, не совместим с целевым типом, указанным в операторе приведения (§5.5, §15.16); в этом случае генерируется исключение `ClassCastException`.
- В автоматически генерируемом приведении, введенном для гарантии корректности операции с недоступным во время выполнения типом (§4.7).
- В присваивании компоненту массива ссылочного типа, когда фактический класс объекта, на который указывает присваиваемое значение, не совместим по присваиванию с фактическим типом времени выполнения компонента массива (§10.5, §15.13, §15.26.1); в этом случае генерируется исключение `ArrayStoreException`.
- Когда исключение не перехвачено ни одной конструкцией `catch` инструкции `try` (§14.20); в этом случае поток управления, в котором произошла генерация исключения, сначала пытается вызвать обработчик неперехваченного исключения (§11.3), а затем прекращает работу.

§15.6. Нормальное и преждевременное завершение вычисления

Каждое выражение имеет нормальный режим вычисления, в котором выполняются определенные шаги вычислений. В следующих разделах описывается нормальный режим вычислений каждого вида выражений.

Если все шаги выполняются без генерации исключений, выражение *завершается нормально*.

Если же при вычислении выражения генерируется исключение, то выражение *завершается преждевременно*. Преждевременное завершение всегда имеет связанную с ним причину, которая всегда выполняет инструкцию `throw` с заданным значением.

Исключения времени выполнения генерируются предопределенными операторами следующим образом.

- Выражение создания экземпляра (§15.9.4), выражение создания массива (§15.10.2), выражение ссылки на метод (§15.13.3), выражение инициализатора массива (§10.6), выражение оператора конкатенации строк (§15.18.1) и лямбда-выражение (§15.27.4) в случае недостаточного количества памяти генерируют исключение `OutOfMemoryError`.
- Выражение создания массива (§15.10.2) в случае, когда любое из выражений размерности оказывается меньше нуля, генерирует исключение `NegativeArraySizeException`.
- Выражение доступа к массиву (§15.10.4) в случае, если значение ссылки на массив имеет значение `null`, генерирует исключение `NullPointerException`.

- Выражение доступа к массиву (§15.10.4) в случае, когда выражение индекса массива отрицательно или не меньше значения `length` массива, генерирует исключение `ArrayIndexOutOfBoundsException`.
- Выражение доступа к полю (§15.11) в случае, когда выражение ссылки на объект имеет значение `null`, генерирует исключение `NullPointerException`.
- Выражение вызова метода (§15.12), вызывающее метод экземпляра в случае нулевой целевой ссылки, генерирует исключение `NullPointerException`.
- Выражение приведения (§15.16) генерирует исключение `ClassCastException` в случае, когда приведение во время выполнения недопустимо.
- Оператор целочисленного деления (§15.17.2) или получения целочисленного остатка (§15.17.3) генерирует исключение `ArithmeticException` в случае, когда значение выражения правого операнда равно нулю.
- Присваивание компоненту массива ссылочного типа (§15.26.1), выражение вызова метода (§15.12), префиксный или постфиксный оператор инкремента (§15.14.2, §15.15.1) или декремента (§15.14.3, §15.15.2) могут генерировать исключение `OutOfMemoryError` как результат преобразования упаковки (§5.1.7).
- Присваивание компоненту массива ссылочного типа (§15.26.1) генерирует исключение `ArrayStoreException` в случае, когда присваиваемое значение не совместимо с типом компонента массива (§10.5).

Выражение вызова метода может также привести к генерации исключения, если происходит исключение, которое приводит к преждевременному завершению тела метода.

Выражение создания экземпляра класса также может привести к генерации исключения, если происходит исключение, которое приводит к преждевременному завершению конструктора.

В процессе вычисления выражения могут также иметь место различные ошибки связывания и виртуальной машины. В силу природы таких ошибок их трудно предсказывать и обрабатывать.

Если произошло исключение, то вычисление одного или нескольких выражений может завершиться до завершения всех шагов нормального режима вычисления; такие выражения называются завершившимися преждевременно.

Если вычисление выражения требует вычисления подвыражения, то преждевременное завершение подвыражения всегда приводит к немедленному преждевременному завершению самого выражения по той же причине, и все последующие шаги нормального режима вычисления не выполняются.

Термины “завершается нормально” и “завершается преждевременно” применимы также к выполнению инструкций (§14.1). Инструкция может завершиться преждевременно по многим причинам, а не только из-за генерации исключения.

§15.7. Порядок вычисления

Язык программирования Java гарантирует, что операнды операторов вычисляются в определенном *порядке вычисления*, а именно — слева направо.

Рекомендуется не полагаться на данную спецификацию. Обычно код яснее, когда каждое выражение содержит не более одного побочного действия в качестве внешней операции и когда код не зависит от того, какое исключение возникает как следствие вычисления выражения слева направо.

§15.7.1. Левый операнд вычисляется первым

Левый операнд бинарного оператора полностью вычисляется до того, как будет вычислена любая часть правого операнда.

Если оператор представляет собой оператор составного присваивания (§15.26.2), то вычисление левого операнда включает как запоминание переменной, описываемой левой частью, так и извлечение и сохранение значения этой переменной для использования в соответствующей бинарной операции.

Если вычисление левого операнда бинарного оператора завершается преждевременно, никакая часть правого операнда не вычисляется.

Пример 15.7.1-1. Левый операнд вычисляется первым

В приведенной далее программе оператор * имеет левый операнд, содержащий присваивание переменной, а правый операнд содержит ссылку на эту же переменную. Значение, получаемое по ссылке, будет отражать тот факт, что первым выполняется присваивание.

```
class Test1 {
    public static void main(String[] args) {
        int i = 2;
        int j = (i=3) * i;
        System.out.println(j);
    }
}
```

Вывод этой программы имеет вид

```
9
```

Вычисление оператором * значения 6 вместо 9 не разрешается.

Пример 15.7.1-2. Неявный левый операнд в операторе составного присваивания

В приведенной далее программе две инструкции присваивания выполняют выборку и запоминание значения левого операнда, равного 9, перед тем как будет вычислен правый операнд оператора сложения, и в этой точке переменная устанавливается равной 3.

```
class Test2 {
    public static void main(String[] args) {
        int a = 9;
        a += (a = 3);     // Первый пример
        System.out.println(a);
        int b = 9;
        b = b + (b = 3); // Второй пример
        System.out.println(b);
```

```
        }
    }
```

Вывод этой программы имеет вид

```
12
12
```

Не разрешается, чтобы любое из присваиваний (составное для `a`, простое для `b`) давало результат 6.

Смотрите также пример в §15.26.2.

Пример 15.7.1-3. Преждевременное завершение вычисления левого операнда

```
class Test3 {
    public static void main(String[] args) {
        int j = 1;
        try {
            int i = forgetIt() / (j = 2);
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Now j = " + j);
        }
    }
    static int forgetIt() throws Exception {
        throw new Exception("I'm outta here!");
    }
}
```

Вывод этой программы имеет вид

```
java.lang.Exception: I'm outta here!
Now j = 1
```

Иначе говоря, левый операнд `forgetIt()` оператора `/` генерирует исключение до вычисления правого операнда и встроенного в него присваивания значения 2 переменной `j`.

§15.7.2. Вычисление операндов до операции

Язык программирования Java гарантирует, что все операнды оператора (за исключением условных операторов `&&`, `||` и `? :`) полностью вычисляются до того, как будет выполнена любая часть самого оператора.

Если бинарный оператор представляет собой целочисленное деление `/` (§15.17.2) или получение целочисленного остатка `%` (§15.17.3), то может быть сгенерировано исключение `ArithmeticException`, но это исключение генерируется только после того, как будут вычислены оба операнда бинарного оператора и только если эти вычисления завершатся нормально.

Пример 15.7.2-1. Вычисление операндов до операции

```
class Test {
    public static void main(String[] args) {
```

```
        int divisor = 0;
        try {
            int i = 1 / (divisor * loseBig());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int loseBig() throws Exception {
        throw new Exception("Shuffle off to Buffalo!");
    }
}
```

Вывод этой программы имеет вид

```
java.lang.Exception: Shuffle off to Buffalo!
```

но не

```
java.lang.ArithmeticException: / by zero
```

поскольку никакая часть операции деления, включая генерацию исключения, сигнализирующего о делении на нуль, не может произойти до того, как завершится вызов `loseBig`, несмотря на то, что реализация компилятора может быть способна обнаружить или сделать вывод о том, что результатом операции деления, определенно, будет генерация исключения “деление на нуль”.

§15.7.3. Вычисления со скобками и приоритеты

Язык программирования Java соблюдает порядок вычислений, явно указанный с помощью скобок, и неявно — с помощью приоритета операторов.

> Реализация языка программирования не может использовать алгебраические тождества, такие как закон ассоциативности, чтобы переписать выражение в более удобном с точки зрения вычислений порядке, если только не доказано, что замена выражения эквивалентна с точки зрения как вычисляемого значения, так и побочных действий, даже при наличии нескольких потоков вычислений (с использованием поточной модели вычисления из §17), для всех возможных значений, которые могут участвовать в вычислениях.

В случае вычислений с плавающей точкой это правило применимо также к значениям “бесконечность” и “не-число” (NaN).

> Например, `!(x<y)` нельзя переписать как `x>=y`, поскольку эти выражения имеют разные значения, если одно из чисел, `x` или `y` (или оба), представляет собой NaN.

В частности, вычисления с плавающей точкой, которые выглядят математически ассоциативными, обычно не являются ассоциативными с вычислительной точки зрения. Такие вычисления нельзя наивно переупорядочивать.

> Например, не является корректным решение компилятора Java переписать `4.0*x*0.5` как `2.0*x`; хотя в данном случае округление и не является проблемой, существуют такие больш*ие* значения `x`, для которых первое выражение дает бесконечность (из-за переполнения), а второе производит конечный результат.

Так что, например, тестовая программа

```
strictfp class Test {
    public static void main(String[] args) {
        double d = 8e+307;
        System.out.println(4.0 * d * 0.5);
        System.out.println(2.0 * d);
    }
}
```

выводит

```
Infinity
1.6e+308
```

поскольку первое выражение приводит к переполнению, а второе — нет.

В отличие от чисел с плавающей точкой целочисленное сложение и умножение *являются* доказанно ассоциативными в языке программирования Java.

Например, выражение `a+b+c`, где `a`, `b` и `c` являются локальными переменными (это упрощение позволяет избежать вопросов, связанных с многопоточностью и переменными, объявленными как `volatile`), всегда дает один и тот же ответ как при вычислении его как `(a+b)+c`, так и как `a+(b+c)`; если выражение `b+c` встречается рядом в коде, интеллектуальный компилятор Java может быть способен использовать это общее подвыражение.

§15.7.4. Списки аргументов вычисляются слева направо

В вызове метода или конструктора или в выражении создания экземпляра класса выражения аргументов могут находиться в скобках, разделенные запятыми. Каждое выражение аргумента должно быть полностью вычислено до вычисления любой части аргумента, находящегося справа от него.

Если вычисление выражения аргумента завершается преждевременно, никакая часть никакого аргумента справа не вычисляется.

Пример 15.7.4-1. Порядок вычисления в вызове метода

```
class Test1 {
    public static void main(String[] args) {
        String s = "going, ";
        print3(s, s, s = "gone");
    }
    static void print3(String a, String b, String c) {
        System.out.println(a + b + c);
    }
}
```

Вывод данной программы имеет вид

```
going, going, gone
```

поскольку присваивание строки `"gone"` переменной `s` происходит после вычисления первых двух аргументов метода `print3`.

Пример 15.7.4-2. Преждевременное завершение выражения аргумента

```
class Test2 {
    static int id;
    public static void main(String[] args) {
        try {
            test(id = 1, oops(), id = 3);
        } catch (Exception e) {
            System.out.println(e + ", id=" + id);
        }
    }
    static int test(int a, int b, int c) {
        return a + b + c;
    }
    static int oops() throws Exception {
        throw new Exception("oops");
    }
}
```

Вывод данной программы имеет вид

```
java.lang.Exception: oops, id=1
```

поскольку присваивание значения 3 **переменной** id **не выполняется.**

§15.7.5. Порядок вычисления других выражений

Порядок вычисления некоторых выражений этими общими правилами не охватывается, поскольку эти выражения могут привести к исключительным условиям в моменты, которые должны быть указаны. Детальные пояснения порядка вычислений следующих выражений можно найти в указанных разделах:

- выражения создания экземпляров класса (§15.9.4);
- выражения создания массивов (§15.10.2);
- выражения доступа к массивам (§15.10.4);
- выражения вызовов методов (§15.12.4);
- выражения ссылки на метод (§15.13.3);
- присваивания, включающие компоненты массива (§15.26);
- лямбда-выражения (§15.27.4).

§15.8. Первичные выражения

Первичные выражения включают большинство простейших видов выражений, из которых строятся все прочие: литералы, создания объектов, доступ к полям, вызовы методов, ссылки на методы, обращение к массивам. Выражения в скобках также синтаксически рассматриваются как первичные выражения.

Primary:
    PrimaryNoNewArray

ArrayCreationExpression

PrimaryNoNewArray:
Literal
ClassLiteral
`this`
TypeName `.` `this`
`(` *Expression* `)`
ClassInstanceCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

Эта часть грамматики языка программирования Java необычна по двум причинам. Во-первых, можно было бы ожидать, что простые имена, такие как имена локальных переменных и параметров методов, должны быть первичными выражениями. По техническим причинам имена группируются вместе с первичными выражениями немного позже, когда вводятся постфиксные выражения (§15.14).

Технические причины должны обеспечивать синтаксический анализ программ Java слева направо с предпросмотром только одного токена. Рассмотрим выражения `(z[3])` и `(z[])`. Первое представляет собой обращение к массиву, заключенное в скобки (§15.13), а второе — начало приведения (§15.16). В момент, когда предпросматриваемым символом является `[`, синтаксический анализ слева направо сворачивает `z` в нетерминал *Name*. В контексте приведения предпочтительнее не сворачивать имя в *Primary*, но если бы нетерминал *Name* был одной из альтернатив *Primary*, то мы не могли бы сказать, следует ли выполнять свертку (т.е. мы не могли бы определить, является ли текущая ситуация обращением к массиву в скобках или приведением) без предпросмотра двух токенов, т.е. токена, следующего за `[`. Представленная здесь грамматика избегает описанной проблемы, поддерживая *Name* и *Primary* отдельно и допуская некоторые другие синтаксические правила (для *ClassInstanceCreationExpression*, *MethodInvocation*, *ArrayAccess*, *PostfixExpression*, но не для *FieldAccess*, поскольку здесь идентификатор используется непосредственно). Эта стратегия эффективно откладывает вопрос о том, следует ли рассматривать *Name* как *Primary*, пока не будет возможности исследовать больший контекст.

Вторая необычность позволяет избежать потенциальной грамматической неоднозначности в выражении "`new int[3][3]`", которое в Java всегда означает единственное создание многомерного массива, но которое без соответствующих грамматических ухищрений могло бы также интерпретироваться как "`(new int[3])[3]`".

Эта неоднозначность устраняется путем разбиения ожидаемого определения *Primary* на *Primary* и *PrimaryNoNewArray*. (Это можно сравнить с разделением *Statement* на *Statement* и *StatementNoShortIf* (§14.5), чтобы избежать проблемы "висячего else".)

§15.8.1. Лексические литералы

Литерал (§3.10) означает фиксированное, неизменное значение.

Для удобства здесь показаны продукции из раздела (§3.10).

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

Тип литерала определяется следующим образом.

- Тип целочисленного литерала (§3.10.1), который заканчивается на `L` или `l`, — `long` (§4.2.1).

  Тип любого иного целочисленного литерала — `int` (§4.2.1).
- Тип литерала с плавающей точкой (§3.10.2), который заканчивается на `F` или `f`, — `float`, а его значение должно быть элементом набора значений float (§4.2.3).

  Тип любого другого литерала с плавающей точкой — `double`, а его значение должно быть элементом набора значений double (§4.2.3).
- Типом булева литерала (§3.10.3) является `boolean` (§4.2.5).
- Типом символьного литерала (§3.10.4) является `char` (§4.2.1).
- Типом строкового литерала (§3.10.5) является `String` (§4.3.3).
- Типом нулевого литерала `null` (§3.10.7) является нулевой тип (§4.1); его значение представляет собой нулевую ссылку.

  Вычисление лексического литерала всегда завершается нормально.

§15.8.2. Литерал класса

Литерал класса представляет собой выражение, состоящее из имени класса, интерфейса, массива или примитивного типа, или из псевдотипа `void`, за которым следуют `'.'` и токен `class`.

ClassLiteral:
TypeName {`[ ]`} `. class`
NumericType {`[ ]`} `. class`
boolean {`[ ]`} `. class`
`void . class`

Типом *C*`.class`, где *C* представляет собой имя класса, интерфейса или тип массива (§4.3), является `Class<`*C*`>`.

Типом *p*`.class`, где *p* представляет собой имя примитивного типа (§4.2), является `Class<`*B*`>`, где *B* представляет собой тип выражения типа *p* после преобразования упаковки (§5.1.7).

Типом `void.class` (§8.4.5) является `Class<Void>`.

Если именованный тип является переменной типа (§4.4) или параметризованным типом (§4.5), или массивом, тип элемента которого представляет собой переменную типа или параметризованный тип, то генерируется ошибка времени компиляции.

Если именованный тип не обозначает тип, который является доступным (§6.6) и находится в области видимости (§6.3) в точке, где появляется литерал класса, генерируется ошибка времени компиляции.

Литерал класса вычисляется в объект `Class` именованного типа (или типа `void`), определенный определяющим загрузчиком класса (§12.2) класса или текущего экземпляра.

§15.8.3. `this`

Ключевое слово `this` может использоваться только в следующих контекстах:

- в теле метода экземпляра или метода по умолчанию (§8.7.4, §9.4.3),
- в теле конструктора класса (§8.8.7),
- в инициализаторе экземпляра класса (§8.6),
- в инициализаторе переменной экземпляра класса (§8.3.2),
- для обозначения параметра-получателя (§8.4.1).

Если оно появляется где-то в другом месте, генерируется ошибка времени компиляции.

Ключевое слово `this` может использоваться в лямбда-выражении, только если оно допустимо в контексте, в котором находится само лямбда-выражение. В противном случае генерируется ошибка времени компиляции.

При использовании в качестве первичного выражения ключевое слово `this` обозначает значение, которое представляет собой ссылку на объект, для которого был вызван метод экземпляра (§15.12), или на создаваемый объект. Значение, обозначаемое ключевым словом `this` в теле лямбда-выражения, — то же самое, что и обозначаемое ключевым словом `this` в окружающем контексте.

> Ключевое слово `this` также используется в специальной инструкции явного вызова конструктора (§8.8.7.1).

Типом `this` является тип класса или интерфейса T, в котором находится это ключевое слово `this`.

> Метод по умолчанию обеспечивает единственную возможность обращения к `this` внутри интерфейса. (Все прочие методы интерфейсов являются либо `abstract`, либо `static`, так что они не предоставляют возможности обращения к `this`.) Таким образом, `this` может иметь тип интерфейса.

Во время выполнения классом фактического объекта, на который указывает `this`, может быть тип T или класс, являющийся подтипом T.

Пример 15.8.3-1. Выражение `this`

```
class IntVector {
    int[] v;
```

```
boolean equals(IntVector other) {
    if (this == other)
        return true;
    if (v.length != other.v.length)
        return false;
    for (int i = 0; i < v.length; i++) {
        if (v[i] != other.v[i]) return false;
    }
    return true;
  }
}
```

Здесь класс `IntVector` реализует метод `equals`, который сравнивает два вектора. Если другой вектор представляет собой тот же объект вектора, что и объект, для которого вызван метод `equals`, то проверка опускает сравнение длин и значений. Метод `equals` реализует эту проверку путем сравнения ссылки на другой объект со значением `this`.

§15.8.4. Квалифицированный `this`

К любому лексически охватывающему экземпляру (§8.1.3) можно обратиться с помощью явно квалифицированного ключевого слова `this`.

Пусть *T* представляет собой тип, описываемый *TypeName*. Пусть *n* является целым числом, таким, что *T* является *n*-м лексически охватывающим объявлением типа класса или интерфейса, в котором находится квалифицированное выражение `this`.

Значение выражения в виде *TypeName*`.this` представляет собой *n*-й лексически охватывающий `this` экземпляр.

Типом выражения является *T*.

Если выражение находится в классе или интерфейсе, который не является внутренним классом класса *T* или самим *T*, генерируется ошибка времени компиляции.

§15.8.5. Выражения в скобках

Выражение в скобках является первичным выражением, тип которого представляет собой тип содержащегося в скобках выражения и значение которого представляет собой значение времени выполнения содержащегося в скобках выражения. Если содержащееся выражение означает переменную, то выражение в скобках также означает эту переменную.

Использование скобок влияет только на *порядок* вычисления, за исключением случаев `(-2147483648)` и `(-9223372036854775808L)`, когда эта запись является корректной, а записи `-(2147483648)` и `-(9223372036854775808L)` — нет.

> Дело в том, что десятичные литералы `2147483648` и `9223372036854775808L` разрешены только как операнды оператора унарного минуса (§3.10.1).

В частности, наличие или отсутствие скобок вокруг выражения (за исключением указанного случая) никоим образом не влияет на

- выбор набора значений (§4.2.3) для выражений типа `float` и `double`;

- то, является ли переменная определенно присвоенной, определенно присвоенной при `true`, определенно присвоенной при `false`, определенно не присвоенной, определенно не присвоенной при `true` или определенно не присвоенной при `false` (§16).

Если заключенное в скобки выражение находится в контексте определенного вида с целевым типом *T* (§5), то оно является выражением, аналогичным появляющемуся в контексте того же вида с целевым типом *T*.

Если содержащееся выражение является поливыражением (§15.2), то заключенное в скобки выражение также является поливыражением. В противном случае это автономное выражение.

§15.9. Выражения создания экземпляра класса

Выражения создания экземпляра класса используются для создания новых объектов, которые являются экземплярами классов.

ClassInstanceCreationExpression:
UnqualifiedClassInstanceCreationExpression
ExpressionName . *UnqualifiedClassInstanceCreationExpression*
Primary . *UnqualifiedClassInstanceCreationExpression*

UnqualifiedClassInstanceCreationExpression:
`new` *[TypeArguments]*
ClassOrInterfaceTypeToInstantiate (*[ArgumentList]*) *[ClassBody]*

ClassOrInterfaceTypeToInstantiate:
{Annotation} Identifier {. {Annotation} Identifier}
[TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
TypeArguments
<>

Далее для удобства приведена продукция из §15.12.

ArgumentList:
Expression {, Expression}

Выражение создания экземпляра класса определяет инстанцируемый класс, за которым могут следовать аргументы типа (§4.5.1) или так называемая *бубна* (<>), если инстанцируемый класс является обобщенным (§8.1.2), после чего следует (возможно, пустой) список фактических значений аргументов конструктора.

Если список аргументов типа класса пуст — имеет вид бубны <>, — то тип аргументов класса выводится. Вполне законно, хотя и не рекомендуется с точки зрения стиля, оставлять пустое пространство между < и > в бубне.

Если конструктор обобщенный (§8.8.4), аргумент типа конструктора может точно так же быть как выведен, так и указан явно. В случае явной передачи аргументы типа конструктора следуют непосредственно за ключевым словом `new`.

Если выражение создания экземпляра класса предоставляет аргументы типа конструктора, но использует <> вместо аргументов типа класса, генерируется ошибка времени компиляции.

> Это правило введено потому, что вывод аргументов типа обобщенного класса может повлиять на ограничения на аргументы типа обобщенного конструктора.

Если *TypeArguments* присутствует сразу же после `new` или сразу перед `(`, то если любой из аргументов типа, использованных в выражении создания экземпляра класса, представляет собой символ подстановки (§4.5.1), генерируется ошибка времени компиляции.

Типы исключений, которые может генерировать выражение создания экземпляра класса, указаны в §11.2.1.

Выражения создания экземпляра класса могут быть двух видов.

- *Неквалифицированные выражения создания экземпляра класса* начинаются с ключевого слова `new`.

  Неквалифицированное выражение создания экземпляра класса может использоваться для создания экземпляра класса независимо от того, является ли этот класс классом верхнего уровня (§7.6), членом (§8.5, §9.5), локальным (§14.3) или анонимным классом (§15.9.5).

- *Квалифицированные выражения создания экземпляра класса* начинаются с выражения *Primary* или *ExpressionName*.

  Квалифицированное выражение создания экземпляра класса позволяет создавать экземпляры внутренних классов-членов и их анонимных подклассов.

Как квалифицированные, так и неквалифицированные выражения создания экземпляров классов могут необязательно заканчиваться телом класса. Такое выражение создания экземпляра класса объявляет *анонимный класс* (§15.9.5) и создает его экземпляр.

Выражение создания экземпляра класса является поливыражением (§15.2), если оно использует бубну в качестве аргументов типа класса, и оно находится в контексте присваивания или контексте вызова (§5.2, §5.3). В противном случае это автономное выражение.

Мы говорим, что класс *инстанцирован*, когда экземпляр класса создан с помощью выражения создания экземпляра класса. Инстанцирование класса включает определение, какой класс инстанцируется (§15.9.1), каковы охватывающие экземпляры (если таковые имеются) вновь созданного экземпляра (§15.9.2), какой конструктор должен быть вызван для создания нового экземпляра (§15.9.3).

§15.9.1. Определение инстанцируемого класса

Если выражение создания экземпляра класса заканчивается телом класса, то инстанцируемый класс является анонимным. В таком случае справедливо следующее.

- Если выражение создания экземпляра является неквалифицированным, то выполняется следующее.

  Нетерминал *ClassOrInterfaceTypeToInstantiate* должен описывать класс, который является доступным, не финальным и не является типом перечисления, или описывать доступный интерфейс. В противном случае генерируется ошибка времени компиляции.

Если *ClassOrInterfaceTypeToInstantiate* заканчивается <>, то генерируется ошибка времени компиляции.

Если *ClassOrInterfaceTypeToInstantiate* завершается *TypeArguments*, то *ClassOrInterfaceTypeToInstantiate* должен обозначать корректно сформированный параметризированный тип (§4.5), иначе генерируется ошибка времени компиляции.

Пусть `T` является типом, который обозначает *ClassOrInterfaceTypeToInstantiate*. Если `T` обозначает класс, то объявлен анонимный непосредственный подкласс класса `T`. Если `T` обозначает интерфейс, то объявлен анонимный непосредственный подкласс класса `Object`, который реализует `T`. В любом случае тело подкласса представляет собой *ClassBody*, заданный выражением создания экземпляра класса.

Инстанцируемый класс является анонимным подклассом.

- Если выражение создания экземпляра является квалифицированным, то выполняется следующее.

  ClassOrInterfaceTypeToInstantiate должен однозначно обозначать внутренний класс, который является доступным, не финальным, не являющимся типом перечисления и являющимся членом типа времени компиляции выражения *Primary* или *Expression Name*. В противном случае генерируется ошибка времени компиляции.

  Если *ClassOrInterfaceTypeToInstantiate* заканчивается <>, то генерируется ошибка времени компиляции.

  Если *ClassOrInterfaceTypeToInstantiate* завершается *TypeArguments*, то *ClassOrInterfaceTypeToInstantiate* должен обозначать корректно сформированный параметризированный тип, иначе генерируется ошибка времени компиляции.

  Пусть `T` является типом, который обозначает *ClassOrInterfaceTypeToInstantiate*. При этом объявлен анонимный непосредственный подкласс класса `T`. Тело подкласса представляет собой *ClassBody*, заданный выражением создания экземпляра класса.

  Инстанцируемый класс является анонимным подклассом.

Если выражение создания экземпляра класса не объявляет анонимный класс, то справедливо следующее.

- Если выражение создания экземпляра класса неквалифицированное, то справедливо следующее.

  Нетерминал *ClassOrInterfaceTypeToInstantiate* должен обозначать класс, который является доступным, не абстрактным и не является типом перечисления. В противном случае генерируется ошибка времени компиляции.

  Если *ClassOrInterfaceTypeToInstantiate* заканчивается <>, но класс, обозначаемый *ClassOrInterfaceTypeToInstantiate*, не обобщенный, то генерируется ошибка времени компиляции.

  Если *ClassOrInterfaceTypeToInstantiate* завершается *TypeArguments*, то *ClassOrInterfaceTypeToInstantiate* должен обозначать корректно сформированный параметризованный класс, иначе генерируется ошибка времени компиляции.

Инстанцируемый класс является классом, обозначаемым *ClassOrInterfaceTypeToInstantiate*.

- Если выражение создания экземпляра класса квалифицированное, то справедливо следующее.

  ClassOrInterfaceTypeToInstantiate должен однозначно обозначать внутренний класс, который является доступным, не абстрактным, не являющимся типом перечисления и являющимся членом типа времени компиляции выражения *Primary* или *ExpressionName*.

  Если *ClassOrInterfaceTypeToInstantiate* заканчивается <>, и класс, обозначаемый *ClassOrInterfaceTypeToInstantiate*, не обобщенный, то генерируется ошибка времени компиляции.

  Если *ClassOrInterfaceTypeToInstantiate* завершается *TypeArguments*, то *ClassOrInterfaceTypeToInstantiate* должен обозначать корректно сформированный параметризованный класс, иначе генерируется ошибка времени компиляции.

  Инстанцируемый класс является классом, обозначаемым *ClassOrInterfaceTypeToInstantiate*.

§15.9.2. Определение охватывающих экземпляров

Пусть `C` — инстанцируемый класс и пусть `i` — создаваемый экземпляр. Если `C` представляет собой внутренний класс, то `i` может иметь *непосредственно охватывающий экземпляр* (§8.1.3), определяемый следующим образом.

- Если `C` представляет собой анонимный класс, то справедливо следующее.
  - ✦ Если выражение создания экземпляра класса находится в статическом контексте, то `i` не имеет непосредственно охватывающего экземпляра.
  - ✦ В противном случае непосредственно охватывающим экземпляром `i` является `this`.
- Если `C` представляет собой локальный класс, то справедливо следующее.
  - ✦ Если `C` находится в статическом контексте, то `i` не имеет непосредственно охватывающего экземпляра.
  - ✦ В противном случае, если выражение создания экземпляра класса находится в статическом контексте, то генерируется ошибка времени компиляции.
  - ✦ В противном случае пусть `O` — непосредственно охватывающий класс для `C`. Пусть *n* — целое число, такое, что `O` является *n*-м лексически охватывающим объявлением типа для класса, в котором находится выражение создания экземпляра класса.

    Непосредственно охватывающим экземпляром `i` является *n*-й лексически охватывающий экземпляр `this` (§8.1.3).
- Если `C` является внутренним классом-членом, то справедливо следующее.
  - ✦ Если выражение создания экземпляра класса является неквалифицированным, то:

— если выражение создания экземпляра класса находится в статическом контексте, генерируется ошибка времени компиляции;

— в противном случае, если `C` является членом охватывающего класса, то пусть `O` — непосредственно охватывающий класс, членом которого является `C`, и пусть *n* — целое число, такое, что `O` является *n*-м лексически охватывающим объявлением типа для класса, в котором находится выражение создания экземпляра класса.

Непосредственно охватывающим экземпляром `i` является *n*-й лексически охватывающий экземпляр `this`;

— в противном случае генерируется ошибка времени компиляции.

✦ Если выражение создания экземпляра класса квалифицированное, то непосредственно охватывающим экземпляром `i` является объект, который представляет собой значение выражения *Primary* или *ExpressionName*.

Если `C` является анонимным классом, а непосредственный суперкласс `S` класса `C` представляет собой внутренний класс, то `i` может иметь *непосредственно охватывающий экземпляр по отношению к* `S`. Он определяется следующим образом.

- Если `S` является локальным классом, то
  - ✦ если `S` находится в статическом контексте, то `i` не имеет непосредственно охватывающего экземпляра по отношению к `S`;
  - ✦ в противном случае, если выражение создания экземпляра класса находится в статическом контексте, генерируется ошибка времени компиляции;
  - ✦ в противном случае пусть `O` — непосредственно охватывающий класс класса `S`. Пусть *n* — целое число, такое, что `O` является *n*-м лексически охватывающим объявлением типа класса, в котором находится выражение создания экземпляра класса.

    Непосредственно охватывающим экземпляром `i` по отношению к `S` является *n*-й лексически охватывающий экземпляр `this`.
- Если `S` является внутренним классом-членом, справедливо следующее.
  - ✦ Если выражение создания экземпляра класса неквалифицированное, то:

    — если выражение создания экземпляра класса находится в статическом контексте, генерируется ошибка времени компиляции;

    — в противном случае, если `S` является членом охватывающего класса, то пусть `O` — непосредственно охватывающий класс, членом которого является `S`, и пусть *n* — целое число, такое, что `O` является *n*-м лексически охватывающим объявлением типа для класса, в котором находится выражение создания экземпляра класса.

    Непосредственно охватывающим экземпляром `i` по отношению к `S` является *n*-й лексически охватывающий экземпляр `this`;

— в противном случае генерируется ошибка времени компиляции.

- ✦ Если выражение создания экземпляра класса квалифицированное, то непосредственно охватывающим экземпляром `i` по отношению к `S` является объект, представляющий собой значение выражения *Primary* или *ExpressionName*.

§15.9.3. Выбор конструктора и его аргументов

Пусть `C` представляет собой инстанцируемый класс. Для создания экземпляра `i` класса `C` во время компиляции с помощью приведенных далее правил выбирается конструктор `C`.

Сначала определяются фактические аргументы вызова конструктора.

- Если `C` является анонимным классом с непосредственным суперклассом `S`, то
  - ✦ если `S` не является внутренним классом или если `S` является локальным классом, который находится в статическом контексте, то аргументы в списке аргументов, если таковые имеются, являются аргументами конструктора в порядке, в котором они находятся в выражении;
  - ✦ в противном случае первым аргументом конструктора является непосредственно охватывающий `i` экземпляр по отношению к `S` (§15.9.2), а за ним следуют аргументы из списка аргументов выражения создания экземпляра класса, если таковые имеются, в порядке, в котором они находятся в выражении.
- Если `C` представляет собой локальный класс или внутренний `private` член-класс, то аргументами конструктора являются аргументы из списка аргументов выражения создания экземпляра класса, если таковые имеются, в том порядке, в котором они находятся в выражении создания экземпляра класса.
- Если `C` представляет собой внутренний член-класс, не объявленный как `private`, то первым аргументом конструктора является непосредственно охватывающий `i` экземпляр (§8.8.1, §15.9.2), а последующие аргументы конструктора представляют собой аргументы из списка аргументов выражения создания экземпляра класса, если таковые имеются, в том порядке, в котором они находятся в выражении создания экземпляра класса.
- В противном случае аргументами конструктора являются аргументы в списке аргументов выражения создания экземпляра класса, если таковые имеются, в порядке, в котором они находятся в выражении.

Затем определяются конструктор `C` и соответствующий тип результата и конструкция `throws`.

- Если выражение создания экземпляра класса использует <> для сокрытия аргументов типа класса, то для целей разрешения перегрузки и вывода аргумента типа определяется список методов $m_1 ... m_n$.

  Пусть $c_1 ... c_n$ представляют собой конструкторы класса `C`. Пусть `#m` — автоматически сгенерированное имя, отличное от имен всех конструкторов и методов в `C`. Для всех j $(1 \le j \le n)$ m_j определяется в терминах c_j следующим образом.

  - ✦ Сначала для инстанцирования типов в c_j определяется подстановка θ_j.

Пусть $F_1...F_p$ — параметры типов C и пусть $G_1...G_q$ — параметры типов (если таковые имеются) для c_j. Пусть $X_1...X_p$ и $Y_1...Y_q$ представляют собой переменные типов с различными именами, которые не находятся в области видимости в теле C.

θ_j представляет собой $[F_1:=X_1, \ldots, F_p:=X_p, G_1:=Y_1, \ldots, G_q:=Y_q]$.

- ✦ Модификаторами m_j являются таковые для c_j.
- ✦ Параметрами типов m_j являются $X_1...X_p$, $Y_1...Y_q$. Граница каждого параметра, если таковая имеется, представляет собой подстановку θ_j, примененную к соответствующей границе параметра в C или c_j.
- ✦ Возвращаемым типом m_j является подстановка θ_j, примененная к $C<F_1,...,F_p>$.
- ✦ Именем m_j является `#m`.
- ✦ (Возможно, пустой) список типов аргументов m_j представляет собой подстановку θ_j, примененную к типам аргументов c_j.
- ✦ (Возможно, пустой) список типов генерируемых исключений m_j представляет собой подстановку θ_j, примененную к типам генерируемых исключений c_j.
- ✦ Тело m_j роли не играет.

Для выбора конструктора мы временно рассматриваем $m_1...m_n$ как члены C. Затем выбирается один из $m_1...m_n$, определяемый аргументами выражения создания экземпляра класса, с использованием описанного в §15.12.2 процесса.

Если единственного наиболее подходящего применимого и доступного метода нет, генерируется ошибка времени компиляции.

В противном случае, когда m_j представляет собой выбранный метод, выбранным конструктором является c_j. Результирующий тип и конструкция `throws` c_j те же, что и возвращаемый тип и конструкция `throws`, определенные для m_j (§15.12.2.6).

- В противном случае выражение создания экземпляра класса не использует <> для сокрытия аргументов типа.

  Пусть T представляет собой тип, обозначаемый C, за которым следуют любые аргументы типа класса в выражении. Процесс, описанный в §15.12.2, модифицированный для обработки конструкторов, используется для выбора одного из конструкторов T и определяет его конструкцию `throws`.

  Как и в случае вызова метода, если нет единственного наиболее подходящего применимого и доступного конструктора, генерируется ошибка времени компиляции.

  В противном случае результирующим типом является T.

Если аргумент выражения создания экземпляра класса не совместим с его целевым типом, получаемым из типа вызова (§15.12.2.6), генерируется ошибка времени компиляции.

Если объявление времени компиляции допускает вызов с переменной арностью (§15.12.2.4), то там, где тип последнего формального параметра типа вызова конструктора представляет собой $F_n[\,]$, если тип, который представляет собой затирание F_n, недоступен в точке вызова, генерируется ошибка времени компиляции.

Типом выражения создания экземпляра класса является результирующий тип конструктора, выбранного, как описано выше.

Обратите внимание, что тип выражения создания экземпляра класса может быть типом анонимного класса; в этом случае вызываемый конструктор представляет собой анонимный конструктор (§15.9.5.1).

§15.9.4. Вычисление времени выполнения выражения создания экземпляра класса

Во время выполнения программы выражение создания экземпляра класса вычисляется следующим образом.

Сначала, если выражение создания экземпляра класса представляет собой выражение создания экземпляра квалифицированного класса, вычисляется квалифицированное первичное выражение. Если результат вычисления квалифицированного выражения равен `null`, генерируется исключение `NullPointerException`, и выражение создания экземпляра класса завершается преждевременно. Если квалифицированное выражение завершается преждевременно, выражение создания экземпляра класса завершается преждевременно по той же причине.

Затем выделяется память для нового экземпляра класса. Если имеющейся памяти для объекта недостаточно, вычисление выражения создания экземпляра класса завершается преждевременно путем генерации исключения `OutOfMemoryError`.

Новый объект содержит новые экземпляры всех полей, объявленных в указанном типе класса и всех его суперклассах. При создании каждого нового поля оно инициализируется своим значением по умолчанию (§4.12.5).

После этого слева направо вычисляются фактические аргументы конструктора. Если любое из вычислений аргумента завершается преждевременно, все выражения аргументов справа от него не вычисляются, и выражение создания экземпляра класса завершается преждевременно по той же причине.

Затем вызывается выбранный конструктор указанного типа класса. В результате его вызова вызывается как минимум по одному конструктору каждого суперкласса типа класса. Этот процесс может управляться с помощью инструкций явного вызова конструкторов (§8.8) и подробно описан в §12.5.

Значение выражения создания экземпляра класса представляет собой ссылку на вновь созданный объект указанного класса. При каждом выполнении этого выражения создается новый объект.

Пример 15.9.4-1. Порядок вычислений и обнаружение нехватки памяти

Если вычисление выражения создания экземпляра класса обнаруживает недостаточность памяти для выполнения операции, генерируется исключение `OutOfMemoryError`. Эта проверка осуществляется до вычислений любого из аргументов.

Например, тестовая программа

```
class List {
    int value;
    List next;
    static List head = new List(0);
    List(int n) { value = n; next = head; head = this; }
```

```
    }
    class Test {
        public static void main(String[] args) {
            int id = 0, oldid = 0;
            try {
                for (;;) {
                    ++id;
                    new List(oldid = id);
                }
            } catch (Error e) {
                List.head = null;
                System.out.println(e.getClass() + ", " +
(oldid==id));
            }
        }
    }
```

выводит

```
class java.lang.OutOfMemoryError, false
```

поскольку условие нехватки памяти обнаруживается до вычисления выражения аргумента `oldid = id`.

Сравните это поведение с трактовкой выражений создания массивов, в которых условие нехватки памяти проверяется после вычисления выражений размерностей (§15.10.2).

§15.9.5. Объявления анонимных классов

Объявление анонимного класса автоматически порождается из выражения создания экземпляра класса компилятором Java.

Анонимный класс не может быть объявлен как `abstract` (§8.1.1.1).

Анонимный класс неявно всегда объявлен как `final` (§8.1.1.2).

Анонимный класс всегда является внутренним классом (§8.1.3); он никогда не бывает `static` (§8.1.1, §8.5.1).

§15.9.5.1. Анонимные конструкторы

Анонимный класс не может иметь явно объявленный конструктор. Вместо этого для анонимного класса неявно объявляется анонимный конструктор. Вид анонимного конструктора анонимного класса *C* с непосредственным суперклассом *S* определяется следующим образом.

- Если *S* не является внутренним классом или если *S* представляет собой локальный класс, находящийся в статическом контексте, то анонимный конструктор имеет один формальный параметр для каждого фактического аргумента выражения создания экземпляра класса, в котором объявлен *C*.

  Фактические аргументы выражения создания экземпляра класса используются для определения конструктора *cs* класса *S* с использованием тех же правил, что и при

вызовах методов (§15.12). Тип каждого формального параметра анонимного конструктора должен быть идентичен соответствующему формальному параметру `cs`.

Тело конструктора состоит из явного вызова конструктора (§8.8.7.1) вида `super(...)`, где фактические аргументы являются формальными параметрами конструктора в порядке, в котором они были объявлены.

- В противном случае первый формальный параметр конструктора `C` представляет значение непосредственно охватывающего экземпляра `i` по отношению к `S` (§15.9.2, §15.9.3). Тип этого параметра представляет собой тип класса, который непосредственно охватывает объявление `S`.

  Конструктор имеет дополнительный формальный параметр для каждого фактического аргумента выражения создания экземпляра класса, которое объявляет анонимный класс. *n*-й формальный параметр `e` соответствует *n* – 1-му фактическому аргументу.

  Фактические параметры выражения создания экземпляра класса используются для определения конструктора `cs` класса `S` с использованием тех же правил, что и для вызовов методов (§15.12). Тип каждого формального параметра анонимного конструктора должен быть идентичен соответствующему формальному параметру `cs`.

  Тело конструктора состоит из явного вызова конструктора (§8.8.7.1) вида `o.super(...)`, где `o` является первым формальным параметром конструктора, а фактические аргументы являются последующими формальными параметрами конструктора в порядке, в котором они были объявлены.

Во всех случаях конструкция `throws` анонимного конструктора должна перечислять все проверяемые исключения, генерируемые инструкцией явного вызова конструктора суперкласса, содержащейся в анонимном конструкторе, и все проверяемые исключения, генерируемые всеми инициализаторами экземпляра или инициализаторами переменных экземпляров анонимного класса.

Обратите внимание, что сигнатура анонимного конструктора может ссылаться на недоступный тип (например, если таковой тип имеется в сигнатуре конструктора суперкласса `cs`). Само по себе это не приводит к каким-то ошибкам времени компиляции или времени выполнения.

§15.10. Выражения создания массивов и доступа к ним

§15.10.1. Выражения создания массивов

Выражение создания массива используется для создания новых массивов (§10).

ArrayCreationExpression:
`new` *PrimitiveType DimExprs [Dims]*
`new` *ClassOrInterfaceType DimExprs [Dims]*
`new` *PrimitiveType Dims ArrayInitializer*
`new` *ClassOrInterfaceType Dims ArrayInitializer*

DimExprs:
DimExpr {DimExpr}

DimExpr:
{Annotation} [*Expression*]
Далее для удобства приведена продукция из §4.3.
Dims:
{Annotation} [ ] *{{Annotation}* [ ]*}*

Выражение создания массива создает объект, который представляет собой новый массив, элементы которого имеют тип, указанный в *PrimitiveType* или *ClassOrInterfaceType*.

Если *ClassOrInterfaceType* не обозначает доступный во время выполнения тип (§4.7), генерируется ошибка времени компиляции. В противном случае *ClassOrInterfaceType* может именовать любой именованный ссылочный тип, даже тип класса `abstract` (§8.1.1.1) или тип интерфейса.

> Из приведенных выше правил вытекает, что тип элемента в выражении создания массива не может быть параметризованным, если только все аргументы типа параметризованного типа не являются неограниченными символами подстановки.

Тип каждого выражения размерности в *DimExpr* должен быть типом, конвертируемым (§5.1.8) в целочисленный тип, иначе генерируется ошибка времени компиляции.

Каждое выражение размерности проходит унарное числовое повышение (§5.6.1). Повышенным типом должен быть `int`, иначе генерируется ошибка времени компиляции.

Типом выражения создания массива является тип массива, который может обозначаться копией выражения создания массива, из которого удалены ключевое слово `new`, все выражения *DimExpr* и инициализатор массива.

> Например, типом выражения создания массива
> ```
> new double[3][3][]
> ```
> является
> ```
> double[][][]
> ```

§15.10.2. Вычисление времени выполнения выражений создания массивов

Во время выполнения вычисление выражения создания массива ведет себя следующим образом.

- Если выражения размерностей отсутствуют, то должен иметься инициализатор массива. Вновь выделенный массив будет инициализирован значениями, предоставленными инициализатором массива, описанным в §10.6. Значение инициализатора массива становится значением выражения создания массива.
- В противном случае инициализатора массива нет, и происходит следующее.
  - ✦ Сначала слева направо вычисляются выражения размерностей. Если вычисление некоторого выражения завершается преждевременно, все выражения справа от него не вычисляются.

- Затем выполняется проверка выражений размерностей. Если значение некоторого из выражений *DimExpr* меньше нуля, генерируется исключение `NegativeArraySizeException`.
- После этого для нового массива выделяется память. Если памяти для нового массива недостаточно, вычисление выражения создания массива завершается преждевременно путем генерации исключения `OutOfMemoryError`.
- Далее, если имеется единственное выражение *DimExpr*, создается одномерный массив указанной длины и каждый компонент массива инициализируется его значением по умолчанию (§4.12.5).
- В противном случае, если имеется *n* выражений *DimExpr*, создание массива выполняется с помощью набора вложенных циклов глубиной *n* – 1 для создания требуемых массивов массивов.

  Многомерные массивы не обязаны быть массивами одной и той же длины на каждом уровне.

Пример 15.10.2-1. Вычисление создания массива

В выражении создания массива с одним или несколькими выражениями размерности каждое выражение размерности полностью вычисляется до любой части любого выражения справа от него. Таким образом, исходный текст

```
class Test1 {
    public static void main(String[] args) {
        int i = 4;
        int ia[][] = new int[i][i=3];
        System.out.println(
            "[" + ia.length + "," + ia[0].length + "]");
    }
}
```

выводит

```
[4,3]
```

поскольку первая размерность вычисляется как равная `4` до второго выражения, устанавливающего значение переменной `i` равным `3`.

Если вычисление выражения размерности завершается преждевременно, никакая часть никакого выражения размерности справа не вычисляется. Таким образом, исходный текст

```
class Test2 {
    public static void main(String[] args) {
        int[][] a = { { 00, 01 }, { 10, 11 } };
        int i = 99;
        try {
            a[val()][i = 1]++;
        } catch (Exception e) {
            System.out.println(e + ", i=" + i);
        }
    }
```

```
    static int val() throws Exception {
        throw new Exception("unimplemented");
    }
}
```

выводит

```
java.lang.Exception: unimplemented, i=99
```

поскольку присваивание, которое устанавливает значение переменной `i` равным `1`, не выполняется.

Пример 15.10.2-2. Создание многомерного массива

Объявление

```
float[][] matrix = new float[3][3];
```

по поведению эквивалентно коду

```
float[][] matrix = new float[3][];
for (int d = 0; d < matrix.length; d++)
    matrix[d] = new float[3];
```

а код

```
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

эквивалентен

```
Age[][][][][] Aquarius = new Age[6][][][][];
for (int d1 = 0; d1 < Aquarius.length; d1++) {
    Aquarius[d1] = new Age[10][][][];
    for (int d2 = 0; d2 < Aquarius[d1].length; d2++) {
        Aquarius[d1][d2] = new Age[8][][];
        for (int d3 = 0; d3 < Aquarius[d1][d2].length; d3++) {
            Aquarius[d1][d2][d3] = new Age[12][];
        }
    }
}
```

Здесь *d*, *d1*, *d2* и *d3* заменяются именами, которые не были объявлены локально. Таким образом, единственное выражение `new` на самом деле создает один массив длиной 6, 6 массивов длиной 10, 6 × 10 = 60 массивов длиной 8 и 6 × 10 × 8 = 480 массивов длиной 12. В этом примере пятое измерение представляет собой массивы, содержащие фактические элементы массива (ссылки на объекты `Age`), инициализированные нулевыми ссылками. Эти массивы могут быть заполнены позже другим кодом, например

```
Age[] Hair = { new Age("quartz"), new Age("topaz") };
Aquarius[1][9][6][9] = Hair;
```

Треугольная матрица может быть создана следующим образом.

```
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
    triang[i] = new float[i+1];
```

Если вычисление выражения создания массива обнаруживает недостаточность памяти для выполнения операции создания массива, генерируется исключение `OutOfMemory`

`Error`. Если выражение создания массива не содержит инициализатор массива, то данная проверка выполняется только после того, как вычисление всех выражений размерности завершается нормально. Если выражение создания массива содержит инициализатор массива, то исключение `OutOfMemoryError` может быть сгенерировано, когда в процессе вычисления выражения инициализатора переменной выделяется память для объекта ссылочного типа или когда выделяется пространство для массива, предназначенного для хранения значений инициализатора (возможно, вложенного) массива.

Пример 15.10.2-3. Вычисление выражения размерности и исключение `OutOfMemoryError`

```
class Test3 {
    public static void main(String[] args) {
        int len = 0, oldlen = 0;
        Object[] a = new Object[0];
        try {
            for (;;) {
                ++len;
                Object[] temp = new Object[oldlen = len];
                temp[0] = a;
                a = temp;
            }
        } catch (Error e) {
            System.out.println(e + ", " + (oldlen==len));
        }
    }
}
```

Вывод данной программы имеет вид

```
java.lang.OutOfMemoryError, true
```

поскольку после вычисления выражения размерности `oldlen = len` обнаруживается нехватка памяти.

Сравните этот пример с выражениями создания экземпляров классов (§15.9), в которых условие нехватки памяти обнаруживается до вычисления выражений аргументов (§15.9.4).

§15.10.3. Выражения обращения к массиву

Выражение обращения к массиву ссылается на переменную, которая представляет собой компонент массива.

ArrayAccess:
ExpressionName [*Expression*]
PrimaryNoNewArray [*Expression*]

Выражение обращения к массиву содержит два подвыражения, *выражение ссылки на массив* (перед левой квадратной скобкой) и *выражение индекса* (в квадратных скобках).

> Обратите внимание, что выражение ссылки на массив может быть именем или любым первичным выражением, которое не является выражением создания массива (§15.10).

Тип выражения ссылки на массив должен быть типом массива (назовем его `T[]`, массив, компоненты которого имеют тип *T*), иначе генерируется ошибка времени компиляции.

Над выражением индекса выполняется унарное числовое повышение (§5.6.1). Повышенный тип должен представлять собой `int`, иначе генерируется ошибка времени компиляции.

Тип выражения доступа к массиву является результатом применения преобразования при фиксации (§5.1.10) к *T*.

Результатом выражения доступа к массиву является переменная типа *T*, а именно — переменная в массиве, выбранная в соответствии со значением выражения индекса.

Эта результирующая переменная, которая является компонентом массива, никогда не рассматривается как `final`, даже если выражение ссылки массива обозначает `final`-переменную.

§15.10.4. Вычисление времени выполнения выражения обращения к массиву

Выражение обращения к массиву вычисляется с помощью следующей процедуры.

- Сначала вычисляется выражение ссылки на массив. Если это вычисление завершается преждевременно, то обращение к массиву завершается преждевременно по той же причине и выражение индекса не вычисляется.
- В противном случае вычисляется выражение индекса. Если это вычисление завершается преждевременно, обращение к массиву завершается преждевременно по той же причине.
- В противном случае, если значение выражения обращения к массиву равно `null`, генерируется исключение `NullPointerException`.
- В противном случае значение выражения обращения к массиву в действительности ссылается на массив. Если значение выражения индекса меньше нуля или не меньше значения `length` массива, генерируется исключение `ArrayIndexOutOfBoundsException`.
- В противном случае результат обращения к массиву представляет собой переменную типа *T* внутри массива, выбранную в соответствии со значением выражения индекса.

Пример 15.10.4-1. Ссылка на массив вычисляется первой

> При обращении к массиву выражение слева от квадратных скобок полностью вычисляется до того, как будет вычислена любая часть выражения в квадратных скобках. Например, в (выглядящем монстрообразно) выражении `a[(a=b)[3]]` выражение `a` полностью вычисляется до выражения `(a=b)[3]`; это означает, что исходное значение `a` выбирается и сохраняется во время вычисления выражения `(a=b)[3]`. К массиву, на который ссылается исходное значение `a`, выполняется

обращение со значением индекса, равным третьему элементу другого массива (возможно, того же самого), на который ссылается переменная `b`, а после вычисления этого индекса — также и переменная `a`.

```
class Test1 {
    public static void main(String[] args) {
        int[] a = { 11, 12, 13, 14 };
        int[] b = { 0, 1, 2, 3 };
        System.out.println(a[(a=b)[3]]);
    }
}
```

Таким образом, вывод приведенной программы имеет вид

```
14
```

поскольку наше монстрообразное выражение вычисляется как `a[b[3]]` или `a[3]`, или `14`.

Пример 15.10.4-2. Преждевременное завершение вычисления ссылки на массив

Если вычисление выражения слева от квадратных скобок завершается преждевременно, никакая часть выражения в квадратных скобках не вычисляется.

```
class Test2 {
    public static void main(String[] args) {
        int index = 1;
        try {
            skedaddle()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
    static int[] skedaddle() throws Exception {
        throw new Exception("Ciao");
    }
}
```

Таким образом, вывод приведенной программы имеет вид

```
java.lang.Exception: Ciao, index=1
```

поскольку присваивание значения `2` переменной `index` никогда не происходит.

Пример 15.10.4-3. Ссылка `null` на массив

Если выражение обращения к массиву дает в качестве ссылки на массив значение `null`, генерируется исключение `NullPointerException` времени выполнения, но только после того, как будут вычислены все части выражения обращения к массиву, и только если эти вычисления завершатся нормально.

```
class Test3 {
    public static void main(String[] args) {
        int index = 1;
        try {
            nada()[index=2]++;
```

```
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
    static int[] nada() { return null; }
}
```

Таким образом, вывод приведенной программы имеет вид

```
java.lang.NullPointerException, index=2
```

поскольку присваивание значения 2 переменной `index` осуществляется до проверки равенства выражения ссылки на массив значению `null`. В качестве другого примера программа

```
class Test4 {
    public static void main(String[] args) {
        int[] a = null;
        try {
            int i = a[vamoose()];
            System.out.println(i);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int vamoose() throws Exception {
        throw new Exception("Twenty-three skidoo!");
    }
}
```

всегда выводит

```
java.lang.Exception: Twenty-three skidoo!
```

Исключение `NullPointerException` никогда не генерируется, поскольку выражение индекса должно быть полностью вычислено до того, как будет вычислена любая часть обращения к выражению, включая проверку на равенство ссылки на массив значению `null`.

§15.11. Выражения обращения к полю

Выражение обращения к полю может обращаться к полю объекта или массива, ссылка на который является либо значением выражения, либо специальным ключевым словом `super`.

FieldAccess:
Primary `.` *Identifier*
`super` `.` *Identifier*
TypeName `.` `super` `.` *Identifier*

Смысл выражения обращения к полю определяется с применением тех же правил, что и для квалифицированных имен (§6.5.6.2), но ограниченных тем фактом, что выражение не может обозначать пакет, тип класса или тип интерфейса.

Возможно также обращение к полю текущего экземпляра или текущего класса с помощью простого имени (§6.5.6.1).

§15.11.1. Обращение к полю с помощью первичного выражения

Типом *Primary* должен быть ссылочный тип `T`, иначе генерируется ошибка времени компиляции.

Смысл выражения обращения к полю определяется следующим образом.

- Если идентификатор именует несколько доступных (§6.6) полей-членов типа `T`, обращение к полю неоднозначное и генерируется ошибка времени компиляции.
- Если идентификатор не именует доступное поле-член в типе `T`, обращение к полю не определено и генерируется ошибка времени компиляции.
- В противном случае идентификатор именует единственное доступное поле в типе `T`, и типом выражения обращения к полю является тип поля-члена после преобразования при фиксации (§5.1.10).

Во время выполнения результат выражения обращения к полю вычисляется следующим образом (в предположении, что программа корректна в смысле анализа определенных присваиваний, т.е. каждая неинициализированная переменная, объявленная как `final`, определенно присваивается до обращения).

- Если поле объявлено как `static`, происходит следующее.
  - ✦ Вычисляется выражение *Primary*, и его результат игнорируется. Если вычисление выражения *Primary* завершается преждевременно, выражение обращения к полю завершается преждевременно по той же причине.
  - ✦ Если поле является непустым (инициализированным) полем, объявленным как `final`, то результатом является значение указанной переменной класса в классе или интерфейсе, являющемся типом выражения *Primary*.
  - ✦ Если поле не объявлено как `final` или является пустым полем, объявленным как `final`, и обращение к полю находится в статическом инициализаторе или инициализаторе переменной класса, то результатом является переменная, а именно — указанная переменная класса в классе, являющемся типом выражения *Primary*.
- Если поле не объявлено как `static`, происходит следующее.
  - ✦ Вычисляется выражение *Primary*. Если вычисление выражения *Primary* завершается преждевременно, выражение обращения к полю завершается преждевременно по той же причине.
  - ✦ Если значением выражения *Primary* является `null`, генерируется исключение `NullPointerException`.
  - ✦ Если поле является непустым полем, объявленным как `final`, то результат представляет собой значение именованного поля-члена типа `T`, принадлежащего объекту, на который указывает значение *Primary*.

- Если поле не объявлено как `final` или является пустым полем, объявленным как `final`, и обращение к полю находится в конструкторе или инициализаторе переменной экземпляра, то результатом является переменная, а именно — именованное поле-член типа *T*, принадлежащее объекту, на который указывает значение *Primary*.

Обратите внимание, что для определения используемого поля используется только тип выражения *Primary*, но не класс фактического объекта, к которому выполняется обращение во время выполнения.

Пример 15.11.1-1. Статическое связывание обращений к полю

```
class S            { int x = 0; }
class T extends S { int x = 1; }
class Test1 {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
                        + t.getClass() + " at run time.";
    }
}
```

Вывод этой программы имеет вид

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=0 when s holds a class T at run time.
```

Последняя строка показывает, что в действительности поле, к которому осуществляется обращение, не зависит от класса времени выполнения объекта, к которому выполняется обращение; даже если `s` содержит ссылку на объект класса `T`, выражение `s.x` ссылается на поле `x` класса `S`, потому что типом выражения `s` является `S`. Объекты класса `T` содержат два поля с именем `x`, одно для класса `T` и одно для его суперкласса `S`.

Это отсутствие динамического поиска для обращения к полю обеспечивает эффективную работу программ с простой реализацией языка. Мощь позднего связывания и перекрытия доступна, но только при использовании методов экземпляра. Рассмотрим тот же пример, используя для доступа к полям методы экземпляра.

```
class S            { int x = 0; int z() { return x; } }
class T extends S { int x = 1; int z() { return x; } }
class Test2 {
    public static void main(String[] args) {
        T t = new T();
```

```
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
        s = t;
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
    static String when(String name, Object t) {
        return " when " + name + " holds a "
                        + t.getClass() + " at run time.";
    }
}
```

Теперь вывод программы имеет вид

```
t.z()=1 when t holds a class T at run time.
s.z()=0 when s holds a class S at run time.
s.z()=1 when s holds a class T at run time.
```

Последняя строка показывает, что в действительности метод, который осуществляет обращение, *зависит* от класса времени выполнения объекта, к которому выполняется обращение. Когда `s` содержит ссылку на объект класса `T`, выражение `s.z()` указывает на метод `z` класса `T`, несмотря на тот факт, что типом выражения `s` является `S`. Метод `z` класса `T` перекрывает метод `z` класса `S`.

Пример 15.11.1-2. Ссылающаяся переменная не имеет значения для обращения к статическому полю

Приведенная программа демонстрирует, что для доступа к переменной класса (`static`) можно использовать нулевую ссылку — это не приведет к генерации исключения.

```
class Test3 {
    static String mountain = "Chocorua";
    static Test3 favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }
}
```

Эта программа успешно компилируется и выполняется; ее вывод имеет вид

```
Mount Chocorua
```

Несмотря на то что результат вызова `favorite()` равен `null`, исключение `NullPointerException` не генерируется. Выведенное слово `"Mount "` демонстрирует, что выражение *Primary* действительно полностью вычисляется во время выполнения программы, независимо от того факта, что для определения целевого поля, к которому выполняется обращение, требуется только его тип, но не значение (поскольку поле `mountain` объявлено как `static`).

§15.11.2. Обращение к методам суперкласса с помощью ключевого слова `super`

Выражение вида `super.`*Identifier* обращается к полю с именем *Identifier* текущего объекта, но текущий объект рассматривается как экземпляр суперкласса текущего класса.

Выражение вида *T*`.super.`*Identifier* обращается к полю с именем *Identifier* лексически охватывающего экземпляра, соответствующего *T*, но этот экземпляр рассматривается как экземпляр суперкласса *T*.

Применение ключевого слова `super` корректно только в методе экземпляра, инициализаторе экземпляра или конструкторе класса, или в инициализаторе переменной экземпляра класса. Если оно встречается в ином контексте, генерируется ошибка времени компиляции.

> Это в точности те же ситуации, в которых ключевое слово `this` может использоваться в объявлении класса (§15.8.3).

Если выражение с ключевым словом `super` находится в объявлении класса `Object`, генерируется ошибка времени компиляции, так как класс `Object` не имеет суперкласса.

Предположим, что выражение обращения к полю `super.`*f* находится в классе *C* и что непосредственным суперклассом *C* является класс *S*. Если *f* в *S* доступно из класса *C* (§6.6), то `super.`*f* рассматривается в точности так, как если бы это было выражение `this.`*f* в теле класса *S*. В противном случае генерируется ошибка времени компиляции.

> Таким образом, `super.`*f* может обращаться к полю *f*, доступному в классе *S*, даже если это поле скрыто объявлением поля *f* в классе *C*.

Предположим, что выражение обращения к полю *T*`.super.`*f* находится в классе *C*, а непосредственный суперкласс класса, обозначаемого *T*, представляет собой класс, полностью квалифицированное имя которого — *S*. Если *f* в *S* доступно из *C*, то *T*`.super.`*f* рассматривается так, как если бы выражение `this.`*f* находилось в теле класса *S*. В противном случае генерируется ошибка времени компиляции.

> Таким образом, *T*`.super.`*f* может обращаться к полю *f*, доступному в классе *S*, даже если это поле скрыто объявлением поля *f* в классе *T*.

Если текущий класс не является внутренним классом класса *T* или самим классом *T*, генерируется ошибка времени компиляции.

Пример 15.11.2-1. Выражение `super`

```
interface I            { int x = 0; }
class T1 implements I  { int x = 1; }
class T2 extends T1    { int x = 2; }
class T3 extends T2 {
    int x = 3;
    void test() {
        System.out.println("x=\t\t"          + x);
        System.out.println("super.x=\t\t"    + super.x);
        System.out.println("((T2)this).x=\t" + ((T2)this).x);
```

```
        System.out.println("((T1)this).x=\t" + ((T1)this).x);
        System.out.println("((I)this).x=\t"  + ((I)this).x);
    }
}
class Test {
    public static void main(String[] args) {
        new T3().test();
    }
}
```

Вывод программы имеет вид

```
x=                  3
super.x=            2
((T2)this).x=       2
((T1)this).x=       1
((I)this).x=        0
```

В классе `T3` выражение `super.x` рассматривается так, как если бы оно представляло собой `((T2)this).x`, где `x` имеет доступ уровня пакета. Обратите внимание, что `super.x` не определяется в терминах приведения в силу сложностей доступа к членам суперкласса, объявленным как `protected`.

§15.12. Выражения вызовов методов

Выражение вызова метода используется для вызова метода класса или экземпляра.

MethodInvocation:
MethodName (*[ArgumentList]*)
TypeName . *[TypeArguments] Identifier* (*[ArgumentList]*)
ExpressionName . *[TypeArguments] Identifier* (*[ArgumentList]*)
Primary . *[TypeArguments] Identifier* (*[ArgumentList]*)
`super` . *[TypeArguments] Identifier* (*[ArgumentList]*)
TypeName . `super` . *[TypeArguments] Identifier* (*[ArgumentList]*)

ArgumentList:
Expression {, *Expression*}

Разрешение имени метода во время компиляции является более сложным, чем разрешение имени поля, из-за возможности перегрузки методов. Вызов метода во время выполнения также является более сложным, чем доступ к полю, из-за возможности перекрытия метода экземпляра.

Определение метода, который будет вызываться выражением вызова метода, включает несколько этапов. В следующих трех разделах описана обработка вызова метода времени компиляции. Определение типа выражения вызова метода описано в §15.12.3.

Типы исключений, которые может генерировать выражение вызова метода, определяются в §11.2.1.

Если имя слева от крайней справа ".", находящееся перед (в *MethodInvocation*, не может быть классифицировано как *TypeName* или *ExpressionName* (§6.5.2), генерируется ошибка времени компиляции.

Если *TypeArguments* присутствует слева от *Identifier*, то, если любой из аргументов типа представляет собой символ подстановки (§4.5.1), генерируется ошибка времени компиляции.

Выражение вызова метода является поливыражением, если выполняются все перечисленные ниже условия.

- Вызов находится в контексте присваивания или контексте вызова (§5.2, §5.3).
- Если вызов квалифицированный (т.е. имеет любой вид *MethodInvocation* за исключением первого), то вызов скрывает *TypeArguments* слева от *Identifier*.
- Вызываемый метод, определенный, как описано в следующих подразделах, является обобщенным (§8.4.4) и имеет возвращаемый тип, который упоминает как минимум один из параметров типов метода.

  В противном случае выражение вызова метода является автономным.

§15.12.1. Этап 1 времени компиляции: определение класса или интерфейса для поиска

Первый этап обработки вызова метода времени компиляции состоит в выводе имени вызываемого метода, а также того, в каком классе или интерфейсе следует искать определения методов с этим именем.

Имя метода определяется нетерминалом *MethodName* или *Identifier*, который непосредственно предшествует левой скобке *MethodInvocation*.

Для поиска класса или интерфейса следует рассмотреть шесть случаев, в зависимости от того, что предшествует левой скобке *MethodInvocation*.

- Если это *MethodName*, т.е. просто *Identifier*, то

  если *Identifier* находится в области видимости объявления видимого метода с данным именем (§6.3, §6.4.1), то:

  - ✦ если имеется объявление охватывающего типа, членом которого является данный метод, то пусть T представляет собой наиболее глубоко вложенное объявление такого типа. Класс или интерфейс, в котором выполняется поиск, — T;

    Такая стратегия поиска называется "правилом гребенки". Она эффективно ищет методы в иерархии суперклассов вложенного класса перед тем, как искать методы в охватывающем классе и иерархии его суперклассов. Смотрите пример в §6.5.7.1.

  - ✦ в противном случае объявление видимого метода может находиться в области видимости в силу одного или нескольких объявлений единственного статического импорта или объявления статического импорта по требованию. Класса или интерфейса для поиска не имеется, так как вызываемый метод определяется позже (§15.12.2).

- Если это *TypeName* . *[TypeArguments] Identifier*, то искомый тип представляет собой тип, обозначаемый с помощью *TypeName*.

- Если это *ExpressionName* . *[TypeArguments] Identifier*, то искомый класс или интерфейс представляет собой объявленный тип `T` переменной, обозначаемой *ExpressionName*, если `T` представляет собой тип класса или интерфейса или верхнюю границу `T`, если `T` является переменной типа.
- Если это *Primary* . *[TypeArguments] Identifier*, то пусть `T` представляет собой тип выражения *Primary*. Искомый класс или интерфейс представляет собой `T`, если `T` является типом класса или интерфейса или верхней границей `T`, если `T` является переменной типа.

  Если `T` не является ссылочным типом, генерируется ошибка времени компиляции.
- Если это `super` . *[TypeArguments] Identifier*, то искомый класс является суперклассом класса, объявление которого содержит вызов метода.

  Пусть `T` является объявлением типа, непосредственно охватывающим вызов метода. Если `T` представляет собой класс `Object` или `T` является интерфейсом, генерируется ошибка времени компиляции.
- Если это *TypeName* . `super` . *[TypeArguments] Identifier*, то имеем следующее.
  - ✦ Если *TypeName* не обозначает ни класс, ни интерфейс, генерируется ошибка времени компиляции.
  - ✦ Если *TypeName* обозначает класс, `C`, то искомый класс является суперклассом `C`.

    Если `C` не является лексически охватывающим объявлением типа текущего класса или если `C` представляет собой класс `Object`, генерируется ошибка времени компиляции.

    Пусть `T` является объявлением типа, непосредственно охватывающим вызов метода. Если `T` представляет собой класс `Object`, генерируется ошибка времени компиляции.
  - ✦ В противном случае *TypeName* обозначает искомый интерфейс, `I`.

    Пусть `T` является объявлением типа, непосредственно охватывающим вызов метода. Если `I` не является непосредственным суперинтерфейсом для `T` или если существует некоторый другой непосредственный суперкласс или непосредственный суперинтерфейс для `T`, `J`, такой, что `J` является подтипом `I`, генерируется ошибка времени компиляции.

Синтаксис *TypeName* . `super` перегружен: традиционно *TypeName* ссылается на лексически охватывающее объявление типа, который является классом, а целевым является суперкласс данного класса, как если бы вызов представлял собой неквалифицированный `super` в лексически охватывающем объявлении типа.

```
class Superclass {
    void foo() { System.out.println("Hi"); }
}

class Subclass1 extends Superclass {
    void foo() { throw new UnsupportedOperationException(); }
    Runnable tweak = new Runnable() {
        void run() {
```

```
            Subclass1.super.foo(); // Получам поведение
        }                          //  'println'
    };
}
```

Для поддержки вызовов методов по умолчанию в суперинтерфейсах *TypeName* может также ссылаться на непосредственный суперинтерфейс текущего класса или интерфейса, и целью является этот суперинтерфейс.

```
interface Superinterface {
    default void foo() { System.out.println("Hi"); }
}

class Subclass2 implements Superinterface {
    void foo() { throw new UnsupportedOperationException(); }
    void tweak() {
        Superinterface.super.foo(); // Получаем поведение
    }                               //  'println'
}
```

Синтаксис не поддерживает объединение этих видов, т.е. вызов метода суперинтерфейса лексически охватывающего объявления типа, который представляет собой класс, как если бы вызов имел вид *InterfaceName* `.super` в лексически охватывающем объявлении типа.

```
class Subclass3 implements Superinterface {
    void foo() { throw new UnsupportedOperationException(); }

    Runnable tweak = new Runnable() {
        void run() {
            Subclass3.Superinterface.super.foo(); // Неверно
        }
    };
}
```

Обходным путем является введение в лексически охватывающее объявление типа `private`-метода, который выполняет вызов интерфейса `super`.

§15.12.2. Этап 2 времени компиляции: определение сигнатуры метода

На втором этапе выполняется поиск методов-членов в определенном на первом этапе типе. На этом шаге имя метода и типы выражений аргументов используются для обнаружения *доступных* и *применимых* методов, т.е. объявлений, которые могут быть корректно вызваны с заданными аргументами.

Может найтись несколько таких методов, и в этом случае выбирается *наиболее точно подходящий*. Дескриптор (сигнатура плюс возвращаемый тип) наиболее точно подходящего метода используется во время выполнения программы для диспетчеризации метода.

Метод *применим*, если он применим с помощью строгого вызова (§15.12.2.2), нестрогого вызова (§15.12.2.3) или вызова с переменной арностью (§15.12.2.4).

Некоторые выражения аргументов, которые содержат неявно типизированные лямбда-выражения (§15.27.1) или неточные[1] ссылки на метод (§15.13.1), игнорируются проверками применимости, поскольку их значение не может быть определено до выбора целевого типа.

Хотя вызов метода может быть поливыражением, только его выражения аргументов — а *не* целевой тип вызова — влияют на выбор применимых методов.

Процесс определения применимости начинается с определения потенциально применимых методов (§15.12.2.1).

Остальная часть процесса разбивается на три фазы, чтобы гарантировать совместимость с версиями языка программирования Java до версии Java SE 5.0. Это следующие фазы.

1. Первая фаза (§15.12.2.2) выполняет разрешение перегрузки без применения преобразований упаковки и распаковки или применения вызова методов переменной арности. Если в этой фазе применимый метод не найден, выполняется переход ко второй фазе.

   > Этим гарантируется, что любые вызовы, которые были корректны в языке программирования Java до версии Java SE 5.0, не рассматриваются как неоднозначные из-за добавления методов переменной арности и неявной упаковки и/или распаковки. Однако объявление метода переменной арности (§8.4.1) может изменить метод, выбранный для данного выражения вызова, поскольку метод переменной арности рассматривается в первой фазе как метод фиксированной арности. Например, объявление `m(Object...)` в классе, который уже объявил `m(Object)`, приводит к тому, что `m(Object)` больше не выбирается для некоторых выражений вызова (таких, как `m(null)`), так как `m(Object[])` более точный.

2. Вторая фаза (§15.12.2.3) выполняет разрешение перегрузки при разрешенной упаковке и распаковке, но все еще исключенном применении вызовов методов с переменной арностью. Если применимые методы в процессе этой фазы не найдены, выполняется переход к третьей фазе.

   > Это гарантирует, что метод с переменной арностью не будет выбран, если в наличии имеется применимый метод с фиксированной арностью.

3. Третья фаза (§15.12.2.4) разрешает перегрузку с использованием методов с переменной арностью, упаковкой и распаковкой.

Принятие решения о применимости метода в случае обобщенных методов (§8.4.4) требует анализа аргументов типов. Аргументы типа могут быть переданы явно или неявно. Если они переданы неявно, их границы должны быть выведены (§18) из выражений аргументов.

Если идентифицировано несколько применимых методов в процессе выполнения одной из трех фаз проверки применимости, то выбирается наиболее точно подходящий, как определено в разделе §15.12.2.5.

---

[1] Ссылка точная, если имеется только одно возможное объявление времени компиляции с единственным возможным типом вызова; в противном случае ссылка неточная. — *Примеч. ред.*

Для проверки применимости типы аргументов вызова не могут в общем случае служить входными данными для анализа. Это связано со следующим.

- Аргументы вызова метода могут быть поливыражениями.
- Поливыражения не могут быть типизированы в отсутствие целевого типа.
- Разрешение перегрузки должно быть завершено до того, как будут известны целевые типы аргументов.

Вместо этого входные данные для проверки применимости представляют собой список выражений аргументов, который *может* быть проверен на совместимость с потенциальными целевыми типами, даже если конечные типы выражений неизвестны.

Обратите внимание, что разрешение перегрузки не зависит от целевого типа. Тому имеются две причины.

- Во-первых, это делает модель более доступной и менее подверженной ошибкам для пользователя (программиста). Значение имени метода (т.е. объявление, соответствующее имени) является слишком фундаментальным для того, чтобы смысл программы зависел от тонких контекстуальных подсказок. (В отличие от этого другие поливыражения могут иметь различное поведение в зависимости от целевого типа; но вариации в поведении всегда ограничены и по существу эквивалентны, хотя и нельзя давать никаких гарантий по поводу поведения произвольного набора методов, которые разделяют имя и арность.)
- Во-вторых, это позволяет другим свойствам — например, таким, как является ли метод поливыражением (§15.12) или как классифицировать условное выражение (§15.25) — зависеть от значения имени метода, даже до того, как будет известен целевой тип.

Пример 15.12.12-1. Применимость методов

```
class Doubler {
            static int two()       { return two(1); }
    private static int two(int i) { return 2*i;    }
}
class Test extends Doubler {
    static long two(long j) { return j+j; }
    public static void main(String[] args) {
        System.out.println(two(3));
        System.out.println(Doubler.two(3)); // Ошибка
                                            // времени компиляции
    }
}
```

В случае вызова метода `two(1)` в классе `Doubler` имеется два доступных метода с именем `two`, но только второй из них применим, так что именно он и будет вызван во время выполнения программы.

В случае вызова метода `two(3)` в классе `Test` имеется два применимых метода, но только один из них в классе `Test` является доступным, так что именно он и будет вызван во время выполнения программы (аргумент `3` преобразуется в тип `long`).

В случае вызова метода `Doubler.two(3)` поиск метода с именем `two` выполняется в классе `Doubler`, а не в классе `Test`; единственный применимый метод не является доступным, так что этот вызов метода приводит к ошибке времени компиляции.

Вот еще один пример.

```
class ColoredPoint {
    int x, y;
    byte color;
    void setColor(byte color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37); // Ошибка времени компиляции
    }
}
```

Здесь ошибка времени компиляции возникает при втором вызове `setColor`, поскольку во время компиляции не удается найти применимый метод. Типом литерала `37` является `int`, а `int` не может быть преобразован в `byte` преобразованием вызова. Преобразование присваивания, которое используется в инициализации переменной `color`, выполняет неявное преобразование константы из типа `int` в `byte`, разрешенное, поскольку значение `37` достаточно мал*о* для представления типом `byte`; но такое преобразование типа не разрешено для преобразования вызова метода.

Если бы метод `setColor` был объявлен как принимающий `int` вместо `byte`, то оба вызова были бы корректны; первый вызов был бы разрешен в силу того, что преобразование вызова может выполнять расширяющее преобразование `byte` в `int`. Однако в таком случае в теле `setColor` потребуется сужающее приведение.

```
void setColor(int color) { this.color = (byte)color; }
```

Вот пример неоднозначности перегрузки. Рассмотрим следующую программу.

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static void test(ColoredPoint p, Point q) {
        System.out.println("(ColoredPoint, Point)");
    }
    static void test(Point p, ColoredPoint q) {
        System.out.println("(Point, ColoredPoint)");
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        test(cp, cp); // Ошибка времени компиляции
    }
}
```

В этом примере генерируется ошибка времени компиляции. Проблема в том, что имеется два объявления `test`, которые применимы и доступны, и ни одно из них не является более точно подходящим, чем другое. Следовательно, вызов метода неоднозначен.

Если добавить третье определение `test`

```
static void test(ColoredPoint p, ColoredPoint q) {
    System.out.println("(ColoredPoint, ColoredPoint)");
}
```

то оно окажется более точно подходящим, чем два другие, и вызов метода перестанет быть неоднозначным.

Пример 15.12.2-2. При выборе метода возвращаемый тип не рассматривается

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static int test(ColoredPoint p) {
        return p.color;
    }
    static String test(Point p) {
        return "Point";
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        String s = test(cp); // Ошибка времени компиляции
    }
}
```

Здесь наиболее точно подходящим объявлением метода `test` является объявление с типом параметра `ColoredPoint`. Поскольку тип результата метода — `int`, генерируется ошибка времени компиляции, поскольку `int` не может быть преобразован в `String` преобразованием присваивания. Этот пример демонстрирует, что возвращаемые типы методов не участвуют в разрешении перегрузки, так что второй метод `test`, возвращающий `String`, не может быть выбран, несмотря на то что он имеет возвращаемый тип, обеспечивающий компиляцию программы без ошибок.

Пример 15.12.2-3. Выбор наиболее точно подходящего метода

Наиболее точно подходящий метод выбирается во время компиляции; его дескриптор определяет, какой именно метод будет выполнен во время выполнения программы. Если к коду добавляется новый метод, то код, который был скомпилирован со старым определением класса, может этот новый метод не использовать, даже если при полной перекомпиляции выбирается именно этот новый метод.

Так, например, рассмотрим два модуля компиляции. Первый — для класса `Point`.

```
package points;
public class Point {
    public int x, y;
```

```
    public Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return toString(""); }
    public String toString(String s) {
        return "(" + x + "," + y + s + ")";
    }
}
```

А второй — для класса `ColoredPoint`.

```
package points;
public class ColoredPoint extends Point {
    public static final int
        RED = 0, GREEN = 1, BLUE = 2;
    public static String[] COLORS =
        { "red", "green", "blue" };
    public byte color;
    public ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = (byte)color;
    }

    /** Копирует все существенные поля аргумента в
        данный объект ColoredPoint. */
    public void adopt(Point p) { x = p.x; y = p.y; }
    public String toString() {
        String s = "," + COLORS[color];
        return super.toString(s);
    }
}
```

Теперь рассмотрим третий модуль компиляции, использующий класс `Colored Point`.

```
import points.*;
class Test {
    public static void main(String[] args) {
        ColoredPoint cp =
            new ColoredPoint(6, 6, ColoredPoint.RED);
        ColoredPoint cp2 =
            new ColoredPoint(3, 3, ColoredPoint.GREEN);
        cp.adopt(cp2);
        System.out.println("cp: " + cp);
    }
}
```

Вывод данной программы имеет вид

```
cp: (3,3,red)
```

Программист, писавший класс `Test`**, ожидал бы увидеть здесь слово** `green`**, поскольку фактический аргумент,** `ColoredPoint`**, имеет поле** `color`**, а** `color`**, казалось бы, относится к "существенным полям". (Конечно, документация пакета** `points` **должна быть гораздо более точной!)**

Заметим, кстати, что наиболее точно подходящий метод (в действительности — единственный применимый) для вызова `adopt` имеет сигнатуру, которая указывает, что это метод с одним параметром и этот параметр имеет тип `Point`. Эта сигнатура становится частью бинарного представления класса `Test`, сгенерированного компилятором Java, и используется вызовом метода во время выполнения.

Предположим, об этой ошибке было сообщено, и после соответствующего обсуждения производитель пакета `points` решил исправить ее, добавив метод в класс `ColoredPoint`.

```
public void adopt(ColoredPoint p) {
    adopt((Point)p);
    color = p.color;
}
```

Если теперь запустить старый бинарный файл `Test` с новым бинарным файлом `ColoredPoint`, вывод останется прежним.

```
cp: (3,3,red)
```

Дело в том, что в старом бинарном файле `Test` все еще указан дескриптор "один параметр типа `Point`; возвращаемый тип `void`", связанный с вызовом метода `cp.adopt(cp2)`. Если исходный текст `Test` перекомпилировать, то компилятор Java обнаружит, что теперь есть два применимых метода `adopt` и что сигнатура более точно подходящего представляет собой "один параметр типа `ColoredPoint`; возвращаемый тип `void`"; после этого запуск программы даст ожидаемый вывод.

```
cp: (3,3,green)
```

Но если принять определенные меры, то программист пакета `points` может исправить класс `ColoredPoint` так, чтобы он работал как со вновь скомпилированным, так и со старым кодом. Для этого надо добавить определенный код к старому методу `adopt` для того, чтобы старый код, который его вызывает, мог работать с аргументами `ColoredPoint`.

```
public void adopt(Point p) {
    if (p instanceof ColoredPoint)
        color = ((ColoredPoint)p).color;
    x = p.x; y = p.y;
}
```

В идеале исходный код должен быть перекомпилирован при изменении кода, от которого он зависит. Однако в среде, в которой различные классы поддерживаются различными организациями, это не всегда осуществимо. Такое "оборонительное программирование" с тщательным вниманием к проблемам эволюции класса может сделать обновленный код гораздо более надежным. Подробное обсуждение двоичной совместимости и эволюции типов можно найти в §13.

§15.12.2.1. Идентификация потенциально применимых методов

В определенном на первом этапе времени компиляции (§15.12.1) классе или интерфейсе выполняется поиск всех потенциально применимых для данного вызова методов-членов; в поиск включаются члены, унаследованные от суперклассов и суперинтерфейсов.

Кроме того, если выражение вызова метода имеет вид *MethodName* — т.е. единственный *Identifier*, — процесс поиска потенциально применимых методов изучает также все методы-члены, которые импортированы объявлением единственного статического импорта и объявлением объявления статического импорта по требованию в модуле компиляции, в котором находится вызов метода (§7.5.3, §7.5.4), и не затенены в месте вызова метода.

Метод-член *потенциально применим* для вызова метода тогда и только тогда, когда выполняются все перечисленные далее условия.

- Имя члена идентично имени метода в вызове метода.
- Член доступен (§6.6) классу или интерфейсу, в котором находится вызов метода.

> Является ли метод-член доступным при вызове метода, зависит от модификатора доступа (`public`, никакого (доступ пакета), `protected` или `private`) в объявлении члена и от того, где находится вызов метода.

- Если член является методом фиксированной арности со значением арности n, арность вызова метода имеет значение n, и для всех i ($1 \le i \le n$) i-й аргумент вызова метода *потенциально совместим* (как определено ниже) с типом i-го параметра метода.
- Если член является методом переменной арности со значением арности n, то для всех i ($1 \le i \le n - 1$) i-й аргумент вызова метода *потенциально совместим* с типом i-го параметра метода; а там, где n-й параметр метода имеет тип `T[]`, справедливо одно из следующего списка.
  - ✦ Арность вызова метода имеет значение, равное $n - 1$.
  - ✦ Арность вызова метода имеет значение, равное n, и n-й аргумент вызова метода потенциально совместим с `T` или `T[]`.
  - ✦ Арность вызова метода имеет значение, равное m, где $m > n$, и для всех i ($n \le i \le m$) i-й аргумент вызова метода потенциально совместим с `T`.
- Если вызов метода включает аргументы с явным типом и член представляет собой обобщенный метод, то количество аргументов типа равно количеству параметров типа метода.

> Это положение подразумевает, что необобщенный метод может быть потенциально применим в вызове, который предоставляет аргументы явного типа. Он и в самом деле может оказаться применимым, — в таком случае аргументы типа будут просто игнорироваться.
>
> Это правило вытекает из проблем совместимости и принципов взаимозаменяемости. Так как интерфейсы или суперклассы могут быть обобщаемы независимо от их подтипов, мы можем перекрыть обобщенный метод необобщенным. Однако перекрывающий (необобщенный) метод должен быть применим к вызову обобщенного метода, включая вызовы, которые явно передают аргументы типа. В противном случае подтип не будет взаимозаменяем с его обобщенным супертипом.

Если поиск не находит как минимум один потенциально применимый метод, генерируется ошибка времени компиляции.

Выражение *потенциально совместимо* с целевым типом в соответствии со следующими правилами.

- Лямбда-выражение (§15.27) потенциально совместимо с типом функционального интерфейса (§9.8), если выполняются все следующие условия.
  - ✦ Арность типа функции целевого типа та же, что и арность лямбда-выражения.
  - ✦ Если тип функции целевого типа имеет возвращаемый тип `void`, то тело лямбда-выражения представляет собой либо выражение инструкции (§14.8), либо void-совместимый блок (§15.27.2).
  - ✦ Если тип функции целевого типа имеет (не-`void`) возвращаемый тип, то тело лямбда-выражения представляет собой либо выражение, либо блок, совместимый со значением (§15.27.2).
- Выражение ссылки на метод (§15.13) потенциально совместимо с типом функционального интерфейса, если при арности типов типа функции, равной *n*, имеется как минимум один применимый метод для выражения ссылки на метод с арностью *n* (§15.13.1) и выполняется одно из приведенных далее условий.
  - ✦ Выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Identifier* и как минимум один потенциально применимый метод является 1) статическим и поддерживает арность *n* или 2) не статическим и поддерживает арность *n* – 1.
  - ✦ Выражение ссылки на метод имеет иной вид и как минимум один потенциально применимый метод не является статическим.
- Лямбда-выражение или выражение ссылки на метод потенциально совместимо с переменной типа, если переменная типа представляет собой параметр типа метода-кандидата.
- Выражение в скобках (§15.8.5) потенциально совместимо с типом, если содержащееся в нем выражение потенциально совместимо с этим типом.
- Условное выражение (§15.25) потенциально совместимо с типом, если выражения и второго, и третьего его операндов потенциально совместимы с этим типом.
- Выражение создания экземпляра класса, выражение вызова метода или выражение автономного вида (§15.2) потенциально совместимо с любым типом.

> Определение потенциальной применимости выходит за рамки базовой проверки арности и должно также учитывать наличие и “форму” целевых типов функционального интерфейса. В некоторых случаях, включающих вывод аргумента типа, лямбда-выражение, появляющееся в качестве аргумента метода вызова, не может быть корректно типизировано, пока не будет выполнено разрешение перегрузки. Эти правила позволяют по-прежнему принимать во внимание вид лямбда-выражения, отбрасывая очевидно неверные целевые типы, которые в противном случае могут вызвать ошибки неоднозначности.

§15.12.2.2. Фаза 1: идентификация методов соответствующей арности, применимых строгим вызовом

Выражение аргумента рассматривается как *подходящее для применимости* (pertinent to applicability) для потенциально применимого метода m, если только оно не имеет один из перечисленных далее видов.

- Неявно типизированное лямбда-выражение (§15.27.1).
- Неточное выражение ссылки на метод (§15.13.1).
- m представляет собой обобщенный метод и вызов метода не предоставляет явных аргументов типа, явно типизированное лямбда-выражение или выражение точной ссылки на метод, для которого соответствующий целевой тип (полученный из сигнатуры m) является параметром типа m.
- Явно типизированное лямбда-выражение, тело которого представляет собой выражение, не являющееся подходящим для применимости.
- Явно типизированное лямбда-выражение, тело которого представляет собой блок, в котором как минимум одно выражение результата не является подходящим для применимости.
- Выражение в скобках (§15.8.5), которое не подходит для применимости.
- Условное выражение (§15.25), второй или третий операнд которого не подходит для применимости.

Пусть m — потенциально применимый метод (§15.12.2.1) с арностью n и типами формальных параметров F_1 ... F_n и пусть $e_1,...,e_n$ представляют собой фактические выражения аргументов вызова метода. Тогда справедливо следующее.

- Если m является обобщенным методом и если вызов метода не предоставляет явных аргументов типа, то применимость метода выводится так, как указано в §18.5.1.
- Если m является обобщенным методом и если вызов метода предоставляет явные аргументы типа, то пусть R_1 ... R_p ($p \geq 1$) — параметры типа m и пусть B_l является объявленной границей R_l ($1 \leq l \leq p$). Пусть также $U_1,...,U_p$ являются явными аргументами типа, заданными в вызове метода. Тогда m *применим строгим вызовом*, если верны оба следующих условия.
  - ✦ Для $1 \leq i \leq n$, если e_i подходящий для применимости, то e_i совместим в контексте строгого вызова с $F_i[R_1:=U_1, ..., R_p:=U_p]$.
  - ✦ Для $1 \leq l \leq p$, U_l <: $B_l[R_1:=U_1, ..., R_p:=U_p]$.
- Если m не является обобщенным методом, то m *применим строгим вызовом*, если для $1 \leq i \leq n$ либо e_i совместим в контексте строгого вызова с F_i, либо e_i не подходящий для применимости.

Если не найден метод, применимый строгим вызовом, то поиск применимых методов продолжается с помощью фазы 2 (§15.12.2.3).

В противном случае среди методов, применимых строгим вызовом, выбирается наиболее подходящий метод (§15.12.2.5).

> Значение неявно типизированного лямбда-выражения или неточного выражения ссылки на метод является достаточно неопределенным до разрешения целевого типа. Поэтому аргументы, содержащие эти выражения, не рассматриваются как *подходящие для применимости*; они просто игнорируются (за исключением их ожидаемой арности) до окончания разрешения перегрузки.

§15.12.2.3. Фаза 2: идентификация методов соответствующей арности, применимых нестрогим вызовом

Пусть m — потенциально применимый метод (§15.12.2.1) с арностью n и типами формальных параметров F_1 ... F_n и пусть $e_1,...,e_n$ представляют собой фактические выражения аргументов вызова метода. Тогда справедливо следующее.

- Если m является обобщенным методом и вызов метода не предоставляет явных аргументов типа, то применимость метода выводится так, как указано в §18.5.1.
- Если m является обобщенным методом и вызов метода предоставляет явные аргументы типа, то пусть R_1 ... R_p ($p \geq 1$) — параметры типа m и пусть B_1 является объявленной границей R_1 ($1 \leq l \leq p$). Пусть также $U_1,...,U_p$ являются явными аргументами типа, заданными в вызове метода. Тогда m *применим нестрогим вызовом*, если верны оба следующих условия.
  - ✦ Для $1 \leq i \leq n$, если e_i подходящий для применимости (§15.12.2.2), e_i совместим в контексте нестрогого вызова с $F_i[R_1:=U_1, ..., R_p:=U_p]$.
  - ✦ Для $1 \leq l \leq p$, $U_l <: B_l[R_1:=U_1, ..., R_p:=U_p]$.
- Если m не является обобщенным методом, то m *применим нестрогим вызовом*, если для $1 \leq i \leq n$ либо e_i совместим в контексте нестрогого вызова с F_i, либо e_i не подходящий для применимости.

Если не найден метод, применимый нестрогим вызовом, то поиск применимых методов продолжается с помощью фазы 3 (§15.12.2.4).

В противном случае среди методов, применимых нестрогим вызовом, выбирается наиболее подходящий метод (§15.12.2.5).

§15.12.2.4. Фаза 3: идентификация методов, применимых вызовом переменной арности

Если метод переменной арности имеет типы формальных параметров F_1, ..., F_{n-1}, $F_n[]$, определим *i-й тип параметра переменной арности* следующим образом.

- При $i \leq n-1$ i-й тип параметра переменной арности представляет собой F_i.
- При $i \geq n-1$ i-й тип параметра переменной арности представляет собой F_n.

Пусть m — потенциально применимый метод (§15.12.2.1) с переменной арностью и пусть T_1, ..., T_k — первые k типов параметров переменной арности m, а e_1, ..., e_k — фактические выражения аргументов вызова метода. Тогда справедливо следующее.

- Если m является обобщенным методом и вызов метода не предоставляет явных аргументов типа, то применимость метода выводится так, как указано в §18.5.1.

- Если m является обобщенным методом и вызов метода предоставляет явные аргументы типа, то пусть R_1 ... R_p ($p \geq 1$) — параметры типа m и пусть B_l является объявленной границей R_l ($1 \leq l \leq p$). Пусть также $U_1,\ldots,U_p$ являются явными аргументами типа, заданными в вызове метода. Тогда m *применим вызовом переменной арности*, если верны оба следующих условия.
  - ✦ Для $1 \leq i \leq k$, если e_i подходящий для применимости (§15.12.2.2), то e_i совместим в контексте нестрогого вызова с $T_i\,[R_1{:=}U_1, \ldots, R_p{:=}U_p]$.
  - ✦ Для $1 \leq l \leq p$ U_l <: $B_l\,[R_1{:=}U_1, \ldots, R_p{:=}U_p]$.
- Если m не является обобщенным методом, то m *применим вызовом переменной арности*, если для $1 \leq i \leq k$ либо e_i совместим в контексте нестрогого вызова с T_i, либо e_i не подходящий для применимости.

Если не найден метод, применимый вызовом переменной арности, генерируется ошибка времени компиляции.

В противном случае среди методов, применимых вызовом переменной арности, выбирается наиболее подходящий (§15.12.2.5).

§15.12.2.5. Выбор наиболее подходящего метода

Если имеется более одного метода-члена, являющегося как доступным, так и применимым для вызова метода, необходимо выбрать один из них для предоставления дескриптора для диспетчеризации метода времени выполнения. Язык программирования Java использует правило, в соответствии с которым выбирается *наиболее подходящий* метод.

Неформально интуитивно ясно, что один метод подходит более другого, если любой вызов, обработанный первым методом, может быть передан другому без ошибок времени компиляции. В таких случаях, как аргумент явно типизированного лямбда-выражения (§15.27.1) или вызов переменной арности (§15.12.2.4), разрешена определенная гибкость при адаптации одной сигнатуры к другой.

Один применимый метод m_1 *более подходящий* для вызова с выражениями аргументов e_1, ..., e_k, чем другой применимый метод m_2, если выполняется любое из перечисленных ниже условий.

- m_2 является обобщенным и m_1 является более подходящим, чем m_2, для выражений аргументов e_1, ..., e_k согласно выводу, описанному в §18.5.4.
- m_2 не является обобщенным, m_1 и m_2 применимы строгим или нестрогим вызовом, m_1 имеет типы формальных параметров S_1, ..., S_n, а m_2 имеет типы формальных параметров T_1, ..., T_n и тип S_i является *более подходящим*, чем T_i, для аргумента e_i для всех i ($1 \leq i \leq n$, $n = k$).
- m_2 не является обобщенным, m_1 и m_2 применимы вызовом переменной арности, первыми k типами параметров переменной арности m_1 являются S_1, ..., S_k, а первыми k типами параметров переменной арности m_2 являются T_1, ..., T_k и тип S_i является *более подходящим*, чем T_i, для аргумента e_i для всех i ($1 \leq i \leq k$). Кроме того, если m_2 имеет $k+1$ параметр, то тип $k+1$-го параметра переменной арности m_1 является подтипом $k+1$-го параметра переменной арности m_2.

Вышеуказанные условия являются единственными обстоятельствами, при которых один метод может быть более подходящим, чем другой.

Тип S *более подходящий*, чем тип T, для *любого* выражения, если $S <: T$ (§4.10).

Тип функционального интерфейса S *более подходящий*, чем тип функционального интерфейса T, для выражения `e`, если T не является подтипом S и справедливо одно из приведенных далее условий (здесь $U_1 \ldots Uk$ и R_1 представляют собой типы параметров и возвращаемый тип типа функции фиксации S, а $V_1 \ldots V_k$ и R_2 являются типами параметров и возвращаемым типом типа функции T).

- Если `e` представляет собой явно типизированное лямбда-выражение (§15.27.1), то верно одно из следующих утверждений.
  - R_2 представляет собой `void`.
  - $R_1 <: R_2$.
  - R_1 и R_2 представляют собой типы функциональных интерфейсов и R_1 более подходящий, чем R_2, для каждого результирующего выражения `e`.

    Результирующее выражение лямбда-выражения с блоком в качестве тела определено в §15.27.2; результирующее выражение лямбда-выражения с телом-выражением представляет собой просто само это тело.
  - R_1 является примитивным типом, R_2 — ссылочным типом, а каждое результирующее выражение `e` является автономным выражением (§15.2) примитивного типа.
  - R_1 является ссылочным типом, R_2 — примитивным типом, а каждое результирующее выражение `e` является либо автономным выражением ссылочного типа, либо поливыражением.
- Если `e` представляет собой выражение точной ссылки на метод (§15.13.1), то i) для всех i ($1 \le i \le k$) U_i совпадает с V_i, и ii) верно одно из следующих утверждений.
  - R_2 представляет собой `void`.
  - $R_1 <: R_2$.
  - R_1 является примитивным типом, R_2 является ссылочным типом, а объявление времени компиляции ссылки на метод имеет возвращаемый тип, являющийся примитивным.
  - R_1 является ссылочным типом, R_2 является примитивным типом, а объявление времени компиляции ссылки на метод имеет возвращаемый тип, являющийся ссылочным.
- Если `e` представляет собой выражение в скобках, то одно из этих условий применяется рекурсивно к содержащемуся в нем выражению.
- Если `e` представляет собой условное выражение, то одно из этих условий применяется рекурсивно ко второму и третьему операндам.

Метод m_1 *строго более подходящий*, чем другой метод m_2, тогда и только тогда, когда m_1 более подходящий, чем m_2, а m_2 не является боле подходящим, чем m_1.

Метод называется *максимально подходящим* для вызова метода, если он доступен, применим и не имеется других доступных и применимых методов, которые при этом являются строго более подходящими.

Если имеется только один максимально подходящий метод, то этот метод фактически и является *наиболее подходящим*; он обязательно должен быть более подходящим, чем любой другой доступный метод, являющийся применимым. Этот метод затем подвергается другим проверкам времени компиляции, описанным в §15.12.3.

Возможно, что наиболее подходящего метода нет, поскольку два или более методов являются максимально подходящими. В таком случае выполняются следующие действия.

- Если все максимально подходящие методы имеют эквивалентные в плане перекрытия (§8.4.2) сигнатуры, то:
  - ✦ если ровно один из максимально подходящих методов является конкретным (не объявлен как `abstract`), он является и наиболее подходящим;
  - ✦ в противном случае, если все максимально подходящие методы являются абстрактными либо методами по умолчанию и если сигнатуры всех максимально подходящих методов имеют одно и то же затирание (§4.6), то наиболее подходящий метод выбирается произвольным образом среди подмножества максимально подходящих методов, которые имеют наиболее подходящий возвращаемый тип.

    В этом случае наиболее подходящий метод рассматривается как абстрактный. Кроме того, наиболее подходящий метод рассматривается как генерирующий проверяемое исключение тогда и только тогда, когда исключение или его затирание объявлено в конструкциях `throws` каждого из максимально подходящих методов.
- В противном случае мы говорим, что вызов метода *неоднозначен*, и генерируется ошибка времени компиляции.

§15.12.2.6. Тип вызова метода

Тип вызова наиболее подходящего доступного и применимого метода представляет собой тип метода (§8.2), выражающий целевые типы аргументов вызова, результат вызова (возвращаемый тип или `void`) и типы исключений вызова. Он определяется следующим образом.

- Если выбранный метод является обобщенным и вызов метода не предоставляет явных аргументов типа, тип вызова выводится так, как описано в §18.5.2.
- Если выбранный метод является обобщенным и вызов метода предоставляет явные аргументы типа, то пусть P_i — параметры типа метода и пусть T_i — явные аргументы типа, предоставленные вызовом метода ($1 \le i \le p$). Тогда:
  - ✦ если непроверяемое преобразование было необходимо для того, чтобы метод был применим, типы параметров типа вызова получаются путем применения подстановки $[P_1:=T_1, ..., P_p:=T_p]$ к типам параметров типа метода, а возвращаемый тип и типы исключений получаются затиранием возвращаемого типа и типов исключений типа метода;

  - ✦ если непроверяемое преобразование не было необходимо для того, чтобы метод был применим, то тип вызова получается путем применения подстановки $[P_1:=T_1, ..., P_p:=T_p]$ к типу метода.
- Если выбранный метод не является обобщенным, то:
  - ✦ если непроверяемое преобразование было необходимо для того, чтобы метод был применим, типы параметров типа вызова представляют собой типы параметров типа метода, а возвращаемый тип и типы исключений получаются затиранием возвращаемого типа и типов исключений типа метода;
  - ✦ в противном случае, если выбранный метод представляет собой метод `getClass` класса `Object` (§4.3.2), тип вызова совпадает с типом метода, за исключением того, что возвращаемый тип представляет собой `Class<? extends` *T*`>`, где *T* — тип, поиск которого выполнялся, как описано в §15.12.1;
  - ✦ в противном случае тип вызова совпадает с типом метода.

§15.12.3. Этап 3 времени компиляции: подходит ли выбранный метод

Если для вызова метода имеется наиболее подходящее объявление метода, оно называется *объявлением времени компиляции* вызова метода.

Если аргумент вызова метода не совместим с его целевым типом, выведенным из типа вызова объявления времени компиляции, генерируется ошибка времени компиляции.

Если объявление времени компиляции применимо вызовом переменной арности, то, когда тип последнего формального параметра типа вызова метода представляет собой $F_n[]$, в случае, если тип, который представляет собой затирание F_n, не доступен в точке вызова, генерируется ошибка времени компиляции.

Если объявление времени компиляции представляет собой `void`, то вызов метода должен быть выражением верхнего уровня (т.е. *Expression* в инструкции выражения или в части *ForInit* или *ForUpdate* инструкции `for`), иначе генерируется ошибка времени компиляции. Такой вызов метода не производит значения и, таким образом, может использоваться только в ситуации, когда значение не требуется.

Кроме того, суждение о том, является ли подходящим объявление времени компиляции, может зависеть от вида выражения вызова метода перед левой скобкой.

- Если этот вид представляет собой *MethodName* — т.е. простой *Identifier* — и объявление времени компиляции является методом экземпляра, то справедливо следующее.
  - ✦ Если вызов метода находится в статическом контексте (§8.1.3), генерируется ошибка времени компиляции.
  - ✦ В противном случае пусть *C* представляет собой непосредственно охватывающий класс, членом которого является объявление времени компиляции. Если вызов метода не является непосредственно охватываемым классом *C* или внутренним классом класса *C*, генерируется ошибка времени компиляции.

- Если вызов имеет вид *TypeName* . [*TypeArguments*] *Identifier*, то объявление времени компиляции должно иметь модификатор `static`, иначе генерируется ошибка времени компиляции.
- Если вызов имеет вид *ExpressionName* . *[TypeArguments] Identifier* или *Primary* . *[Type Arguments] Identifier*, то объявление времени компиляции не должно быть статическим методом, объявленным в интерфейсе, иначе генерируется ошибка времени компиляции.
- Если вызов имеет вид `super` . *[TypeArguments] Identifier*, то:
  - ✦ если объявление времени компиляции абстрактное, генерируется ошибка времени компиляции;
  - ✦ если вызов метода находится в статическом контексте, генерируется ошибка времени компиляции.
- Если вызов имеет вид *TypeName* . `super` . *[TypeArguments] Identifier*, то:
  - ✦ если объявление времени компиляции абстрактное, генерируется ошибка времени компиляции;
  - ✦ если вызов метода находится в статическом контексте, генерируется ошибка времени компиляции;
  - ✦ если *TypeName* обозначает класс C, то, если вызов метода не является непосредственно охватываемым классом C или внутренним классом класса C, генерируется ошибка времени компиляции;
  - ✦ если *TypeName* обозначает интерфейс, то пусть T — объявление типа, непосредственно охватывающее вызов метода. Если имеется метод, отличный от объявления времени компиляции, который перекрывает (§9.4.1) объявление времени компиляции в непосредственном суперклассе или суперинтерфейсе T, то генерируется ошибка времени компиляции.

В случае, когда интерфейс перекрывает метод, объявленный в интерфейсе-"деде" (т.е. в суперинтерфейсе суперинтерфейса), это правило не дает дочернему интерфейсу "опустить" перекрытие, просто добавив "деда" в свой список непосредственных суперинтерфейсов. Соответствующий способ доступа к функциональности "деда" —использовать непосредственный суперинтерфейс, и только если этот суперинтерфейс обеспечивает доступ к искомому поведению. (В качестве альтернативы разработчик может определить собственный дополнительный суперинтерфейс, который демонстрирует желаемое поведение с помощью вызова метода `super`.)

Типы параметров времени компиляции и *результат времени компиляции* определяются следующим образом.

- Если объявление времени компиляции вызова метода *не* является полиморфным в смысле сигнатуры методом, то типами параметров времени компиляции являются типы формальных параметров объявлений времени компиляции, а результат времени компиляции представляет собой результат, выбранный для объявления времени компиляции (§15.12.2.6).

- Если объявление времени компиляции вызова метода является полиморфным в смысле сигнатуры методом, то выполняется следующее.
  - ✦ Типы параметров времени компиляции являются статическими типами фактических выражений аргументов. Выражение аргумента, представляющее собой литерал `null` (§3.10.7), рассматривается как имеющее статический тип `void`.
  - ✦ Результат времени компиляции определяется следующим образом.
    - — Если выражение вызова метода является инструкцией выражения, результат времени компиляции представляет собой `void`.
    - — В противном случае, если выражение вызова метода является операндом выражения приведения (§15.16), результат времени компиляции представляет собой затирание типа выражения приведения (§4.6).
    - — В противном случае результат времени компиляции является объявленным возвращаемым типом полиморфного в смысле сигнатуры метода, `Object`.

Метод является *полиморфным в смысле сигнатуры* тогда, когда выполняются все приведенные далее условия.

- Он объявлен в классе `java.lang.invoke.MethodHandle`.
- Он принимает единственный параметр переменной арности (§8.4.1), объявленный тип которого — `Object[]`.
- Он имеет возвращаемый тип `Object`.
- Он объявлен как `native`.

> **В Java SE 8 единственными полиморфными в смысле сигнатуры методами являются методы `invoke` и `invokeExact` класса `java.lang.invoke.MethodHandle`.**

Затем с вызовом метода для использования во время выполнения программы связывается следующая информация времени компиляции.

- Имя метода.
- Квалифицированный тип вызова метода (§13.1).
- Количество параметров и типы параметров в порядке их указания.
- Результат времени компиляции, или `void`.
- Режим вызова, вычисленный следующим образом.
  - ✦ Если квалифицированный тип объявления метода является классом, то:
    - — если объявление времени компиляции имеет модификатор `static`, то режим вызова — `static`;
    - — в противном случае, если объявление времени компиляции имеет модификатор `private`, режим вызова — `nonvirtual`;
    - — в противном случае, если часть вызова метода до левой скобки имеет вид `super .` *Identifier* или вид *TypeName* `. super .` *Identifier*, режим вызова — `super`;
    - — в противном случае режим вызова — `virtual`.

- ✦ Если квалифицированный тип вызова метода является интерфейсом, то режим вызова — `interface`.

Если результат типа вызова объявления времени компиляции — не `void`, то тип выражения вызова метода получается путем применения преобразования при фиксации (§5.1.10) к возвращаемому типу типа вызова объявления времени компиляции.

§15.12.4. Вычисление вызова метода времени выполнения

Вызов метода во время выполнения требует осуществления пяти шагов. На первом шаге может быть вычислена *целевая ссылка*. На втором вычисляются выражения аргументов. На третьем проверяется доступность вызываемого метода. На четвертом шаге выясняется местонахождение фактического кода выполняемого метода. Наконец на пятом шаге создается новый кадр активации, при необходимости осуществляется синхронизация и управление передается коду метода.

§15.12.4.1. Вычисление (при необходимости) целевой ссылки

Имеется шесть случаев, которые следует рассмотреть, в зависимости от вида вызова метода.

- Если вызов метода имеет вид *MethodName* — т.е. просто *Identifier* — то:
  - ✦ если режим вызова — `static`, целевой ссылки нет;
  - ✦ в противном случае пусть T является объявлением охватывающего типа, членом которого является метод, и пусть n — целое число, такое, что T представляет собой n-е лексически охватывающее объявление типа для класса, объявление которого непосредственно содержит вызов метода. Тогда целевая ссылка представляет собой n-й лексически охватывающий экземпляр `this`.

    Если n-й лексически охватывающий экземпляр `this` не существует, генерируется ошибка времени компиляции.
- Если вызов метода имеет вид *TypeName* . *[TypeArguments] Identifier*, то целевой ссылки не существует.
- Если вызов метода имеет вид *ExpressionName* . *[TypeArguments] Identifier*, то:
  - ✦ если режим вызова — `static`, целевой ссылки не существует. Вычисляется выражение *ExpressionName*, но результат вычисления игнорируется;
  - ✦ в противном случае целевая ссылка представляет собой значение, обозначаемое нетерминалом *ExpressionName*.
- Если вызов метода имеет вид *Primary* . *[TypeArguments] Identifier*, то:
  - ✦ если режим вызова — `static`, целевой ссылки нет. Выполняется вычисление выражения *Primary*, но вычисленный результат игнорируется;
  - ✦ в противном случае выполняется вычисление выражения *Primary*, и полученный результат используется в качестве целевой ссылки.

В любом случае, если вычисление выражения *Primary* завершается преждевременно, то никакая часть никакого выражения аргумента не вычисляется, и вызов метода завершается преждевременно по той же причине.

- Если вызов метода имеет вид `super` . *[TypeArguments] Identifier*, то целевая ссылка имеет значение `this`.
- Если вызов метода имеет вид *TypeName* . `super` . *[TypeArguments] Identifier*, то если *TypeName* обозначает класс, то целевая ссылка имеет значение *TypeName*`.this`; в противном случае целевая ссылка имеет значение `this`.

Пример 15.12.4.1-1. Целевые ссылки и статические методы

Когда целевая ссылка вычисляется и игнорируется из-за режима вызова `static`, проверка ссылки на равенство `null` не выполняется.

```
class Test1 {
    static void mountain() {
        System.out.println("Monadnock");
    }
    static Test1 favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        favorite().mountain();
    }
}
```

Вывод программы имеет вид

```
Mount Monadnock
```

Хотя здесь `favorite()` возвращает `null`, исключение `NullPointerException` не генерируется.

Пример 15.12.4.1-2. Порядок вычислений при вызове метода

В качестве части вызова метода экземпляра (§15.12) вычисляется выражение, обозначающее объект. Это выражение полностью вычисляется до вычисления любой части любого выражения аргумента вызова метода.

```
class Test2 {
    public static void main(String[] args) {
        String s = "one";
        if (s.startsWith(s = "two"))
            System.out.println("oops");
    }
}
```

В этом примере первым вычисляется `s` перед `.startsWith`, до выражения аргумента `s = "two"`. Следовательно, в качестве целевой ссылки запоминается ссылка на строку `"one"` перед тем, как локальная переменная `s` изменяется и начинает ссылаться на строку `"two"`. В результате метод `startsWith` вызывается для

> целевого объекта `"one"` с аргументом `"two"`, так что результатом вызова является значение `false`, поскольку строка `"one"` не начинается со строки `"two"`. Отсюда следует, что тестовая программа не выведет `"oops"`.

§15.12.14.2. Вычисление аргументов

Процесс вычисления списка аргументов различен для метода с фиксированной и с переменной арностью (§8.4.1).

Если вызываемый метод является методом с переменной арностью m, он обязательно имеет $n > 0$ формальных параметров. Последний формальный параметр m обязательно имеет тип T`[]` для некоторого T и m обязательно вызывается с $k \geq 0$ фактическими выражениями аргументов.

Если m вызывается с $k \neq n$ фактическими выражениями аргументов или если m вызывается с $k = n$ фактическими выражениями аргументов и типом k-го выражения аргумента, несовместимого по присваиванию с T`[]`, то список аргументов (e_1, ..., e_{n-1}, e_n, ..., e_k) вычисляется, как если бы он был записан как (e_1, ..., e_{n-1}, `new` $|T[]|${ e_n, ..., e_k }), где $|T[]|$ обозначает затирание (§4.6) T`[]`.

> В предыдущих абзацах описывается обработка взаимодействия параметризованных типов и типов массивов в виртуальной машине Java с затертыми обобщенными типами. А именно: если тип элемента T параметра переменного массива недоступен во время выполнения, например, `List<String>`, то следует принять специальные меры касательно выражения создания массива (§15.10), поскольку тип элемента созданного массива должен быть доступен во время выполнения. Путем затирания типа массива последнего выражения списка аргументов мы гарантируем получение типа элемента, доступного во время выполнения. Затем, поскольку выражение создания массива находится в контексте вызова (§5.3), возможно непроверяемое преобразование (§5.1.9) из типа массива с типом элемента, доступным во время выполнения, в тип массива с типом элемента, недоступным во время выполнения, в частности — параметра переменной арности. От компилятора Java при таком преобразовании требуется выдать предупреждение времени компиляции о непроверяемом типе. Эталонная реализация компилятора Java от Oracle выдает в этом случае более информативное предупреждение о *создании непроверяемого обобщенного массива*.

Теперь выражения аргументов (возможно, переписанные, как описано выше) вычисляются и дают *значения аргументов*. Каждое значение аргумента соответствует ровно одному из n формальных параметров метода.

Выражения аргументов, если таковые имеются, вычисляются по порядку слева направо. Если вычисление любого выражения аргумента завершается преждевременно, то ни одна часть ни одного аргумента справа от него не вычисляется, и вызов метода завершается преждевременно по той же причине. Результат вычисления j-го выражения аргумента представляет собой значение j-го аргумента для $1 \leq j \leq n$. Вычисление после этого продолжается с использованием этих значений аргументов, как описано ниже.

§15.12.4.3. Проверка доступности типа и метода

Пусть C — класс, содержащий вызов метода, пусть T — квалифицированный тип вызова метода (§13.1) и пусть m — имя метода, определенное во время компиляции (§15.12.3).

Реализация языка программирования Java должна убедиться (как часть связывания), что метод *m* имеется в типе *T*. Если это не так, генерируется исключение `NoSuchMethodError` (которое является подклассом `IncompatibleClassChangeError`).

Если режим вызова — `interface`, то реализация должна также проверить, что тип целевой ссылки реализует указанный интерфейс. Если тип целевой ссылки интерфейс не реализует, генерируется исключение `IncompatibleClassChangeError`.

Реализация также должна убедиться в процессе связывания, что тип *T* и метод *m* доступны.

- Для типа *T*
  - ✦ Если *T* находится в том же пакете, что и *C*, то *T* доступен.
  - ✦ Если *T* находится в пакете, отличном от пакета *C*, и *T* объявлен как `public`, то *T* доступен.
  - ✦ Если *T* находится в пакете, отличном от пакета *C*, и *T* объявлен как `protected`, то *T* доступен тогда и только тогда, когда *C* является подклассом *T*.
- Для метода *m*
  - ✦ Если *m* объявлен как `public`, то *m* доступен. (Все члены интерфейсов являются `public` (§9.2).)
  - ✦ Если *m* объявлен как `protected`, то *m* доступен тогда и только тогда, когда либо *T* находится в том же пакете, что и *C*, либо *C* представляет собой *T* или подкласс *T*.
  - ✦ Если *m* имеет доступ по умолчанию (пакета), то *m* доступен тогда и только тогда, когда *T* находится в том же пакете, что и *C*.
  - ✦ Если *m* объявлен как `private`, то *m* доступен тогда и только тогда, когда *C* представляет собой *T* или *C* охватывает *T*, или *T* охватывает *C*, или *T* и *C* оба охватывают третий класс.

Если либо *T*, либо *m* недоступно, генерируется исключение `IllegalAccessError` (§12.3).

§15.12.4.4. Выяснение местонахождения метода

Стратегия поиска метода зависит от режима вызова.

Если режим вызова — `static`, целевая ссылка не требуется и перекрытие не разрешено. Вызывается метод *m* класса *T*.

В противном случае вызывается метод экземпляра и имеется целевая ссылка. Если целевая ссылка — `null`, генерируется исключение `NullPointerException`. В противном случае целевая ссылка ссылается на *целевой объект* и используется в качестве значения ключевого слова `this` в вызываемом методе. Далее будут рассмотрены четыре других варианта режима вызова.

Если режим вызова — `nonvirtual`, перекрытие не разрешено. Вызывается метод *m* класса *T*.

В противном случае, если режим вызова — `virtual`, и *T*, и *m* вместе указывают сигнатуру полиморфного метода (§15.12.3), целевым объектом является экземпляр `java.lang.invoke.MethodHandle`. Дескриптор метода инкапсулирует *тип*, соответствует информации, связанной с вызовом метода во время компиляции (§15.12.3). Детали этого

соответствия можно найти в *The Java Virtual Machine Specification, Java SE 8 Edition* и документации к API платформы Java SE. Если соответствие успешно, немедленно и непосредственно вызывается *целевой метод*, инкапсулированный дескриптором метода, а *процедура из §15.12.4.5 не выполняется*.

В противном случае режимом вызова является `interface`, `virtual` или `super` и при этом возможно перекрытие. В данной ситуации используется *динамический поиск метода*. Процесс динамического поиска метода начинается с класса *S*, определяемого следующим образом.

- Если режим вызова — `interface` или `virtual`, то изначально *S* является фактическим классом времени выполнения *R* целевого объекта.

  > Это справедливо, даже если целевой объект является экземпляром массива. (Обратите внимание, что для режима вызова `interface` *R* с необходимостью реализует *T*; в случае режима вызова `virtual` *R* с необходимостью является либо *T*, либо подклассом *T*.)

- Если режимом вызова является `super`, то *S* изначально является квалифицирующим типом (§13.1) вызова метода.

Динамический поиск метода использует для поиска в классе *S* (а при необходимости и в суперклассах и суперинтерфейсах класса *S*) метода *m* описанную ниже процедуру.

Пусть *X* является типом времени компиляции целевой ссылки вызова метода.

1. Если класс *S* содержит объявление метода с именем *m* с тем же дескриптором (тем же количеством параметров, теми же типами параметров и тем же возвращаемым типом), требуемым определенным во время компиляции вызовом метода (§15.12.3), то выполняется следующее.

   - Если режим вызова — `super` или `interface`, то это и есть вызываемый метод, и процедура завершает работу.
   - Если режим вызова — `virtual`, а объявление в *S* перекрывает (§8.4.8.1) *X*.*m*, то метод, объявленный в *S*, и есть вызываемый метод, и процедура завершает работу.

2. В противном случае, если *S* имеет суперкласс, шаги 1 и 2 этой же процедуры поиска применяются рекурсивно с использованием вместо *S* непосредственного суперкласса *S*; вызываемый метод, если таковой имеется, представляет собой результат рекурсивного вызова процедуры поиска.

3. Если на двух предыдущих шагах метод не найден, поиск подходящего метода продолжается в суперинтерфейсе *S*.

Множество методов-кандидатов рассматривается с использованием следующих свойств: i) каждый метод объявлен в (непосредственном или косвенном) суперинтерфейсе *S*; ii) каждый метод имеет имя и дескриптор, требуемые вызовом метода; iii) все методы не являются статическими; iv) для каждого метода, объявляющий интерфейс которого — *I*, не имеется других методов, удовлетворяющих свойствам i–iii, объявленных в подынтерфейсе *I*.

Если это множество содержит метод по умолчанию, этот метод является вызываемым. В противном случае в качестве вызываемого метода из множества выбирается абстрактный метод.

Динамический поиск метода может привести к следующим ошибкам.

- Если вызываемый метод является абстрактным, генерируется исключение `AbstractMethodError`.
- Если вызываемый метод является методом по умолчанию и если в множестве кандидатов на описанном выше шаге 3 имеется более одного метода по умолчанию, генерируется исключение `IncompatibleClassChangeError`.
- Если режим вызова — `interface`, а выбранный метод не является `public`, генерируется исключение `IllegalAccessError`.

Описанная выше процедура (если она завершается без ошибок) находит доступный, не являющийся абстрактным метод для вызова при условии, что все классы и интерфейсы в программе согласованно скомпилированы. Однако, если это не так, могут возникнуть различные ошибки. Спецификация поведения виртуальной машины Java при этих условиях приведена в *The Java Virtual Machine Specification, Java SE 8 Edition*.

Заметим, что процесс динамического поиска, явно описанный здесь, зачастую реализуется неявно, например как побочный результат создания и использования таблиц диспетчеризации методов для каждого класса или создания других структур для каждого класса, используемых для эффективной диспетчеризации.

Пример 15.12.4.4-1. Перекрытие и вызов метода

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
            clear();
    }
    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }
}
class ColoredPoint extends Point {
    int color;
    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }
}
```

Здесь подкласс `ColoredPoint` расширяет абстракцию (здесь — общий метод) `clear`, определенную в суперклассе `Point`. Это делается с помощью перекрытия метода `clear` собственным методом, который вызывает метод `clear` суперкласса, с использованием вызова вида `super.clear()`.

Затем этот метод вызывается всякий раз, когда целевым объектом вызова `clear` является `ColoredPoint`. Даже метод `move` в классе `Point` вызывает метод `clear` класса `ColoredPoint` в случае, когда классом `this` является `ColoredPoint`, что и демонстрирует приведенная далее тестовая программа.

```
class Test1 {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20):");
        p.move(20, 20);
        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20):");
        cp.move(20, 20);
        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:");
        p.move(20, 20);
    }
}
```

Ее вывод имеет вид

```
p.move(20,20):
        Point clear
cp.move(20,20):
        ColoredPoint clear
        Point clear
p.move(20,20), p colored:
        ColoredPoint clear
        Point clear
```

Перекрытие иногда называют "позднее связывание для собственных ссылок" (late-bound self-reference); в данном примере это означает, что ссылка на `clear` в теле `Point.move` (которая в действительности является синтаксическим сокращением для `this.clear`) вызывает метод, выбранный "поздно" (во время выполнения программы, на основе класса времени выполнения объекта, на который указывает `this`), а не "рано" (во время компиляции, на основании только типа `this`). Этим программисту предоставляется мощная возможность расширения абстракций, а сама описанная технология является ключевой идеей объектно-ориентированного программирования.

Пример 15.12.4.4-2. Вызов метода с использованием ключевого слова `super`

Обратиться к перекрытому методу экземпляра суперкласса можно с помощью ключевого слова `super`, которое используется для обращения к членам непосредственного суперкласса, минуя любые перекрывающие объявления в классе, содержащем вызов метода.

При доступе к переменной экземпляра `super` означает то же, что и приведение `this` (§15.11.2), но эта эквивалентность не выполняется для вызовов методов. Это демонстрируется приведенным ниже примером.

```
class T1 {
    String s() { return "1"; }
```

```
}
class T2 extends T1 {
    String s() { return "2"; }
}
class T3 extends T2 {
    String s() { return "3"; }
    void test() {
        System.out.println("s()=\t\t" + s());
        System.out.println("super.s()=\t" + super.s());
        System.out.println("((T2)this).s()=\t" + ((T2)this).s());
        System.out.println("((T1)this).s()=\t" + ((T1)this).s());
    }
}
class Test2 {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}
```

Вывод данной программы имеет вид

```
s()=            3
super.s()=      2
((T2)this).s()= 3
((T1)this).s()= 3
```

Приведение к типам `T1` и `T2` не изменяет вызываемый метод, поскольку вызываемый метод экземпляра выбирается в соответствии с классом времени выполнения объекта, на который указывает `this`. Приведение не изменяет класс объекта; это только проверка того, что класс совместим с определенным типом.

§15.12.4.5. Создание кадра, синхронизация, передача управления

Итак, некоторый метод *m* в некотором классе *S* идентифицирован как метод, который должен быть вызван.

Теперь создается новый *кадр активации*, содержащий целевую ссылку (если таковая имеется), значения аргументов (если таковые имеются) и достаточное количество памяти для локальных переменных и стека для вызываемого метода, а также дополнительную информацию, которая может понадобиться реализации (указатель на стек, программный счетчик, ссылка на предыдущий кадр активации и т.п.). Если памяти для создания такого кадра активации недостаточно, генерируется исключение `StackOverflowError`.

Вновь созданный кадр активации становится текущим кадром активации. Следствием этого являются присвоение значений аргументов соответствующим вновь созданным переменным параметров метода и доступность целевой ссылки как `this`, если целевая ссылка имеется. Прежде чем каждое значение аргумента будет присвоено соответствующей переменной параметра, он подвергается преобразованию вызова (§5.3), который включает в себя все необходимые преобразования набора значений (§5.1.13).

Если затирание (§4.6) типа вызываемого метода отличается по сигнатуре от затирания типа объявления времени компиляции вызова метода (§15.12.3), то, если любое из значений аргументов представляет собой объект, который не является экземпляром подкласса или подынтерфейса затирания типа соответствующего формального параметра в объявлении времени компиляции вызова метода, генерируется исключение `ClassCastException`.

Если метод *m* является `native`, но необходимый машинный зависящий от реализации бинарный код не загружен или не может быть динамически связан, генерируется исключение `UnsatisfiedLinkError`.

Если метод *m* не является `synchronized`, управление передается телу вызываемого метода *m*.

Если метод *m* является `synchronized`, то объект перед передачей управления должен быть заблокирован. Никакие дальнейшие действия не могут выполняться до тех пор, пока текущий поток не захватит блокировку. Если имеется целевая ссылка, то целевой объект должен быть заблокирован; в противном случае должен быть заблокирован объект `Class` для класса *S*, в котором находится метод *m*. Затем управление передается телу вызываемого метода *m*. Объект автоматически разблокируется по завершении выполнения тела метода независимо от того, нормальное ли это завершение или преждевременное. Поведение блокировки и разблокировки в точности такое же, как если бы тело метода было встроено в инструкцию `synchronized` (§14.19).

Пример 15.12.4.5-1. Сигнатура вызываемого метода имеет затирание, отличное от затирания сигнатуры метода времени компиляции

Рассмотрим объявления

```
abstract class C<T> {
    abstract T id(T x);
}
class D extends C<String> {
    String id(String x) { return x; }
}
```

Теперь рассмотрим вызов

```
C c = new D();
c.id(new Object()); // Генерация исключения ClassCastException
```

Затирание фактически вызываемого метода, `D.id()`, отличается сигнатурой от объявления метода времени компиляции, `C.id()`. Первый принимает аргумент типа `String`, в то время как второй принимает аргумент типа `Object`. Вызов завершается генерацией исключения `ClassCastException` до того, как будет выполнено тело метода.

Такие ситуации могут возникнуть, только если программа приводит к предупреждению о непроверяемом типе времени компиляции (§4.8, §5.1.9, §5.5.2, §8.4.1, §8.4.8.3, §8.4.8.4, §9.4.1.2, §15.12.4.2).

Реализации могут обеспечивать эти семантики путем создания *методов-мостов*. В приведенном выше примере в классе `D` будет создан следующий метод-мост.

```
Object id(Object x) { return id((String) x); }
```

Это именно тот метод, который будет в действительности вызываться виртуальной машиной Java в ответ на показанный выше вызов `c.id(new Object())`, и этот метод будет выполнять приведение и генерировать исключение, как и требуется.

§15.13. Выражения ссылки на метод

Выражение ссылки на метод используется для обозначения вызова метода без реального выполнения этого вызова. Некоторые разновидности выражений ссылки на метод также позволяют рассматривать создание экземпляра класса (§15.9) или создание массива (§15.10), как если бы это были вызовы методов.

MethodReference:
ExpressionName `::` *[TypeArguments] Identifier*
ReferenceType `::` *[TypeArguments] Identifier*
Primary `::` *[TypeArguments] Identifier*
`super` `::` *[TypeArguments] Identifier*
TypeName `.` `super` `::` *[TypeArguments] Identifier*
ClassType `::` *[TypeArguments]* `new`
ArrayType `::` `new`

Если справа от `::` присутствует *TypeArguments*, то, если любой из аргументов типа представляет собой символ подстановки (§4.5.1), генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *ExpressionName* `::` *[TypeArguments] Identifier* или *Primary* `::` *[TypeArguments] Identifier*, то, если тип *ExpressionName* или *Primary* не является ссылочным, генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид `super` `::` *[TypeArguments] Identifier*, то пусть *T* представляет собой объявление типа, непосредственно охватывающее выражение ссылки на метод. Если *T* представляет собой класс `Object` или *T* является интерфейсом, генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, то выполняется следующее.

- Если *TypeName* обозначает класс, *C*, то, если *C* не является классом, лексически охватывающим текущий класс, или если *C* является классом `Object`, генерируется ошибка времени компиляции.
- Если *TypeName* обозначает интерфейс, *I*, то пусть *T* представляет собой объявление типа, непосредственно охватывающее выражение ссылки на метод. Если *I* не является непосредственным суперинтерфейсом *T* или если существует некоторый другой непосредственный суперкласс или непосредственный суперинтерфейс *T*, *J*, такой, что *J* является подтипом *I*, генерируется ошибка времени компиляции.
- Если *TypeName* обозначает переменную типа, то генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид `super` `::` *[TypeArguments] Identifier* или *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, то, если выражение находится в статическом контексте, генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *ClassType* `::` *[TypeArguments]* `new`, то:

- *ClassType* должен обозначать класс, являющийся доступным, не абстрактным и не являющийся типом перечисления, иначе генерируется ошибка времени компиляции;
- если *ClassType* обозначает параметризованный тип (§4.5), то, если любой из его аргументов типа является символом подстановки, генерируется ошибка времени компиляции;
- если *ClassType* обозначает несформированный тип (§4.8), то, если после `::` присутствует *TypeArguments*, генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *ArrayType* `::` `new`, то *ArrayType* должен означать тип, доступный во время выполнения (§4.7), иначе генерируется ошибка времени компиляции.

Целевая ссылка метода экземпляра (§15.12.4.1) может быть предоставлена выражением ссылки на метод с использованием *ExpressionName*, *Primary* или `super` или может быть предоставлена позже, при вызове метода. Непосредственно охватывающий экземпляр нового экземпляра внутреннего класса (§15.9.2) должен быть предоставлен лексически охватывающим `this` экземпляром (§8.1.3).

Если одно и то же имя имеет более чем один метод-член типа или если класс имеет более одного конструктора, соответствующий метод или конструктор выбирается на основе типа функционального интерфейса, целевого для выражения, как определено в §15.13.1.

Если метод или конструктор обобщенный, соответствующие аргументы типа могут быть либо выведены, либо предоставлены явно. Аналогично аргументы типа обобщенного типа, упомянутые в выражении ссылки на метод, могут быть предоставлены явно или выведены.

Выражения ссылки на метод всегда являются поливыражениями (§15.2).

Если выражение ссылки на метод находится в программе в месте, отличном от контекста присваивания (§5.2), контекста вызова (§5.3) и контекста приведения (§5.5), генерируется ошибка времени компиляции.

Вычисление выражения ссылки на метод дает экземпляр типа функционального интерфейса (§9.8). Вычисление ссылки на метод *не* приводит к выполнению соответствующего метода; это может произойти позже, когда будет вызван соответствующий метод функционального интерфейса.

Вот некоторые выражения ссылки на метод, вначале без целевой ссылки, а затем с ней.

```
String::length                  // Метод экземпляра
System::currentTimeMillis       // Статический метод
List<String>::size              // Явные аргументы типа
                                // обобщенного типа
List::size                      // Выведенные аргументы типа
                                // обобщенного типа
int[]::clone
T::tvarMember

System.out::println
"abc"::length
foo[x]::bar
```

```
(test ? list.replaceAll(String::trim) : list) :: iterator
super::toString
```

Вот еще несколько выражений ссылок на методы.

```
String::valueOf       // Требуется разрешение перегрузки
Arrays::sort          // Аргументы типа выводятся из контекста
Arrays::<String>sort // Явные аргументы типа
```

Несколько выражений ссылок на методы, которые представляют отложенное создание объекта или массива.

```
ArrayList<String>::new    // Конструктор параметризованного
типа
ArrayList::new            // Выведенные аргументы типа
                          // обобщенного класса
Foo::<Integer>new         // Явные аргументы типа для
                          // обобщенного конструктора
Bar<String>::<Integer>new // Обобщенный класс,
                          // обобщенный конструктор
Outer.Inner::new          // Конструктор внутреннего класса
int[]::new                // Создание массива
```

Невозможно определить конкретную сигнатуру, соответствующую, например, `Arrays::sort(int[])`. Вместо этого функциональный интерфейс предоставляет типы аргументов, которые используются в качестве входных данных для алгоритма разрешения перегрузки (§15.12.2). Этого должно хватать в подавляющем большинстве случаев; при редкой необходимости для более точного управления можно использовать лямбда-выражения.

Использование синтаксиса аргументов типа в имени класса перед разделителем (`List<String>::size`) приводит к проблеме синтаксического анализа, заключающейся в необходимости различать < как скобку для аргумента типа и < как оператор “меньше”. Теоретически это не хуже, чем допустить аргументы типа в выражениях приведения. Разница в том, что приведение осуществляется, только когда встречается токен (; с добавлением выражений ссылки на метод начало *каждого* выражения потенциально представляет собой параметризованный тип.

§15.13.1. Объявление времени компиляции ссылки на метод

Объявление времени компиляции ссылки на метод представляет собой метод, на который ссылается выражение. В частных случаях объявления времени компиляции фактически не существует, но имеется воображаемый метод, который представляет создание экземпляра класса или создание массива. Выбор объявления времени компиляции зависит от целевого типа функции выражения, так же как объявление времени компиляции вызова метода зависит от аргументов вызова (§15.12).

Поиск объявления времени компиляции отражает процесс вызова метода из §15.12.1 и §15.12.2.

- Сначала определяется искомый тип, т.е. тип, в котором будет осуществляться поиск.

- ✦ Если выражение ссылки на метод имеет вид *ExpressionName* : : *[TypeArguments] Identifier* или *Primary* : : *[TypeArguments] Identifier*, искомым типом является тип выражения, предшествующего токену : :.
- ✦ Если выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Identifier*, искомым типом является результат преобразования при фиксации (§5.1.10), примененного к *ReferenceType*.
- ✦ Если выражение ссылки на метод имеет вид `super` : : *[TypeArguments] Identifier*, искомым типом является тип суперкласса класса, объявление которого содержит ссылку на метод.
- ✦ Если выражение ссылки на метод имеет вид *TypeName* . `super` : : *[TypeArguments] Identifier*, то, если *TypeName* обозначает класс, искомым типом является тип суперкласса именованного класса; в противном случае *TypeName* обозначает интерфейс, а искомым типом является тип соответствующего суперинтерфейса класса или интерфейса, объявление которого содержит ссылку на метод.
- ✦ Для двух остальных видов (включающих : : `new`) метод, на который осуществляется ссылка, является воображаемым и искомого типа нет.

- Затем для данного целевого типа функции с *n* параметрами определяется множество потенциально применимых методов.
  - ✦ Если выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Identifier*, потенциально применимые методы представляют собой методы-члены типа, в котором выполняется поиск, имеющие подходящее имя (задаваемое *Identifier*), доступность, арность (*n* или $n-1$) и арность аргументов типа (получаемую из *[TypeArguments]*), как определено в §15.12.2.1.

    Две различные арности, *n* и $n-1$, рассматриваются для учета возможности, что данный вид выражения может ссылаться как на статический метод, так и на метод экземпляра.

  - ✦ Если выражение ссылки на метод имеет вид *ClassType* : : *[TypeArguments]* `new`, потенциально применимые методы представляют собой множество воображаемых методов, соответствующих конструкторам *ClassType*.

    Если *ClassType* является несформированным типом, но не является нестатическим типом-членом несформированного типа, то кандидаты в воображаемые методы-члены описаны в §15.9.3 для выражения создания экземпляра класса, которое использует <> для сокрытия (пропуска) аргументов типа класса.

    В противном случае кандидатами в воображаемые методы-члены являются конструкторы *ClassType*, рассматриваемые так, как если бы это были методы с возвращаемым типом *ClassType*. Среди этих кандидатов выбираются методы с подходящими доступностью, арностью (*n*) и арностью аргументов типов (получаемой из *[TypeArguments]*), как описано в §15.12.2.1.
  - ✦ Если выражение ссылки на метод имеет вид *ArrayType* : : `new`, рассматривается единственный воображаемый метод. Этот метод имеет единственный параметр типа `int`, возвращает *ArrayType* и не имеет конструкции `throws`. Если $n = 1$, это

единственный применимый метод; в противном случае потенциально применимых методов нет.

- ✦ Для всех прочих видов выражений потенциально применимыми методами являются методы-члены типа, в котором выполняется поиск и которые имеют подходящие имя (задаваемое *Identifier*), доступность, арность (*n*) и арность аргументов типа (получаемую из *[TypeArguments]*), как описано в §15.12.2.1.

- Наконец, если потенциально применимых методов нет, то нет и объявления времени компиляции.

В противном случае для данного целевого типа функции с типами параметров P_1, ..., P_n и множества потенциально применимых методов объявление времени компиляции выбирается следующим образом.

- ✦ Если выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Id entifier*, то выполняются два поиска наиболее подходящего применимого метода. Каждый поиск выполняется так, как указано в разделах с §15.12.2.2 по §15.12.2.5, с приведенными ниже пояснениями. Каждый поиск может дать в результате метод или, в случае ошибки, как указано в разделах с §15.12.2.2 по §15.12.2.5, остаться безрезультатным.

  При первом поиске ссылка на метод рассматривается так, как если бы она была вызовом с выражениями аргументов типов P_1, ..., P_n; аргументы типов, если таковые имеются, задаются выражением ссылки на метод.

  При втором поиске, если P_1, ..., P_n не пусты и P_1 является подтипом *ReferenceType*, выражение ссылки на метод рассматривается так, как если бы оно было выражением вызова метода с выражениями аргументов типов P_2, ..., P_n. Если *ReferenceType* представляет собой несформированный тип и если имеется параметризация этого типа, G<...>, которая представляет собой супертип для P_1, то тип, в котором выполняется поиск, представляет собой результат применения преобразования при фиксации (§5.1.10), примененного к G<...>; в противном случае тип, в котором выполняется поиск, совпадает с типом, в котором выполняется первый поиск. И вновь, аргументы типа, если таковые имеются, задаются выражением ссылки на метод.

  Если первый поиск дает статический метод и в процессе второго поиска не находится нестатического метода, применимого согласно §15.12.2.2, §15.12.2.3 или §15.12.2.4, то объявление времени компиляции представляет собой результат первого поиска.

  В противном случае, если при первом поиске не находится статического метода, применимого согласно §15.12.2.2, §15.12.2.3 или §15.12.2.4, а второй поиск дает нестатический метод, то объявление времени компиляции представляет собой результат второго поиска.

  В противном случае объявления времени компиляции не имеется.

- ✦ Для всех прочих видов выражений ссылки на метод выполняется один поиск наиболее подходящего метода. Поиск выполняется так, как указано в разделах с §15.12.2.2 по §15.12.2.5, с пояснениями, приведенными ниже.

Ссылка на метод рассматривается так, как если бы она была вызовом с выражениями аргументов типов P_1, ..., P_n; аргументы типов, если таковые имеются, задаются выражением ссылки на метод.

Если поиск заканчивается ошибкой, как определено в разделах с §15.12.2.2 по §15.12.2.5, или если наиболее подходящий применимый метод является статическим, объявления времени компиляции не имеется.

В противном случае объявление времени компиляции представляет собой наиболее подходящий применимый метод.

Если выражение ссылки на метод имеет вид *ReferenceType* `::` *[TypeArguments] Identifier*, объявление времени компиляции статическое и *ReferenceType* не является простым или квалифицированным именем (§6.2), генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид `super` `::` *[TypeArguments] Identifier* или *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, и объявление времени компиляции является абстрактным, генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, а *TypeName* обозначает интерфейс и существует метод, отличный от объявления времени компиляции, который перекрывает (§8.4.8, §9.4.1) объявление времени компиляции из непосредственного суперкласса или непосредственного суперинтерфейса типа, объявление которого непосредственно охватывает выражение ссылки на метод, то генерируется ошибка времени компиляции.

Если выражение ссылки на метод имеет вид *ClassType* `::` *[TypeArguments]* `new`, и при определении охватывающего экземпляра для *ClassType*, как определено в §15.9.2 (рассматривая выражение ссылки на метод как если бы это было выражение создания экземпляра класса), генерируется ошибка времени компиляции.

Выражение ссылки на метод вида *ReferenceType* `::` *[TypeArguments] Identifier* может рассматриваться различными способами. Если *Identifier* указывает на метод экземпляра, то неявное лямбда-выражение имеет дополнительный параметр по сравнению с ситуацией, когда *Identifier* указывает на статический метод. Возможна ситуация, когда *ReferenceType* имеет применимые методы обоих видов, так что алгоритм поиска, описанный выше, идентифицирует их по отдельности, поскольку в каждом случае имеются свои типы параметров.

Примером неоднозначности является следующий код.

```
interface Fun<T,R> { R apply(T arg); }
class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    void test() {
        Fun<C, Integer> f1 = C::size;
            // Ошибка: метод экземпляра size()
            // или статический метод size(Object)?
    }
}
```

Эта неоднозначность не может быть разрешена путем предоставления применимого метода экземпляра, который был бы более подходящим, чем применимый статический метод.

```
interface Fun<T,R> { R apply(T arg); }
class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    int size(C arg) { return 0; }
    void test() {
        Fun<C, Integer> f1 = C::size;
            // Ошибка: метод экземпляра size()
            // или статический метод size(Object)?
    }
}
```

Поиск достаточно интеллектуален, чтобы игнорировать неоднозначности, в которых все применимые методы (из обоих поисков) являются методами экземпляров.

```
interface Fun<T,R> { R apply(T arg); }
class C {
    int size() { return 0; }
    int size(Object arg) { return 0; }
    int size(C arg) { return 0; }
    void test() {
        Fun<C, Integer> f1 = C::size;
            // OK: ссылка на метод экземпляра size()
    }
}
```

Для удобства, когда имя обобщенного типа используется для ссылки на метод экземпляра (где первым параметром становится получатель), для определения аргументов типа используется целевой тип. Это облегчает использование чего-то наподобие `Pair::first` вместо `Pair<String,Integer>::first`. Аналогично ссылка на метод наподобие `Pair::new` рассматривается как "бубновое" создание экземпляра (`new Pair<>()`). Поскольку "бубна" неявная, такой вид *не* инстанцирует несформированный тип; фактически не существует способа выразить ссылку на конструктор несформированного типа.

Для некоторых выражений ссылок на методы имеется только одно возможное объявление времени компиляции с единственным возможным типом вызова (§15.12.2.6), независимо от целевого типа функции. Такие выражения ссылок на методы называются *точными* (exact). Выражение ссылки на метод, не являющееся точным, называется *неточным* (inexact).

Выражение ссылки на метод, заканчивающееся *Identifier*, является точным, если оно удовлетворяет всем приведенным далее условиям.

- Если выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Identifier*, то *ReferenceType* не обозначает несформированный тип.

- Тип, в котором выполняется поиск, имеет ровно один метод-член с именем *Identifier*, который доступен для класса или интерфейса, в котором находится выражение ссылки на метод.
- Этот метод не является методом переменной арности (§8.4.1).
- Если этот метод обобщенный (§8.4.4), то выражение ссылки на метод предоставляет *TypeArguments*.

Выражение ссылки на метод вида *ClassType* `::` *[TypeArguments]* `new` является точным, если оно удовлетворяет всем приведенным далее условиям.

- Тип, обозначаемый *ClassType*, не является несформированным или представляет собой нестатический тип-член несформированного типа.
- Тип, обозначаемый *ClassType*, имеет ровно один конструктор, являющийся доступным для класса или интерфейса, в котором находится выражение ссылки на метод.
- Этот конструктор не является конструктором переменной арности.
- Если этот конструктор обобщенный, то выражение ссылки на метод предоставляет *TypeArguments*.

Выражение ссылки на метод вида *ArrayType* `::` `new` всегда точное.

§15.13.2. Тип ссылки на метод

Выражение ссылки на метод совместимо в контексте присваивания, контексте вызова или контексте приведения с целевым типом T, если T является типом функционального интерфейса (§9.8) и выражение *конгруэнтно* с типом функции *базового целевого типа*, производного от T.

Базовый целевой тип является производным от T следующим образом.

- Если T является типом функционального интерфейса, параметризованным с использованием символов подстановки, то базовый целевой тип представляет собой параметризацию без символов подстановки (§9.9) типа T.
- В противном случае базовый целевой тип представляет собой T.

Выражение ссылки на метод *конгруэнтно* типу функции, если выполняются оба приведенные ниже условия.

- Тип функции определяет единственное объявление времени компиляции, соответствующее ссылке.
- Истинно одно из следующий утверждений.
  - ✦ Результатом типа функции является `void`.
  - ✦ Результатом типа функции является R, а результатом применения преобразования при фиксации (§5.1.10) к возвращаемому типу типа вызова (§15.12.2.6) выбранного объявления времени компиляции является R' (где R представляет собой целевой тип, который может использоваться для вывода R'); ни R, ни R' не является `void`, и R' совместим с R в контексте присваивания.

Предупреждение времени компиляции о непроверенных типах выдается тогда, когда для того, чтобы объявление времени компиляции было применимо, требуется непроверяемое преобразование, и это преобразование приводит к предупреждению о непроверенных типах в контексте вызова.

Предупреждение времени компиляции о непроверенных типах выдается тогда, когда для того, чтобы описанный выше возвращаемый тип R' был совместим с возвращаемым типом R типа функции, требуется непроверяемое преобразование, и это преобразование приводит к предупреждению о непроверенных типах в контексте присваивания.

Если выражение ссылки на метод совместимо с целевым типом T, то тип выражения U является базовым целевым типом, выведенным из T.

Если любой класс или интерфейс, упомянутый либо U, либо типом функции U, является недоступным для класса или интерфейса, в котором находится выражение ссылки на метод, генерируется ошибка времени компиляции.

Для каждого нестатического метода-члена m типа U, если тип функции U имеет подсигнатуру сигнатуры m, воображаемый метод, тип метода которого является типом функции U, считается перекрывающим m, и может произойти любая ошибка времени компиляции или предупреждение времени компиляции о непроверенных типах, описанные в §8.4.8.3.

Для каждого проверяемого типа исключения X, перечисленного в списке конструкции `throws` типа вызова объявления времени компиляции, X или суперкласс X должен быть упомянут в конструкции `throws` типа функции U, иначе генерируется ошибка времени компиляции.

> Ключевая идея, стоящая за определением совместимости, заключается в том, что ссылка на метод является совместимой тогда и только тогда, когда совместимо эквивалентное лямбда-выражение `(x,y,z)->exp.<T1,T2>method(x,y,z)`. (Это утверждение неформальное, так как имеются спорные вопросы, затрудняющие или делающие невозможным формальное определение семантики в терминах такой перезаписи.)
>
> Эти правила совместимости предоставляют удобную возможность преобразования из одного функционального интерфейса в другой.
>
> ```
> Task t = () -> System.out.println("hi");
> Runnable r = t::invoke;
> ```
>
> Реализация может быть оптимизирована таким образом, что, когда производный от лямбда-выражения объект передается и преобразуется в различные типы, это не приводит к большому числу уровней логики адаптации вокруг ядра тела лямбда-выражения.
>
> В отличие от лямбда-выражения ссылка на метод может быть конгруэнтна с типом обобщенной функции (т.е. с типом функции, имеющим параметры типа). Это связано с тем, что лямбда-выражение должно быть способно объявлять параметры типов, но никакой синтаксис эту возможность не поддерживает; в то время как в случае ссылки на метод такое объявление не является необходимым. Например, приведенная далее программа является корректной.
>
> ```
> interface ListFactory {
>     <T> List<T> make();
> ```

```
    }

    ListFactory lf = ArrayList::new;
    List<String> ls = lf.make();
    List<Number> ln = lf.make();
```

§15.13.3. Вычисление времени выполнения ссылок на методы

Вычисление выражения ссылки на метод во время выполнения похоже на вычисление выражения создания экземпляра класса, поскольку в результате нормального завершения оно производит ссылку на объект. Вычисление выражения ссылки на метод отличается от вычисления самого метода.

Во-первых, если выражение ссылки на метод начинается с *ExpressionName* или *Primary*, вычисляется это подвыражение. Если результатом вычисления подвыражения является значение `null`, генерируется исключение `NullPointerException`, и вычисление выражения ссылки на метод завершается преждевременно. Если подвыражение завершается преждевременно, выражение ссылки на метод завершается преждевременно по той же причине.

Далее либо для нового экземпляра класса с указанными ниже свойствами выделяется память и выполняется инициализация, либо создается ссылка на существующий экземпляр класса с указанными ниже свойствами. Если должен быть создан новый экземпляр, но памяти для выделения объекту недостаточно, вычисление выражения ссылки на метод завершается преждевременно генерацией исключения `OutOfMemoryError`.

Значение выражения ссылки на метод представляет собой ссылку на экземпляр класса со следующими свойствами.

- Этот класс реализует тип целевого функционального интерфейса и, если целевой тип является типом пересечения, в этом пересечении должны быть упомянуты все прочие типы интерфейсов.
- Если выражение ссылки на метод имеет тип *U*, то для каждого нестатического метода-члена *m* типа *U* выполняется следующее.

  Если тип функции *U* имеет подсигнатуру сигнатуры *m*, класс объявляет *метод вызова* (*вызываемый метод*), который перекрывает *m*. Тело метода вызова вызывает метод, на который указывает ссылка, создает экземпляр класса или массив, как описано ниже. Если результат метода вызова не является `void`, тело возвращает результат вызываемого метода или создания объекта после всех необходимых преобразований присваивания (§5.2).

  Если затирание типа перекрытого метода отличается сигнатурой от затирания типа функции *U*, то перед вызовом метода или созданием объекта тело метода вызова проверяет, является ли значение каждого аргумента экземпляром подкласса или подынтерфейса затирания соответствующего типа параметра в типе функции *U*. Если нет, то генерируется исключение `ClassCastException`.
- Класс не перекрывает другие методы типа функционального интерфейса иди других типов интерфейсов, упомянутых выше, хотя может перекрывать методы класса `Object`.

Тело метода вызова зависит от вида выражения ссылки на метод следующим образом.

- Если выражение ссылки на метод имеет вид *ExpressionName* : : *[TypeArguments] Identifier* или *Primary* : : *[TypeArguments] Identifier*, то тело метода вызова рассматривается как выражение вызова метода для объявления времени компиляции, которое представляет собой объявление времени компиляции выражения ссылки на метод. Вычисление времени выполнения выражения вызова метода определяется в §15.12.4.3–§15.12.4.5, где:
  - ✦ режим вызова выводится из объявления времени компиляции, как определено в §15.12.3;
  - ✦ целевая ссылка представляет собой значение *ExpressionName* или *Primary*, определяемое при вычислении выражения ссылки на метод;
  - ✦ аргументы выражения вызова метода являются формальными параметрами метода вызова.
- Если выражение ссылки на метод имеет вид *ReferenceType* : : *[TypeArguments] Ident ifier*, то тело метода вызова аналогично действует как выражение вызова метода для объявления времени компиляции, которое представляет собой объявление времени компиляции выражения ссылки на метод. Вычисление времени выполнения выражения вызова метода определяется в §15.12.4.3–§15.12.4.5, где:
  - ✦ режим вызова выводится из объявления времени компиляции, как определено в §15.12.3;
  - ✦ если объявление времени компиляции представляет собой метод экземпляра, то целевая ссылка является первым формальным параметром метода вызова; в противном случае целевой ссылки не имеется;
  - ✦ если объявление времени компиляции представляет собой метод экземпляра, то аргументами выражения вызова метода (если таковые имеются) являются второй и последующие формальные параметры метода вызова; в противном случае аргументами выражения вызова метода являются формальные параметры метода вызова.
- Если выражение ссылки на метод имеет вид `super` : : *[TypeArguments] Identifier* или *TypeName* . `super` : : *[TypeArguments] Identifier*, тело метода вызова действует как выражение вызова метода для объявления времени компиляции, которое представляет собой объявление времени компиляции выражения ссылки на метод; вычисление времени выполнения выражения вызова метода определяется в §15.12.4.3–§15.12.4.5, где:
  - ✦ режим вызова — `super`;
  - ✦ если выражение ссылки на метод начинается с нетерминала *TypeName*, который именует класс, целевой ссылкой является значение *TypeName* . `this` в точке, в которой вычисляется ссылка на метод; в противном случае целевая ссылка представляет собой значение `this` в точке, в которой вычисляется ссылка на метод;
  - ✦ аргументами выражения вызова метода являются формальные параметры метода вызова.
- Если выражение ссылки на метод имеет вид *ClassType* : : *[TypeArguments]* `new`, тело метода вызова действует как выражение создания экземпляра класса вида `new` *[TypeA*

rguments] ClassType(A_1, ..., A_n), где аргументы A_1, ..., A_n представляют собой формальные параметры метода вызова и где

- ✦ охватывающий экземпляр для нового объекта, если таковой имеется, выводится из местоположения выражения ссылки на метод, как определено в §15.9.2;
- ✦ вызываемый конструктор представляет собой конструктор, который соответствует объявлению времени компиляции ссылки на метод (§15.13.1).

- Если выражение ссылки на метод имеет вид *Type* `[]`^{k} `::new` ($k \geq 1$), то тело метода вызова действует как выражение создания массива вида `new` *Type* `[`*size*`]` `[]`$^{k-1}$, где *size* представляет собой единственный параметр метода вызова. (Запись `[]`^{k} означает последовательность из k пар квадратных скобок.)

Если тело метода вызова действует как выражение вызова метода, то типы параметров времени компиляции и результат времени компиляции вызова метода определяются так, как определено в §15.12.3. Для целей определения результата времени компиляции выражение вызова метода представляет собой инструкцию выражения, если результатом метода вызова является `void`, и *Expression* инструкции `return`, если результатом метода вызова является не `void`.

Результатом этого определения в случае, когда объявление времени компиляции ссылки на метод полиморфно в смысле сигнатуры, является следующее.

- Типами параметров вызова метода являются типы соответствующих аргументов.
- Вызов метода либо представляет собой `void`, либо имеет возвращаемый тип `Object` в зависимости от того, является ли метод вызова, охватывающий вызываемый метод, `void`-методом или имеет возвращаемый тип.

Хронометраж вычисления выражения ссылки на метод более сложен, чем для лямбда-выражений (§15.27.4). Когда выражение ссылки на метод имеет выражение (а не тип), предшествующее разделителю `::`, подвыражение вычисляется немедленно. Результат вычисления сохраняется до тех пор, пока не будет вызван метод соответствующего типа функционального интерфейса; при этом сохраненный результат используется в качестве целевой ссылки для вызова. Это означает, что выражение, предшествующее разделителю `::`, вычисляется только тогда, когда программа встречает выражение ссылки на метод, и не вычисляется заново при последующих вызовах типа функционального интерфейса.

Интересно сопоставить эту трактовку `null` с трактовкой во время вызова метода. Когда вычисляется выражение вызова метода, возможна ситуация, когда нетерминал *Primary*, квалифицирующий вызов, после вычисления оказывается равным `null`, но при этом исключение `NullPointerException` не генерируется. Это происходит, когда вызываемый метод является статическим (несмотря на синтаксис вызова, предполагающий метод экземпляра). Поскольку применимый метод для выражения ссылки на метод, квалифицированного с помощью *Primary*, не может быть статическим (§15.13.1), вычисление выражения ссылки на метод проще — *Primary* со значением `null` всегда вызывает исключение `NullPointerException`.

§15.14. Постфиксные выражения

Постфиксные выражения используют постфиксные операторы ++ и --. Имена в качестве первичных выражений (§15.8) не рассматриваются, но в грамматике обрабатываются отдельно во избежание некоторых неоднозначностей. Они становятся взаимозаменяемыми только здесь, на уровне приоритетов постфиксных выражений.

PostfixExpression:
Primary
ExpressionName
PostIncrementExpression
PostDecrementExpression

§15.14.1. Имена выражений

Правила вычисления имен выражений приведены в §6.5.6.

§15.14.2. Оператор постфиксного инкремента ++

Постфиксное выражение, за которым следует оператор ++, является постфиксным выражением инкремента.

PostIncrementExpression:
PostfixExpression ++

Результатом постфиксного выражения должна быть переменная типа, преобразуемого (§5.1.8) в числовой тип, иначе генерируется ошибка времени компиляции.

Типом выражения постфиксного инкремента является тип переменной. Результатом выражения постфиксного инкремента является не переменная, а значение.

Во время выполнения программы, если вычисление выражения операнда завершается преждевременно по некоторой причине, вычисление выражения постфиксного инкремента завершается преждевременно по той же причине, и инкремент не выполняется. В противном случае к значению переменной прибавляется значение `1`, и сумма сохраняется в той же переменной. Перед сложением выполняется бинарное числовое повышение (§5.6.2) значения `1` и значения переменной. При необходимости перед сохранением над суммой выполняется сужающее примитивное преобразование (§5.1.3) и/или преобразование упаковки (§5.1.7) в тип переменной. Значением выражения постфиксного инкремента является значение переменной *перед* сохранением нового значения.

> Обратите внимание, что упомянутое выше бинарное числовое повышение может включать преобразование распаковки (§5.1.8) и преобразование набора значений (§5.1.13). При необходимости преобразование набора значений применяется к сумме перед ее сохранением в переменной.

Переменная, объявленная как `final`, не может подвергаться операции инкремента, поскольку, когда доступ к такой `final`-переменной используется в качестве выражения, результатом является значение, а не переменная. Таким образом, ее нельзя использовать как операнд оператора постфиксного инкремента.

§15.14.3. Постфиксный оператор декремента --

Постфиксное выражение, за которым следует оператор --, является постфиксным выражением декремента.

PostDecrementExpression:
PostfixExpression --

Результатом постфиксного выражения должна быть переменная типа, преобразуемого (§5.1.8) в числовой тип, иначе генерируется ошибка времени компиляции.

Типом выражения постфиксного декремента является тип переменной. Результатом выражения постфиксного декремента является не переменная, а значение.

Во время выполнения программы, если вычисление выражения операнда завершается преждевременно по некоторой причине, вычисление выражения постфиксного декремента завершается преждевременно по той же причине, и декремент не выполняется. В противном случае из значения переменной вычитается значение `1`, и разность сохраняется в той же переменной. Перед вычитанием выполняется бинарное числовое повышение (§5.6.2) значения `1` и значения переменной. При необходимости перед сохранением над разностью выполняется сужающее примитивное преобразование (§5.1.3) и/или преобразование упаковки (§5.1.7) в тип переменной. Значением выражения постфиксного декремента является значение переменной *перед* сохранением нового значения.

> Обратите внимание, что упомянутое выше бинарное числовое повышение может включать преобразование распаковки (§5.1.8) и преобразование набора значений (§5.1.13). При необходимости преобразование набора значений применяется к разности перед ее сохранением в переменной.

Переменная, объявленная как `final`, не может подвергаться операции декремента, поскольку, когда доступ к такой `final`-переменной используется в качестве выражения, результатом является значение, а не переменная. Таким образом, ее нельзя использовать как операнд оператора постфиксного декремента.

§15.15. Унарные операторы

Операторы `+`, `-`, `++`, `--`, `~`, `!` и оператор приведения (§15.16) называются *унарными операторами*.

UnaryExpression:
PreIncrementExpression
PreDecrementExpression
`+` *UnaryExpression*
`-` *UnaryExpression*
UnaryExpressionNotPlusMinus

PreIncrementExpression:
`++` *UnaryExpression*

PreDecrementExpression:
`--` *UnaryExpression*

UnaryExpressionNotPlusMinus:
PostfixExpression
`~` *UnaryExpression*
`!` *UnaryExpression*
CastExpression

Далее для удобства приведена продукция из §15.16.

CastExpression:
`(` *PrimitiveType* `)` *UnaryExpression*
`(` *ReferenceType {AdditionalBound}* `)` *UnaryExpressionNotPlusMinus*
`(` *ReferenceType {AdditionalBound}* `)` *LambdaExpression*

Выражения с унарными операторами группируются справа налево, так что `-~x` означает то же, что и `-(~x)`.

Эта часть грамматики содержит некоторые уловки, чтобы избежать двух потенциальных синтаксических неоднозначностей.

Первая потенциальная неоднозначность возникает в выражениях наподобие `(p)+q`, которые выглядят с точки зрения программиста на C или C++ либо как приведение к типу `p` результата применения унарного `+` к `q`, либо как результат сложения двух величин, `p` и `q`. В C и C++ синтаксический анализатор решает эту задачу, выполняя в процессе синтаксического анализа ограниченный семантический анализ, так что ему становится известно, чем является `p` — именем типа или именем переменной.

У Java подход иной. Результат оператора `+` должен быть числовым, а все имена типов, включаемые в приведения числовых значений, являются известными ключевыми словами. Таким образом, если `p` представляет собой ключевое слово, именующее примитивный тип, то `(p)+q` имеет смысл только как приведение унарного выражения. Однако если `p` не является ключевым словом, именующим примитивный тип, то `(p)+q` может иметь смысл только как бинарная арифметическая операция. Аналогичное замечание применимо и к оператору `-`. Показанная выше грамматика разделяет *CastExpression* на две продукции для того, чтобы выявить это различие. Нетерминал *UnaryExpression* включает все унарные операции, но нетерминал *UnaryExpressionNotPlusMinus* исключает все унарные операторы, которые могут одновременно быть бинарными операторами; в языке программирования Java таковыми являются `+` и `-`.

Вторая потенциальная неоднозначность заключается в том, что выражение `(p)++` может, с точки зрения программиста на C или C++, быть либо постфиксным инкрементом выражения в скобках, либо началом приведения, например, в `(p)++q`. Как и ранее, синтаксические анализаторы C и C++ знают, представляет ли собой `p` имя типа или имя переменной. Но синтаксический анализатор, использующий предпросмотр только на один токен и не выполняющий семантического анализа в процессе анализа синтаксического, не в состоянии сказать при предпросмотре токена `++`, следует ли рассматривать `(p)` как выражение *Primary* или оставить его нетронутым для дальнейшего рассмотрения как часть *CastExpression*.

В языке программирования Java результат оператора ++ должен быть числовым, а все имена типов, включаемые в приведения числовых значений, являются известными ключевыми словами. Таким образом, если `p` представляет собой ключевое слово, именующее примитивный тип, то `(p)++` может иметь смысл только как приведение выражения префиксного инкремента, применяемого к операнду, такому как `q`, следующему за ++. Однако если `p` не является ключевым словом, именующим примитивный тип, то `(p)++` может иметь смысл только как постфиксный инкремент `p`. Аналогичное замечание применимо и к оператору `--`. Таким образом, нетерминал *UnaryExpressionNotPlusMinus* исключает также применения префиксных операторов ++ и `--`.

§15.15.1. Оператор префиксного инкремента ++

Унарное выражение, которому предшествует оператор ++, представляет собой выражение префиксного инкремента.

Результатом этого унарного выражения должна быть переменная типа, преобразуемого (§5.1.8) в числовой тип, иначе генерируется ошибка времени компиляции.

Типом выражения префиксного инкремента является тип переменной. Результатом выражения префиксного инкремента является не переменная, а значение.

Во время выполнения программы, если вычисление выражения операнда завершается преждевременно по некоторой причине, вычисление выражения префиксного инкремента завершается преждевременно по той же причине, и инкремент не выполняется. В противном случае к значению переменной прибавляется значение `1`, и сумма сохраняется в той же переменной. Перед сложением выполняется бинарное числовое повышение (§5.6.2) значения `1` и значения переменной. При необходимости перед сохранением над суммой выполняется сужающее примитивное преобразование (§5.1.3) и/или преобразование упаковки (§5.1.7) в тип переменной. Значением выражения префиксного инкремента является значение переменной *после* сохранения нового значения.

> Обратите внимание, что упомянутое выше бинарное числовое повышение может включать преобразование распаковки (§5.1.8) и преобразование набора значений (§5.1.13). При необходимости преобразование набора значений применяется к сумме перед ее сохранением в переменной.

Переменная, объявленная как `final`, не может подвергаться операции инкремента, поскольку, когда доступ к такой `final`-переменной используется в качестве выражения, результатом является значение, а не переменная. Таким образом, ее нельзя использовать как операнд оператора префиксного инкремента.

§15.15.2. Оператор префиксного декремента --

Унарное выражение, которому предшествует оператор `--`, представляет собой выражение префиксного декремента.

Результатом этого унарного выражения должна быть переменная типа, преобразуемого (§5.1.8) в числовой тип, иначе генерируется ошибка времени компиляции.

Типом выражения префиксного декремента является тип переменной. Результатом выражения префиксного декремента является не переменная, а значение.

Во время выполнения программы, если вычисление выражения операнда завершается преждевременно по некоторой причине, вычисление выражения префиксного декремента завершается преждевременно по той же причине, и декремент не выполняется. В противном случае из значения переменной вычитается значение `1`, и разность сохраняется в той же переменной. Перед вычитанием выполняется бинарное числовое повышение (§5.6.2) значения `1` и значения переменной. При необходимости перед сохранением над разностью выполняется сужающее примитивное преобразование (§5.1.3) и/или преобразование упаковки (§5.1.7) в тип переменной. Значением выражения префиксного декремента является значение переменной *после* сохранения нового значения.

> Обратите внимание, что упомянутое выше бинарное числовое повышение может включать преобразование распаковки (§5.1.8) и преобразование набора значений (§5.1.13). При необходимости преобразование набора значений применяется к разности перед ее сохранением в переменной.

Переменная, объявленная как `final`, не может подвергаться операции декремента, поскольку, когда доступ к такой `final`-переменной используется в качестве выражения, результатом является значение, а не переменная. Таким образом, ее нельзя использовать как операнд оператора префиксного декремента.

§15.15.3. Оператор унарного +

Типом операнда выражения оператора унарного + должен быть тип, преобразуемый (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

К операнду применяется унарное числовое повышение (§5.6.1). Типом выражения унарного + является повышенный тип операнда. Результатом выражения унарного + является не переменная, а значение, даже если результат выражения операнда является переменной.

Во время выполнения значением выражения унарного + является повышенное значение операнда.

§15.15.4. Оператор унарного -

Типом операнда выражения оператора унарного − должен быть тип, преобразуемый (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

К операнду применяется унарное числовое повышение (§5.6.1).

Типом выражения унарного − является повышенный тип операнда.

Обратите внимание, что унарное числовое повышение выполняет преобразование набора значений (§5.1.13). Из какого бы набора значений ни выбиралось значение повышаемого операнда, выполняется операция унарного минуса и результат выбирается из того же набора значений. Затем над этим результатом выполняется преобразование набора значений.

Во время выполнения программы значение выражения унарного минуса представляет собой арифметическое изменение знака повышенного значения операнда.

Для целочисленных значений изменение знака эквивалентно вычитанию из нуля. Язык программирования Java использует для представления целых чисел дополнительный код, а так как диапазон значений в дополнительном коде не симметричен, так что изменение знака максимального отрицательного значения `int` или `long` дает то же самое максимальное отрицательное значение. В этом случае возникает переполнение, но исключение не генерируется. Для всех целочисленных значений `x`, `-x` эквивалентно `(~x)+1`.

Для значений с плавающей точкой изменение знака *не* эквивалентно вычитанию из нуля, поскольку если `x` равно `+0.0`, то `0.0-x` равно `+0.0`, но `-x` представляет собой `-0.0`. Унарный минус просто инвертирует знак числа с плавающей точкой. Особые случаи, представляющие интерес, показаны ниже.

- Если операнд — NaN, результат равен NaN. (Вспомним, что значение NaN знака не имеет (§4.2.3).)
- Если операнд равен бесконечности, то результат равен бесконечности с противоположным знаком.
- Если операнд равен нулю, результат равен нулю с противоположным знаком.

§15.15.5. Оператор побитового дополнения ~

Типом операнда выражения унарного оператора ~ должен быть тип, преобразуемый (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

К операнду применяется унарное числовое повышение (§5.6.1). Типом выражения унарного оператора побитового дополнения является повышенный тип операнда.

Во время выполнения значением выражения унарного оператора побитового дополнения является побитовое дополнение повышенного значения операнда. Во всех случаях `~x` эквивалентно `(-x)-1`.

§15.15.6. Оператор логического дополнения !

Типом операнда выражения унарного оператора `!` должен быть `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

Типом выражения унарного логического дополнения является `boolean`.

Во время выполнения операнд при необходимости подвергается преобразованию распаковки (§5.1.8). Значение выражения унарного логического дополнения равно `true`, если (возможно, преобразованное) значение операнда равно `false`, и `false`, если (возможно, преобразованное) значение операнда равно `true`.

§15.16. Выражения приведения

Выражения приведения преобразуют во время выполнения программы значение одного числового типа в аналогичное значение другого числового типа или подтверждает во время компиляции, что типом выражения является `boolean`, или проверяет во время выполнения, что ссылочное значение указывает на объект, класс которого совместим с определенным ссылочным типом или списком ссылочных типов.

Скобки и тип или список типов, который они содержат, иногда называют *оператором приведения*.

CastExpression:
(*PrimitiveType*) *UnaryExpression*
(*ReferenceType {AdditionalBound}*) *UnaryExpressionNotPlusMinus*
(*ReferenceType {AdditionalBound}*) *LambdaExpression*

Далее для удобства приведена продукция из §4.4.

AdditionalBound:
& *InterfaceType*

Если оператор приведения содержит список типов — т.е. *ReferenceType*, за которым следует один или несколько *AdditionalBound*, — то должны выполняться все перечисленные ниже условия, иначе генерируется ошибка времени компиляции.

- *ReferenceType* должен обозначать тип класса или интерфейса.
- Затирания (§4.6) всех перечисленных типов должны быть попарно различны.
- Никакие два перечисленные типа не могут быть подтипами различных параметризаций одного и того же обобщенного интерфейса.

Целевой тип для контекста приведения (§5.5), вводимый выражением приведения, представляет собой либо *PrimitiveType* или *ReferenceType* (если за ним не следуют члены *AdditionalBound*) в операторе приведения, либо тип пересечения, описываемый *ReferenceType* и членами *AdditionalBound*, находящимися в операторе приведения.

Типом выражения приведения является результат применения преобразования при фиксации (§5.1.10) к этому целевому типу.

> Приведение может использоваться для явной пометки лямбда-выражения или выражения ссылки на метод конкретным целевым типом. Для предоставления надлежащего уровня гибкости целевой тип может быть списком типов, описывающим тип пересечения, при условии, что пересечение порождает функциональный интерфейс (§9.8).

Результатом выражения приведения является не переменная, а значение, даже если результат выражения операнда представляет собой переменную.

Оператор приведения не влияет на выбор набора значений (§4.2.3) для значения типа `float` или типа `double`. Следовательно, приведение к типу `float` в выражении, не являющемся FP-строгим (§15.4), не обязательно приводит к преобразованию его значения в элемент набора значений float, а приведение к типу `double` в выражении, не являющемся FP-строгим (§15.4), не обязательно приводит к преобразованию его значения в элемент набора значений double.

Если тип времени компиляции операнда не может быть приведен к типу, указанному в операторе приведения в соответствии с правилами преобразования приведения (§5.5), генерируется ошибка времени компиляции.

В противном случае во время выполнения значение операнда преобразуется (при необходимости) с помощью преобразования приведения в тип, указанный в операторе приведения.

Если во время выполнения обнаруживается недопустимость приведения, генерируется исключение `ClassCastException`.

> Некоторые приведения приводят к ошибкам времени компиляции. Некоторые приведения, как можно доказать во время компиляции, всегда корректны во время выполнения. Например, всегда корректно преобразование значения типа класса в тип его суперкласса; такое приведение не должно требовать выполнения специальных действий во время выполнения. Наконец для некоторых приведений во время компиляции нельзя доказать ни их корректность, ни их некорректность. Такие приведения должны проверяться во время выполнения. Подробно об этом можно прочесть в §5.5.

§15.17. Мультипликативные операторы

Операторы *, / и % называются *мультипликативными операторами.*

MultiplicativeExpression:
UnaryExpression
MultiplicativeExpression * *UnaryExpression*
MultiplicativeExpression / *UnaryExpression*
MultiplicativeExpression % *UnaryExpression*

Все мультипликативные операторы имеют одинаковый приоритет и синтаксически левоассоциативны (группируются слева направо).

Тип каждого операнда мультипликативного оператора должен быть типом, преобразуемым (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

Над операндами выполняется бинарное числовое повышение (§5.6.2).

> Обратите внимание, что бинарное числовое повышение выполняет преобразование набора значений (§5.1.13) и может выполнять преобразование распаковки (§5.1.8).

Типом мультипликативного выражения является повышенный тип его операндов.

Если повышенный тип представляет собой `int` или `long`, применяется целочисленная арифметика.

Если повышенный тип представляет собой `float` или `double`, применяется арифметика с плавающей точкой.

§15.17.1. Оператор умножения *

Бинарный оператор * выполняет умножение, возвращая произведение операндов.

Умножение является коммутативной операцией, если выражения операндов не имеют побочных действий.

Целочисленное умножение ассоциативно, если все операнды имеют один и тот же тип.

Умножение с плавающей точкой ассоциативным не является.

Если происходит переполнение при целочисленном умножении, то результат представляет собой младшие биты математического произведения, представленного в некотором достаточно большом числе в дополнительном формате. В результате при переполнении знак результата может не быть тем же, что и знак математического произведения значений двух операндов.

Результат умножения с плавающей точкой определяется правилами арифметики IEEE 754.

- Если один из операндов представляет собой NaN, результатом является NaN.
- Если результат не является NaN, знак результата положителен, если оба операнда имеют один и тот же знак, и отрицателен, если операнды имеют разные знаки.
- Умножение бесконечности на нуль дает NaN.
- Умножение бесконечности на конечное значение дает знаковую бесконечность. Знак определяется приведенным выше правилом.
- В остальных случаях, когда ни один операнд не является ни бесконечностью, ни NaN, вычисляется точное математическое произведение. Затем выбирается набор значений с плавающей точкой.
  - ✦ Если выражение умножения является FP-строгим (§15.4):
    - — если тип выражения умножения — `float`, должен быть выбран набор значений float;
    - — если тип выражения умножения — `double`, должен быть выбран набор значений double.
  - ✦ Если выражение умножения не является FP-строгим:
    - — если тип выражения умножения — `float`, то может быть выбран как набор значений float, так и расширенный набор значений float; это зависит от конкретной реализации;
    - — если тип выражения умножения — `double`, то может быть выбран как набор значений double, так и расширенный набор значений double; это зависит от конкретной реализации.

Затем из выбранного набора значений выбирается значение, представляющее произведение.

Если величина произведения слишком велика для представления, мы говорим о переполнении при выполнении операции; результатом в этом случае является бесконечность соответствующего знака.

В противном случае произведение округляется до ближайшего значения из выбранного набора значений с использованием режима округления к ближайшему IEEE 754. Язык программирования Java требует поддержки постепенной потери значимости, определенной IEEE 754 (§4.2.4).

Независимо от возможного факта переполнения, потери значимости или потери информации, вычисление оператора умножения * никогда не приводит к генерации исключения времени выполнения.

§15.17.2. Оператор деления /

Бинарный оператор / выполняет деление, давая частное своих операндов. Левый операнд представляет собой *делимое*, а правый — *делитель*.

Результат целочисленного деления округляется по направлению к 0, т.е. частное, полученное для целочисленных после бинарного числового повышения (§5.6.2) операндов *n* и *d*, представляет собой целочисленное значение *q*, абсолютная величина которого — максимально возможная, удовлетворяющая соотношению $|d \cdot q| \leq |n|$. Кроме того, *q* положительно, когда $|n| \geq |d|$ и *n* и *d* имеют одинаковые знаки, и отрицательно, когда $|n| \geq |d|$ и *n* и *d* имеют противоположные знаки.

Имеется один особый случай, не подчиняющийся этому правилу: если делимое является отрицательным целым числом с наибольшим абсолютным значением для своего типа, а делитель равен `-1`, то возникает целочисленное переполнение и результат равен делимому. Несмотря на переполнение исключение в этом случае не генерируется. С другой стороны, если значение делителя при целочисленном делении равно `0`, генерируется исключение `ArithmeticException`.

Результат деления с плавающей точкой определяется правилами арифметики IEEE 754.

- Если один из операндов представляет собой NaN, результатом является NaN.
- Если результат не является NaN, знак результата положителен, если оба операнда имеют один и тот же знак, и отрицателен, если операнды имеют разные знаки.
- Деление бесконечности на бесконечность дает NaN.
- Деление бесконечности на конечное значение дает знаковую бесконечность. Знак определяется приведенным выше правилом.
- Деление конечного значения на бесконечность дает знаковый нуль. Знак определяется приведенным выше правилом.
- Деление нуля на нуль дает NaN; деление нуля на другое конечное значение дает знаковый нуль. Знак определяется приведенным выше правилом.
- Деление ненулевого конечного значения на нуль дает знаковую бесконечность. Знак определяется приведенным выше правилом.
- В остальных случаях, когда ни один операнд не является ни бесконечностью, ни NaN, вычисляется точное математическое частное. Затем выбирается набор значений с плавающей точкой.
  - ✦ Если выражение деления является FP-строгим (§15.4):
    - — если тип выражения деления — `float`, должен быть выбран набор значений float;
    - — если тип выражения деления — `double`, должен быть выбран набор значений double.
  - ✦ Если выражение деления не является FP-строгим:
    - — если тип выражения деления — `float`, то может быть выбран как набор значений float, так и расширенный набор значений float; это зависит от конкретной реализации;
    - — если тип выражения деления — `double`, то может быть выбран как набор значений double, так и расширенный набор значений double; это зависит от конкретной реализации.

Затем из выбранного набора значений выбирается значение, представляющее частное.

Если величина частного слишком велика для представления, мы говорим о переполнении при выполнении операции; результатом в этом случае является бесконечность соответствующего знака.

В противном случае частное округляется до ближайшего значения из выбранного набора значений с использованием режима округления к ближайшему IEEE 754. Язык программирования Java требует поддержки постепенной потери значимости, определенной IEEE 754 (§4.2.4).

Независимо от возможного факта переполнения, потери значимости, деления на нуль или потери информации, вычисление оператора деления / с плавающей точкой никогда не приводит к генерации исключения времени выполнения.

§15.17.3. Оператор получения остатка %

Бинарный оператор % дает остаток от предполагаемого деления его операндов; левый операнд представляет собой *делимое*, а правый — *делитель*.

В C и C++ оператор получения остатка применим только к целочисленным операндам, но в языке программирования Java он допускает и операнды с плавающей точкой.

Операция получения остатка для целочисленных после бинарного числового повышения (§5.6.2) операндов дает значение, такое, что `(a/b)*b+(a%b)` равно `a`.

Это тождество выполняется даже в том частном случае, когда делимое является отрицательным целым числом с наибольшим абсолютным значением для своего типа, а делитель равен `-1` (остаток равен `0`).

Это следует из правила, что результат операции получения остатка может быть отрицателен, только если отрицательно делимое, и может быть положительным, только если делимое положительно. Кроме того, по абсолютному значению результат всегда меньше делителя.

Если значение делителя в операции получения целочисленного остатка равен `0`, генерируется исключение `ArithmeticException`.

Пример 15.17.3-1. Оператор целочисленного остатка

```
class Test1 {
    public static void main(String[] args) {
        int a = 5%3; // 2
        int b = 5/3; // 1
        System.out.println("5%3 дает " + a +
                " (при этом 5/3 дает " + b + ")");
        int c = 5%(-3); // 2
        int d = 5/(-3); // -1
        System.out.println("5%(-3) дает " + c +
                " (при этом 5/(-3) дает " + d + ")");
        int e = (-5)%3; // -2
        int f = (-5)/3; // -1
        System.out.println("(-5)%3 дает " + e +
                " (при этом (-5)/3 дает " + f + ")");
```

```
        int g = (-5)%(-3); // -2
        int h = (-5)/(-3); // 1
        System.out.println("(-5)%(-3) дает " + g +
                " (при этом (-5)/(-3) дает " + h + ")");
    }
}
```

Вывод этой программы имеет вид

```
5%3 дает 2 (при этом 5/3 дает 1)
5%(-3) дает 2 (при этом 5/(-3) дает -1)
(-5)%3 дает -2 (при этом (-5)/3 дает -1)
(-5)%(-3) дает -2 (при этом (-5)/(-3) дает 1)
```

Результат операции получения остатка с плавающей точкой, вычисляемый с помощью оператора `%`, *не* совпадает с результатом операции вычисления остатка, определенной IEEE 754. Операция вычисления остатка IEEE 754 вычисляет остаток на основе округляющего, а не отсекающего деления, так что ее поведение *не* соответствует обычному целочисленному оператору получения остатка. Вместо этого язык программирования Java определяет оператор `%` для чисел с плавающей точкой, который ведет себя аналогично оператору получения целочисленного остатка; его можно сравнить с библиотечной функцией C `fmod`. Операция получения остатка IEEE 754 может быть вычислена с помощью библиотечной подпрограммы `Math.IEEEremainder`.

Результат получения остатка с плавающей точкой определяется правилами арифметики IEEE 754.

- Если один из операндов представляет собой NaN, результатом является NaN.
- Если результат не является NaN, знак результата равен знаку делимого.
- Если делимое является бесконечностью или делитель — нулем (или и то, и другое одновременно), результат операции равен NaN.
- Если делимое конечно, а делитель бесконечен, результат равен делимому.
- Если делимое — нуль, а делитель конечен, то результат равен делимому.
- В остальных случаях, когда в качестве операндов не участвуют ни нуль, ни бесконечность, ни NaN, остаток с плавающей точкой r от деления делимого n на делитель d определяется математическим соотношением $r = n - (d \cdot q)$, где q — целое число, которое является отрицательным, только если отрицательно n/d, и положительным, только если положительно n/d, и абсолютное значение которого максимально возможное среди не превышающих истинное математическое частное n и d.

Вычисление оператора получения остатка с плавающей точкой `%` никогда не генерирует исключение времени выполнения, даже если правый операнд равен нулю. При вычислении невозможны переполнение, потеря значимости или потеря точности.

Пример 15.17.3-2. Оператор получения остатка с плавающей точкой

```
class Test2 {
    public static void main(String[] args) {
        double a = 5.0%3.0; // 2.0
        System.out.println("5.0%3.0 равно " + a);
```

```
        double b = 5.0%(-3.0); // 2.0
        System.out.println("5.0%(-3.0) равно " + b);
        double c = (-5.0)%3.0; // -2.0
        System.out.println("(-5.0)%3.0 равно " + c);
        double d = (-5.0)%(-3.0); // -2.0
        System.out.println("(-5.0)%(-3.0) равно " + d);
    }
}
```

Вывод этой программы имеет вид

```
5.0%3.0 равно 2.0
5.0%(-3.0) равно 2.0
(-5.0)%3.0 равно -2.0
(-5.0)%(-3.0) равно -2.0
```

§15.18. Аддитивные операторы

Операторы + и − называются *аддитивными операторами.*

AdditiveExpression:
MultiplicativeExpression
AdditiveExpression + *MultiplicativeExpression*
AdditiveExpression − *MultiplicativeExpression*

Аддитивные операторы имеют одинаковые приоритеты и синтаксически левоассоциативны (группируются слева направо).

Если один из операндов оператора + имеет тип `String`, то операция представляет собой конкатенацию строк.

В противном случае тип каждого операнда оператора + должен быть преобразуемым (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

В любом случае тип каждого операнда бинарного оператора − должен быть преобразуемым (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

§15.18.1. Оператор конкатенации строк

Если только одно выражение операнда имеет тип `String`, то ко второму операнду для получения строки во время выполнения программы применяется строковое преобразование (§5.1.11).

Результатом конкатенации строк является ссылка на объект типа `String`, который представляет собой конкатенацию двух строк-операндов. Во вновь созданной строке символы левого операнда предшествуют символам правого операнда.

Объект `String` создается заново (§12.5), если только выражение не является константным выражением (§15.28).

> Реализация может пойти по пути выполнения преобразования и конкатенации как единого шага, избегая создания и уничтожения промежуточного объекта `String`.

Для увеличения производительности многократных конкатенаций строк компилятор Java может использовать класс `StringBuffer` или подобную методику для снижения количества промежуточных объектов `String`, создаваемых при вычислении выражения.

Для примитивных типов реализация может также прибегнуть к оптимизации, заключающейся в отказе от промежуточных объектных типов, выполняя преобразование в строку непосредственно для примитивного типа.

Пример 15.18.1-1. Конкатенация строк

Например, выражение

```
"The square root of 2 is " + Math.sqrt(2)
```

приводит к результату

```
"The square root of 2 is 1.4142135623730952"
```

Оператор + синтаксически левоассоциативный, независимо от того, что он выполняет — конкатенацию строк или сложение чисел. В некоторых случаях для получения желаемого результата нужна внимательность. Например, выражение

```
a + b + c
```

всегда рассматривается как

```
(a + b) + c
```

так что результатом выражения

```
1 + 2 + " fiddlers"
```

является строка

```
"3 fiddlers"
```

а результатом выражения

```
"fiddlers " + 1 + 2
```

строка

```
"fiddlers 12"
```

Пример 15.18.1-2. Конкатенация строк и условные выражения

```
class Bottles {
    static void printSong(Object stuff, int n) {
        String plural = (n == 1) ? "" : "s";
loop: while (true) {
            System.out.println(n + " bottle" + plural
                    + " of " + stuff + " on the wall,");
            System.out.println(n + " bottle" + plural
                    + " of " + stuff + ";");
            System.out.println("You take one down "
                    + "and pass it around:");
            --n;
            plural = (n == 1) ? "" : "s";
            if (n == 0)
                break loop;
            System.out.println(n + " bottle" + plural
```

```
                    + " of " + stuff + " on the wall!");
            System.out.println();
        }
        System.out.println("No bottles of " +
                stuff + " on the wall!");
    }
    public static void main(String[] args) {
        printSong("slime", 3);
    }
}
```

В этом шуточном примере метод `printSong` выводит версию известной детской песенки. Вот какой вид имеет результат работы данной программы:

```
3 bottles of slime on the wall,
3 bottles of slime;
You take one down and pass it around:
2 bottles of slime on the wall!

2 bottles of slime on the wall,
2 bottles of slime;
You take one down and pass it around:
1 bottle of slime on the wall!

1 bottle of slime on the wall,
1 bottle of slime;
You take one down and pass it around:
No bottles of slime on the wall!
```

Обратите внимание на аккуратную генерацию с помощью условного выражения слова `"bottle"`, где речь идет об одном предмете, и `"bottles"` — где о нескольких; обратите также внимание на то, как оператор конкатенации строк использован для разбиения длинной константной строки

`"You take one down and pass it around:"`

на две части, чтобы избежать слишком длинных строк в исходном тексте.

§15.18.2. Аддитивные операторы (+ и –) для числовых типов

Бинарный оператор +, будучи применен к двум операндам числового типа, выполняет их сложение, возвращая сумму операндов.

Бинарный оператор – выполняет вычитание, возвращая разность двух числовых операндов.

Над операндами производится бинарное числовое повышение (§5.6.2).

> Обратите внимание, что бинарное числовое повышение выполняет преобразование набора значений (§5.1.13) и может выполнять преобразование распаковки (§5.1.8).

Типом аддитивного выражения с числовыми операндами является повышенный тип его операндов.

Если этот повышенный тип — `int` или `long`, применяется целочисленная арифметика.

Если этот повышенный тип — `float` или `double`, применяется арифметика с плавающей точкой.

Сложение является коммутативной операцией, если выражения операндов не имеют побочных действий.

Целочисленное сложение ассоциативно, если все операнды — одного типа.

Сложение с плавающей точкой ассоциативным не является.

Если целочисленное сложение приводит к переполнению, то результат представляет собой младшие биты математической суммы, представленной в некотором достаточно большом числе в дополнительном формате. В случае переполнения знак результата не совпадает со знаком математической суммы значений двух операндов.

Результат сложения с плавающей точкой определяется правилами арифметики IEEE 754.

- Если один из операндов представляет собой NaN, результатом является NaN.
- Сумма двух бесконечностей противоположного знака представляет собой NaN.
- Сумма двух бесконечностей одинакового знака представляет собой бесконечность с тем же знаком.
- Сумма бесконечности и конечного значения равна бесконечному операнду.
- Сумма двух нулей противоположного знака равна положительному нулю.
- Сумма двух нулей одинакового знака равна нулю с этим знаком.
- Сумма нуля и ненулевого конечного значения равна ненулевому операнду.
- Сумма двух конечных значений с одинаковым абсолютным значением и разными знаками равна положительному нулю.
- В остальных случаях, когда в качестве операндов не участвуют ни нуль, ни бесконечность, ни NaN и когда операнды имеют одинаковые знаки или разные абсолютные значения, вычисляется точная математическая сумма. Затем выбирается набор значений с плавающей точкой.
  - ✦ Если выражение сложения является FP-строгим (§15.4):
    - — если тип выражения сложения — `float`, должен быть выбран набор значений float;
    - — если тип выражения сложения — `double`, должен быть выбран набор значений double.
  - ✦ Если выражение сложения не является FP-строгим:
    - — если тип выражения сложения — `float`, то может быть выбран как набор значений float, так и расширенный набор значений float; это зависит от конкретной реализации;
    - — если тип выражения сложения — `double`, то может быть выбран как набор значений double, так и расширенный набор значений double; это зависит от конкретной реализации.

Затем из выбранного набора значений выбирается значение, представляющее сумму.

Если величина суммы слишком велика для представления, мы говорим о переполнении при выполнении операции; результатом в этом случае является бесконечность соответствующего знака.

В противном случае сумма округляется до ближайшего значения из выбранного набора значений с использованием режима округления к ближайшему IEEE 754. Язык программирования Java требует поддержки постепенной потери значимости, определенной IEEE 754 (§4.2.4).

Бинарный оператор `-`, будучи примененным к двум операндам числового типа, выполняет их вычитание, возвращая разность операндов; левый операнд представляет собой *уменьшаемое*, а правый — *вычитаемое*.

В случае как целочисленной арифметики, так и арифметики с плавающей точкой `a-b` всегда дает тот же результат, что и `a+(-b)`.

Обратите внимание, что в случае целочисленной арифметики вычитание из нуля эквивалентно изменению знака. Однако для операндов с плавающей точкой вычитание из нуля *не* эквивалентно изменению знака, поскольку если `x` равно `+0.0`, то `0.0-x` равно `+0.0`, а `-x` равно `-0.0`.

Независимо от возможного факта переполнения, потери значимости, деления на нуль или потери информации, вычисление числового аддитивного оператора никогда не приводит к генерации исключения времени выполнения.

§15.19. Операторы сдвига

Операторы `<<` (левый сдвиг), `>>` (знаковый правый сдвиг) и `>>>` (беззнаковый правый сдвиг) называются *операторами сдвига*. Левый операнд оператора сдвига представляет собой сдвигаемое значение, а правый — величину сдвига.

ShiftExpression:
AdditiveExpression
ShiftExpression `<<` *AdditiveExpression*
ShiftExpression `>>` *AdditiveExpression*
ShiftExpression `>>>` *AdditiveExpression*

Операторы сдвига синтаксически левоассоциативны (группируются слева направо).

Унарное числовое повышение (§5.6.1) выполняется над каждым оператором отдельно. (Бинарное числовое повышение (§5.6.2) над операндами *не* производится.)

Если тип каждого операнда оператора сдвига после числового повышения не является примитивным целочисленным типом, генерируется ошибка времени компиляции.

Типом выражения сдвига является повышенный тип левого операнда.

Если повышенный тип левого операнда представляет собой `int`, в качестве величины сдвига используются только пять младших битов правого операнда, как если бы правый операнд подвергся действию побитового логического оператора И `&` (§15.22.1) со значением маски `0x1f` (`0b11111`). Таким образом, фактически используемая величина сдвига всегда находится в диапазоне от `0` до `31` включительно.

Если повышенный тип левого операнда представляет собой `long`, в качестве величины сдвига используются только шесть младших битов правого операнда, как если бы

правый операнд подвергся действию побитового логического оператора И `&` (§15.22.1) со значением маски `0x3f` (`0b111111`). Таким образом, фактически используемая величина сдвига всегда находится в диапазоне от `0` до `63` включительно.

Во время выполнения операции сдвига выполняются над целочисленным представлением значения левого операнда в дополнительном коде.

Значение *n* << *s* представляет собой *n*, сдвинутое влево на *s* битовых позиций; это эквивалентно (даже при переполнении) умножению на 2^s.

Значение *n* >> *s* представляет собой *n*, сдвинутое вправо на *s* битовых позиций с распространением знака. Получающееся в результате значение равно $floor\left(n/2^s\right)$. Для неотрицательных значений *n* это эквивалентно отсекающему целочисленному делению, вычисляемому оператором целочисленного деления `/`, на 2^s.

Значение *n* >>> *s* представляет собой *n*, сдвинутое вправо на *s* битовых позиций с распространением нуля.

- Если *n* положительное, результат совпадает с результатом *n* >> *s*.
- Если *n* отрицательное и тип левого операнда — `int`, то результат равен результату выражения (*n* >> *s*) + (2 << ~*s*).
- Если *n* отрицательное и тип левого операнда — `long`, то результат равен результату выражения (*n* >> *s*) + (2L << ~*s*).

> Добавленный член (2 << ~*s*) или (2L << ~*s*) отменяет распространение знакового бита.
>
> Обратите внимание, что в силу неявного маскирования правого операнда оператора сдвига ~*s* в качестве величины сдвига эквивалентно 31-*s* при сдвиге значения типа `int` и 63-*s* — при сдвиге значения типа `long`.

§15.20. Операторы отношения

Операторы числового сравнения <, >, <= и >=, а также оператор `instanceof` называются *операторами отношения*.

RelationalExpression:
ShiftExpression
RelationalExpression < *ShiftExpression*
RelationalExpression > *ShiftExpression*
RelationalExpression <= *ShiftExpression*
RelationalExpression >= *ShiftExpression*
RelationalExpression `instanceof` *ReferenceType*

Операторы отношения синтаксически левоассоциативны (группируются слева направо).

> Однако этот факт приносит мало пользы. Например, a<b<c трактуется как (a<b)<c, что приводит к ошибке времени компиляции, поскольку типом a<b всегда является `boolean`, а < не является оператором, работающим со значениями типа `boolean`.

Тип выражения отношения всегда представляет собой `boolean`.

§15.20.1. Числовые операторы сравнения <, >, <= и >=

Тип каждого операнда оператора числового сравнения должен быть преобразуем (§5.1.8) в примитивный числовой тип, иначе генерируется ошибка времени компиляции.

Над операндами выполняется бинарное числовое повышение (§5.6.2).

> Обратите внимание, что бинарное числовое повышение выполняет преобразование набора значений (§5.1.13), а также может выполнять преобразование распаковки (§5.1.8).

Если повышенный тип операндов представляет собой `int` или `long`, то выполняется знаковое целочисленное сравнение.

Если повышенный тип операндов представляет собой `float` или `double`, выполняется сравнение с плавающей точкой.

Со значениями с плавающей точкой сравнение осуществляется точно, независимо от того, из какого набора значений выбраны представляющие их значения.

Результат сравнения с плавающей точкой определяется спецификацией стандарта IEEE 754.

- Если один из операндов равен NaN, результат сравнения равен `false`.
- Все отличные от NaN значения упорядочены, при этом отрицательная бесконечность меньше всех конечных значений, а положительная бесконечность больше всех конечных значений.
- Положительный нуль и отрицательный нуль рассматриваются как равные.

> Например, `-0.0<0.0` равно `false`, но `-0.0<=0.0` равно `true`.
>
> Заметим, однако, что методы `Math.min` и `Math.max` рассматривают отрицательный нуль как строго меньший положительного нуля.

С учетом этих соображений для чисел с плавающей точкой, для целочисленных операндов и для операндов с плавающей точкой, отличных от NaN, выполняются следующие правила.

- Значение, возвращаемое оператором <, равно `true`, если значение левого операнда меньше значения правого операнда; в противном случае возвращаемое значение равно `false`.
- Значение, возвращаемое оператором <=, равно `true`, если значение левого операнда меньше или равно значению правого операнда; в противном случае возвращаемое значение равно `false`.
- Значение, возвращаемое оператором >, равно `true`, если значение левого операнда больше значения правого операнда; в противном случае возвращаемое значение равно `false`.
- Значение, возвращаемое оператором >=, равно `true`, если значение левого операнда больше или равно значению правого операнда; в противном случае возвращаемое значение равно `false`.

§15.20.2. Оператор сравнения типа `instanceof`

Тип операнда *RelationalExpression* оператора `instanceof` должен быть ссылочным типом или типом `null`; в противном случае генерируется ошибка времени компиляции.

Если *ReferenceType*, находящийся справа от оператора `instanceof` не обозначает тип, доступный во время выполнения (§4.7), генерируется ошибка времени компиляции.

Если приведение (§15.16) *RelationalExpression* к *ReferenceType* отвергается с ошибкой времени компиляции, то и выражение отношения `instanceof` генерирует ошибку времени компиляции. В такой ситуации результат выражения `instanceof` не может быть истинным.

Во время выполнения результат оператора `instanceof` равен `true`, если значение *RelationalExpression* не равно `null` и ссылка может быть приведена к *ReferenceType* без генерации исключения `ClassCastException`. В противном случае результат равен `false`.

Пример 15.20.2-1. Оператор `instanceof`

```
class Point   { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
        Point   p = new Point();
        Element e = new Element();
        if (e instanceof Point) { // Ошибка времени компиляции
            System.out.println("I get your point!");
            p = (Point)e;         // Ошибка времени компиляции
        }
    }
}
```

При компиляции этой программы возникают две ошибки времени компиляции. Приведение `(Point)e` некорректно, поскольку ни экземпляр класса `Element`, ни любого из его возможных подклассов (здесь не показанных) не могут быть экземпляром некоторого подкласса класса `Point`. Выражение `instanceof` некорректно по тем же самым причинам. Если бы, с другой стороны, класс `Point` был подклассом класса `Element` (правда, это было бы странно):

```
class Point extends Element { int x, y; }
```

то приведение было бы возможным, хотя и требовало бы проверки времени выполнения, и выражение `instanceof` имело бы смысл и было корректным. Приведение `(Point)e` не генерировало бы исключение, поскольку не выполнялось бы, если бы значение `e` не могло быть корректно приведено к типу `Point`.

§15.21. Операторы равенства

Операторы == (равно) и != (не равно) называются *операторами равенства*.

EqualityExpression:
RelationalExpression

EqualityExpression `==` *RelationalExpression*
EqualityExpression `!=` *RelationalExpression*

Операторы равенства синтаксически левоассоциативны (группируются слева направо).

> Однако этот факт, по сути, бесполезен. Например, `a==b==c` трактуется, как `(a==b)==c`. Тип результата `a==b` всегда `boolean`, так что `c` должен иметь тип `boolean`, иначе генерируется ошибка времени компиляции. Таким образом, `a==b==c` не проверяет тот факт, что все три значения, `a`, `b` и `c`, равны между собой.

Операторы равенства коммутативны, если выражения операндов не имеют побочных эффектов.

Операторы равенства аналогичны операторам отношения, за исключением того, что они имеют более низкий приоритет. Таком образом, `a<b==c<d` равно `true`, когда `a<b` и `c<d` имеют одно и то же значение истинности.

Операторы равенства могут использоваться для сравнения двух операндов, преобразуемых (§5.1.8) в числовой тип, или двух операндов типа `boolean` или `Boolean`, или двух операндов, каждый из которых — либо ссылочный тип, либо тип `null`. Во всех прочих случаях генерируется ошибка времени компиляции.

Тип выражения равенства всегда представляет собой `boolean`.

Во всех случаях `a!=b` дает тот же результат, что и `!(a==b)`.

§15.21.1. Числовые операторы равенства == и !=

Если оба операнда оператора равенства имеют числовой тип или один из них числового типа, а второй преобразуем (§5.1.8) в числовой тип, над операндами производится бинарное числовое повышение (§5.6.2).

> Обратите внимание, что бинарное числовое повышение выполняет преобразование набора значений (§5.1.13), а также может выполнять преобразование распаковки (§5.1.8).

Если повышенный тип операндов представляет собой `int` или `long`, то выполняется знаковая целочисленная проверка равенства.

Если повышенный тип операндов представляет собой `float` или `double`, выполняется проверка равенства с плавающей точкой.

Со значениями с плавающей точкой сравнение осуществляется точно, независимо от того, из какого набора значений выбраны представляющие их значения.

Результат проверки равенства с плавающей точкой определяется спецификацией стандарта IEEE 754.

- Если один из операндов равен NaN, результат оператора `==` равен `false`, но результат оператора `!=` равен `true`.

  Проверка `x!=x` дает `true` тогда и только тогда, когда значение `x` равно NaN.

> Для проверки того, что значение равно NaN, можно также использовать методы `Float.isNaN` и `Double.isNaN`.

- Положительный нуль и отрицательный нуль рассматриваются как равные.

  Например, `-0.0==0.0` равно `true`.

- В остальных случаях два различных значения с плавающей точкой операторами равенства рассматриваются как неравные.

  В частности, это относится к значениям, представляющим положительную и отрицательную бесконечность: они равны только самим себе и не равны всем прочим значениям.

С учетом этих соображений для чисел с плавающей точкой, для целочисленных операндов и для операндов с плавающей точкой, отличных от NaN, выполняются следующие правила.

- Значение, возвращаемое оператором `==`, равно `true`, если значение левого операнда равно значению правого операнда; в противном случае возвращаемое значение равно `false`.
- Значение, возвращаемое оператором `!=`, равно `true`, если значение левого операнда не равно значению правого операнда; в противном случае возвращаемое значение равно `false`.

§15.21.2. Логические операторы равенства == и !=

Если оба операнда оператора равенства имеют тип `boolean` или если один операнд имеет тип `boolean`, а второй — тип `Boolean`, то операция представляет собой булеву проверку равенства.

Операторы булева равенства ассоциативны.

Если один из операторов имеет тип `Boolean`, выполняется преобразование распаковки (§5.1.8).

Результатом выполнения проверки `==` является `true`, если оба операнда (после необходимого преобразования распаковки) равны `true` или оба равны `false`; в противном случае результат равен `false`.

Результатом выполнения проверки `!=` является `false`, если оба операнда равны `true` или оба равны `false`; в противном случае результат равен `true`.

> Таким образом, оператор `!=` ведет себя так же, как и `^` (§15.22.2) при применении к операндам типа `boolean`.

§15.21.3. Ссылочные операторы равенства == и !=

Если оба операнда оператора равенства имеют либо ссылочный тип, либо тип `null`, операция является проверкой равенства объектов.

Если тип одного операнда невозможно преобразовать в тип другого с помощью преобразования приведения (§5.5), генерируется ошибка времени компиляции. Значения времени выполнения этих двух операндов обязательно различны.

Во время выполнения результат оператора `==` равен `true`, если оба значения операндов равны `null` или оба указывают на один и тот же объект или массив; в противном случае результат равен `false`.

Результат оператора `!=` равен `false`, если оба значения операндов равны `null` или оба указывают на один и тот же объект или массив; в противном случае результат равен `true`.

При применении оператора `==` для сравнения ссылок на тип `String` такая проверка равенства определяет, указывают ли оба операнда на один и тот же объект `String`. Результат равен `false`, если операнды представляют собой различные объекты `String`, даже если они содержат одинаковые последовательности символов (§3.10.5). Содержимое двух строк, `s` и `t`, можно сравнить на равенство с помощью вызова метода `s.equals(t)`.

§15.22. Побитовые и логические операторы

Побитовые операторы и *логические операторы* включают оператор И `&`, исключающего ИЛИ `^` и включающего ИЛИ `|`.

AndExpression:
EqualityExpression
AndExpression `&` *EqualityExpression*

ExclusiveOrExpression:
AndExpression
ExclusiveOrExpression `^` *AndExpression*

InclusiveOrExpression:
ExclusiveOrExpression
InclusiveOrExpression `|` *ExclusiveOrExpression*

Эти операторы имеют различные приоритеты, наивысший — у оператора `&`, а наинизший — у оператора `|`.

Каждый из этих операторов левоассоциативен (каждый группируется слева направо).

Каждый оператор коммутативен, если выражения операндов не имеют побочных эффектов.

Каждый оператор ассоциативен.

Побитовые и логические операторы могут применяться для двух операндов числового типа или двух операторов типа `boolean`. Во всех других случаях генерируется ошибка времени компиляции.

§15.22.1. Целочисленные побитовые операторы &, ^ и |

Если оба операнда операторов `&`, `^` и `|` имеют тип, преобразуемый (§5.1.8) в примитивный целочисленный тип, сначала над операндами выполняется бинарное числовое повышение (§5.6.2).

Тип результата выражения побитового оператора представляет собой повышенный тип операндов.

В случае оператора `&` результирующее значение представляет собой побитовое И значений операндов.

В случае оператора ^ результирующее значение представляет собой побитовое исключающее ИЛИ значений операндов.

В случае оператора | результирующее значение представляет собой побитовое включающее ИЛИ значений операндов.

> Например, результатом выражения
> `0xff00 & 0xf0f0`
> является
> `0xf000`
> Результатом выражения
> `0xff00 ^ 0xf0f0`
> является
> `0x0ff0`
> Результатом выражения
> `0xff00 | 0xf0f0`
> является
> `0xfff0`

§15.22.2. Булевы логические операторы &, ^ и |

Если оба операнда операторов `&`, `^` и `|` имеют тип `boolean` или `Boolean`, то типом выражения побитового оператора является `boolean`. Во всех случаях при необходимости над операндами выполняется преобразование распаковки (§5.1.8).

В случае оператора `&` результирующее значение равно `true`, если оба значения операндов равны `true`; в противном случае результирующее значение равно `false`.

В случае оператора ^ результирующее значение равно `true`, если значения операндов различны; в противном случае результирующее значение равно `false`.

В случае оператора | результирующее значение равно `false`, если оба значения операндов равны `false`; в противном случае результирующее значение равно `true`.

§15.23. Оператор условного И &&

Оператор условного И `&&` подобен оператору `&` (§15.22.2), но его правый операнд вычисляется только тогда, когда значение левого операнда равно `true`.

ConditionalAndExpression:
InclusiveOrExpression
ConditionalAndExpression `&&` *InclusiveOrExpression*

Оператор условного И левоассоциативен (группируется слева направо).

Оператор условного И полностью ассоциативен как по отношению к побочным эффектам, так и по отношению к результирующему значению. То есть для любых выражений `a`, `b` и `c` вычисление выражения `((a)&&(b))&&(c)` дает тот же результат, с теми же побочными эффектами и в том же порядке, что и при вычислении выражения `(a)&&((b)&&(c))`.

Каждый операнд оператора условного И должен иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

Типом выражения условного И всегда является `boolean`.

Во время выполнения первым вычисляется выражение левого операнда; если результат имеет тип `Boolean`, к нему применяется преобразование распаковки (§5.1.8).

Если результирующее значение равно `false`, значение выражения условного И равно `false`, и выражение правого операнда не вычисляется.

Если значение левого операнда равно `true`, вычисляется выражение правого операнда; если результат имеет тип `Boolean`, к нему применяется преобразование распаковки (§5.1.8). Результирующее значение становится результатом выражения условного И.

Таким образом, оператор `&&` вычисляет тот же результат, что и оператор `&` с операндами типа `boolean` или `Boolean`. Отличие заключается в том, что выражение правого операнда вычисляется только при соответствующем условии, а не всегда.

§15.24. Оператор условного ИЛИ `||`

Оператор условного ИЛИ `||` подобен оператору `|` (§15.22.2), но его правый операнд вычисляется только тогда, когда значение левого операнда равно `false`.

ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression `||` *ConditionalAndExpression*

Оператор условного ИЛИ левоассоциативен (группируется слева направо).

Оператор условного ИЛИ полностью ассоциативен как по отношению к побочным эффектам, так и по отношению к результирующему значению. То есть для любых выражений *a*, *b* и *c* вычисление выражения `((`*a*`)||(`*b*`))||(`*c*`)` дает тот же результат с теми же побочными эффектами в том же порядке, что и при вычислении выражения `(`*a*`)||((`*b*`)||(`*c*`))`.

Каждый операнд оператора условного ИЛИ должен иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

Типом выражения условного ИЛИ всегда является `boolean`.

Во время выполнения первым вычисляется выражение левого операнда; если результат имеет тип `Boolean`, к нему применяется преобразование распаковки (§5.1.8).

Если результирующее значение равно `true`, значение выражения условного ИЛИ равно `true`, и выражение правого операнда не вычисляется.

Если значение левого операнда равно `false`, вычисляется выражение правого операнда; если результат имеет тип `Boolean`, к нему применяется преобразование распаковки (§5.1.8). Результирующее значение становится результатом выражения условного ИЛИ.

Таким образом, оператор `||` вычисляет тот же результат, что и оператор `|` с операндами типа `boolean` или `Boolean`. Отличие заключается в том, что выражение правого операнда вычисляется только при соответствующем условии, а не всегда.

§15.25. Условный оператор ? :

Условный оператор `?:` использует булево значение одного выражения для принятия решения о том, какое из двух других выражений должно быть вычислено.

ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression `?` *Expression* `:` *ConditionalExpression*

Условный оператор правоассоциативен (группируется справа налево). Таким образом, `a?b:c?d:e?f:g` означает то же, что и `a?b:(c?d:(e?f:g))`.

Условный оператор имеет три выражения операндов. Символ `?` находится между первым и вторым выражениями, а символ `:` находится между вторым и третьим выражениями.

Первое выражение должно иметь тип `boolean` или `Boolean`, иначе генерируется ошибка времени компиляции.

Если выражение второго или третьего операнда представляет собой вызов `void`-метода, генерируется ошибка времени компиляции.

> Согласно грамматике инструкций выражений (§14.8) условное выражение не должно находиться в контексте, где может находиться вызов `void`-метода.

Имеется три вида условных выражений, классифицируемых в соответствии с выражениями второго и третьего операндов: *логические (булевы) условные выражения*, *числовые условные выражения* и *ссылочные условные выражения*. Правила классификации приведены ниже.

- Если выражения и второго, и третьего операнда являются *булевыми выражениями*, то условное выражение является булевым условным выражением.

  Булевыми выражениями являются следующие.

  - ✦ Выражение автономного вида (§15.2), имеющее тип `boolean` или `Boolean`.
  - ✦ Выражение типа `boolean` в скобках (§15.8.5).
  - ✦ Выражение создания экземпляра класса (§15.9) для класса `Boolean`.
  - ✦ Выражение вызова метода (§15.12), для которого выбранный наиболее подходящий метод (§15.12.2.5) имеет возвращаемый тип `boolean` или `Boolean`.

  > Обратите внимание, что в случае обобщенного метода это тип *до* инстанцирования аргументов типов.

  - ✦ Условное выражение типа `boolean`.
- Если выражения и второго, и третьего операндов являются *числовыми выражениями*, то условное выражение является числовым условным выражением.

  Числовыми выражениями являются следующие.

  - ✦ Выражение автономного вида (§15.2) с типом, преобразуемым в числовой тип (§4.2, §5.1.8).
  - ✦ Числовое выражение в скобках (§15.8.5).

  - ✦ Выражение создания экземпляра класса (§15.9) для класса, преобразуемого в числовой тип.
  - ✦ Выражение вызова метода (§15.12), для которого выбранный наиболее подходящий метод (§15.12.2.5) имеет возвращаемый тип, преобразуемый в числовой тип.
  - ✦ Числовое условное выражение.
- В противном случае условное выражение является ссылочным условным выражением.

Процесс определения типа условного выражения зависит от вида условного выражения, как описано в последующих разделах.

В приведенных далее табл. 15.25-А–15.25-Д приведены сформулированные выше правила, которые дают тип условного выражения для всех возможных типов второго и третьего операндов. `bnp(..)` означает применение бинарного числового повышения. Запись "*T*`|bnp(..)`" используется там, где один операнд является константным выражением типа `int` и может быть представлен типом *T* и где бинарное числовое повышение используется, если операнд не может быть представлен типом *T*. Тип операнда `Object` означает любой ссылочный тип, отличный от типа null и от восьми классов-оболочек: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.

Во время выполнения сначала вычисляется выражение первого операнда. При необходимости к результату применяется преобразование распаковки.

Получающееся в результате значение типа `boolean` затем используется для выбора выражения второго или третьего операнда.

- Если значение первого операнда — `true`, выбирается выражение второго операнда.
- Если значение первого операнда — `false`, выбирается выражение третьего операнда.

После этого вычисляется выбранное выражение операнда, а результат вычисления преобразуется в тип условного выражения, определяемый приведенными выше правилами.

Это преобразование может включать преобразование упаковки или распаковки (§5.1.7, §5.1.8).

При каждом конкретном вычислении условного выражения не выбранное выражение операнда не вычисляется.

§15.25.1. Булевы условные выражения

Булевы условные выражения являются автономными выражениями (§15.2).

Тип булева условного выражения определяется следующим образом.

- Если и второй, и третий операнды имеют тип `Boolean`, условное выражение имеет тип `Boolean`.
- В противном случае условное выражение имеет тип `boolean`.

Таблица 15.25-А. Тип условного выражения (примитивный третий операнд, часть I)

Третий→	byte	short	char	int
Второй↓				
byte	byte	short	bnp(byte,char)	byte|bnp(byte,int)
Byte	byte	short	bnp(Byte,char)	byte|bnp(Byte,int)
short	short	short	bnp(short,char)	short|bnp(short,int)
Short	short	short	bnp(Short,char)	short|bnp(Short,int)
char	bnp(char,byte)	bnp(char,short)	char	char|bnp(char,int)
Character	bnp(Character,byte)	bnp(Character,short)	char	char|bnp(Character,int)
int	byte|bnp(int,byte)	short|bnp(int,short)	char|bnp(int,char)	int
Integer	bnp(Integer,byte)	bnp(Integer,short)	bnp(Integer,char)	int
long	bnp(long,byte)	bnp(long,short)	bnp(long,char)	bnp(long,int)
Long	bnp(Long,byte)	bnp(Long,short)	bnp(Long,char)	bnp(Long,int)
float	bnp(float,byte)	bnp(float,short)	bnp(float,char)	bnp(float,int)
Float	bnp(Float,byte)	bnp(Float,short)	bnp(Float,char)	bnp(Float,int)
double	bnp(double,byte)	bnp(double,short)	bnp(double,char)	bnp(double,int)
Double	bnp(Double,byte)	bnp(Double,short)	bnp(Double,char)	bnp(Double,int)
boolean	lub(Boolean,Byte)	lub(Boolean,Short)	lub(Boolean,Character)	lub(Boolean,Integer)
Boolean	lub(Boolean,Byte)	lub(Boolean,Short)	lub(Boolean,Character)	lub(Boolean,Integer)
null	lub(null,Byte)	lub(null,Short)	lub(null,Character)	lub(null,Integer)
Object	lub(Object,Byte)	lub(Object,Short)	lub(Object,Character)	lub(Object,Integer)

Таблица 15.25-Б. Тип условного выражения (примитивный третий операнд, часть II)

Третий→	long	float	double	boolean
Второй↓				
byte	bnp(byte,long)	bnp(byte,float)	bnp(byte,double)	lub(Byte,Boolean)
Byte	bnp(Byte,long)	bnp(Byte,float)	bnp(Byte,double)	lub(Byte,Boolean)
short	bnp(short,long)	bnp(short,float)	bnp(short,double)	lub(Short,Boolean)
Short	bnp(Short,long)	bnp(Short,float)	bnp(Short,double)	lub(Short,Boolean)
char	bnp(char,long)	bnp(char,float)	bnp(char,double)	lub(Character,Boolean)
Character	bnp(Character,long)	bnp(Character,float)	bnp(Character,double)	lub(Character,Boolean)

Окончание табл. 15.25-Б

Третий→	long	float	double	boolean
Второй↓				
int	bnp(int,long)	bnp(int,float)	bnp(int,double)	lub(Integer,Boolean)
Integer	bnp(Integer,long)	bnp(Integer,float)	bnp(Integer,double)	lub(Integer,Boolean)
long	long	bnp(long,float)	bnp(long,double)	lub(Long,Boolean)
Long	long	bnp(Long,float)	bnp(Long,double)	lub(Long,Boolean)
float	bnp(float,long)	float	bnp(float,double)	lub(Float,Boolean)
Float	bnp(Float,long)	float	bnp(Float,double)	lub(Float,Boolean)
double	bnp(double,long)	bnp(double,float)	double	lub(Double,Boolean)
Double	bnp(Double,long)	bnp(Double,float)	double	lub(Double,Boolean)
boolean	lub(Boolean,Long)	lub(Boolean,Float)	lub(Boolean,Double)	boolean
Boolean	lub(Boolean,Long)	lub(Boolean,Float)	lub(Boolean,Double)	boolean
null	lub(null,Long)	lub(null,Float)	lub(null,Double)	lub(null,Boolean)
Object	lub(Object,Long)	lub(Object,Float)	lub(Object,Double)	lub(Object,Boolean)

Таблица 15.25-В. Тип условного выражения (ссылочный третий операнд, часть I)

Третий→	Byte	Short	Character	Integer
Второй↓				
byte	byte	short	bnp(byte,Character)	bnp(byte,Integer)
Byte	Byte	short	bnp(Byte,Character)	bnp(Byte,Integer)
short	short	short	bnp(short,Character)	bnp(short,Integer)
Short	short	Short	bnp(Short,Character)	bnp(Short,Integer)
char	bnp(char,Byte)	bnp(char,Short)	char	bnp(char,Integer)
Character	bnp(Character,Byte)	bnp(Character,Short)	Character	bnp(Character,Integer)
int	byte|bnp(int,Byte)	short|bnp(int,Short)	char|bnp(int,Character)	int
Integer	bnp(Integer,Byte)	bnp(Integer,Short)	bnp(Integer,Character)	Integer
long	bnp(long,Byte)	bnp(long,Short)	bnp(long,Character)	bnp(long,Integer)
Long	bnp(Long,Byte)	bnp(Long,Short)	bnp(Long,Character)	bnp(Long,Integer)
float	bnp(float,Byte)	bnp(float,Short)	bnp(float,Character)	bnp(float,Integer)
Float	bnp(Float,Byte)	bnp(Float,Short)	bnp(Float,Character)	bnp(Float,Integer)

Окончание табл. 15.25-В

Третий→	Byte	Short	Character	Integer
Второй↓				
double	bnp(double,Byte)	bnp(double,Short)	bnp(double,Character)	bnp(double,Integer)
Double	bnp(Double,Byte)	bnp(Double,Short)	bnp(Double,Character)	bnp(Double,Integer)
boolean	lub(Boolean,Byte)	lub(Boolean,Short)	lub(Boolean,Character)	lub(Boolean,Integer)
Boolean	lub(Boolean,Byte)	lub(Boolean,Short)	lub(Boolean,Character)	lub(Boolean,Integer)
null	Byte	Short	Character	Integer
Object	lub(Object,Byte)	lub(Object,Short)	lub(Object,Character)	lub(Object,Integer)

Таблица 15.25-Г. Тип условного выражения (ссылочный третий операнд, часть II)

Третий→	Long	Float	Double	Boolean
Второй↓				
byte	bnp(byte,Long)	bnp(byte,Float)	bnp(byte,Double)	lub(Byte,Boolean)
Byte	bnp(Byte,Long)	bnp(Byte,Float)	bnp(Byte,Double)	lub(Byte,Boolean)
short	bnp(short,Long)	bnp(short,Float)	bnp(short,Double)	lub(Short,Boolean)
Short	bnp(Short,Long)	bnp(Short,Float)	bnp(Short,Double)	lub(Short,Boolean)
char	bnp(char,Long)	bnp(char,Float)	bnp(char,Double)	lub(Character,Boolean)
Character	bnp(Character,Long)	bnp(Character,Float)	bnp(Character,Double)	lub(Character,Boolean)
int	bnp(int,Long)	bnp(int,Float)	bnp(int,Double)	lub(Integer,Boolean)
Integer	bnp(Integer,Long)	bnp(Integer,Float)	bnp(Integer,Double)	lub(Integer,Boolean)
long	long	bnp(long,Float)	bnp(long,Double)	lub(Long,Boolean)
Long	Long	bnp(Long,Float)	bnp(Long,Double)	lub(Long,Boolean)
float	bnp(float,Long)	float	bnp(float,Double)	lub(Float,Boolean)
Float	bnp(Float,Long)	Float	bnp(Float,Double)	lub(Float,Boolean)
double	bnp(double,Long)	bnp(double,Float)	double	lub(Double,Boolean)
Double	bnp(Double,Long)	bnp(Double,Float)	Double	lub(Double,Boolean)
boolean	lub(Boolean,Long)	lub(Boolean,Float)	lub(Boolean,Double)	boolean
Boolean	lub(Boolean,Long)	lub(Boolean,Float)	lub(Boolean,Double)	Boolean
null	Long	Float	Double	Boolean
Object	lub(Object,Long)	lub(Object,Float)	lub(Object,Double)	lub(Object,Boolean)

Таблица 15.25-Д. Тип условного выражения (ссылочный третий операнд, часть III)

Третий→ Второй↓	`null`	`Object`
`byte`	`lub(Byte,null)`	`lub(Byte,Object)`
`Byte`	`Byte`	`lub(Byte,Object)`
`short`	`lub(Short,null)`	`lub(Short,Object)`
`Short`	`Short`	`lub(Short,Object)`
`char`	`lub(Character,null)`	`lub(Character,Object)`
`Character`	`Character`	`lub(Character,Object)`
`int`	`lub(Integer,null)`	`lub(Integer,Object)`
`Integer`	`Integer`	`lub(Integer,Object)`
`long`	`lub(Long,null)`	`lub(Long,Object)`
`Long`	`Long`	`lub(Long,Object)`
`float`	`lub(Float,null)`	`lub(Float,Object)`
`Float`	`Float`	`lub(Float,Object)`
`double`	`lub(Double,null)`	`lub(Double,Object)`
`Double`	`Double`	`lub(Double,Object)`
`boolean`	`lub(Boolean,null)`	`lub(Boolean,Object)`
`Boolean`	`Boolean`	`lub(Boolean,Object)`
`null`	`null`	`lub(null,Object)`
`Object`	`Object`	`Object`

§15.25.2. Числовые условные выражения

Числовые условные выражения являются автономными выражениями (§15.2). Тип числового условного выражения определяется следующим образом.

- Если второй и третий операнды имеют один и тот же тип, то этот тип является типом условного выражения.
- Если один из второго и третьего операндов имеет примитивный тип *T*, а тип другого является результатом применения преобразования упаковки (§5.1.7) к *T*, то типом условного выражения является *T*.
- Если один из операндов имеет тип `byte` или `Byte`, а второй имеет тип `short` или `Short`, то типом условного выражения является `short`.
- Если один из операндов имеет тип *T*, где *T* представляет собой `byte`, `short` или `char`, а второй операнд является константным выражением (§15.28) типа `int`, значение которого представимо типом *T*, то типом условного выражения является *T*.
- Если один из операндов имеет тип *T*, где *T* представляет собой `Byte`, `Short` или `Character`, а второй операнд является константным выражением типа `int`, значение которого представимо типом *U*, который является результатом применения преобразования распаковки к *T*, то типом условного выражения является *U*.
- В противном случае к типам операндов применяется бинарное числовое повышение (§5.6.2), и типом условного выражения является повышенный тип второго и третьего операндов.

Обратите внимание, что бинарное числовое повышение выполняет преобразование набора значений (§5.1.13) и может выполнять преобразование распаковки (§5.1.8).

§15.25.3. Ссылочные условные выражения

Ссылочное условное выражение является поливыражением, если оно находится в контексте присваивания или контексте вызова (§5.2, §5.3). В противном случае оно является автономным выражением.

Если ссылочное условное поливыражение находится в контексте определенного вида с целевым типом T, выражения второго и третьего его операндов аналогично находятся в контексте того же вида с целевым типом T.

Тип ссылочного условного поливыражения совпадает с его целевым типом.

Тип ссылочного условного автономного выражения определяется следующим образом.

- Если второй и третий операнды имеют один и тот же тип (который может быть типом null), то он является типом условного выражения.
- Если тип одного из второго и третьего операндов является типом null, а тип второго операнда является ссылочным типом, то тип условного выражения является этим ссылочным типом.
- В противном случае второй и третий операнды имеют типы S_1 и S_2 соответственно. Пусть T_1 является типом, который является результатом применения преобразования упаковки к S_1, а T_2 — типом, который является результатом применения преобразования упаковки к S_2. Тип условного выражения представляет собой результат применения преобразования при фиксации (§5.1.10) к lub(T_1, T_2).

Поскольку ссылочные условные выражения могут быть поливыражениями, они могут "передавать" контекст своим операндам. Это позволяет лямбда-выражениям и выражениям ссылки на метод выступать в роли операндов:

```
return ... ? (x -> x) : (x -> -x);
```

Это также позволяет использовать дополнительную информацию для улучшения проверки типов вызовов обобщенных методов. До Java SE 8 следующее присваивание было корректно типизированным:

```
List<String> ls = Arrays.asList();
```

а следующее — нет:

```
List<String> ls = ... ? Arrays.asList() : Arrays.
asList("a","b");
```

Приведенные выше правила делают оба присваивания корректно типизированными.

Обратите внимание, что ссылочное условное выражение не обязано *содержать* поливыражение в качестве операнда для того, чтобы *являться* поливыражением. Оно является поливыражением просто в силу контекста, в котором находится. Например, в приведенном далее коде условное выражение является поливыражением, а каждый операнд рассматривается как целевой для контекста присваивания `Class<? super Integer>`:

```
 Class<? super Integer> choose(boolean b,
                               Class<Integer> c1,
```

```
                                        Class<Number> c2) {
    return b ? c1 : c2;
}
```

Если условное выражение не является поливыражением, может произойти ошибка времени компиляции, если его типом является lub(`Class<Integer>`,`Class<Number>`) = `Class<? extends Number>`, который не совместим с возвращаемым типом `choose`.

§15.26. Операторы присваивания

Имеется 12 *операторов присваивания*; все они синтаксически правоассоциативны (группируются справа налево). Таким образом, `a=b=c` означает `a=(b=c)`, где сначала значение `c` присваивается переменной `b`, а затем значение `b` присваивается переменной `a`.

AssignmentExpression:
    ConditionalExpression
    Assignment

Assignment:
    LeftHandSide AssignmentOperator Expression

LeftHandSide:
    ExpressionName
    FieldAccess
    ArrayAccess

AssignmentOperator: одно из
`=  *=  /=  %=  +=  -=  <<=  >>=  >>>=  &=  ^=  |=`

Результатом первого операнда оператора присваивания должна быть переменная, иначе генерируется ошибка времени компиляции.

Этот операнд может быть именованной переменной, такой как локальная переменная или поле текущего объекта или класса, или вычисленной переменной, такой как результат обращения к полю (§15.11) или к массиву (§15.10.3).

Типом выражения присваивания является тип переменной после преобразования при фиксации (§5.1.10).

Во время выполнения результат выражения присваивания представляет собой значение переменной после выполнения присваивания. Результатом выражения присваивания не является сама переменная.

Переменной, объявленной как `final`, не может быть присвоено значение (за исключением случая, когда она определенно не присвоена (§16)), поскольку, когда в качестве выражения используется обращение к такой `final`-переменной, результатом является значение, а не переменная, так что оно не может быть использовано в качестве первого операнда оператора присваивания.

§15.26.1. Простой оператор присваивания =

Если тип правого операнда не может быть преобразован в тип переменной с помощью преобразования присваивания (§5.2), генерируется ошибка времени компиляции.

Во время выполнения выражение вычисляется одним из трех способов.

Если выражение левого операнда является выражением обращения к полю (§15.11) `e.f`, возможно, в одной или нескольких парах скобок, то выполняется следующее.

- Сначала вычисляется выражение `e`. Если вычисление `e` завершается преждевременно, выражение присваивания завершается преждевременно по той же причине.
- Затем вычисляется правый операнд. Если вычисление правого выражения завершается преждевременно, выражение присваивания завершается преждевременно по той же причине.
- Затем, если поле, обозначаемое `e.f`, не является `static` и указанный выше результат вычисления `e` равен `null`, генерируется исключение `NullPointerException`.
- В противном случае переменной, обозначаемой `e.f`, присваивается значение вычисленного выше правого операнда.

Если левый операнд представляет собой выражение доступа к массиву (§15.10.3), возможно, в одной или нескольких парах скобок, выполняется следующее.

- Сначала вычисляется подвыражение ссылки выражения обращения к массиву левого операнда. Если это вычисление завершается преждевременно, то выражение присваивания завершается преждевременно по той же причине; подвыражение индекса (выражения обращения к массиву левого операнда) и правый операнд при этом не вычисляются и присваивание не выполняется.
- В противном случае вычисляется подвыражение индекса выражения обращения к массиву левого операнда. Если это вычисление завершается преждевременно, то выражение присваивания завершается преждевременно по той же причине; правый операнд при этом не вычисляется и присваивание не выполняется.
- В противном случае вычисляется правый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине и присваивание не выполняется.
- В противном случае, если значение подвыражения ссылки на массив равно `null`, присваивание не выполняется и генерируется исключение `NullPointerException`.
- В противном случае значение подвыражения ссылки на массив действительно указывает на массив. Если значение подвыражения индекса меньше нуля или больше или равно длине массива (поле `length`), присваивание не выполняется и генерируется исключение `ArrayIndexOutOfBoundsException`.
- В противном случае значение подвыражения индекса используется для выбора компонента массива, на который указывает значение подвыражения ссылки на массив.

  Этот компонент представляет собой переменную; обозначим ее тип как *SC*. Пусть также *TC* представляет собой тип левого операнда оператора присваивания, определенный во время компиляции. Тогда возможны два варианта.

- Если *TC* является примитивным типом, то *SC* с необходимостью тот же, что и *TC*.

  Значение правого операнда преобразуется в тип выбранного компонента массива и подвергается преобразованию набора значений (§5.1.13) в соответствующий стандартный набор значений (не в расширенный набор значений), а результат преобразования сохраняется в компоненте массива.
- Если *TC* является ссылочным типом, *SC* может не совпадать с *TC*, а быть типом, расширяющим или реализующим *TC*.

  Пусть *RC* представляет собой класс объекта, на который указывает значение правого операнда во время выполнения.

  Компилятор Java может быть способен доказать во время компиляции, что компонент массива имеет в точности тип *TC* (например, *TC* может быть `final`). Но если компилятор Java не может доказать во время компиляции, что компонент массива имеет в точности тип *TC*, то должна быть выполнена проверка времени выполнения, гарантирующая, что класс *RC* совместим по присваиванию (§5.2) с фактическим типом *SC* компонента массива.

  Эта проверка подобна сужающему приведению (§5.5, §15.16) с тем отличием, что, если проверка не пройдена, генерируется исключение `ArrayStoreException`, а не `ClassCastException`.

  Если класс *RC* не является присваиваемым типу *SC*, присваивание не выполняется и генерируется исключение `ArrayStoreException`.

  В противном случае ссылочное значение правого операнда сохраняется в выбранном компоненте массива.

В противном случае требуется выполнение трех шагов.

- Сначала, чтобы получить переменную, вычисляется левый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине; правый операнд не вычисляется и присваивание не выполняется.
- В противном случае вычисляется правый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине и присваивание не выполняется.
- В противном случае значение правого операнда преобразуется в тип левой переменной и подвергается преобразованию набора значений (§5.1.13) в соответствующий стандартный набор значений (не в расширенный набор значений), а результат преобразования сохраняется в переменной.

Пример 15.26.1-1. Простое присваивание компоненту массива

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow          extends RuntimeException { }
class RightHandSideThrow  extends RuntimeException { }

class IllustrateSimpleArrayAssignment {
    static Object[] objects = { new Object(), new Object() };
```

```
    static Thread[] threads = { new Thread(), new Thread() };

    static Object[] arrayThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() {
        throw new IndexThrow();
    }
    static Thread rightThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }
    static void testFour(Object[] x, int j, Object y) {
        String sx = x == null ? "null" : name(x[0]) + "s";
        String sy = name(y);
        System.out.println();
        try {
            System.out.print(sx + "[throw]=throw => ");
            x[indexThrow()] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[throw]=" + sy + " => ");
            x[indexThrow()] = y;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]=throw => ");
            x[j] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]=" + sy + " => ");
            x[j] = y;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
    }
    public static void main(String[] args) {
    try {
        System.out.print("throw[throw]=throw => ");
        arrayThrow()[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
```

```
            System.out.print("throw[throw]=Thread => ");
            arrayThrow()[indexThrow()] = new Thread();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]=throw => ");
            arrayThrow()[1] = rightThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]=Thread => ");
            arrayThrow()[1] = new Thread();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }

        testFour(null, 1, new StringBuffer());
        testFour(null, 9, new Thread());
        testFour(objects, 1, new StringBuffer());
        testFour(objects, 1, new Thread());
        testFour(objects, 9, new StringBuffer());
        testFour(objects, 9, new Thread());
        testFour(threads, 1, new StringBuffer());
        testFour(threads, 1, new Thread());
        testFour(threads, 9, new StringBuffer());
        testFour(threads, 9, new Thread());
        }
}
```

Вывод этой программы имеет следующий вид.

```
throw[throw]=throw => ArrayReferenceThrow
throw[throw]=Thread => ArrayReferenceThrow
throw[1]=throw => ArrayReferenceThrow
throw[1]=Thread => ArrayReferenceThrow

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=StringBuffer => Okay!

Objects[throw]=throw => IndexThrow
```

```
Objects[throw]=Thread => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=Thread => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=StringBuffer => ArrayIndexOutOfBoundsException

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=Thread => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=StringBuffer => ArrayStoreException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=Thread => Okay!

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=StringBuffer => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=Thread => ArrayIndexOutOfBoundsException
```

Наиболее интересный случай — тринадцатый с конца.

```
Threads[1]=StringBuffer => ArrayStoreException
```

Он указывает, что при попытке сохранить ссылку на `StringBuffer` **в массив, компоненты которого имеют тип** `Thread`**, генерируется исключение** `ArrayStoreException`**. Код с точки зрения типов во время компиляции корректен: слева у присваивания тип** `Object[]`**, а справа — тип** `Object`**. Во время выполнения первый фактический аргумент метода** `testFour` **представляет собой ссылку на экземпляр "массива объектов типа** `Thread`**", а третий фактический аргумент представляет собой ссылку на экземпляр класса** `StringBuffer`**.**

§15.26.2. Составные операторы присваивания

Выражение составного присваивания вида *E1 op= E2* эквивалентно *E1 = (T) ((E1) op (E2))*, где *T* представляет собой тип *E1*, с тем отличием, что *E1* вычисляется только однократно.

Например, следующий код корректен.

```
short x = 3;
x += 4.6;
```

И результат, сохраненный в `x`, имеет значение `7`, поскольку этот код эквивалентен следующему.

```
short x = 3;
x = (short)(x + 4.6);
```

Во время выполнения программы выражение вычисляется одним из двух возможных путей.

Если выражение левого операнда не является выражением обращения к массиву, выполняется следующее.

- Сначала для получения переменной вычисляется левый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине; правый операнд не вычисляется и присваивание не выполняется.
- В противном случае значение левого операнда сохраняется, после чего вычисляется правый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине и присваивание не выполняется.
- В противном случае сохраненное значение левой переменной и значение правого операнда используются для выполнения бинарной операции, указанной в операторе составного присваивания. Если эта операция завершается преждевременно, выражение присваивания завершается преждевременно по той же причине, и присваивание не выполняется.
- В противном случае результат бинарной операции преобразуется в тип левой переменной и над ним выполняется преобразование набора значений (§5.1.13) в соответствующий стандартный набор значений (не в расширенный набор значений), а результат преобразования сохраняется в переменной.

Если выражение левого операнда представляет собой выражение обращения к массиву (§15.10.3), то выполняются следующие действия.

- Сначала вычисляется подвыражение ссылки на массив выражения обращения к массиву левого операнда. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине; подвыражение индекса (выражения обращения к массиву левого операнда) и правый операнд не вычисляются и присваивание не выполняется.
- В противном случае вычисляется подвыражение индекса выражения обращения к массиву левого операнда. Если это вычисление завершается преждевременно, то выражение присваивания завершается преждевременно по той же причине; правый операнд не вычисляется и присваивание не выполняется.
- В противном случае, если значение подвыражения ссылки на массив равно `null`, присваивание не выполняется и генерируется исключение `NullPointerException`.

- В противном случае значение подвыражения ссылки на массив в действительности указывает на массив. Если значение подвыражения индекса меньше нуля или не меньше длины массива, то присваивание не выполняется и генерируется исключение `ArrayIndexOutOfBoundsException`.
- В противном случае значение подвыражения индекса используется для выбора компонента массива, на который указывает значение подвыражения ссылки на массив. Значение этого компонента сохраняется, после чего вычисляется правый операнд. Если это вычисление завершается преждевременно, выражение присваивания завершается преждевременно по той же причине, и присваивание не выполняется.

  В случае простого оператора присваивания вычисление правого операнда выполняется перед проверкой подвыражения ссылки на массив и подвыражения индекса, но в случае составного оператора присваивания вычисление правого операнда выполняется после этих проверок.

- В противном случае рассмотрим компонент массива, выбранный на предыдущем шаге, значение которого было сохранено. Этот компонент представляет собой переменную; назовем ее "тип S". Пусть тип T — тип левого операнда оператора присваивания, определенный во время компиляции.

  ✦ Если T является примитивным типом, то S с необходимостью совпадает с T.

  Сохраненное значение компонента массива и значение правого операнда используются для выполнения бинарной операции, указанной в составном операторе присваивания.

  Если эта операция завершается преждевременно (единственная возможность этого — целочисленное деление на нуль, см. §15.17.2), то выражение присваивания завершается преждевременно по той же причине, и присваивание не выполняется.

  В противном случае результат бинарной операции преобразуется в тип выбранного компонента массива и над ним выполняется преобразование набора значений (§5.1.13) в соответствующий стандартный набор значений (не в расширенный набор значений), а результат преобразования сохраняется в компоненте массива.

  ✦ Если T представляет собой ссылочный тип, то он должен представлять собой `String`. Поскольку класс `String` является `final`-классом, S также должно быть типом `String`.

  Поэтому необходимая иногда для оператора простого присваивания проверка времени выполнения никогда не требуется для составного оператора присваивания.

  Сохраненное значение компонента массива и значение правого операнда используются для выполнения бинарной операции (конкатенации строк), указанной в составном операторе присваивания (который с необходимостью представляет собой +=). Если эта операция завершается преждевременно, выражение присваивания завершается преждевременно по той же причине, и присваивание не выполняется.

  В противном случае результат бинарной операции типа `String` сохраняется в компоненте массива.

Пример 15.26.2-1. Составное присваивание компоненту массива

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow          extends RuntimeException { }
class RightHandSideThrow  extends RuntimeException { }

class IllustrateCompoundArrayAssignment {
    static String[] strings = { "Simon", "Garfunkel" };
    static double[] doubles = { Math.E, Math.PI };

    static String[] stringsThrow() {
        throw new ArrayReferenceThrow();
    }
    static double[] doublesThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() {
        throw new IndexThrow();
    }
    static String stringThrow() {
        throw new RightHandSideThrow();
    }
    static double doubleThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }

    static void testEight(String[] x, double[] z, int j) {
        String sx = (x == null) ? "null" : "Strings";
        String sz = (z == null) ? "null" : "doubles";
        System.out.println();
        try {
            System.out.print(sx + "[throw]+=throw => ");
            x[indexThrow()] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=throw => ");
            z[indexThrow()] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[throw]+=\"heh\" => ");
            x[indexThrow()] += "heh";
            System.out.println("Okay!");
```

```
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[throw]+=12345 => ");
            z[indexThrow()] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=throw => ");
            x[j] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=throw => ");
            z[j] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sx + "[" + j + "]+=\"heh\" => ");
            x[j] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print(sz + "[" + j + "]+=12345 => ");
            z[j] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
    }

    public static void main(String[] args) {
        try {
            System.out.print("throw[throw]+=throw => ");
            stringsThrow()[indexThrow()] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=throw => ");
            doublesThrow()[indexThrow()] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=\"heh\" => ");
            stringsThrow()[indexThrow()] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[throw]+=12345 => ");
            doublesThrow()[indexThrow()] += 12345;
            System.out.println("Okay!");
```

```
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            stringsThrow()[1] += stringThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=throw => ");
            doublesThrow()[1] += doubleThrow();
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=\"heh\" => ");
            stringsThrow()[1] += "heh";
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        try {
            System.out.print("throw[1]+=12345 => ");
            doublesThrow()[1] += 12345;
            System.out.println("Okay!");
        } catch (Throwable e) { System.out.println(name(e)); }
        testEight(null, null, 1);
        testEight(null, null, 9);
        testEight(strings, doubles, 1);
        testEight(strings, doubles, 9);
    }
}
```

Вывод данной программы имеет следующий вид.

```
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+="heh" => ArrayReferenceThrow
throw[throw]+=12345 => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+="heh" => ArrayReferenceThrow
throw[1]+=12345 => ArrayReferenceThrow

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[1]+=throw => NullPointerException
null[1]+=throw => NullPointerException
null[1]+="heh" => NullPointerException
null[1]+=12345 => NullPointerException

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
```

```
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[9]+=throw => NullPointerException
null[9]+=throw => NullPointerException
null[9]+="heh" => NullPointerException
null[9]+=12345 => NullPointerException

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
Strings[1]+="heh" => Okay!
doubles[1]+=12345 => Okay!

Strings[throw]+=throw => IndexThrow
doubles[throw]+=throw => IndexThrow
Strings[throw]+="heh" => IndexThrow
doubles[throw]+=12345 => IndexThrow
Strings[9]+=throw => ArrayIndexOutOfBoundsException
doubles[9]+=throw => ArrayIndexOutOfBoundsException
Strings[9]+="heh" => ArrayIndexOutOfBoundsException
doubles[9]+=12345 => ArrayIndexOutOfBoundsException
```

Наиболее интересные случаи — одиннадцатый и двенадцатый с конца.

```
Strings[1]+=throw => RightHandSideThrow
doubles[1]+=throw => RightHandSideThrow
```

Это случаи, когда фактически сгенерированное исключение представляет собой исключение, сгенерированное правой частью оператора. Более того, это единственные такие случаи среди всех. Это показывает, что вычисление правого операнда действительно происходит после проверок ссылки массива на равенство `null` и выхода индекса за пределы диапазона.

Пример 15.26.2-2. Значение левого операнда составного присваивания сохраняется до вычисления правого операнда

```
class Test {
    public static void main(String[] args) {
        int k = 1;
        int[] a = { 1 };
        k += (k = 4) * (k + 2);
        a[0] += (a[0] = 4) * (a[0] + 2);
        System.out.println("k==" + k + " and a[0]==" + a[0]);
    }
}
```

Вывод данной программы имеет следующий вид.

```
k==25 and a[0]==25
```

Значение 1 переменной k сохраняется составным оператором присваивания += перед тем, как вычисляется правый операнд (k = 4)*(k + 2). Вычисление этого правого операнда присваивает значение 4 переменной k, вычисляет значение 6 выражения k + 2, а затем умножает 4 на 6, что дает 24. Эта величина добавляется к сохраненному значению 1, что дает 25, а затем сохраняется в переменной k оператором +=. Идентичный анализ применим и к случаю с использованием a[0].

Вкратце, инструкции

```
k += (k = 4) * (k + 2);
a[0] += (a[0] = 4) * (a[0] + 2);
```

ведут себя в точности так же, как и инструкции

```
k = k + (k = 4) * (k + 2);
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```

§15.27. Лямбда-выражения

Лямбда-выражение подобно методу: оно предоставляет список формальных параметров и тело — выражение или блок, — выраженное с использованием этих параметров.

LambdaExpression:

LambdaParameters -> *LambdaBody*

Лямбда-выражение всегда является поливыражением (§15.2).

Если лямбда-выражение находится в программе в месте, отличном от контекста присваивания (§5.2), контекста вызова (§5.3) и контекста приведения (§5.5), генерируется ошибка времени компиляции.

Вычисление лямбда-выражения создает экземпляр функционального интерфейса (§9.8). Вычисление лямбда-выражения *не* приводит к вычислению тела выражения; оно может осуществиться позже, когда будет вызван соответствующий метод функционального интерфейса.

Вот несколько примеров лямбда-выражений.

```
()-> {}                  // Параметров нет; результат void
()-> 42                  // Параметров нет; тело выражения
()-> null                // Параметров нет; тело выражения
()-> { return 42;}       // Параметров нет; тело блока с возвратом
()-> { System.gc();} // Параметров нет; тело блока void

()-> {                   // Сложное тело блока с возвратом
    if (true) return 12;
    else {
        int result = 15;
        for (int i = 1; i < 10; i++)
            result *= i;
        return result;
    }
}

(int x) -> x+1       // Единственный параметр объявленного типа
```

```
(int x) -> {          // Единственный параметр объявленного типа
    return x+1; }
(x) -> x+1            // Единственный параметр объявленного типа
x -> x+1              // Для единственного параметра с выводимым
                      // типом не обязательны

(String s) ->         // Единственный параметр объявленного типа
    s.length()
(Thread t) ->         // Единственный параметр объявленного типа
    { t.start(); }
s -> s.length()       // Единственный параметр выводимого типа
t -> { t.start();}    // Единственный параметр выводимого типа
(int x, int y) ->     // Несколько параметров объявленного типа
    x+y
(x, y) -> x+y         // Несколько параметров выводимого типа
(x, int y) -> x+y     // Неверно: нельзя смешивать объявленные
                      // и выводимые типы
(x, final y) -> x+y   // Неверно: модификаторы с выводимыми
                      // типами недопустимы
```

Этот синтаксис обладает тем преимуществом, что минимизирует "скобочный шум" вокруг простых лямбда-выражений, что особенно полезно, когда лямбда-выражение является аргументом метода или когда тело представляет собой иное лямбда-выражение. Он также позволяет четко различать выражения и инструкции, избегая неоднозначностей и чрезмерной ориентации на токены '`;`'. При необходимости добавления излишних скобок для визуального различия полного лямбда-выражения и его тела обеспечивается естественная поддержка скобок (как и в прочих случаях, когда не очевиден приоритет операторов).

Синтаксис обладает определенными сложностями при анализе. Язык программирования Java всегда имел неоднозначность с типами и выражениями после токена '`(`': за ним может следовать как приведение, так и выражение в скобках. Ситуация ухудшилась с применением обобщенными типами бинарных операторов '`<`' и '`>`' в типах. Лямбда-выражения вводят новую сложность: токены, следующие за '`(`', могут описывать тип, выражение или список параметров лямбда-выражения. Некоторые токены (аннотации, `final`) являются уникальными и могут появляться только в списке параметров, хотя имеются и некоторые иные случаи, которые должны интерпретироваться только как списки параметров (два имени в строке, запятая '`,`', не вложенная между '`<`' и '`>`'). Иногда неоднозначность не может быть разрешена до достижения '`->`' после '`)`'. Простейший способ решения этой задачи — с использованием конечного автомата: каждое состояние представляет подмножество возможных интерпретаций (тип, выражение или параметры), и когда автомат переходит в состояние, в котором множество состоит из одного элемента, синтаксический анализатор знает, с чем имеет дело. Однако это решение не очень элегантно отображается на грамматику с фиксированным предпросмотром.

Не имеется специальной нуль-арной записи: лямбда-выражение с нулевым количеством аргументов записывается как `() -> ...`. Напрашивающийся очевидный

синтаксис для данного частного случая `->  . . .` не работает, поскольку вносит неоднозначность между списками аргументов и приведениями: `(x)  ->  . . .`.

Лямбда-выражения не могут объявлять параметры типов. Хотя семантически это и имело бы смысл, естественный синтаксис (список параметров типов, предшествующий списку параметров) вносит массу неоднозначностей. Рассмотрим, например, фрагмент

```
foo( (x) < y , z > (w) -> v )
```

Это может быть вызов `foo` с одним аргументом (обобщенное лямбда-выражение с приведением к типу `x`), вызовом `foo` с двумя аргументами, каждый из которых является результатом сравнения, причем второе сравнение — `z` с лямбда-выражением. (Строго говоря, лямбда-выражение не имеет смысла в качестве операнда оператора отношения >, но это тонкое предположение, на котором строится грамматика.)

Имеется прецедент разрешения неоднозначности при применении приведений, который по сути запрещает применение - и + после приведения к непримитивному типу (§15.15), но распространение этого подхода на лямбда-выражения в общем случае требует существенного изменения грамматики.

§15.27.1. Параметры лямбда-выражения

Формальными параметрами лямбда-выражения могут быть либо объявленные, либо выводимые типы. Эти стили нельзя смешивать: недопустимо лямбда-выражение, объявляющее типы некоторых своих параметров и требующее выведения типов других. Модификаторы могут иметь только параметры с объявленными типами.

LambdaParameters:
    Identifier
    (*[FormalParameterList]*)
    (*InferredFormalParameterList*)

InferredFormalParameterList:
    Identifier {, Identifier}

Далее для удобства приведены продукции из §4.3, §8.3 и §8.4.1.

FormalParameterList:
    FormalParameters , LastFormalParameter
    LastFormalParameter

FormalParameters:
    FormalParameter {, FormalParameter}
    ReceiverParameter {, FormalParameter}

FormalParameter:
    {VariableModifier} UnannType VariableDeclaratorId

LastFormalParameter:

{VariableModifier} UnannType {Annotation} ... VariableDeclaratorId
FormalParameter

VariableModifier: одно из
Annotation `final`

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} [ ] *{{Annotation}* [ ]*}*

В *FormalParameters* лямбда-выражения не допускаются параметры-получатели, определенные в §8.4.1.

Лямбда-выражение, формальные параметры которого имеют объявленные типы, называется *явно типизированным*, в то время как лямбда-выражение, формальные параметры которого имеют выводимые типы, называется *неявно типизированным*. Лямбда-выражение с нулевым количеством параметров является явно типизированным.

Если формальные параметры имеют выводимые типы, то эти типы выводятся (§15.27.3) из типа функционального интерфейса, являющегося целевым для лямбда-выражения.

Синтаксис формальных параметров с объявленными типами совпадает с синтаксисом формальных параметров объявления метода (§8.4.1).

Объявленный тип формального параметра обозначается нетерминалом *UnannType*, который находится в спецификаторе параметра и за которым в декларатоpе следуют *Identifier*, а за ним — любое количество пар квадратных скобок, за исключением случая параметра переменной арности, объявленный тип которого является типом массива с типом компонента *UnannType*, который находится в спецификаторе параметра.

Нет никакой разницы между следующими списками параметров лямбда-выражений.

```
(int... x) -> ..
(int[] x) -> ..
```

В соответствии с правилами перекрытия может использоваться абстрактный метод функционального интерфейса как фиксированной арности, так и переменной арности. Поскольку лямбда-выражения никогда не вызываются непосредственно, применение `int...` там, где функциональный интерфейс использует `int[]`, никак не влияет на окружающую программу. В теле лямбда-выражения параметр переменной арности рассматривается как параметр с типом массива.

Правила применения модификаторов аннотаций в объявлениях формальных параметров описаны в §9.7.4 и §9.7.5.

Если `final` встречается в объявлении формального параметра в качестве модификатора более одного раза, генерируется ошибка времени компиляции.

Использование для параметров переменной арности смешанной записи для массивов (§10.2) приводит к ошибке времени компиляции.

Область видимости и затенение объявления формального параметра описаны в §6.3 и §6.4.

Если лямбда-выражение объявляет два формальных параметра с одним именем (т.е. в их объявлениях упоминается один и тот же *Identifier*), генерируется ошибка времени компиляции.

Если параметр лямбда-выражения имеет имя "_" (т.е. единственный символ подчеркивания), генерируется ошибка времени компиляции.

> Не рекомендуется использовать в качестве имени переменной "_" ни в каком контексте. Будущие версии языка программирования Java могут зарезервировать это имя как ключевое слово и/или придать ему специальную семантику.

Если параметр-получатель (§8.4.1) встречается в *FormalParameters* лямбда-выражения, генерируется ошибка времени компиляции.

Если в теле лямбда-выражения выполняется присваивание формальному параметру, объявленному как `final`, генерируется ошибка времени компиляции.

При вызове лямбда-выражения (посредством выражения вызова метода (§15.12)) значения выражений фактических аргументов инициализируют вновь создаваемые переменные параметров, каждая своего объявленного или выведенного типа, перед выполнением тела лямбда-выражения. Нетерминал *Identifier*, находящийся в *VariableDeclaratorId* или *InferredFormalParameterList*, может использоваться в качестве простого имени в теле лямбда-выражения для обращения к формальному параметру.

Параметр лямбда-выражения типа `float` всегда содержит элемент набора значений float (§4.2.3); аналогично параметр лямбда-выражения типа `double` всегда содержит элемент набора значений double. Параметр лямбда-выражения типа `float` не может содержать элемент набора значений float с расширенным показателем степени, который не является элементом набора значений float; аналогично параметр лямбда-выражения типа `double` не может содержать элемент набора значений double с расширенным показателем степени, который не является элементом набора значений double.

> При выведении типов параметров лямбда-выражений одно и то же тело лямбда-выражения может интерпретироваться различными способами в зависимости от контекста, в котором оно находится. В частности, типы выражений в теле лямбда-выражения, проверяемые исключения, генерируемые телом лямбда-выражения, и корректность типов кода тела лямбда-выражения зависят от правильности выведенных типов параметров. Отсюда следует, что вывод типов параметров должен происходить "до" попыток проверки типов в теле лямбда-выражения.

§15.27.2. Тело лямбда-выражения

Тело лямбда-выражения представляет собой либо единственное выражение, либо блок (§14.2). Подобно телу метода, тело лямбда-выражения описывает код, который будет выполнен при вызове.

LambdaBody:
    Expression
    Block

В отличие от кода в объявлениях анонимных классов, значения имен и ключевых слов `this` и `super`, находящихся в теле лямбда-выражения, наряду с доступностью объявлений, на которые они ссылаются, являются теми же, что и в окружающем контексте (за исключением того, что параметры лямбда-выражения вводят новые имена).

> Прозрачность `this` (явная и неявная) в теле лямбда-выражения — т.е. его рассмотрение так же, как и в окружающем контексте — обеспечивает большую гибкость реализаций и предотвращает зависимость смысла неквалифицированных имен в теле лямбда-выражения от разрешения перегрузки.
>
> Говоря с практической точки зрения, необходимость лямбда-выражения обращаться к самому себе (либо при рекурсивном вызове, либо для вызова других его методов) достаточно необычна; гораздо более распространено использование имен для обращения к сущностям в охватывающем классе, которые иначе оказываются затененными (`this`, `toString()`). Если лямбда-выражению необходимо обращаться к самому себе (с помощью `this`), следует использовать ссылку на метод или анонимный внутренний класс.

Блок тела лямбда-выражения является *void-совместимым*, если все инструкции возврата в блоке имеют вид `return;`.

Блок тела лямбда-выражения является *совместимым со значением*, если он не может завершиться нормально (§14.21) и каждая инструкция возврата в блоке имеет вид `return` *Expression*`;`.

Если блок тела лямбда-выражения не является ни void-совместимым, ни совместимым со значением, генерируется ошибка времени компиляции.

В блоке тела лямбда-выражения, совместимом со значением, *выражение результата* представляет собой любое выражение, которое может производить значение вызова. В частности, для каждой инструкции вида `return` *Expression*`;`, содержащейся в теле, *Expression* представляет собой выражение результата.

> Следующие лямбда-выражения являются void-совместимыми.
>
> ```
> () -> {}
> () -> { System.out.println("done"); }
> ```
>
> Следующие лямбда-выражения являются совместимыми со значениями.
>
> ```
> () -> { return "done"; }
> () -> { if (...) return 1; else return 0; }
> ```
>
> Эти лямбда-выражения обладают обоими свойствами совместимости.
>
> ```
> () -> { throw new RuntimeException(); }
> () -> { while (true); }
> ```
>
> Эти лямбда-выражения не обладают ни одним свойством совместимости.
>
> ```
> () -> { if (...) return "done"; System.out.println("done"); }
> ```
>
> Обработка void-совместимых и совместимых со значениями тел лямбда-выражений и смысла имен в телах лямбда-выражений совместно служат минимизации зависимости определенного целевого типа в данном контексте, что является полезным как для реализаций, так и для понимания программистами. В то время как выражения могут быть присвоены различным типам во время разрешения перегрузки

в зависимости от целевого типа, значение неквалифицированных имен и базовая структура тела лямбда-выражения не изменяются.

Обратите внимание, что определение void-совместимости и совместимости со значениями не является строго структурным свойством: "может завершиться нормально" зависит от значений константных выражений, и они могут включать имена, которые ссылаются на константные переменные.

Любая локальная переменная, формальный параметр или параметр исключения, использованный, но не объявленный в лямбда-выражении, должен либо быть объявлен как `final`, либо быть фактически финальным (§4.12.4), иначе при попытке его использования генерируется ошибка времени компиляции.

Любая локальная переменная, использованная, но не объявленная в теле лямбда-выражения, должна быть определенно присвоенной (§16) до тела лямбда-выражения, иначе генерируется ошибка времени компиляции.

Аналогичные правила использования переменных применяются в теле внутреннего класса (§8.1.3). Ограничение на фактически финальные переменные запрещают доступ для динамического изменения локальных переменных, фиксация которых может привести к проблемам параллельности. По сравнению с ограничением `final` это снижает ограничения для программистов.

Ограничения на фактически финальные переменные включают переменные стандартных циклов, но не переменные расширенного цикла `for`, которые рассматриваются как разные на каждой итерации цикла (§14.14.2).

Следующие тела лямбда-выражений демонстрируют использование фактически финальных переменных.

```
void m1(int x) {
    int y = 1;
    foo(() -> x+y);
    // Корректно: и x, и y фактически финальные.
}
void m2(int x) {
    int y;
    y = 1;
    foo(() -> x+y);
    // Корректно: и x, и y фактически финальные.
}
void m3(int x) {
    int y;
    if (...) y = 1;
    foo(() -> x+y);
    // Неверно: y фактически финальная,
    // но не определенно присвоенная.
}
void m4(int x) {
    int y;
    if (...) y = 1; else y = 2;
    foo(() ->; x+y);
```

```
        // Корректно: и x, и y фактически финальные.
    }
    void m5(int x) {
        int y;
        if (...) y = 1;
        y = 2;
        foo(() -> x+y);
        // Неверно: y не фактически финальная.
    }
    void m6(int x) {
        foo(() -> x+1);
        x++;
        // Неверно: x не фактически финальная.
    }
    void m7(int x) {
        foo(() -> x=1);
        // Неверно: x не фактически финальная.
    }
    void m8() {
        int y;
        foo(() -> y=1);
        // Неверно: y не является определенно
        // присвоенной до лямбда-выражения.
    }
        void m9(String[] arr) {
        for (String s : arr) {
            foo(() -> s);
            // Корректно: s фактически финальная
            // (является новой переменной на каждой итерации)
        }
    }
    void m10(String[] arr) {
        for (int i = 0; i < arr.length; i++) {
            foo(() -> arr[i]);
            // Неверно: i не является фактически финальной
            // (она не финальная и увеличивается)
        }
    }
```

§15.27.3. Тип лямбда-выражения

Лямбда-выражение является совместимым в контексте присваивания, контексте вызова или контексте приведения с целевым типом T, если T представляет собой тип функционального интерфейса (§9.8) и выражение *конгруэнтно* типу функции *базового целевого типа*, производного от T.

Базовый целевой тип получается из T следующим образом.

- Если *T* представляет собой тип функционального интерфейса, параметризованного символами подстановки, а лямбда-выражение явно типизировано, то базовый целевой тип выводится так, как описано в §18.5.3.
- Если *T* представляет собой тип функционального интерфейса, параметризованного символами подстановки, а лямбда-выражение типизировано неявно, то базовый целевой тип представляет собой параметризацию *T* без символов подстановки (§9.9).
- В противном случае базовый целевой тип представляет собой *T*.

Лямбда-выражение *конгруэнтно* с типом функции, если истинны все следующие условия.

- Тип функции не имеет параметров типа.
- Количество параметров лямбда-выражения совпадает с количеством типов параметров типа функции.
- Если лямбда-выражение явно типизировано, типы его формальных параметров совпадают с типами параметров типа функции.
- Если предполагается, что параметры лямбда-выражения имеют те же типы, что и типы параметров типа функции, то:
  - ✦ если тип результата функции — `void`, тело лямбда-выражения представляет собой либо выражение инструкции, либо `void`-совместимый блок;
  - ✦ если тип результата функции представляет собой (не `void`) тип *R*, то либо 1) тело лямбда-выражения представляет собой выражение, совместимое с *R* в контексте присваивания, либо 2) тело лямбда-выражения представляет собой совместимый со значением блок, а каждое выражение результата (§15.27.2) совместимо с *R* в контексте присваивания.

Если лямбда-выражение совместимо с целевым типом *T*, то тип выражения, *U*, представляет собой базовый целевой тип, производный от *T*.

Если любой класс или интерфейс, упомянутый *U* или типом функции *U*, не является доступным из класса или интерфейса, в котором находится лямбда-выражение, генерируется ошибка времени компиляции.

Для каждого нестатического метода-члена *m* типа *U*, если тип функции *U* имеет подсигнатуру сигнатуры *m*, воображаемый метод, тип метода которого представляет собой тип функции *U*, считается перекрывающим *m*, и может быть сгенерирована любая ошибка времени компиляции или предупреждение о непроверенном типе, определенные в §8.4.8.3.

Проверяемое исключение, которое может быть сгенерировано в теле лямбда-выражения, может привести к ошибке времени компиляции, как указано в §11.2.3.

> Типы параметров явно типизированных лямбда-выражений обязаны точно соответствовать типу функции. Хотя можно было бы быть и более гибким — например, допуская упаковку или вариации типов, — этот вид обобщенности кажется излишним и не согласуется со способом работы перекрытия в объявлениях классов. Программист при написании лямбда-выражения должен точно знать, какой тип функции является целевым, так что он должен точно знать, какая сигнатура долж

на быть перекрыта. (В отличие от лямбда-выражений это не так для ссылок на методы, так что при их использовании разрешается бо́льшая гибкость.) Кроме того, бо́льшую гибкость типам параметров будут добавлять сложности вывода типа и разрешения перегрузки.

Обратите внимание, что, в то время как в контексте строгого вызова упаковка запрещена, упаковка результата лямбда-выражения всегда *разрешена*, т.е. выражение результата находится в контексте присваивания независимо от контекста, охватывающего лямбда-выражение. Однако, если явно типизированное лямбда-выражение является аргументом перегруженного метода, сигнатура метода, который обходится без упаковки или распаковки результата лямбда-выражения, является предпочтительной при выборе наиболее подходящего метода (§15.12.2.5).

Если тело лямбда-выражения представляет собой выражение инструкции (т.е. выражение, которое может использоваться отдельно в качестве инструкции), то оно совместимо с типом функции, возвращающей `void`; любой его результат просто игнорируется. Так, например, оба приведенных ниже фрагмента корректны.

```
// Predicate имеет булев результат
java.util.function.Predicate<String> p = s -> list.add(s);
// Consumer имеет результат void
java.util.function.Consumer<String> c = s -> list.add(s);
```

Вообще говоря, лямбда-выражение вида `() ->` *expr*, где *expr* представляет собой выражение инструкции, интерпретируется либо как `() -> {` `return` *expr;* `}`, либо как `() -> {` *expr;* `}` в зависимости от целевого типа.

§15.27.4. Вычисление лямбда-выражений во время выполнения

Во время выполнения вычисление лямбда-выражения аналогично вычислению выражения создания экземпляра класса, поскольку в качестве нормального завершения производит ссылку на объект. Вычисление лямбда-выражения отличается от вычисления тела лямбда-выражения.

При этом либо создается и инициализируется новый экземпляр класса с описанными ниже свойствами, либо используется ссылка на существующий экземпляр класса с описанными ниже свойствами. Если создается новый экземпляр, но при этом не хватает памяти для выделения объекту, вычисление лямбда-выражения завершается преждевременно генерацией исключения `OutOfMemoryError`.

Значением лямбда-выражения является ссылка на экземпляр класса со следующими свойствами.

- Класс реализует целевой тип функционального интерфейса, а если целевой тип представляет собой тип пересечения, то все типы интерфейсов, упомянутые в пересечении.
- Если выражение лямбда-выражения имеет тип U, то для каждого нестатического метода-члена m типа U справедливо следующее.

  Если тип функции U имеет подсигнатуру сигнатуры m, то класс объявляет метод, который перекрывает m. Тело метода вычисляет тело лямбда-выражения, если оно пред-

ставляет собой выражение, или выполняет тело лямбда-выражения, если оно является блоком; если ожидается результат, он возвращается методом.

Если затирание типа перекрытого метода отличается сигнатурой от затирания типа функции *U*, то до вычисления или выполнения тела лямбда-выражения тело метода убеждается, что каждое значение аргумента является экземпляром подкласса или подынтерфейса затирания соответствующего типа параметра в типе функции *U*; если это не так, генерируется исключение `ClassCastException`.

- Класс не перекрывает другие методы целевого типа функционального интерфейса или другие типы интерфейсов, упомянутые выше, хотя может перекрывать методы класса `Object`.

Эти правила призваны обеспечить следующую гибкость реализации языка программирования Java.

- Новый объект не обязан создаваться при каждом вычислении лямбда-выражения.
- Объекты, производимые различными лямбда-выражениями, не обязаны принадлежать разным классам (например, если их тела идентичны).
- Каждый создаваемый при вычислении объект не обязан принадлежать одному и тому же классу (например, фиксация локальных переменных может быть встроенной).
- Если доступен "существующий экземпляр", он не обязан быть созданным предыдущим вычислением лямбда-выражения (например, он может быть создан во время инициализации охватывающего класса).

Если целевой тип функционального интерфейса является подтипом `java.io.Serializable`, получающийся в результате объект автоматически является экземпляром сериализуемого класса. Придание объекту, производному от лямбда-выражения, свойства сериализации может привести к дополнительным накладным расходам и нежелательным последствиям в смысле безопасности, так что порождаемые лямбда-выражениями объекты не должны быть сериализуемыми "по умолчанию".

§15.28. Константные выражения

ConstantExpression:
Expression

Константное выражение времени компиляции представляет собой выражение, обозначающее значение примитивного типа или типа `String`, которое не завершается преждевременно и составлено только с применением следующих сущностей.

- Литералы примитивного типа и литералы типа `String` (§3.10.1–§3.10.5).
- Приведения к примитивным типам и к типу `String` (§15.16).
- Унарные операторы `+`, `-`, `~` и `!` (но не `++` или `--`) (§15.15.3–§15.15.6).
- Мультипликативные операторы `*`, `/` и `%` (§15.17).
- Аддитивные операторы `+` и `-` (§15.18).

- Операторы сдвига <<, >> и >>> (§15.19).
- Операторы отношения <, <=, > и >= (но не `instanceof`) (§15.20).
- Операторы равенства == и != (§15.21).
- Побитовые и логические операторы &, ^ и | (§15.22).
- Операторы условного И && и условного ИЛИ || (§15.23, §15.24).
- Тернарный условный оператор ?: (§15.25).
- Выражения в скобках (§15.8.5), где содержащиеся в скобках выражения являются константными.
- Простые имена (§6.5.6.1), указывающие на константные переменные (§4.12.4).
- Квалифицированные имена (§6.5.6.2) вида *TypeName* . *Identifier*, указывающие на константные переменные (§4.12.4).

Константные выражения типа `String` всегда "интернированы", так что, чтобы совместно использовать единственные экземпляры, используйте метод `String.intern`.

Константные выражения всегда рассматриваются как FP-строгие (§15.4), даже если они находятся в контексте, где неконстантное выражение не рассматривалось бы как FP-строгое.

Константные выражения времени компиляции используются в метках `case` в инструкциях `switch` (§14.11) и имеют особую важность для преобразования присваивания (§5.2) и инициализации класса или интерфейса (§12.4.2). Они также могут управлять возможностью нормального завершения инструкций `while`, `do` и `for` (§14.21), а также типом условного оператора ?: с числовыми операндами.

Пример 15.28-1. Константные выражения

```
true
(short)(1*2*3*4*5*6)
Integer.MAX_VALUE / 2
2.0 * Math.PI
"The integer " + Long.MAX_VALUE + " is mighty big."
```

ГЛАВА 16

Определенное присваивание

Каждая локальная переменная (§14.4) и каждое пустое `final`-поле (§4.12.4, §8.3.1.2) должны иметь *определенно присвоенное* (*является определенно присвоенной*) значение при любом обращении к их значениям.

Обращение к их значениям состоит из простого имени переменной (или, в случае поля, простого имени поля, квалифицированного ключевым словом `this`), находящегося в любом месте выражения, за исключением левого операнда оператора присваивания = (§15.26.1).

Для каждого обращения к локальной переменной или к пустому `final`-полю `x`, `x` должно быть определенно присвоено до этого обращения, иначе генерируется ошибка времени компиляции.

Аналогично каждая пустая `final`-переменная должна быть присвоена не более одного раза; при выполнении присваивания она должна быть *определенно неприсвоенной* (*является определенно присвоенной*).

Такое присваивание происходит по определению тогда и только тогда, когда либо простое имя переменной, либо (в случае поля) простое имя, квалифицированное ключевым словом `this`, находится слева от оператора присваивания.

Для каждого присваивания пустой `final`-переменной эта переменная перед присваиванием должна быть определенно неприсвоенной, иначе генерируется ошибка времени компиляции.

Оставшаяся часть этой главы посвящена точному объяснению терминов “определенно присвоенная до” и “определенно неприсвоенная до”.

Идея, лежащая в основе определенного присваивания, заключается в том, что присваивание локальной переменной или пустому `final`-полю должно осуществляться на любом пути выполнения к доступу к этой переменной или полю. Аналогично идея, лежащая в основе определенной неприсвоенности, заключается в том, что на любом пути выполнения до при пустой `final`-переменной не должно быть другого присваивания.

Анализ принимает во внимание структуру инструкций и выражений; он, кроме того, специальным образом рассматривает операторы `!`, `&&`, `||` и `?:`, а также булевы константные выражения.

За исключением специального рассмотрения условных булевых операторов `&&`, `||` и `?:` и булевых константных выражений значения выражений в анализе потока во внимание не принимаются.

Пример 16-1. Определенное присваивание, рассматривающее структуру инструкций и выражений

Компилятор Java в приведенном коде распознает, что переменная `k` определенно присвоена перед обращением к ней (как к аргументу вызова метода).

```
{
    int k;
    if (v > 0 && (k = System.in.read()) >= 0)
        System.out.println(k);
}
```

Дело в том, что это обращение осуществляется только в том случае, когда значение выражения

```
v > 0 && (k = System.in.read()) >= 0
```

равно `true`, а это значение может быть `true`, только если выполняется присваивание локальной переменной `k` (говоря более строго, не выполняется, а вычисляется).

Аналогично компилятор Java распознает, что в коде

```
{
    int k;
    while (true) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);
}
```

переменная `k` определенно присвоена инструкцией `while`, поскольку выражение условия `true` никогда не может быть значением `false`, так что нормальное завершение инструкции `while` возможно только с помощью инструкции `break`, а `k` определенно присвоена перед инструкцией `break`.

С другой стороны, исходный текст

```
{
    int k;
    while (n < 4) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k); /* k не является "определенно
                              присвоенной" до этой инструкции */
}
```

должен быть отвергнут компилятором Java, поскольку в этом случае при рассмотрении правил определенного присваивания не гарантируется выполнение тела инструкции `while`.

Пример 16-2. Определенное присваивание не рассматривает значения выражений

Компилятор Java при компиляции приведенного исходного текста должен генерировать ошибку времени компиляции

```
{
    int k;
    int n = 5;
    if (n > 2)
        k = 3;
    System.out.println(k); /* k не является "определенно
                              присвоенной" до этой инструкции */
}
```

несмотря на то, что значение `n` известно во время компиляции, и в принципе во время компиляции может быть известно, что присваивание локальной переменной `k` всегда выполняется (или, строго говоря, вычисляется). Компилятор Java должен работать в соответствии с правилами, изложенными в данном разделе. Эти правила распознают только константные выражения; в нашем примере выражение `n > 2` не является константным выражением, определенным в §15.28.

В качестве еще одного примера компилятор Java принимает следующий исходный текст в плане определенного присваивания `k`.

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    else
        k = 4;
    System.out.println(k);
}
```

Дело в том, что указанные в этом разделе правила позволяют сказать, что `k` присваивается независимо от того, имеет ли `flag` значение `true` или `false`. Но эти правила не пропускают следующую вариацию кода.

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    if (!flag)
        k = 4;
    System.out.println(k); /* k не является "определенно
                              присвоенной" до этой инструкции */
}
```

Таким образом, компиляция этой программы должна привести к ошибке времени компиляции.

Пример 16-3. Определенное неприсваивание

Компилятор Java принимает следующий исходный текст в плане определенного присваивания `k`.

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    else {
        k = 4;
        System.out.println(k);
    }
}
```

Дело в том, что указанные в этом разделе правила позволяют сказать, что `k` присваивается не более одного раза (на самом деле — ровно один раз) независимо от того, имеет ли `flag` значение `true` или `false`. Но эти правила не пропускают следующую вариацию кода.

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    if (!flag) {
        k = 4;
        System.out.println(k); /* k не является "определенно
                                  неприсвоенной" до этой инструкции */
    }
}
```

Таким образом, компиляция этого исходного текста должна привести к генерации ошибки времени компиляции.

Для точного описания всех случаев определенного присваивания правила в этом разделе определяют несколько технических терминов:

- является ли переменная *определенно присвоенной до* инструкции или выражения;
- является ли переменная *определенно неприсвоенной до* инструкции или выражения;
- является ли переменная *определенно присвоенной после* инструкции или выражения;
- является ли переменная *определенно неприсвоенной после* инструкции или выражения.

Что касается булевых выражений, то последние два пункта подразделяются на четыре случая:

- является ли переменная *определенно присвоенной после* выражения *при значении true*;
- является ли переменная *определенно неприсвоенной после* выражения *при значении true*;

- является ли переменная *определенно присвоенной после* выражения *при значении false*;
- является ли переменная *определенно неприсвоенной после* выражения *при значении false*.

Здесь *при значении true* и *при значении false* подразумевают значение выражения.

> Например, локальной переменной `k` определенно присвоено значение после вычисления выражения
>
> ```
> a && ((k=m) > 5)
> ```
>
> если выражение равно `true`, но не когда выражение равно `false` (поскольку, если `a` равно `false`, присваивание переменной `k` не обязательно выполняется (строго говоря, вычисляется)).

Фраза "*V* определенно присвоена после *X*" (где *V* представляет собой локальную переменную, а *X* — инструкцию или выражение) означает "*V* определенно присвоена после *X*, если *X* завершается нормально". Если *X* завершается преждевременно, присваивание не должно выполняться, и описанные здесь правила должны принимать этот факт во внимание.

> Своеобразным следствием этого определения является то, что утверждение "*V* определенно присвоено после `break;`" всегда верно! Поскольку инструкция `break` никогда не завершается нормально, верно (и совершенно бессмысленно), что, если инструкция `break` завершилась нормально, *V* было присвоено значение.

Утверждение "*V* определенно неприсвоена после *X*" (где *V* — переменная, а *X* — инструкция или выражение) означает "*V* определенно неприсвоена после нормального завершения *X*".

> Есть еще более бессмысленное следствие этого определения — утверждение "*V* определенно неприсвоена после `break;`" всегда истинно! Поскольку инструкция `break` никогда не завершается нормально, верно (и совершенно бессмысленно), что, если инструкция `break` завершилась нормально, *V* не было присвоено значение. (Впрочем, столь же бессмысленно верно и то, что если инструкция `break` завершилась нормально, то Луна сделана из зеленого сыра...)

В целом для переменной *V* после выполнения инструкции или выражения имеется четыре возможности.

- *V* определенно присвоена и не является определенно неприсвоенной.

  (Анализ потока доказывает, что присваивание *V* имело место.)
- *V* определенно неприсвоена и не является определенно присвоенной.

  (Анализ потока доказывает, что присваивание *V* не имело места.)
- *V* не является определенно присвоенной и не является определенно неприсвоенной.

  (Правила не могут доказать, имело ли место присваивание переменной *V*.)
- *V* определенно присвоена и не определенно неприсвоена.

  (Инструкция или выражение не может завершиться нормально.)

Воспользуемся следующим сокращением в дальнейшем изложении: если правило содержит одно или несколько "[не]присвоена", то это означает два правила, в одном из

которых каждое “[не]присвоена” заменяется на “определенно присвоена”, а во втором каждое “[не]присвоена” заменяется на “определенно неприсвоена”.

Например:

- *V* [не]присвоена после пустой инструкции тогда и только тогда, когда она [не] присвоена перед пустой инструкцией

следует понимать как два правила:

- *V* определенно присвоена после пустой инструкции тогда и только тогда, когда она определенно присвоена до пустой инструкции;
- *V* определенно неприсвоена после пустой инструкции тогда и только тогда, когда она определенно неприсвоена до пустой инструкции.

Во всей оставшейся части этой главы мы будем (если явно не указано иное) обозначать через *V* локальную переменную или пустое `final`-поле, находящиеся в области видимости (§6.3). Аналогично мы будем использовать *a*, *b*, *c* и *e* для представления выражений, а *S* и *T* для представления инструкций. Мы будем использовать фразу “*a* представляет собой *V*” для указания того факта, что *a* представляет собой либо простое имя переменной *V*, либо простое имя *V*, квалифицированное ключевым словом `this` (игнорируя скобки). Утверждение “*a* не является *V*” означает отрицание утверждения “*a* представляет собой *V*”.

Анализ определенного неприсваивания в инструкциях цикла приводит к особой проблеме. Рассмотрим инструкцию `while(`*e*`)` *S*. Для того чтобы определить, является ли *V* определенно неприсвоенной в некотором подвыражении *e*, нам надо определить, является ли *V* определенно неприсвоенной до *e*. Можно доказать по аналогии с правилом для определенного присваивания (§16.2.10), что *V* определенно неприсвоена до *e* тогда и только тогда, когда она определенно неприсвоена до инструкции `while`. Однако такое правило для наших целей не подходит. Если *e* вычисляется равным `true`, будет выполнена инструкция *S*. Позже, если *V* присваивается инструкцией *S*, в следующих итерациях *V* уже будет присвоена при вычислении *e*. При предложенном выше правиле можно выполнить присваивание *V* несколько раз, а это именно то, чего мы стремились с помощью этих правил избежать.

Пересмотренное правило имеет вид “*V* определенно неприсвоена до *e* тогда и только тогда, когда она определенно неприсвоена до инструкции `while` и определенно неприсвоена после *S*”. Однако, когда мы формулируем правило для *S*, находим “*V* определенно неприсвоена до *S* тогда и только тогда, когда она определенно неприсвоена после *e* при значении `true`”. Это приводит к зацикливанию. По сути, *V* определенно неприсвоена *до* условия цикла *e*, только если она неприсвоена *после* всего цикла!

Мы разорвем этот порочный круг с помощью гипотетического анализа условия цикла и тела. Например, если мы считаем, что *V* определенно неприсвоена до *e* (независимо от того, действительно ли *V* определенно неприсвоена до *e*), а затем можем доказать, что *V* была определенно неприсвоена после *e*, то мы знаем, что *e* не присваивает *V*. Это можно сформулировать более формально.

В предположении, что *V* определенно неприсвоена до *e*, *V* определенно неприсвоена после *e*.

Вариации приведенного выше анализа используются для определения обоснованных правил определенного неприсваивания для всех инструкций циклов в языке программирования Java.

§16.1. Определенное присваивание и выражения

§16.1.1. Булевы константные выражения

- *V* [не]присвоена после любого константного выражения (§15.28), значение которого равно `true` при значении false.
- *V* [не]присвоена после любого константного выражения (§15.28), значение которого равно `false` при значении true.
- *V* [не]присвоена после любого константного выражения, значение которого равно `true` при значении true тогда и только тогда, когда *V* [не]присвоена до константного выражения.
- *V* [не]присвоена после любого константного выражения, значение которого равно `false` при значении false тогда и только тогда, когда *V* [не]присвоена до константного выражения.
- *V* [не]присвоена после булева константного выражения `e` тогда и только тогда, когда *V* [не]присвоена после `e` при значении true и *V* [не]присвоена после `e` при значении false.

Это эквивалентно тому, чтобы сказать, что *V* [не]присвоена после `e` тогда и только тогда, когда *V* [не]присвоена до `e`.

Поскольку константное выражение, значение которого равно `true`, никогда не имеет значения `false`, а константное выражение, значение которого равно `false`, никогда не имеет значения `true`, первые два правила бессмысленно выполняются. Они полезны в анализе выражений с операторами `&&` (§16.1.2), `||` (§16.1.3), `!` (§16.1.4) и `?:` (§16.1.5).

§16.1.2. Оператор условного И &&

- *V* [не]присвоена после *a* `&&` *b* (§15.23) при значении true тогда и только тогда, когда *V* [не]присвоена после *b* при значении true.
- *V* [не]присвоена после *a* `&&` *b* при значении false тогда и только тогда, когда *V* [не] присвоена после *a* при значении false и *V* [не]присвоена после *b* при значении false.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до *a* `&&` *b*.
- *V* [не]присвоена до *b* тогда и только тогда, когда *V* [не]присвоена после *a* при значении true.
- *V* [не]присвоена после *a* `&&` *b* тогда и только тогда, когда *V* [не]присвоена после *a* `&&` *b* при значении true и *V* [не]присвоена после *a* `&&` *b* при значении false.

§16.1.3. Оператор условного ИЛИ ||

- *V* [не]присвоена после *a* || *b* (§15.24) при значении true тогда и только тогда, когда *V* [не]присвоена после *a* при значении true и *V* [не]присвоена после *b* при значении true.
- *V* [не]присвоена после *a* || *b* при значении false тогда и только тогда, когда *V* [не]присвоена после *b* при значении false.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до *a* || *b*.
- *V* [не]присвоена до *b* тогда и только тогда, когда *V* [не]присвоена после *a* при значении false.
- *V* [не]присвоена после *a* || *b* тогда и только тогда, когда *V* [не]присвоена после *a* || *b* при значении true и *V* [не]присвоена после *a* || *b* при значении false.

§16.1.4. Оператор логического дополнения !

- *V* [не]присвоена после !*a* (§15.15.6) при значении true тогда и только тогда, когда *V* [не]присвоена после *a* при значении false.
- *V* [не]присвоена после !*a* при значении false тогда и только тогда, когда *V* [не]присвоена после *a* при значении true.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до !*a*.
- *V* [не]присвоена после !*a* тогда и только тогда, когда *V* [не]присвоена после !*a* при значении true и *V* [не]присвоена после !*a* при значении false.

> Это эквивалентно тому, что *V* [не]присвоена после !*a* тогда и только тогда, когда *V* [не]присвоена после *a*.

§16.1.5. Условный оператор ? :

Предположим, что *b* и *c* представляют собой булевы выражения.

- *V* [не]присвоена после *a*?*b*:*c* (§15.25) при значении true тогда и только тогда, когда *V* [не]присвоена после *b* при значении true и *V* [не]присвоена после *c* при значении true.
- *V* [не]присвоена после *a*?*b*:*c* при значении false тогда и только тогда, когда *V* [не] присвоена после *b* при значении false и *V* [не]присвоена после *c* при значении false.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до *a*?*b*:*c*.
- *V* [не]присвоена до *b* тогда и только тогда, когда *V* [не]присвоена после *a* при значении true.
- *V* [не]присвоена до *c* тогда и только тогда, когда *V* [не]присвоена после *a* при значении false.
- *V* [не]присвоена после *a*?*b*:*c* тогда и только тогда, когда *V* [не]присвоена после *a*?*b*:*c* при значении true и *V* [не]присвоена после *a*?*b*:*c* при значении false.

§16.1.6. Условный оператор ? :

Предположим, что *b* и *c* представляют собой выражения, не являющиеся булевыми.

- *V* [не]присвоена после *a*?*b*:*c* (§15.25) тогда и только тогда, когда *V* [не]присвоена после *b* и *V* [не]присвоена после *c*.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до *a*?*b*:*c*.
- *V* [не]присвоена до *b* тогда и только тогда, когда *V* [не]присвоена после *a* при значении true.
- *V* [не]присвоена до *c* тогда и только тогда, когда *V* [не]присвоена после *a* при значении false.

§16.1.7. Прочие выражения типа `boolean`

Предположим, что *e* представляет собой выражение типа `boolean` и не является логическим константным выражением, выражением логического дополнения !*a*, выражением условного И *a*&&*b*, выражением условного ИЛИ *a*||*b* или условным выражением *a*?*b*:*c*.

- *V* [не]присвоена после *e* при значении true тогда и только тогда, когда *V* [не]присвоена после *e*.
- *V* [не]присвоена после *e* при значении false тогда и только тогда, когда *V* [не]присвоена после *e*.

§16.1.8. Выражения присваивания

Рассмотрим выражение присваивания *a*=*b*, *a*+=*b*, *a*-=*b*, *a**=*b*, *a*/=*b*, *a*%=*b*, *a*<<=*b*, *a*>>=*b*, *a*>>>=*b*, *a*&=*b*, *a*|=*b* или *a*^=*b* (§15.26).

- *V* определенно присвоена после выражения присваивания тогда и только тогда, когда
  - ✦ либо *a* представляет собой *V*, либо
  - ✦ *V* определенно присвоена после *b*.
- *V* определенно неприсвоена после выражения присваивания тогда и только тогда, когда *a* не является *V* и *V* определенно неприсвоена после *b*.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до выражения присваивания.
- *V* [не]присвоена до *b* тогда и только тогда, когда *V* [не]присвоена после *a*.

> Заметим, что если *a* представляет собой *V* и *V* не является определенно присвоенной до составного присваивания, такого, как *a*&=*b*, то с необходимостью генерируется ошибка времени компиляции. Первое правило для определенного присваивания, приведенное выше, включает дизъюнктив "*a* представляет собой *V*" даже для составных выражений присваивания, а не только для простых присваиваний, так что *V* будет рассматриваться как определенно присвоенная в более поздних точках исходного текста. Включение дизъюнктива "*a* представляет собой *V*" не влияет на бинарное решение о том, является ли программа приемлемой или приведет к

генерации ошибки времени компиляции, но влияет на то, сколько различных точек исходного текста будут рассматриваться как ошибочные, так что на практике это может повысить качество сообщений об ошибках. Аналогичное замечание применимо и к включению конъюнктива "*a* не является *V*" в первом правиле для определенного неприсваивания, приведенном выше.

§16.1.9. Операторы ++ и --

- *V* определенно присвоена после `++`*a* (§15.15.1), `--`*a* (§15.15.2), *a*`++` (§15.14.2) и *a*`--` (§15.14.3) тогда и только тогда, когда либо *a* представляет собой *V*, либо *V* определенно присвоена после выражения операнда.
- *V* определенно присвоена после `++`*a*, `--`*a*, *a*`++` и *a*`--` тогда и только тогда, когда *a* не является *V* и *V* определенно неприсвоена после выражения операнда.
- *V* [не]присвоена до *a* тогда и только тогда, когда *V* [не]присвоена до `++`*a*, `--`*a*, *a*`++` и *a*`--`.

§16.1.10. Другие выражения

Если выражение не является ни булевым константным выражением, ни одним из выражений преинкремента `++`*a*, предекремента `--`*a*, постинкремента *a*`++`, постдекремента *a*`--`, выражения логического дополнения `!`*a*, выражением условного И *a*`&&`*b*, выражением условного ИЛИ *a*`||`*b*, условным выражением *a*`?`*b*`:`*c*, ни выражением присваивания, применяются следующие правила.

- Если выражение не имеет подвыражений, *V* [не]присвоена после выражения тогда и только тогда, когда *V* [не]присвоена до выражения.

  Этот случай применим к литералам, именам, ключевому слову `this` (квалифицированному и неквалифицированному), выражениям создания экземпляра неквалифицированного класса без аргументов, выражениям создания инициализированного массива, инициализаторы которых не содержат выражений, выражениям обращения к полю неквалифицированного суперкласса, вызовам именованных методов без аргументов и вызовам методов неквалифицированного суперкласса без аргументов.

- Если выражение имеет подвыражения, *V* [не]присвоена после выражения тогда и только тогда, когда *V* [не]присвоена после его крайнего справа непосредственного подвыражения.

  Имеется небольшое тонкое обоснование, лежащее в основе утверждения о том, что может быть известно, что переменная *V* определенно неприсвоена после вызова метода. Взятое само по себе, как есть и без квалификации, такое утверждение не всегда верно, поскольку вызываемый метод может выполнять присваивания. Однако следует помнить, что для целей языка программирования Java концепция определенного неприсваивания применима только к пустым `final`-переменным. Если *V* является пустой локальной `final`-переменной, присваивание *V* может выполнить только метод, которому принадлежит ее объявление. Если *V* является

пустым `final`-полем, присваивание *V* может выполнить только конструктор или инициализатор класса, содержащего объявление *V*; никакой метод не может выполнить присваивание *V*. Наконец отдельно обрабатываются (§16.9) явные вызовы конструкторов (§8.8.7.1); хотя синтаксически они похожи на инструкции выражений, содержащие вызовы методов, они не являются инструкциями выражений, и, следовательно, правила этого раздела к явным вызовам конструкторов неприменимы.

Для любого непосредственного подвыражения *y* выражения *x*, *V* [не]присвоена до *y* тогда и только тогда, когда выполняется одна из следующих ситуаций.

- *y* представляет собой крайнее слева непосредственное подвыражение *x* и *V* [не]присвоена до *x*.
- *y* представляет собой правый операнд бинарного оператора и *V* [не]присвоена после левого операнда.
- *x* представляет собой обращение к массиву, *y* является подвыражением в квадратных скобках и *V* [не]присвоена после подвыражения до квадратных скобок.
- *x* представляет собой первичное выражение вызова метода, *y* представляет собой первое выражение аргумента в выражении вызова метода и *V* [не]присвоена после первичного выражения, которое вычисляет целевой объект.
- *x* представляет собой выражение вызова метода или выражение создания экземпляра класса; *y* представляет собой выражение аргумента, но не первого; и *V* [не]присвоена после выражения аргумента слева от *y*.
- *x* представляет собой выражение создания экземпляра квалифицированного класса, *y* представляет собой выражение первого аргумента в выражении создания экземпляра класса и *V* [не]присвоена после первичного выражения, которое вычисляет квалифицирующий объект.
- *x* представляет собой выражение создания экземпляра массива; *y* представляет собой выражение размерности, но не первое; и *V* [не]присвоена после выражения размерности слева от *y*.
- *x* представляет собой выражение создания экземпляра массива, инициализированного инициализатором массива; *y* представляет собой инициализатор массива в *x*; и *V* [не] присвоена после выражения размерности слева от *y*.

§16.2. Определенное присваивание и инструкции

§16.2.1. Пустые инструкции

- *V* [не]присвоена после пустой инструкции (§14.6) тогда и только тогда, когда она [не] присвоена до пустой инструкции.

§16.2.2. Блоки

- Пустое `final`-поле *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до блока (§14.2), который представляет собой тело любого метода

в области видимости *V* и до объявления любого класса, объявленного в области видимости *V*.

- Локальная переменная *V* определенно неприсвоена (и, кроме того, не является определенно присвоенной) до блока, который представляет собой тело конструктора, метода, инициализатора экземпляра или статического экземпляра, который объявляет *V*.
- Пусть *C* представляет собой класс, объявленный в области видимости *V*. Тогда *V* определенно присвоена до блока, который представляет собой тело любого конструктора, метода, инициализатора экземпляра или статического экземпляра, объявленного в *C*, тогда и только тогда, когда *V* определенно присвоена до объявления *C*.

> Обратите внимание, что нет никаких правил, которые позволили бы нам заключить, что *V* определенно неприсвоена до блока, который представляет собой тело любого конструктора, метода, инициализатора экземпляра или статического инициализатора, объявленного в *C*. Неформально можно заключить, что *V* не является определенно неприсвоенной до блока, который представляет собой тело любого конструктора, метода, инициализатора экземпляра или статического инициализатора, объявленного в *C*, однако в явном указании такого правила необходимости нет.

- *V* [не]присвоена после пустого блока тогда и только тогда, когда *V* [не]присвоена до пустого блока.
- *V* [не]присвоена после непустого блока тогда и только тогда, когда *V* [не]присвоена после последней инструкции этого блока.
- *V* [не]присвоена до первой инструкции блока тогда и только тогда, когда *V* [не]присвоена до этого блока.
- *V* [не]присвоена до любой иной инструкции *S* блока тогда и только тогда, когда *V* [не] присвоена после инструкции, непосредственно предшествующей *S* в блоке.

Мы говорим, что *V* определенно неприсвоена везде в блоке *B* тогда и только тогда, когда:

- *V* определенно неприсвоена до *B*.
- *V* определенно присвоена после *e* в каждом выражении присваивания `V = e`, `V += e`, `V -= e`, `V *= e`, `V /= e`, `V %= e`, `V <<= e`, `V >>= e`, `V >>>= e`, `V &= e`, `V |= e` и `V ^= e`, встречающегося в *B*.
- *V* определенно присвоена до каждого выражения `++V`, `--V`, `V++` и `V--`, встречающегося в *B*.

> Эти условия противоречат интуиции и требуют определенного пояснения. Рассмотрим простое присваивание `V = e`. Если *V* определенно присвоена после *e*, то осуществляется одна из двух ситуаций.
>
> - Присваивание находится в невыполняющемся ("мертвом") коде, и *V* бессмысленно определенно присвоена. В этом случае присваивание фактически не будет иметь места, и можно считать, что *V* не присвоена выражением присваивания.
> - *V* уже была присвоена более ранним выражением до *e*. В этом случае текущее присваивание приведет к ошибке времени компиляции. Так что мы можем

заключить, что если условия отвечают программе, не вызывающей ошибок времени компиляции, то любые присваивания *V* в *B* в действительности не будут иметь места во время выполнения.

§16.2.3. Инструкции объявления локального класса

- *V* [не]присвоена после инструкции объявления локального класса (§14.3) тогда и только тогда, когда *V* [не]присвоена до инструкции объявления локального класса.

§16.2.4. Инструкции объявления локальных переменных

- *V* [не]присвоена после инструкции объявления локальной переменной (§14.4), которая не содержит инициализаторов переменных, тогда и только тогда, когда *V* [не]присвоена до инструкции объявления локальной переменной.
- *V* определенно присвоена после инструкции объявления локальной переменной, которая содержит как минимум один инициализатор переменной, тогда и только тогда, когда либо *V* определенно присвоена после последнего инициализатора переменной в инструкции объявления локальной переменной, либо последний инициализатор переменной в объявлении находится в декларатоpe, объявляющем *V*.
- *V* определенно неприсвоена после инструкции объявления локальной переменной, которая содержит как минимум один инициализатор, тогда и только тогда, когда *V* определенно неприсвоена после последнего инициализатора переменной в инструкции объявления локальной переменной, а последний инициализатор переменной в объявлении не находится в декларатоpe, который объявляет *V*.
- *V* [не]присвоена до первого инициализатора переменной в инструкции объявления локальной переменной тогда и только тогда, когда *V* [не]присвоена до инструкции объявления локальной переменной.
- *V* определенно присвоена до любого инициализатора переменной *e*, отличного от первого, в инструкции объявления локальной переменной тогда и только тогда, когда либо *V* определенно присвоена после инициализатора переменной слева от *e* или выражение инициализатора слева от *e* находится в декларатоpe, объявляющем *V*.
- *V* определенно неприсвоена до любого инициализатора переменной *e*, отличного от первого в инструкции объявления локальной переменной тогда и только тогда, когда *V* определенно неприсвоена после инициализатора переменной слева от *e* и выражение инициализатора слева от *e* не находится в декларатоpe, который объявляет *V*.

§16.2.5. Помеченные инструкции

- *V* [не]присвоена после помеченной инструкции `L: S` (где *L* — метка) (§14.7) тогда и только тогда, когда *V* [не]присвоена после *S* и *V* [не]присвоена до каждой инструкции `break` которая может осуществлять выход из помеченной инструкции `L: S`.
- *V* [не]присвоена до *S* тогда и только тогда, когда *V* [не]присвоена до `L: S`.

§16.2.6. Инструкции выражений

- *V* [не]присвоена после инструкции выражения `e;` (§14.8) тогда и только тогда, когда она [не]присвоена после `e`.
- *V* [не]присвоена до `e` тогда и только тогда, когда она [не]присвоена до `e;`.

§16.2.7. Инструкции `if`

К инструкции `if (e)` *S* (§14.9.1) применимы следующие правила.

- *V* [не]присвоена после `if (e)` *S* тогда и только тогда, когда *V* [не]присвоена после *S* и *V* [не]присвоена после `e` при значении false.
- *V* [не]присвоена до `e` тогда и только тогда, когда *V* [не]присвоена до `if (e)` *S*.
- *V* [не]присвоена до *S* тогда и только тогда, когда *V* [не]присвоена после `e` при значении true.

К инструкции `if (e)` *S* `else` *T* (§14.9.2) применимы следующие правила.

- *V* [не]присвоена после `if (e)` *S* `else` *T* тогда и только тогда, когда *V* [не]присвоена после *S* и *V* [не]присвоена после *T*.
- *V* [не]присвоена до `e` тогда и только тогда, когда *V* [не]присвоена до `if (e)` *S* `else` *T*.
- *V* [не]присвоена до *S* тогда и только тогда, когда *V* [не]присвоена после `e` при значении true.
- *V* [не]присвоена до *T* тогда и только тогда, когда *V* [не]присвоена после `e` при значении false.

§16.2.8. Инструкции `assert`

Приведенные далее правила применимы как к инструкции `assert` e_1, так и к инструкции `assert` e_1 `:` e_2 (§14.10).

- *V* [не]присвоена до e_1 тогда и только тогда, когда *V* [не]присвоена до инструкции `assert`.
- *V* определенно присвоена после инструкции `assert` тогда и только тогда, когда *V* определенно присвоена до инструкции `assert`.
- *V* определенно неприсвоена после инструкции `assert` тогда и только тогда, когда *V* определенно неприсвоена до инструкции `assert` и *V* определенно неприсвоена после e_1 при значении true.

  Следующее правило применимо к инструкции `assert` e_1 `:` e_2.

- *V* [не]присвоена до e_2 тогда и только тогда, когда *V* [не]присвоена после e_1 при значении false.

§16.2.9. Инструкции `switch`

- *V* [не]присвоена после инструкции `switch` (§14.11) тогда и только тогда, когда выполняются все перечисленные далее условия.
  - ✦ Либо в блоке `switch` имеется метка `default`, либо *V* [не]присвоена после выражения `switch`.
  - ✦ Либо нет меток в блоке `switch`, которые не начинают группу инструкции блока (т.е. отсутствуют метки непосредственно перед скобкой "`}`", завершающей блок `switch`), либо *V* [не]присвоена после выражения `switch`.
  - ✦ Либо блок `switch` не содержит блока инструкций блока, либо *V* [не]присвоена после последней инструкции блока из последней группы.
  - ✦ *V* [не]присвоена до каждой из инструкций `break`, которые могут обеспечить выход из инструкции `switch`.
- *V* [не]присвоена до выражения `switch` тогда и только тогда, когда *V* [не]присвоена до инструкции `switch`.

Если блок `switch` содержит как минимум одну группу инструкций блока (блок кода), то применимы также следующие правила.

- *V* [не]присвоена до первой инструкции блока первой группы инструкций блока в блоке `switch` тогда и только тогда, когда *V* [не]присвоена после выражения `switch`.
- *V* [не]присвоена до первой инструкции блока любой группы инструкций блока, отличной от первой, тогда и только тогда, когда *V* [не]присвоена после выражения `switch` и *V* [не]присвоена после предыдущей инструкции блока.

§16.2.10. Инструкция `while`

- *V* [не]присвоена после `while(`*e*`)` *S* (§14.12) тогда и только тогда, когда *V* [не]присвоена после *e* при значении false и *V* [не]присвоена до каждой из инструкций `break`, для которых эта инструкция `while` является целевой.
- *V* определенно присвоена до *e* тогда и только тогда, когда *V* определенно присвоена до инструкции `while`.
- *V* определенно неприсвоена до *e* тогда и только тогда, когда выполняются все приведенные ниже условия.
  - ✦ *V* определенно неприсвоена до инструкции `while`.
  - ✦ В предположении, что *V* определенно неприсвоена до *e*, *V* определенно неприсвоена после *S*.
  - ✦ В предположении, что *V* определенно неприсвоена до *e*, *V* определенно неприсвоена до каждой из инструкций `continue`, для которых эта инструкция `while` является целевой.
- *V* [не]присвоена до *S* тогда и только тогда, когда *V* [не]присвоена после *e* при значении true.

§16.2.11. Инструкция do

- *V* [не]присвоена после `do` *S* `while(e);` (§14.13) тогда и только тогда, когда *V* [не] присвоена после `e` при значении false и *V* [не]присвоена до каждой из инструкций `break`, для которых эта инструкция `do` является целевой.
- *V* определенно присвоена до *S* тогда и только тогда, когда *V* определенно присвоена до инструкции `do`.
- *V* определенно неприсвоена до *S* тогда и только тогда, когда выполняются все приведенные далее условия.
  - *V* определенно неприсвоена до инструкции `do`.
  - В предположении, что *V* определенно неприсвоена до *S*, *V* определенно неприсвоена после `e` при значении true.
- *V* [не]присвоена до `e` тогда и только тогда, когда *V* [не]присвоена после *S* и *V* [не]присвоена до каждой из инструкций `continue`, для которых эта инструкция `do` является целевой.

§16.2.12. Инструкции for

Приведенные здесь правила относятся к базовой инструкции `for` (§14.14.1). Поскольку расширенная инструкция `for` (§14.14.2) определяется трансляцией к базовой инструкции `for`, для нее не требуются никакие специальные правила.

- *V* [не]присвоена после инструкции `for` тогда и только тогда, когда выполняются оба следующие условия.
  - Либо выражение условия отсутствует, либо *V* [не]присвоена после выражения условия при значении false.
  - *V* [не]присвоена до каждой из инструкций `break`, для которых эта инструкция `for` является целевой.
- *V* [не]присвоена до части инициализации инструкции `for` тогда и только тогда, когда *V* [не]присвоена до инструкции `for`.
- *V* определенно присвоена до части условия инструкции `for` тогда и только тогда, когда *V* определенно присвоена после части инициализации инструкции `for`.
- *V* определенно неприсвоена до части условия инструкции `for` тогда и только тогда, когда выполняются все следующие условия.
  - *V* определенно неприсвоена после части инициализации инструкции `for`.
  - В предположении, что *V* определенно неприсвоена до части условия инструкции `for`, *V* определенно неприсвоена после содержащейся в `for` инструкции.
  - В предположении, что *V* определенно неприсвоена до содержащейся в `for` инструкции, *V* определенно неприсвоена до каждой из инструкций `continue`, для которых эта инструкция `for` является целевой.
- *V* [не]присвоена до содержащейся (в `for`) инструкции тогда и только тогда, когда выполняется любое из следующих условий.

  - ✦ Выражение условия имеется в наличии и *V* [не]присвоена после выражения условия при значении true.
  - ✦ Выражение условия отсутствует и *V* [не]присвоена после части инициализации инструкции `for`.
- *V* [не]присвоена до части инкремента инструкции `for` тогда и только тогда, когда *V* [не]присвоена после содержащейся инструкции и *V* [не]присвоена до каждой из инструкций `continue`, для которых эта инструкция `for` является целевой.

§16.2.12.1. Часть инициализации инструкции `for`

- Если часть инициализации инструкции `for` представляет собой инструкцию объявления локальной переменной, применимы правила из §16.2.4.
- В противном случае, если часть инициализации пуста, *V* [не]присвоена после части инициализации тогда и только тогда, когда *V* [не]присвоена до части инициализации.
- В противном случае применимы три правила.
  - ✦ *V* [не]присвоена после части инициализации тогда и только тогда, когда *V* [не]присвоена после последней инструкции выражения в части инициализации.
  - ✦ *V* [не]присвоена до первой инструкции выражения в части инициализации тогда и только тогда, когда *V* [не]присвоена до части инициализации.
  - ✦ *V* [не]присвоена до инструкции выражения *S*, отличной от первой, в части инициализации тогда и только тогда, когда *V* [не]присвоена после инструкции выражения, непосредственно предшествующей *S*.

§16.2.12.2. Часть инкремента инструкции `for`

- Если часть инкремента инструкции `for` пуста, то *V* [не]присвоена после части инкремента тогда и только тогда, когда *V* [не]присвоена до части инкремента.
- В противном случае применимы три правила.
  - ✦ *V* [не]присвоена после части инкремента тогда и только тогда, когда *V* [не]присвоена после последней инструкции выражения в части инкремента.
  - ✦ *V* [не]присвоена до первой инструкции выражения в части инкремента тогда и только тогда, когда *V* [не]присвоена до части инкремента.
  - ✦ *V* [не]присвоена до инструкции выражения *S*, отличной от первой, в части инкремента тогда и только тогда, когда *V* [не]присвоена после инструкции выражения, непосредственно предшествующей *S*.

§16.2.13. Инструкции `break`, `continue`, `return` и `throw`

- По соглашению мы говорим, что *V* [не]присвоена после любой инструкции `break`, `continue`, `return` или `throw` (§14.15, §14.16, §14.17, §14.18).

> Запись “переменная [не]присвоена после” инструкции или выражения в действительности означает “переменная [не]присвоена после инструкции или выражения, завершившегося нормально”. Поскольку инструкции `break`, `continue`, `return`

и `throw` никогда не завершаются нормально, они бессмысленно удовлетворяют этой записи.

- В инструкции `return` с выражением `e` или в инструкции `throw` с выражением `e` *V* [не]присвоена до `e` тогда и только тогда, когда *V* [не]присвоена до инструкции `return` или `throw`.

§16.2.14. Инструкции `synchronized`

- *V* [не]присвоена после `synchronized(e)` *S* (§14.19) тогда и только тогда, когда *V* [не]присвоена после *S*.
- *V* [не]присвоена до `e` тогда и только тогда, когда *V* [не]присвоена до инструкции `synchronized(e)` *S*.
- *V* [не]присвоена до *S* тогда и только тогда, когда *V* [не]присвоена после `e`.

§16.2.15. Инструкции `try`

Эти правила применимы к каждой инструкции `try` (§14.20) независимо от наличия у нее блока `finally`.

- *V* [не]присвоена до блока `try` тогда и только тогда, когда *V* [не]присвоена до инструкции `try`.
- *V* определенно присвоена до блока `catch` тогда и только тогда, когда *V* определенно присвоена до блока `try`.
- *V* определенно неприсвоена до блока `catch` тогда и только тогда, когда выполняются все следующие условия.
  - ✦ *V* определенно неприсвоена после блока `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `return`, принадлежащей блоку `try`.
  - ✦ *V* определенно неприсвоена после `e` каждой инструкции вида `throw e`, принадлежащей блоку `try`.
  - ✦ *V* определенно неприсвоена после каждой инструкции `assert`, находящейся в блоке `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `break`, принадлежащей блоку `try`, результирующая инструкция которой содержит (или представляет собой) инструкцию `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `continue`, принадлежащей блоку `try`, результирующая инструкция которой содержит инструкцию `try`.

  Если инструкция `try` не имеет блока `finally`, применимо также следующее правило.
- *V* [не]присвоена после инструкции `try` тогда и только тогда, когда *V* [не]присвоена после блока `try` и *V* [не]присвоена после каждого блока `catch` в инструкции `try`.

  Если инструкция `try` имеет блок `finally`, применимы также следующие правила.

- *V* определенно присвоена после инструкции `try` тогда и только тогда, когда выполняется как минимум одно из условий.
  - ✦ *V* определенно присвоена после блока `try` и *V* определенно присвоена после каждого блока `catch` в инструкции `try`.
  - ✦ *V* определенно присвоена после блока `finally`.
  - ✦ *V* определенно неприсвоена после инструкции `try` тогда и только тогда, когда *V* определенно неприсвоена после блока `finally`.
- *V* определенно присвоена до блока `finally` тогда и только тогда, когда *V* определенно присвоена до инструкции `try`.
- *V* определенно неприсвоена до блока `finally` тогда и только тогда, когда выполняются все приведенные далее условия.
  - ✦ *V* определенно неприсвоена после блока `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `return`, которая принадлежит блоку `try`.
  - ✦ *V* определенно неприсвоена после `e` в каждой инструкции вида `throw e`, которая принадлежит блоку `try`.
  - ✦ *V* определенно неприсвоена после каждой инструкции `assert`, которая находится в блоке `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `break`, которая принадлежит блоку `try` и целевая инструкция которой содержит (или представляет собой) инструкцию `try`.
  - ✦ *V* определенно неприсвоена до каждой инструкции `continue`, которая принадлежит блоку `try` и целевая инструкция которой содержит инструкцию `try`.
  - ✦ *V* определенно неприсвоена после каждого блока `catch` инструкции `try`.

§16.3. Определенное присваивание и параметры

- Формальный параметр *V* метода или конструктора (§8.4.1, §8.8.1) определенно присвоен (и, кроме того, не является определенно неприсвоенным) до тела метода или конструктора.
- Параметр исключения *V* конструкции `catch` (§14.20) определенно присвоен (и, кроме того, не является определенно неприсвоенным) до тела конструкции `catch`.

§16.4. Определенное присваивание и инициализаторы массива

- *V* [не]присвоена после пустого инициализатора массива (§10.6) тогда и только тогда, когда *V* [не]присвоена до пустого инициализатора массива.

- *V* [не]присвоена после непустого инициализатора массива тогда и только тогда, когда *V* [не]присвоена после инициализатора последней переменной в инициализаторе массива.
- *V* [не]присвоена до инициализатора первой переменной инициализатора массива тогда и только тогда, когда *V* [не]присвоена до инициализатора массива.
- *V* [не]присвоена до любого иного инициализатора `e` инициализатора массива тогда и только тогда, когда *V* [не]присвоена после инициализатора переменной слева от `e` в инициализаторе массива.

§16.5. Определенное присваивание и константы перечислений

Правила, определяющие, когда переменная определенно присвоена или определенно неприсвоена до константы перечисления (§8.9.1), приведены в §16.8.

> Дело в том, что константа перечисления по сути представляет собой поле, объявленное как `static final` (§8.3.1.1, §8.3.1.2), которое инициализировано выражением создания экземпляра класса (§15.9).

- *V* определенно присвоена до объявления тела класса константы перечисления без аргументов, который объявлен в области видимости *V*, тогда и только тогда, когда *V* определенно присвоена до константы перечисления.
- *V* определенно присвоена до объявления тела класса константы перечисления с аргументами, который объявлен в области видимости *V*, тогда и только тогда, когда *V* определенно присвоена после последнего выражения аргумента константы перечисления.

Статус определенного присваивания/неприсваивания любой конструкции в теле класса константы перечисления подчиняется обычным правилам для классов.

- *V* [не]присвоена до первого аргумента константы перечисления тогда и только тогда, когда она [не]присвоена до константы перечисления.
- *V* [не]присвоена до `y` (аргумента константы перечисления, но не первого) тогда и только тогда, когда *V* [не]присвоена после аргумента слева от `y`.

§16.6. Определенное присваивание и анонимные классы

- *V* определенно присвоена до объявления анонимного класса (§15.9.5), который объявлен в области видимости *V* тогда и только тогда, когда *V* определенно присвоена после выражения создания экземпляра класса, которое объявляет анонимный класс.

> Должно быть ясно, что если анонимный класс неявно определяется константой перечисления, применяются правила из §16.5.

§16.7. Определенное присваивание и типы-члены

Пусть *C* является классом и пусть *V* представляет собой пустое поле-член *C*, объявленное как `final`. Тогда

- *V* определенно присвоена (и, кроме того, не является определенно неприсвоенной) до объявления любого типа-члена (§8.5, §9.5) *C*.

Пусть *C* представляет собой класс, объявленный в области видимости *V*. Тогда

- *V* определенно присвоена до объявления типа-члена *C* тогда и только тогда, когда *V* определенно присвоена до объявления *C*.

§16.8. Определенное присваивание и статические инициализаторы

Пусть *C* представляет собой класс, объявленный в области видимости *V*. Тогда

- *V* определенно присвоена до константы перечисления (§8.9.1) или инициализатора статической переменной (§8.3.2) класса *C* тогда и только тогда, когда *V* определенно присвоена до объявления класса *C*.

> Обратите внимание на отсутствие правил, которые могли бы позволить нам заключить, что *V* определенно неприсвоена до инициализатора статической переменной или константы перечисления. Неформально можно сделать вывод о том, что *V* не является определенно неприсвоенной до любого инициализатора статической переменной класса *C*, но в явном формулировании такого правила необходимости нет.

Пусть *C* представляет собой класс, а *V* — пустое поле-член класса *C*, объявленное в *C* как `static final`. Тогда

- *V* определенно неприсвоено (и, кроме того, не является определенно присвоенным) до крайней слева константы перечисления, статического инициализатора (§8.7) или инициализатора статической переменной *C*;
- *V* [не]присвоено до константы перечисления, статического инициализатора или инициализатора статической переменной *C*, отличных от крайних слева, тогда и только тогда, когда *V* [не]присвоено после предшествующей константы перечисления, статического инициализатора или инициализатора статической переменной *C*.

Пусть *C* представляет собой класс, а *V* — пустое поле-член класса *C*, объявленное в суперклассе *C* как `static final`. Тогда

- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до каждой константы перечисления класса *C*;
- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до блока, который представляет собой тело статического инициализатора класса *C*;
- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до каждого инициализатора статической переменной класса *C*.

§16.9. Определенное присваивание, конструкторы и инициализаторы экземпляров

Пусть *C* представляет собой класс, объявленный в области видимости *V*. Тогда

- *V* определенно присвоена до инициализатора переменной экземпляра (§8.3.2) класса *C* тогда и только тогда, когда *V* определенно присвоена до объявления класса *C*.

> Обратите внимание на отсутствие правил, которые могли бы позволить нам заключить, что *V* определенно неприсвоена до инициализатора переменной экземпляра. Неформально можно сделать вывод о том, что *V* не является определенно неприсвоенной до любого инициализатора переменной экземпляра класса *C*, но в явном формулировании такого правила необходимости нет.

Пусть *C* представляет собой класс, а *V* — пустое поле-член класса *C*, объявленное в *C* как `final` и не `static`. Тогда

- *V* определенно неприсвоено (и, кроме того, не является определенно присвоенным) до крайнего слева инициализатора экземпляра (§8.6) или инициализатора переменной экземпляра класса *C*;
- *V* [не]присвоено до инициализатора экземпляра или инициализатора переменной экземпляра класса *C*, отличного от крайнего слева, тогда и только тогда, когда *V* [не] присвоено после предшествующего инициализатора экземпляра или инициализатора переменной экземпляра класса *C*.

В конструкторе (§8.8.7) класса *C* выполняются следующие правила.

- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) после вызова другого конструктора (§8.8.7.1).
- *V* определенно неприсвоено (и, кроме того, не является определенно присвоенным) до явного или неявного вызова конструктора суперкласса (§8.8.7.1).
- Если *C* не имеет инициализаторов экземпляра или инициализаторов переменных экземпляра, то *V* не является определенно присвоенным (и, кроме того, является определенно неприсвоенным) после явного или неявного вызова конструктора суперкласса.
- Если *C* имеет как минимум один инициализатор экземпляра или инициализатор переменной экземпляра, то *V* [не]присвоено после явного или неявного вызова конструктора суперкласса тогда и только тогда, когда *V* [не]присвоено после крайнего справа инициализатора экземпляра или инициализатора переменной экземпляра класса *C*.

Пусть *C* представляет собой класс, а *V* — пустое поле-член класса *C*, объявленное в *C* как `final`. Тогда

- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до блока, который представляет собой тело конструктора или инициализатор экземпляра класса *C*;
- *V* определенно присвоено (и, кроме того, не является определенно неприсвоенным) до каждого инициализатора переменной экземпляра класса *C*.

ГЛАВА 17

Потоки и блокировки

Хотя в предыдущих главах в основном рассматривались вопросы поведения кода при выполнении одной инструкции или одного выражения за раз, т.е. одним *потоком* (*thread*), виртуальная машина Java может поддерживать одновременное выполнение многих потоков. Эти потоки независимо выполняют код, который работает со значениями и объектами, находящимися в совместно используемой основной памяти. Потоки могут поддерживаться с помощью нескольких аппаратных процессоров или путем распределения времени между потоками на одном или нескольких аппаратных процессорах.

Потоки представлены классом `Thread`. Единственный способ создания потока пользователем — создание объекта этого класса; каждый поток ассоциирован с таким объектом. Поток начинает работу, когда вызывается метод `start()` соответствующего объекта `Thread`.

Поведение потоков, в особенности в случае некорректной синхронизации, может быть непонятным и нелогичным. В этой главе описывается семантика многопоточных программ; она включает в себя правила, значения для которых могут быть получены путем чтения из общей памяти, которая обновляется несколькими потоками. Поскольку спецификация похожа на *модели памяти* для различных аппаратных архитектур, эти семантики известны как *модель памяти языка программирования Java*. В ситуациях, когда эти понятия не могут быть спутаны, мы будем говорить просто о "модели памяти".

Эти семантики не поясняют, как именно должны выполняться многопоточные программы. Скорее они описывают поведение, которое могут демонстрировать многопоточные программы. Любая стратегия исполнения, которая приводит к разрешенному поведению, является приемлемой стратегией выполнения.

§17.1. Синхронизация

Язык программирования Java предоставляет несколько механизмов для обмена информацией между потоками. Наиболее фундаментальным среди них является *синхронизация*, реализованная с помощью *мониторов*. Каждый объект в Java связан с монитором, который может *блокировать* или *разблокировать поток*. Одновременно хранить блокировку монитора может только один поток. Любые другие потоки, пытающиеся заблокировать монитор, блокируются до того момента, когда они смогут получить блокировку этого монитора. Поток *t* может блокировать некоторый монитор несколько раз; каждое разблокирование отменяет одну операцию блокировки.

Инструкция `synchronized` (§14.19) вычисляет ссылку на объект; затем она пытается выполнить блокировку монитора объекта; выполнение потока приостанавливается до тех пор, пока блокировка не будет успешно выполнена. После блокировки выполняется тело инструкции `synchronized`. По завершении выполнения тела, как нормальном, так и преждевременном, над тем же монитором автоматически выполняется разблокирование.

Метод, объявленный как `synchronized` (§8.4.3.6), автоматически выполняет блокировку при вызове; его тело не выполняется до тех пор, пока блокирование не будет успешно завершено. Если метод является методом экземпляра, он блокирует монитор, связанный с экземпляром, для которого он вызывается (т.е. объектом, который представляет собой `this` в процессе вызова тела метода). Если метод объявлен как `static`, он блокирует монитор, связанный с объектом `Class`, который представляет класс, в котором объявлен этот метод. Когда выполнение тела метода завершается (нормально или преждевременно), автоматически выполняется разблокирование этого монитора.

Язык программирования Java не предупреждает о взаимных блокировках и не требует обнаружения взаимоблокировок. Программа, в которой потоки хранят (непосредственно или косвенно) блокировки нескольких объектов, должны использовать обычные методы предотвращения взаимоблокировок, при необходимости создавая высокоуровневые блокирующие примитивы, которые исключают взаимоблокировки.

Альтернативные способы синхронизации обеспечивают другие механизмы, такие как чтение и запись переменных, объявленных как `volatile`, и использование классов из пакета `java.util.concurrent`.

§17.2. Множества ожидания и уведомление

Каждый объект, помимо связанного с ним монитора, имеет связанное с ним *множество ожидания* (множество процессов, ожидающих доступ к объекту, — *wait set*). Множество ожидания представляет собой множество потоков.

При создании объекта его множество ожидания пустое. Элементарные действия, которые добавляют поток в множество ожидания и удаляют его оттуда, атомарны. Работа с множествами ожидания осуществляется исключительно с помощью методов `Object.wait`, `Object.notify` и `Object.notifyAll`.

На работу с множествами ожидания может влиять и состояние прерывания потока, а также методы класса `Thread` для работы с прерыванием. Кроме того, методы класса `Thread` для приостановки и объединения с другими потоками имеют свойства, которые являются производными от ожиданий и уведомлений.

§17.2.1. Ожидание

Ожидания (wait actions) осуществляются с помощью вызова метода `wait()` или его вариантов с указанием времени `wait(long millisecs)` и `wait(long millisecs, int nanosecs)`.

> Вызов `wait(long millisecs)` с нулевым параметром или `wait(long millisecs, int nanosecs)` с двумя нулевыми параметрами эквивалентен вызову `wait()`.

Поток *возвращается нормально* из ожидания, если возврат происходит без генерации исключения `InterruptedException`.

Пусть поток *t* представляет собой поток, выполняющий метод `wait` объекта *m*, и пусть *n* — количество блокировок потоком *t* объекта *m*, не соответствующих разблокированиям. Тогда осуществляется одно действие из приведенного списка.

- Если *n* равно нулю (например, поток *t* еще не блокировал целевой объект *m*), генерируется исключение `IllegalMonitorStateException`.
- Если это ожидание с указанием времени и аргумент `nanosecs` не находится в диапазоне `0-999999` или аргумент `millisecs` имеет отрицательное значение, генерируется исключение `IllegalArgumentException`.
- Если поток *t* прерван, то генерируется исключение `InterruptedException`, и статус прерывания *t* устанавливается равным `false`.
- В противном случае выполняется следующая последовательность действий.

  1. Поток *t* добавляется к множеству ожидания объекта *m* и выполняет *n* разблокировок объекта *m*.
  2. Поток *t* не выполняет никакие дальнейшие инструкции до тех пор, пока он не будет удален из множества ожидания объекта *m*. Поток может быть удален из множества ожидания из-за любого из следующих действий и будет продолжен через некоторое время после этого.

     - Действие `notify`, выполненное над *m*, в котором *t* выбран для удаления из множества ожидания.
     - Действие `notifyAll` , выполненное над *m*.
     - Действие `interrupt` , выполненное над *t*.
     - Если это ожидание с указанием времени, внутреннее действие удаляет *t* из множества ожидания *m*, что происходит после как минимум `millisecs` миллисекунд плюс `nanosecs` наносекунд после начала этого ожидания.
     - Внутреннее действие согласно реализации. Реализации разрешено, хотя и не рекомендуется, выполнять “ложное пробуждение”, т.е. удалять потоки из множества ожидания и тем самым позволять возобновление без явной команды поступать таким образом.

> Обратите внимание, что это положение требует от практики программирования на Java использования `wait` только в циклах, которые завершаются только по выполнении некоторого логического условия, которое ожидает поток.

Каждый поток должен определить порядок событий, которые могут привести его к удалению из набора ожидания. Этот порядок может не согласовываться с другими упорядочениями, но поток должен вести себя так, как если бы эти события произошли в этом порядке.

Например, если поток *t* находится в множестве ожидания *m*, а затем происходит как прерывание *t*, так и уведомление *m*, то должен быть определенный порядок этих событий. Если считать, что первым произошло прерывание, то *t* в конечном итоге вернется из `wait` путем генерации исключения `InterruptedException`, и уведомление должен получить некоторый другой поток в множестве ожидания *m* (если таковой существует в момент уведомления). Если же считать, что первым выполняется уведомление, то *t* в конечном итоге вернется из `wait` нормально с незавершенным прерыванием.

3. Поток *t* выполняет *n* блокировок над *m*.
4. Если поток *t* был удален из множества ожидания *m* на шаге 2 из-за прерывания, то состояние прерывания *t* устанавливается равным `false` и метод `wait` генерирует исключение `InterruptedException`.

§17.2.2. Уведомление

Уведомление осуществляется в результате вызова методов `notify` и `notifyAll`.

Пусть поток *t* представляет собой поток, выполняющий любой из методов объекта *m*, и пусть *n* — количество блокировок потоком *t* над *m*, которые не соответствуют разблокированиям. Тогда происходит одно из приведенных ниже действий.

- Если *n* равно нулю, генерируется исключение `IllegalMonitorStateException`.
- Это случай, когда поток *t* еще не блокировал целевой объект *m*.
- Если *n* больше нуля и это действие `notify`, то если множество ожидания *m* не пустое, то из этого множества выбирается и удаляется поток *u*, который является членом текущего множества ожидания *m*.
- Никакие гарантии о том, какой именно поток из множества ожидания выбирается, не предоставляются. Это удаление из множества ожидания обеспечивает возобновление *u* из ожидания. Заметим, однако, что блокирование *u* после возобновления не может быть успешным, пока не пройдет некоторое время после того, как *t* полностью разблокирует монитор *m*.
- Если *n* больше нуля и это действие `notifyAll`, то из множества ожидания *m* удаляются, и тем самым возобновляются, все потоки.
- Заметим, однако, что блокировать монитор, требуемый для возобновления ожидания одновременно будет только один из потоков.

§17.2.3. Прерывания

Прерывание осуществляется путем вызова `Thread.interrupt`, а также методов, которые его вызывают, таких как `ThreadGroup.interrupt`.

Пусть *t* представляет собой поток, вызывающий *u*`.interrupt`, для некоторого потока *u*, где *t* и *u* могут совпадать. Это действие приводит к тому, что состояние прерывания *u* устанавливается равным `true`.

Кроме того, если существует некоторый объект *m*, множество ожидания которого содержит *u*, то *u* удаляется из множества ожидания *m*. Это позволяет *u* восстановиться в ожидании, и в этом случае это ожидание будет генерировать, после повторной блокировки монитора *m*, исключение `InterruptedException`.

Вызов `Thread.isInterrupted` позволяет определить состояние прерывания потока. `static`-метод `Thread.interrupted` может быть вызван потоком для просмотра и сброса собственного статуса прерывания.

§17.2.4. Взаимодействие ожиданий, уведомлений и прерываний

Вышеуказанные спецификации позволяют нам определить несколько свойств, относящихся ко взаимодействию ожиданий, уведомлений и прерываний.

Если поток во время ожидания как уведомлен, так и прерван, возможны следующие варианты.

- Поток нормально возвращается из `wait`, при этом по-прежнему имея ожидающее прерывание (другими словами, вызов `Thread.interrupted` должен вернуть `true`).
- Поток возвращается из `wait` путем генерации исключения `InterruptedException`.

Поток не может сбросить состояние прерывания и при этом нормально вернуться из вызова `wait`.

Аналогично уведомления не могут быть потеряны из-за прерываний. Предположим, что множество *s* потоков находится в множестве ожидания объекта *m* и другой поток выполняет `notify` над объектом *m*. Тогда возможны следующие варианты.

- Как минимум один поток в *s* должен вернуться из `wait` нормально или
- все потоки в *s* должны выйти из `wait` путем генерации исключения `InterruptedException`.

Обратите внимание, что если поток одновременно прерван и разбужен вызовом `notify` и такой поток возвращается из `wait` путем генерации исключения `InterruptedException`, то должен быть уведомлен некоторый другой поток из множества ожидания.

§17.3. `sleep` и `yield`

`Thread.sleep` заставляет текущий выполняющийся поток “заснуть” (временно приостановить выполнение) на указанное в вызове время с учетом точности системных таймеров и планировщиков. Поток не теряет владение ни одним монитором, а возобновление выполнения будет зависеть от планирования и доступности процессоров, на которых выполняется поток.

Важно заметить, что ни `Thread.sleep`, ни `Thread.yield` не обладает семантикой синхронизации. В частности, компилятор не обязан сбрасывать кеши записи из регистров в разделяемую память перед вызовом `Thread.sleep` или `Thread.yield`, так же как

и не обязан перезагружать значения, кешированные в регистрах после вызовов `Thread.sleep` или `Thread.yield`.

> Например, в следующем (некорректном) фрагменте кода предположим, что `this.done` представляет собой поле типа `boolean`, не являющееся `volatile`.
>
> ```
> while (!this.done)
>     Thread.sleep(1000);
> ```
>
> Компилятор имеет право считать поле `this.done` всего лишь один раз и использовать кешированное значение при каждом выполнении цикла. Это означает, что цикл никогда не завершится, даже если другой поток изменит значение `this.done`.

§17.4. Модель памяти

Модель памяти описывает для данной программы и ее выполнения, является ли это выполнение корректным. Модель памяти языка программирования Java работает путем изучения каждого чтения в процессе выполнения и проверки того факта, что наблюдаемые этими чтениями записи корректны в соответствии с определенными правилами.

Модель памяти описывает возможное поведение программы. Реализация языка может генерировать любой код, лишь бы его выполнение приводило к результатам, предсказываемым моделью памяти.

> Тем самым реализатору языка обеспечивается значительная свобода выполнения множества преобразований кода, включая изменение порядка действий и удаление ненужных синхронизаций.

Пример 17.4-1. Некорректно синхронизированная программа может демонстрировать удивительное поведение

> Семантика языка программирования Java позволяет компиляторам и микропроцессорам выполнять оптимизации, которые могут взаимодействовать с некорректно синхронизированным кодом способом, который приводит к парадоксально выглядящему поведению. Вот некоторые примеры того, как некорректно синхронизированные программы могут демонстрировать удивительное поведение.
>
> Рассмотрим, например, программу, выполнение которой отслеживается в табл. 17.1. Эта программа использует локальные переменные `r1` и `r2` и разделяемые переменные `A` и `B`. Изначально `A == B == 0`.

Таблица 17.1. Удивительный результат, вызванный переупорядочением инструкций: исходный код

Поток 1	Поток 2
1: `r2 = A;`	3: `r1 = B;`
2: `B = 1;`	4: `A = 2;`

Может показаться, что результат `r2 == 2` и `r1 == 1` невозможен. Интуитивно кажется, что первой при выполнении должна быть либо инструкция 1, либо инструкция 3. Если первой идет инструкция 1, она не должна видеть результат записи в инструкции 4. Если первой идет инструкция 3, она не должна видеть результат записи в инструкции 2.

Если некоторое выполнение демонстрирует такое поведение, то это говорит нам о том, что инструкция 4 выполняется до инструкции 1, которая выполняется до инструкции 2, которая выполняется до инструкции 3, которая выполняется до инструкции 4. Абсурд!

Однако компилятор имеет право переупорядочивать инструкции в любом потоке, если это не влияет на выполнение такого потока изолированно от других. Если инструкцию 1 поменять местами с инструкцией 2, как показано в табл. 17.2, то легко увидеть, как получить результат `r2 == 2` и `r1 == 1`.

Таблица 17.2. Удивительный результат, вызванный переупорядочением инструкций: разрешенное преобразование компилятором

Поток 1	Поток 2
`B = 1;`	`r1 = B;`
`r2 = A;`	`A = 2;`

Для некоторых программистов такое поведение выглядит некорректным. Однако следует заметить, что данный код некорректно синхронизирован:

- в одном потоке имеется запись,
- в другом потоке имеется чтение той же переменной,
- запись и чтение не упорядочены с помощью синхронизации.

Эта ситуация является примером *гонки данных* (*data race*, §17.4.5). Если код содержит гонку данных, зачастую возможны нелогичные результаты.

Изменение порядка в табл. 17.2 может выполняться несколькими механизмами. Перестановка кода может выполняться как Just-In-Time-компилятором в реализации виртуальной машины Java, так и процессором. Кроме того, иерархия памяти архитектуры, на которой выполняется реализация виртуальной машины Java, может выглядеть так, как будто код переупорядочен. В этой главе все, что может переупорядочивать код, мы будем называть *компилятором*.

Еще один пример удивительных результатов показан в табл. 17.3. Изначально `p == q` и `p.x == 0`. Эта программа также некорректно синхронизирована; она выполняет записи в разделенную память без обеспечения упорядочения этих записей.

Таблица 17.3. Удивительный результат, вызванный подстановкой

Поток 1	Поток 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r1.x;`	

Одна распространенная оптимизация компилятора включает использование значения, считанного в `r2`, в качестве значения `r5`: оба они считывают `r1.x` без промежуточной записи. Эта ситуация показана в табл. 17.4.

Таблица 17.4. Удивительный результат, вызванный подстановкой

Поток 1	Поток 2
`r1 = p;`	`r6 = p;`
`r2 = r1.x;`	`r6.x = 3;`
`r3 = q;`	
`r4 = r3.x;`	
`r5 = r2;`	

Теперь рассмотрим случай, когда присваивание `r6.x` в потоке 2 происходит между первым чтением `r1.x` и чтением `r3.x` в потоке 1. Если компилятор решит повторно использовать значение `r2` для `r5`, затем `r2` и `r5` будут иметь значение `0`, а `r4` будет иметь значение `3`. С точки зрения программиста, значение, хранящееся в `p.x`, изменилось с `0` до `3`, а затем изменяется обратно.

Модель памяти определяет, какие значения могут быть считаны в каждой точке программы. Действия каждого потока в отдельности должны вести себя как подчиняющиеся семантике этого потока, с тем исключением, что значения, которые видит каждое чтение, определяются моделью памяти. Когда мы говорим это, то мы говорим, что программа подчиняется *внутрипоточной семантике*. Эта семантика представляет собой семантику однопоточной программы и позволяет полностью предсказывать поведение потока на основе значений, считываемых потоком. Для того чтобы определить, корректны ли действия потока *t* при выполнении, мы просто рассматриваем реализацию потока *t*, как если бы она выполнялась в однопоточном контексте, как определено в остальной части данной спецификации.

Всякий раз, когда операция потока *t* генерирует межпоточное действие, оно должно соответствовать межпоточному действию *a* потока *t*, которое выполняется следующим в порядке программы. Если *a* представляет собой чтение, то дальнейшее выполнение *t* использует значение, полученное *a* и определенное моделью памяти.

В этом разделе представлена спецификация модели памяти языка программирования Java, за исключением вопросов, имеющих отношение к `final`-полям, описанным в §17.5.

Представленная здесь модель памяти не основана на объектно-ориентированной природе языка программирования Java. Для краткости и простоты в наших примерах мы часто демонстрируем фрагменты кода без определений класса или метода или явного разыменования. Большинство примеров состоят из двух или более потоков, содержащих инструкции с доступом к локальным переменным, разделяемым глобальным переменным или полям экземпляра объекта. Мы обычно используем для обозначения переменных, являющихся локальными для метода или потока, такие имена переменных, как *`r1`* или *`r2`*. Такие переменные недоступны другим потокам.

§17.4.1. Разделяемые переменные

Память, которая может быть совместно использована потоками, называется *разделяемой памятью* (shared memory), или *кучей* (heap memory).

Все поля экземпляров, `static`-поля и элементы массивов хранятся в куче. В этой главе мы используем термин *переменная* как для полей, так и для элементов массивов.

Локальные переменные (§14.4), формальные параметры методов (§8.4.1) и параметры обработчиков исключений (§14.20) никогда между потоками не разделяются и модель памяти влияния на них не оказывает.

Два обращения (чтения и записи) одной и той же переменной являются *конфликтующими*, если по крайней мере одно из них является записью.

§17.4.2. Действия

Межпоточное действие (inter-thread action) представляет собой действие, выполняемое одним потоком, которое может быть обнаружено другим потоком или на которое этот другой поток может влиять непосредственно. Имеется ряд видов межпоточных действий, которые могут выполняться программой.

- *Чтение* (обычное, не синхронизированное). Считывает переменную.
- *Запись* (обычное, не синхронизированное). Записывает переменную.
- *Действия синхронизации*, к которым относятся следующие.
  - ✦ *Синхронизированное чтение*. Считывает переменную как `volatile`.
  - ✦ *Синхронизированная запись*. Записывает переменную как `volatile`.
  - ✦ *Блокировка*. Блокирует монитор.
  - ✦ *Разблокировка*. Разблокирует монитор.
  - ✦ Искусственное первое и последнее действия потока.
  - ✦ Действия, которые запускают поток или обнаруживают завершение потока (§17.4.4).
- *Внешние действия*. Внешнее действие представляет собой действие, которое может наблюдаться вне исполнения и дает результат, основанный на внешнем окружении исполнения.
- *Действия расходимости потока* (§17.4.9). Действие расходимости потока выполняется только потоком, находящимся в бесконечном цикле, в котором не выполняются действия с памятью, действия синхронизации или внешние действия. Если поток выполняет действие расходимости потока, за ним следует бесконечное число действий расходимости других потоков.

> Действия расходимости потока введены для моделирования того, как поток может привести к остановке других потоков и прекратить работу программы.

Данная спецификация рассматривает только межпоточные действия. Внутрипоточные действия рассматривать не требуется (например, сложение двух локальных переменных и сохранение результата в третьей локальной переменной). Как упоминалось ранее,

все потоки должны подчиняться корректной внутрипоточной семантике программ Java. Обычно о внутрипоточных действиях мы говорим кратко как о просто *действиях*.

Действие a описывается кортежем $\langle t, k, v, u \rangle$, состоящим из следующих элементов.

- t — поток, выполняющий действие.
- k — вид действия.
- v — переменная (или монитор), участвующая в действии.

  В случае блокировки v представляет собой блокируемый монитор; в случае разблокирования v представляет собой разблокируемый монитор.

  Если действие представляет собой (синхронизированное или не синхронизированное) чтение, v представляет собой считываемую переменную.

  Если действие представляет собой (синхронизированную или не синхронизированную) запись, v представляет собой записываемую переменную.
- u — произвольный идентификатор, однозначно идентифицирующий действие.

Кортеж внешнего действия содержит дополнительный компонент, в котором находится результат внешнего действия, воспринятый потоком, выполнившим это действие. Это может быть информация об успешности действия, а также любые значения, считанные действием.

Параметры внешнего действия (например, какие байты должны быть записаны и в какой сокет) не являются частью кортежа внешнего действия. Эти параметры настраиваются другими действиями в потоке и могут быть определены путем изучения внутрипоточной семантики. В модели памяти они явно не обсуждаются.

В незавершающих выполнениях не все внешние действия являются наблюдаемыми. Незавершающие выполнения и наблюдаемые действия обсуждаются в §17.4.9.

§17.4.3. Программы и программный порядок

Среди всех межпоточных действий, выполняемых каждым потоком t, *программный порядок* t представляет собой общий порядок, отражающий порядок, в котором эти действия будут выполняться согласно внутрипоточной семантике t.

Множество действий является *последовательно согласованным*, если все действия осуществляются в общем порядке (порядке выполнения), который согласован с программным порядком, и, кроме того, каждое чтение r переменной v получает значение, записанное записью w в переменную v, такое, что

- w находится до r в порядке выполнения и
- нет другой записи w', такой, что w выполняется до w', а w' располагается в порядке выполнения до r.

Последовательная согласованность является очень сильной гарантией видимости и упорядочения при выполнении программы. При последовательно согласованном выполнении над всеми индивидуальными действиями (такими, как чтение и запись) устанавливается общий порядок, согласованный с порядком программы, и каждое отдельное действие атомарно и немедленно видно каждому потоку.

Если в программе отсутствует гонка данных, то все выполнения программы выглядят последовательно согласованными.

Последовательная согласованность и/или отсутствие гонок данных, тем не менее, допускают ошибки, возникающие из-за групп операций, которые должны восприниматься как атомарные, но таковыми не являются.

> Если бы мы должны были использовать последовательную согласованность в качестве нашей модели памяти, многие из оптимизаций компилятора и процессора, которые мы обсуждали, стали бы некорректными. Например, в табл. 17.3, как только произошла запись `3` в `p.x`, все последующие чтения из этого места в памяти должны были бы получать это значение.

§17.4.4. Порядок синхронизации

Каждое выполнение имеет *порядок синхронизации*. Порядок синхронизации представляет собой общий порядок всех действий синхронизации при выполнении программы. Для каждого потока *t* порядок синхронизации действий синхронизации (§17.4.2) в *t* согласован с программным порядком (§17.4.3) потока *t*.

Действия синхронизации вводят отношение *синхронизирован с* над действиями, определенное следующим образом.

- Разблокирование монитора *m синхронизировано со* всеми последующими блокировками *m* (где "последующий" определено в соответствии с порядком синхронизации).
- Запись в `volatile`-переменную *v* (§8.3.1.4) *синхронизировано со* всеми последующими чтениями *v* любым потоком (где "последующий" определено в соответствии с порядком синхронизации).
- Действие, запускающее поток, *синхронизировано с* первым действием запускаемого потока.
- Запись значения по умолчанию (нуль, `false` или `null`) в каждую переменную *синхронизирована с* первым действием в каждом потоке.

> Хотя запись значения по умолчанию в переменную до выделения памяти для содержащего ее объекта может показаться немного странной, концептуально каждый объект создается в начале программы со своими инициализирующими значениями по умолчанию.

- Последнее действие в потоке *T1 синхронизировано с* действием в другом потоке *T2*, который обнаруживает завершение *T1*.

> *T2* может выполнить это с помощью вызова *T1*`.isAlive()` или *T1*`.join()`.

- Если поток *T1* прерывает поток *T2*, прерывание потоком *T1 синхронизировано с* любой точкой, где любой другой поток (включая *T2*) определяет, что поток *T2* был прерван (путем генерации исключения `InterruptedException` или вызова `Thread.interrupted` или `Thread.isInterrupted`).

Исходная вершина ребра *синхронизирован с* называется *освобождением* (release), а ее целевая вершина — *захватом* (acquire).

§17.4.5. Упорядочение *произошло до*

Два действия могут быть упорядочены с помощью отношения *произошло до*. Если одно действие *произошло до* другого, то первое видимо вторым и при упорядочении находится перед ним.

Если у нас имеется два действия, *x* и *y*, мы записываем *hb(x,y)*, чтобы указать, что *x* *произошло до* (happens-before) *y*.

- Если *x* и *y* представляет собой действия одного и того же потока и *x* находится до *y* в программном порядке, то *hb(x,y)*.
- Имеется ребро *произошло до* от конца конструктора объекта до начала финализатора (§12.6) этого объекта.
- Если действие *x синхронизировано со* следующим за ним действием *y*, то мы также имеем *hb(x,y)*.
- Если *hb(x,y)* и *hb(y,z)*, то *hb(x,z)*.

Методы `wait` класса `Object` (§17.2.1) должны блокировать и разблокировать действия, связанные с ними; их отношения *произошло до* определяются связанными с ними действиями.

Следует заметить, что наличие отношения *произошло до* между двумя действиями не обязательно подразумевает, что они имеют место в данном порядке в реализации. Если переупорядочение дает результаты, согласующиеся с корректным выполнением, такое переупорядочение не является некорректным.

> Например, запись значения по умолчанию в каждое поле объекта, созданного потоком, не обязательно должна происходить до начала потока, если отсутствует чтение, способное обнаружить этот факт.

Говоря более конкретно, если два действия связаны отношением *происходят до*, они не обязаны выглядеть происходящими в данном порядке для кода, с которым у них нет отношения *происходит до*. Записи в одном потоке, который находится в состоянии гонки данных с чтением в другом потоке, могут, например, выглядеть как не связанные с порядком этих чтений.

Отношение *происходит до* определяет, когда имеет место гонка данных.

Множество ребер синхронизации *S* является *достаточным*, если оно представляет собой минимальное множество, такое, что транзитивное замыкание *S* и программный порядок определяют все ребра *происходит до* в выполнении программы. Такое множество является единственным.

Из приведенных выше определений вытекает следующее.

- Разблокировка монитора *происходит до* каждой последующей блокировки монитора.
- Запись `volatile`-поля (§8.3.1.4) *происходит до* каждого последующего чтения этого поля.
- Вызов `start()` потока *происходит до* любого действия в запущенном потоке.
- Все действия в потоке *происходят до* того, как любой другой поток успешно вернется из вызова `join()` этого потока.

- Инициализация по умолчанию любого объекта *происходит до* любых других действий (отличных от записи значений по умолчанию) программы.

Если программа содержит два конфликтующих доступа (§17.4.1), не упорядоченных с помощью отношения *происходит до*, мы говорим, что она содержит *гонку данных* (data race).

На семантику операций, отличных от межпоточных действий, таких как чтения длин массивов (§10.7), выполнения проверенных приведений (§5.5, §15.16) и вызовов виртуальных методов (§15.12), гонки данных непосредственно не влияют.

> Следовательно, гонка данных не может привести к некорректному поведению, такому как возврат неверной длины массива.

Программа *корректно синхронизирована* тогда и только тогда, когда все последовательно согласованные выполнения свободны от гонок данных.

Если программа корректно синхронизирована, то все выполнения программы будут последовательно согласованными (§17.4.3).

> Это чрезвычайно сильная гарантия для программистов. Программистам не нужно рассуждать о переупорядочениях, чтобы определить, что их код содержит гонку данных. Также им не нужно рассуждать о переупорядочении при определении, корректно ли синхронизирован их код. После определения того факта, что код корректно синхронизирован, программисту не нужно беспокоиться о влиянии переупорядочения на его код.
>
> Чтобы избежать наблюдения нелогичного поведения программы при переупорядочении, программу следует корректно синхронизировать. Применение корректной синхронизации не гарантирует, что все поведение программы в целом будет корректным. Однако ее использование позволяет программисту более просто разобраться в возможном поведении программы; поведение корректно синхронизированной программы гораздо меньше зависит от возможных переупорядочений. При отсутствии корректного упорядочения возможно очень странное, противоречивое и нелогичное поведение программы.

Мы говорим, что чтение r переменной v может наблюдать запись w переменной v, если при частичном упорядочении выполнения с использованием отношения *происходит до*

- r не находится до w (т.е. не выполняется $hb(r,w)$) и
- нет никаких промежуточных записей w' переменной v (т.е. нет записи w' в переменную v, такой, что $hb(w,w')$ и $hb(w',r)$).

Говоря неформально, чтение r может видеть результат записи w, если не имеется упорядочения *происходит до*, предотвращающего это чтение.

Множество действий A *согласовано с отношением "происходит до"*, если для всех чтений r в A, где $W(r)$ представляет собой запись, видимую чтением r, не выполняется $hb(r,W(r))$ и не существует записи w в A, такой, что $w.v = r.v$, $hb(W(r),w)$ и $hb(w,r)$.

В *согласованном с отношением "происходит до"* множестве действий каждое чтение видит запись, которую ему позволяет видеть упорядочение *происходит до*.

Пример 17.4.5-1. Согласованность *происходит до*

В фрагменте в табл. 17.5 изначально `A == B == 0`. При выполнении можно обнаружить `r2 == 0` и `r1 == 0`, и это поведение оказывается *согласованным с отношением "происходит до"*, поскольку имеется порядок выполнения, который позволяет каждому чтению видеть результат соответствующей записи.

Таблица 17.5. Поведение, разрешенное согласованностью "происходит до", но не последовательной согласованностью

Поток 1	Поток 2
`B = 1;`	`A = 2;`
`r2 = A;`	`r1 = B;`

Поскольку синхронизация здесь отсутствует, каждое чтение может видеть либо запись исходного значения, либо запись, выполненную в другом потоке. Порядок выполнения, который демонстрирует данное поведение, следующий.

```
1: B = 1;
3: A = 2;
2: r2 = A; // Видит исходную запись 0
4: r1 = B; // Видит исходную запись 0
```

Другой порядок выполнения, *согласованный с отношением "происходит до"*, имеет вид

```
1: r2 = A; // Видит запись A = 2
3: r1 = B; // Видит запись B = 1
2: B = 1;
4: A = 2;
```

При этом выполнении чтения видят записи, которые происходят позже в порядке выполнения. Это может выглядеть невозможным, но это допускается согласованностью с отношением *происходит до*. Разрешение чтениям видеть более поздние записи может иногда приводить к неприемлемому поведению.

§17.4.6. Выполнения

Выполнение E описывается кортежем $\langle P, A, po, so, W, V, sw, hb \rangle$, состоящим из перечисленных далее элементов.

- P — программа.
- A — множество действий.
- po — программный порядок, который для каждого потока t представляет собой общий порядок всех действий, выполняемых t в A.
- so — порядок синхронизации, который представляет собой общий порядок всех действий синхронизации в A.
- W — функция видимой записи, которая для каждого чтения r из A дает $W(r)$, запись, видимую чтением r в E.

- *V* — функция записанного значения, которая для каждой записи *w* в *A* дает *V(w)*, значение, записанное записью *w* в *E*.
- *sw* — *синхронизирован с*, частичный порядок действий синхронизации.
- *hb* — *произошел до*, частичный порядок действий.

Обратите внимание, что элементы *синхронизация с* и *произошел до* единственным образом определяются другими компонентами выполнения и правилами корректно сформированных выполнений (§17.4.7).

Выполнение является *согласованным с отношением происходит до*, если его множество действий является *согласованным с отношением происходит до* (§17.4.5).

§17.4.7. Корректно сформированные выполнения

Мы рассматриваем только корректно сформированные выполнения. Выполнение является корректно сформированным, если выполнены следующие условия.

1. Каждое чтение видит запись в ту же переменную в выполнении.

   Все чтения и записи volatile-переменных являются синхронизированными (volatile) действиями. Для всех чтений *r* в *A* мы имеем *W(r)* в *A* и *W(r).v* = *r.v*. Переменная *r.v* является volatile-переменной тогда и только тогда, когда *r* является синхронизированным чтением, а переменная *w.v* является volatile-переменной тогда и только тогда, когда *w* является синхронизированной записью.

2. Упорядочение *происходит до* является частичным.

   Порядок *происходит до* задается транзитивным замыканием ребер *синхронизирован с* и программного порядка. Он должен быть корректным частичным порядком: рефлексивным, транзитивным и антисимметричным.

3. Выполнение подчиняется внутрипоточной согласованности.

   Для каждого потока *t* действия, выполняемые *t* в *A*, являются теми же, которые генерировались бы изолированным потоком в программном порядке, когда каждая запись *w*, записывающая значение *V(w)*, приводит к тому, что каждое чтение *r* видит значение *V(W(r))*. Значения, видимые каждым чтением, определяются моделью памяти. Данный программный порядок должен отражать программный порядок, в котором выполнялись бы действия в соответствии с внутрипоточной семантикой *P*.

4. Выполнение является *согласованным с отношением происходит до* (§17.4.6).
5. Выполнение подчиняется согласованности с порядком синхронизации.

   Для всех синхронизированных чтений *r* в *A* не выполняется *so(r,W(r))*, а также не существует записи *w* в *A*, такой, что *w.v* = *r.v*, *so(W(r),w)* и *so(w,r)*.

§17.4.8. Выполнения и требования причинности

Мы используем запись $f|_d$ для обозначения функции с областью определения *d*. Для всех $x \in d$ выполняется $f|_d(x) = f(x)$, а для всех $x \notin d$ значение $f|_d(x)$ не определено.

> Мы используем запись $p|_d$ для представления ограничения на частичное упорядочение p элементами из d. Для всех $x, y \in d$ $p(x, y)$ тогда и только тогда, когда $p|_d(x, y)$. Если или x, или y не принадлежит d, то утверждение $p|_d(x, y)$ неверно.

Правильность корректно сформированного выполнения $E = \langle P, A, po, so, W, V, sw, hb \rangle$ проверяется путем *фиксирующих* действий из A. Если все действия в A могут быть фиксированы, то выполнение удовлетворяет требованиям причинности модели памяти языка программирования Java.

Начиная с пустого множества C_0 мы выполняем последовательность шагов, в которых предпринимаем действия из множества действий A и добавляем их в множество фиксированных действий C_i для получения нового множества фиксированных действий C_{i+1}. Чтобы продемонстрировать разумность этого подхода, для каждого C_i нам надо продемонстрировать выполнение E, содержащее C_i, соответствующее определенным условиям.

Формально выполнение E удовлетворяет *требованиям причинности модели памяти языка программирования Java* тогда и только тогда, когда существует следующее.

- Множества действий $C_0, C_1, \ldots$, такие, что
  - C_0 является пустым множеством;
  - C_i является истинным подмножеством C_{i+1};
  - $A = \bigcup(C_0, C_1, \ldots)$.

  Если множество A конечное, то последовательность $C_0, C_1, \ldots$ конечна и заканчивается множеством $C_n = A$.

  Если множество A бесконечное, то последовательность $C_0, C_1, \ldots$ может быть бесконечной, и в этом случае объединение всех элементов этой бесконечной последовательности равно A.
- Корректно сформированные выполнения $E_1, \ldots$, где $E_i = \langle P, A_i, po_i, so_i, W_i, V_i, sw_i, hb_i \rangle$.

Для данных множеств действий $C_0, \ldots$ и выполнений $E_1, \ldots$ каждое действие в C_i должно разделять один и тот же относительный порядок "происходит до" и порядок синхронизации как в E_i, так и в E. Вот как это формулируется формально.

1. C_i является подмножеством A_i.
2. $hb_i|_{C_i} = hb|_{C_i}$.
3. $so_i|_{C_i} = so|_{C_i}$.

Значения, записанные записями в C_i, должны быть одинаковыми в E_i и E. Одни и те же записи в E_i и E должны видеть только чтения в C_{i-1}. Формально это выглядит следующим образом.

4. $V_i|_{C_i} = V|_{C_i}$.
5. $W_{i-1}|_{C_i} = W|_{C_i}$.

Все чтения в E_i, не входящие в C_{i-1}, должны видеть записи, которые произошли до них. Каждое чтение r в $C_i - C_{i-1}$ должно видеть записи в C_{i-1} как в E_i, так и в E, но может видеть запись в E_i, отличную от записи в E. Формально это можно записать следующим образом.

6. Для любого чтения r в $A_i - C_{i-1}$ мы имеем $hb_i(W_i(r), r)$.

7. Для любого чтения r в $(C_i - C_{i-1})$ мы имеем $W_i(r)$ в C_{i-1} и $W(r)$ в C_{i-1}.

Для данного множества синхронизированных ребер в E_i, если имеется пара "освобожден–захвачен", которая происходит до (§17.4.5) фиксируемого вами действия, эта пара должна присутствовать во всех E_j, где $j \geq i$. Формально это можно записать следующим образом.

8. Пусть ssw_i представляет собой ребра sw_i, которые также находятся в транзитивном сокращении hb_i, но не в *po*. Назовем ssw_i *достаточными ребрами "синхронизирован с"* для E_i. Если $ssw_i(x, y)$ и $hb_i(y, z)$, и z принадлежит C_i, то $sw_j(x, y)$ для всех $j \geq i$.

   Если действие y фиксировано, все внешние действия, которые произошли до y, также фиксированы.

9. Если y находится в C_i, x представляет собой внешнее действие и $hb_i(x, y)$, то x находится в C_i.

Пример 17.4.8-1. Согласованности "происходит до" недостаточно

Согласованность "происходит до" представляет собой необходимое, но не достаточное множество ограничений. Простое обеспечение согласованности "происходит до" допускает неприемлемое поведение, которое нарушает предъявляемые к программе требования. Например, согласованность "происходит до" допускает значения, выглядящие "взятыми с потолка". Это можно видеть при подробном изучении табл. 17.6.

Таблица 17.6. Согласованности "происходит до" недостаточно

Поток 1	Поток 2
`r1 = x;`	`r2 = y;`
`if (r1 != 0) y = 1;`	`if (r2 != 0) x = 1;`

Код, приведенный в табл. 17.6, корректно синхронизирован. Это может показаться удивительным, так как он не выполняет никаких действий по синхронизации. Вспомните, однако, что программа корректно синхронизирована, если при последовательно согласованном выполнении в ней нет гонки данных. Если приведенный код выполняется последовательно согласованно, то каждое действие будет происходить в программном порядке и не будет выполнена ни одна из операций записи. Поскольку записи не происходят, гонки данных быть не может, так что программа корректно синхронизирована.

Так как программа правильно синхронизирована, единственное ее допустимое поведение — последовательно согласованное. Однако имеется выполнение программы, которое хотя и является согласованным в смысле "происходит до", не является последовательно согласованным.

```
r1 = x; // Видит запись x = 1
y = 1;
r2 = y; // Видит запись y = 1
x = 1;
```

Этот результат согласованности “происходит до”: не имеется отношения “происходит до”, которое бы не позволяло получить такой результат. Однако это очевидно неприемлемо: не существует последовательно согласованного выполнения, которое привело бы к такому поведению. Тот факт, что мы позволяем чтению видеть запись, которая происходит позже в порядке выполнения, иногда может привести к неприемлемому поведению.

Хотя разрешение чтениям видеть записи, происходящие позже в порядке выполнения, иногда является нежелательным, иногда оно просто необходимо. Как мы видели выше, в табл. 17.5, требуется, чтобы некоторые чтения видели записи, происходящие позже в порядке выполнения. Поскольку в каждом потоке первым идет чтение, первым действием в порядке выполнения должно быть чтение. Если это чтение не может видеть запись, которая происходит позднее, то оно не может видеть никакое значение, отличное от начального значения переменной, которую оно читает. Это явно не отражает все поведения.

Мы говорим о ситуации, когда чтение может видеть будущие записи, как о *причинности*, из-за проблем, которые возникают в случаях, подобных приведенному в табл. 17.6. В этом случае чтения приводят к осуществлению записей, а записи приводит к осуществлению чтений. Не существует “первого действия” в этой цепочке. Поэтому наша модель памяти нуждается в согласованном способе определения того, какие именно чтения могут видеть ранние записи.

Примеры, такие как приведенный в табл. 17.6, демонстрируют, что спецификация должна быть осторожна при утверждении, что чтение может видеть записи, выполняющиеся позже (с учетом того, что если чтение видит запись, которая выполняется позже, то это представляет тот факт, что запись на самом деле выполняется ранее).

Модель памяти принимает в качестве входных данных такое выполнение и программу и определяет, является ли это выполнение программы корректным. Она делает это путем постепенного наращивания множества “фиксированных” действий, которые отражают то, какие действия были выполнены программой. Обычно следующее фиксируемое действие будет отражать следующее действие, которое может быть выполнено при последовательно согласованном выполнении. Однако, чтобы отразить чтения, которые должны видеть более поздние записи, мы разрешаем некоторым действиям быть фиксированными ранее, чем другим действиям, которые “происходят до” них.

Очевидно, некоторые действия могут быть фиксированы ранее, а некоторые — нет. Если, к примеру, одна из записей в табл. 17.6 была фиксирована до чтения этой переменной, чтение может видеть эту запись, и может получиться результат “с потолка”. Неформально мы разрешаем действию быть фиксированным ранее, если знаем, что действие может происходить без предположения о гонке данных. В табл. 17.6 мы не можем выполнить раннюю запись, потому что записи не могут произойти, если только чтения не увидят результат гонки данных (должны сработать условия в операторах `if`).

§17.4.9. Наблюдаемое поведение и незавершающее выполнение

Поведение программ, которые всегда завершаются в пределах некоторого ограниченного периода времени, может быть (неформально) понято просто в терминах допустимых выполнений. Для программ, которые могут не завершаться в пределах некоторого ограниченного периода времени, возникают более тонкие вопросы.

Наблюдаемое поведение программы определяется конечными множествами внешних действий, которые программа может выполнить. Программа, которая, например, просто вечно печатает `"Hello"`, описывается множеством поведений, которые для любого неотрицательного целого числа *i* включают поведение, заключающееся в выводе `"Hello"` *i* раз.

Завершение программы не моделируется явно как поведение, но программа может быть легко расширена для генерации дополнительного внешнего действия *execution Termination*, которое происходит, когда все потоки завершены.

Мы также определяем специальное действие *зависание* (hang). Если поведение описывается с помощью множества внешних действий, включая действия *зависания*, оно описывает поведение, когда после наблюдения внешних действий программа может работать неограниченное количество времени без выполнения каких-либо дополнительных внешних действий или завершения работы. Программы могут “зависнуть”, если все потоки блокируются или если программа может выполнять неограниченное количество действий без выполнения каких-либо внешних действий.

Поток может быть заблокирован в различных ситуациях, например когда он пытается захватить блокировку или выполнить внешнее действие (например, чтение), которое зависит от внешних данных.

Выполнение может привести к тому, что поток блокируется на неопределенный срок, и выполнение не завершается. В таких случаях действия, порожденные заблокированным потоком, должны состоять из всех действий, сгенерированных этим потоком, включая действие, вызвавшее блокировку потока, но без действий, которые были бы сгенерированы потоком после этого действия.

Говоря о наблюдаемых поведениях, мы должны говорить о множествах наблюдаемых действий.

Если *O* является множеством наблюдаемых действий для выполнения *E*, то множество *O* должно быть подмножеством *A* действий *E* и содержать только конечное число действий, даже если *A* содержит их бесконечное количество. Кроме того, если действие *y* находится в *O* и либо $hb(x, y)$, либо $so(x, y)$, то *x* находится в *O*.

Обратите внимание, что множество наблюдаемых действий не ограничено внешними действиями. Вместо этого наблюдаемыми внешними действиями считаются только внешние действия, которые находятся в множестве наблюдаемых действий.

Поведение *B* является допустимым поведением программы *P* тогда и только тогда, когда *B* является конечным множеством внешних действий и выполняется одно из следующих условий.

- Существует выполнение *E* программы *P*, множество *O* наблюдаемых действий *E*, и *B* представляет собой множество внешних действий в *O*. (Если некоторые потоки в *E* за-

вершаются в заблокированном состоянии и O содержит все действия в E, то B может также содержать действие *зависания*.)

- Существует множество действий O, такое, что B состоит из действия *зависания* плюс все внешние действия в O; для всех $k \geq |O|$ существует выполнение E программы P с действиями A; и существует множество действий O', такое, что справедливо следующее.
  - ✦ И O, и O' являются подмножествами A, удовлетворяющими требованиям для множеств наблюдаемых действий.
  - ✦ $O \subseteq O' \subseteq A$.
  - ✦ $|O'| \geq k$.
  - ✦ $O' - O$ не содержит внешних действий.

> Обратите внимание, что поведение B не описывает порядок, в котором наблюдаются внешние действия в B, но такие ограничения могут налагать другие (внутренние) ограничения на то, как внешние действия создаются и выполняются.

§17.5. Семантика поля, объявленного как `final`

Поля, объявленные как `final`, инициализируются однократно и при нормальных условиях никогда не изменяют своего значения. Подробная семантика `final`-полей несколько отличается от семантики нормальных полей. В частности, компиляторы обладают большой свободой перемещения чтений `final`-полей через барьеры синхронизации и вызовы произвольных или неизвестных методов. Соответственно, компиляторам разрешается хранить значения `final`-полей кешированными в регистрах и не загружать их из памяти в ситуациях, когда поле, не являющееся `final`, было бы перезагружено.

`final`-поля также позволяют программистам реализовывать поточно-безопасные (thread-safe) неизменяемые объекты без синхронизации. Такой объект виден как неизменный всем потокам, даже если для передачи ссылок на неизменяемые объекты между строками используется гонка данных. Это может обеспечить гарантии безопасности против неправильного применения неизменяемого класса некорректным или злонамеренным кодом. Для обеспечения гарантии неизменности `final`-поля должны использоваться корректным образом.

Объект считается *полностью инициализированным*, когда завершается его конструктор. Поток, который может видеть ссылку на объект только после полной его инициализации, гарантированно видит корректно инициализированные значения `final`-полей этого объекта.

Модель использования для `final`-поля проста: задать `final`-поля объекта в его конструкторе и не записывать ссылку на создаваемый объект туда, где другой поток может ее увидеть до завершения работы конструктора объекта. Если следовать этому правилу, то когда объект будет виден другим потоком, этот поток всегда будет видеть корректно построенную версию `final`-полей этого объекта. Он будет также видеть версии любого объекта или массива, на которые ссылаются эти `final`-поля, как минимум актуальными в той же мере, что и `final`-поля.

Пример 17.5-1. `final`-поля в модели памяти Java

Приведенная ниже программа иллюстрирует, как `final`-поля соотносятся с обычными полями.

```
class FinalFieldExample {
    final int x;
    int y;

    static FinalFieldExample f;

    public FinalFieldExample() {
        x = 3;
        y = 4;
    }

    static void writer() {
        f = new FinalFieldExample();
    }

    static void reader() {
        if (f != null) {
            int i = f.x; // Гарантированно видит 3
            int j = f.y; // Может видеть 0
        }
    }
}
```

Класс `FinalFieldExample` имеет поле `final int x` и поле `int y`, не являющееся `final`-полем. Один поток может выполнять метод `writer`, а второй — метод `reader`.

Поскольку метод `writer` записывает `f` *после* завершения конструктора объекта, метод `reader` гарантированно видит корректно инициализированное значение `f.x`: он считывает значение `3`. Однако `f.y` не является `final`; таким образом, не гарантируется, что метод `reader` увидит его значение `4`.

Пример 17.5-2. Поля `final` и безопасность

`final`-поля созданы для того, чтобы обеспечить необходимые гарантии безопасности. Рассмотрим следующую программу. Один поток (который мы будем называть "поток 1") выполняет код

```
Global.s = "/tmp/usr".substring(4);
```

в то время как другой поток ("поток 2") выполняет код

```
String myS = Global.s;
if (myS.equals("/tmp")) System.out.println(myS);
```

Объекты `String` разработаны таким образом, что должны быть неизменяемыми, а строковые операции не прибегают к синхронизации. В то время как реализация `String` не имеет гонок данных, такие гонки данных, включающие использование объектов типа `String`, могут быть в другом коде, а модель памяти обеспечивает

слабые гарантии для программ с гонками данных. В частности, если бы поля класса `String` не были объявлены как `final`, то было бы возможно (хотя и маловероятно), что поток 2 мог бы изначально видеть значение по умолчанию смещения строкового объекта, равное `0`, что позволило бы ему определить объект как эквивалентный строке `"/tmp"`. Более поздняя операция с объектом `String` могла бы увидеть корректное значение смещения — `4`, так что объект `String` воспринимался бы как `"/usr"`. Многие возможности языка программирования Java, связанные с безопасностью, зависят от того, что объекты `String` воспринимаются как истинно неизменяемые, даже если вредоносный код использует гонки данных для передачи ссылок на `String` между потоками.

§17.5.1. Семантика `final`-полей

Пусть *o* представляет собой объект, а *c* — конструктор *o*, в котором записывается `final`-поле *f*. Действие *заморозки* (freeze) над `final`-полем *f* объекта *o* имеет место по завершении *c* нормально или преждевременно.

Заметим, что если конструктор вызывает другой конструктор и вызванный конструктор устанавливает значение `final`-поля, то заморозка `final`-поля осуществляется в конце этого вызванного конструктора.

Для каждого выполнения на поведение чтений влияют два дополнительных частичных упорядочения, цепочка разыменований *dereferences()* и цепочка памяти *mc()*, которые рассматриваются как часть выполнения (и, таким образом, фиксированы для любого конкретного выполнения). Эти частичные упорядочения должны удовлетворять следующим ограничениям (которые не обязаны иметь единственное решение).

- Цепочка разыменований. Если действие *a* представляет собой чтение или запись поля или элемента объекта *o* потоком *t*, который не инициализировал *o*, то должно существовать некоторое чтение *r* потоком *t*, которое видит адрес *o*, такой, что *r присутствует в цепочке разыменований dereferences(r,a)*.
- Цепочка памяти. Имеется несколько ограничений на упорядочение цепочки памяти.
  - ✦ Если *r* представляет собой чтение, которое видит запись *w*, то должно быть *mc(w,r)*.
  - ✦ Если *r* и *a* представляют собой действия, такие, что *dereferences(r,a)*, то должно быть *mc(r,a)*.
  - ✦ Если *w* представляет собой запись адреса объекта *o* потоком *t*, который не инициализирует *o*, то должно существовать некоторое чтение *r* потоком *t*, которое видит адрес *o*, такое, что *mc(r,w)*.

Для данной записи *w*, заморозки *f*, действия *a* (которое не является чтением `final`-поля), чтения r_1 `final`-поля, замороженного *f*, и чтения r_2, такого, что *hb(w,f)*, *hb(f,a)*, $mc(a,r_1)$ и $dereferences(r_1,r_2)$, когда нам надо определить, какие значения могут быть видимы чтением r_2, мы рассматриваем $hb(w,r_2)$. (Это упорядочение *происходит до* транзитивно не замыкается с другим упорядочением *происходит до*.)

Заметим, что упорядочение *dereferences* рефлексивное и что r_1 может совпадать с r_2.

Для чтений `final`-полей единственными записями, которые происходят до чтения `final`-поля, являются те, которые порождены семантикой `final`-поля.

§17.5.2. Чтение `final`-полей в процессе конструирования

Чтение `final`-поля объекта в потоке, который конструирует этот объект, упорядочено по отношению к инициализации этого поля в конструкторе с помощью обычных правил *происходит до*. Если чтение осуществляется после того, как поле установлено в конструкторе, оно видит присвоенное значение `final`-поля, в противном случае оно видит значение по умолчанию.

§17.5.3. Последующая модификация `final`-полей

В некоторых случаях, таких, как десериализация, системе требуется изменить `final`-поля объекта после построения. `final`-поля могут быть изменены с помощью рефлексии и других средств, зависящих от реализации. Единственным шаблоном, где это имеет разумную семантику, является шаблон, в котором выполняется построение объекта с последующим обновлением `final`-полей. Объект не должен быть видимым другим потокам, а `final`-поля — быть считанными до тех пор, пока не будут выполнены все обновления этих полей. Заморозка `final`-поля осуществляется как в конце конструктора, в котором устанавливается значение `final`-поля, так и непосредственно после каждого изменения `final`-поля посредством рефлексии или иного специального механизма.

Но даже в этом случае есть целый ряд осложнений. Если `final`-поле инициализируется константным выражением (§15.28) в объявлении поля, изменения в `final`-поле могут не быть наблюдаемыми, так как использование данного поля заменяется во время компиляции значением константного выражения.

Еще одна проблема заключается в том, что спецификация допускает агрессивную оптимизацию `final`-поля. В рамках потока допустимо переупорядочение чтений `final`-поля и тех его изменений, которые не происходят в конструкторе.

Пример 17.5.3-1. Агрессивная оптимизация `final`-полей

```
class A {
    final int x;
    A() {
        x = 1;
    }
    int f() {
        return d(this,this);
    }
    int d(A a1, A a2) {
        int i = a1.x;
        g(a1);
        int j = a2.x;
        return j - i;
    }
    static void g(A a) {
        // Использует рефлексию для изменения значения
        // a.x на новое значение 2
    }
}
```

> В методе `d` компилятору позволено свободно переупорядочивать чтения `x` и вызов `g`. Таким образом, выражение `new A().f()` может вернуть `-1`, `0` или `1`.

Реализация может предоставить способ выполнения блока кода в *безопасном для* `final`*-полей контексте*. Если объект создан в безопасном для `final`-полей контексте, чтения `final`-поля этого объекта не будут переупорядочиваться с модификациями `final`-поля, которые осуществляются в данном в безопасном для `final`-полей контексте.

Безопасный для `final`-полей контекст имеет дополнительную защиту. Если поток видел некорректно опубликованную ссылку на объект, позволяющую потоку видеть значение по умолчанию `final`-поля, а затем, в безопасном для `final`-полей контексте, читает корректно опубликованную ссылку на объект, это будет гарантировать корректность значения `final`-поля. В данном формализме код, выполняемый в безопасном для `final`-полей контексте, рассматривается как отдельный поток (только для целей семантики `final`-поля).

В реализации компилятор не должен перемещать доступ к `final`-полю в безопасный для `final`-полей контекст или из него (хотя его можно перемещать в пределах выполнения такого контекста, если сам объект не создается в данном контексте).

> Одним из мест, где целесообразно применение безопасного для `final`-полей контекста, является исполнитель `java.util.concurrent.Executors` или пул потоков. Путем выполнения каждого объекта `Runnable` в отдельном безопасном для `final`-полей контексте исполнитель гарантирует, что некорректный доступ одним объектом `Runnable` к объекту *o* не удаляет гарантии `final`-поля для других объектов `Runnable`, обрабатываемых тем же исполнителем.

§17.5.4. Поля, защищенные от записи

Обычно поле, являющееся `final` и `static`, не может быть изменено. Однако поля `System.in`, `System.out` и `System.err` представляют собой поля `static final`, которые для совместимости с прежними версиями должны изменяться с помощью методов `System.setIn`, `System.setOut` и `System.setErr`. Мы называем эти поля *защищенными от записи*, чтобы отличать их от обычных `final`-полей.

Компилятор должен рассматривать эти поля не так, как остальные `final`-поля. Например, чтение обычного `final`-поля имеет "иммунитет" к синхронизации: проблемы с блокировкой или синхронизированное чтение не влияют на то, какое значение будет считано из `final`-поля. Поскольку значение поля, защищенного от записи, может изменяться, события синхронизации должны влиять на них. Следовательно, семантика требует, чтобы эти поля рассматривались как обычные поля, которые не могут быть изменены пользовательским кодом, если только этот код не находится в классе `System`.

§17.6. Разрыв слова

Одним из соображений при реализации виртуальной машины Java является то, что каждое поле и элемент массива рассматривается как отдельный объект; обновления одного поля или элемента не должны влиять на чтения или обновления любого другого поля или

элемента. В частности, два потока, которые по отдельности обновляют соседние элементы массива байтов, не должны влиять один на другой и не должны требовать синхронизации для обеспечения последовательной согласованности.

Некоторые процессоры не предоставляют возможности записи отдельного байта. Было бы неверным подходом реализовывать обновление байтового массива на таком процессоре путем простого чтения целого слова, исправления в нем одного байта и перезаписи слова в память. Эта проблема известна под названием *разрыва слова*, и на процессорах, которые не могут легко изменять один отдельный байт, требуется некоторый другой подход.

Пример 17.6-1. Обнаружение разрыва слова

Приведенная далее программа представляет собой тест, обнаруживающий разрыв слова.

```
public class WordTearing extends Thread {
    static final int LENGTH = 8;
    static final int ITERS = 1000000;
    static byte[] counts = new byte[LENGTH];
    static Thread[] threads = new Thread[LENGTH];

    final int id;
    WordTearing(int i) {
        id = i;
    }

    public void run() {
        byte v = 0;
        for (int i = 0; i < ITERS; i++) {
            byte v2 = counts[id];
            if (v != v2) {
                System.err.println("Word-Tearing found: " +
                                   "counts[" + id + "] = "+ v2 +
                                   ", should be " + v);
                return;
            }
            v++;
            counts[id] = v;
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < LENGTH; ++i)
            (threads[i] = new WordTearing(i)).start();
    }
}
```

Смысл заключается в том, что байты не должны перезаписываться при записи соседних байтов.

§17.7. Неатомарное рассмотрение `double` и `long`

Для целей модели памяти языка программирования Java отдельная запись значения `long` или `double`, не являющегося `volatile`, рассматривается как две отдельные записи — по одной в каждую 32-битовую половину. В результате возможна ситуация, когда поток видит первые 32 бита 64-битового значения от одной записи, а вторые 32 бита — от другой.

Записи и чтения значений `long` и `double`, являющихся `volatile`, всегда атомарны.

Записи и чтения ссылок всегда атомарны, независимо от того, как они реализованы: как 32- или 64-битовые значения.

> Некоторые реализации могут счесть удобным разделение одного действия записи 64-битового значения `long` или `double` на две записи смежных 32-битовых значений. Ради эффективности это поведение зависит от реализации; реализация виртуальной машины Java может выполнять запись значений `long` и `double` как атомарно, так и разделяя ее на две части.
>
> Реализации виртуальной машины Java рекомендуется избегать разделения 64-битовых значений на части везде, где это возможно. Программистам же предлагается объявлять совместно используемые 64-битовые значения как `volatile` или корректно синхронизировать свои программы во избежание возможных осложнений.

ГЛАВА 18

Вывод типов

Множеству анализов времени компиляции требуется умозаключение о типах, которые пока что не известны. Основными среди них являются проверка применимости обобщенного метода (§18.5.1) и вывод типа вызова обобщенного метода (§18.5.2). В общем случае процесс получения информации о неизвестных типах именуется *выводом типа* (type inference).

На верхнем уровне вывод типа можно разделить на три процесса.

- *Приведение* (reduction) получает утверждение о совместимости выражения или типа, именуемое *формулой ограничения* (constraint formula), и приводит (сокращает) его к набору *границ* (bounds) для *переменных вывода* (inference variables). Зачастую формула ограничений приводит к *другим* формулам ограничений, которые, в свою очередь, должны быть рекурсивно приведены. Следование данной процедуре приводит к идентификации этих дополнительных формул ограничения и в конечном итоге к выражению условий через набор границ, при которых выбор выводимых типов будет делать истинной каждую формулу ограничения.
- *Объединение* (incorporation) поддерживает набор границ переменных вывода, гарантируя, что они согласуются с вновь добавляемыми границами. Поскольку границы для одной переменной могут иногда влиять на возможный выбор для другой переменной, этот процесс устанавливает (фиксирует) границы между такими независимыми переменными.
- *Разрешение* (resolution) исследует границы переменных вывода и определяет *инстанцирование* (instantiation), совместимое с этими границами. Оно также принимает решение о порядке, в котором выполняется разрешение взаимозависимых переменных вывода.

Эти процессы тесно взаимодействуют: приведение может запустить объединение, объединение может привести к дальнейшему приведению, а разрешение — к дальнейшему объединению.

- В §18.1 более точно определены концепции, используемые в качестве промежуточных результатов, и обозначения, используемые для их выражения.
- В §18.2 подробно описано приведение.
- В §18.3 подробно описано объединение.
- В §18.4 подробно описано разрешение.

- В §18.5 определяется, как эти инструменты вывода используются для решения некоторых задач анализа времени компиляции.

По сравнению с Java SE 7 Edition в вывод типов внесены важные изменения:

- Добавлена поддержка лямбда-выражений и ссылок на методы в качестве аргументов вызова методов.
- Обобщено определение вывода в терминах поливыражений, которые могут не иметь точно определенных типов до завершения вывода. Это оказывает заметное влияние на улучшение вывода для вложенных обобщенных методов и вызовов конструкторов с подставленными обобщенными-типами (оператор “бубна”).
- Описано, как использовать вывод для работы с целевыми типами функциональных интерфейсов, параметризированных символами подстановки, и для анализа наиболее подходящих методов.
- Уточнено различие между проверкой применимости для вызова (которая включает только аргументы вызова) и выводом типа вызова (включающим целевой тип).
- Для получения лучших результатов разрешение всех переменных вывода, даже имеющих нижние границы, откладывается до вывода типа вызова.
- Улучшено поведение вывода для взаимозависимых (или самозависимых) переменных.
- Устранены ошибки и потенциальные источники путаницы. Эта версия более осторожно и точно обрабатывает различия между определенными контекстами преобразования и субтипированиями и описывает приведение путем распараллеливания соответствующих отношений, не являющихся отношениями вывода. Все случаи преднамеренного отхода от отношений, не являющихся отношениями вывода, теперь явно определены.
- Заложен фундамент для будущего развития: усовершенствования или новые применения вывода будет проще интегрировать в спецификацию языка.

§18.1. Концепции и обозначения

В этом разделе определяются термины *переменные вывода*, *формулы ограничений* и *границы*, которые будут использоваться во всей этой главе. Здесь также представлены соответствующие обозначения.

§18.1.1. Переменные вывода

Переменные вывода представляют собой *метапеременные* для типов, т.е. специальные имена, которые позволяют вести абстрактные рассуждения о типах. Чтобы отличать их от *переменных типа*, переменные вывода представлены греческими буквами, главным образом — буквой α.

Термин “тип” в этой главе используется достаточно свободно и включает “типообразный” синтаксис, содержащий переменные вывода. Термин *истинный тип* (proper type) исключает такие типы, упоминающие переменные вывода. Утверждения, включающие

переменные вывода, являются утверждениями о каждом истинном типе, который может быть получен путем замены каждой переменной вывода истинным типом.

§18.1.2. Формулы ограничений

Формулы ограничений являются утверждениями о совместимости или субтипировании, которые могут включать переменные вывода. Эти формулы могут принимать один из следующих видов.

- ‹*Expression* → *T*›: выражение совместимо в контексте нестрогого вызова с типом *T* (§5.3).
- ‹*S* → *T*›: тип *S* совместим в контексте нестрогого вызова с типом *T* (§5.3).
- ‹*S* <: *T*›: ссылочный тип *S* является подтипом ссылочного типа *T* (§4.10).
- ‹*S* <= *T*›: аргумент типа *S* содержится в аргументе типа *T* (§4.5.1).
- ‹*S* = *T*›: ссылочный тип *S* совпадает со ссылочным типом *T* (§4.3.4), или аргумент типа *S* совпадает с аргументом типа *T*.
- ‹*LambdaExpression* $\rightarrow_{throws}$ *T*›: проверяемое исключение, генерируемое телом лямбда-выражения *LambdaExpression*, объявлено конструкцией `throws` типа функции, производного от *T*.
- ‹*MethodReference* $\rightarrow_{throws}$ *T*›: проверяемое исключение, генерируемое в методе по ссылке, объявлено конструкцией `throws` типа функции, производного от *T*.

Примеры формул ограничений.

- Из `Collections.singleton("hi")` мы получаем формулу ограничения ‹"hi" → α›. Посредством приведения мы получаем формулу ограничения ‹`String` <: α›.
- Из `Arrays.asList(1,2.0)` мы получаем формулы ограничений ‹`1` → α› и ‹`2.0` → α›. Посредством приведения мы получаем формулы ограничений ‹`int` → α› и ‹`double` → α›, а затем — ‹`Integer` <: α› и ‹`Double` <: α›.
- Из целевого типа вызова конструктора `List<Thread> lt = new ArrayList<>()` мы получаем формулу ограничения ‹`ArrayList<`α`>` → `List<Thread>`›. Посредством приведения мы получаем формулу ограничения ‹α <= `Thread`›, а затем — ‹α = `Thread`›.

§18.1.3. Границы

Во время процесса вывода поддерживается множество *границ* переменных вывода. Граница имеет один из следующих видов.

- *S* = *T*, где переменной вывода является по крайней мере одно из *S* или *T*: *S* представляет собой то же, что и *T*.
- *S* <: *T*, где переменной вывода является по крайней мере одно из *S* или *T*: *S* представляет собой подтип *T*.
- *false*: не существует корректного выбора переменных вывода.

- $G<\alpha_1,...,\alpha_n>$ = capture($G<A_1,...,A_n>$): переменные $\alpha_1,...,\alpha_n$ представляют результат преобразования при фиксации (§5.1.10), примененного к $G<A_1,...,A_n>$ (где $A_1,...,A_n$ могут быть типами или символами подстановки и могут ссылаться на переменные вывода).
- `throws` α: переменная вывода α находится в конструкции `throws`.

Подстановка переменной вывода *удовлетворяет* границе, если после применения подстановки утверждение (условие границы) истинно. Граница *false* не может быть удовлетворена никогда.

Некоторые границы (условия) связывают переменную вывода с истинным типом. Пусть T — истинный тип. Для заданной границы вида $\alpha = T$ или $T = \alpha$ мы говорим, что T является *инстанцированием* α. Аналогично для заданной границы вида $\alpha <: T$ мы говорим, что T является *истинной верхней границей* α, а для заданной границы вида $T <: \alpha$ мы говорим, что T является *истинной нижней границей* α.

Другие границы связывают две переменные вывода, или переменную вывода с типом, который содержит переменные вывода. Такие границы вида $S = T$ или $S <: T$ называются *зависимостями* (dependencies).

Граница вида $G<\alpha_1,...,\alpha_n>$ = capture($G<A_1,...,A_n>$) указывает, что $\alpha_1,...,\alpha_n$ являются заполнителями для результата преобразования при фиксации. Это необходимо, потому что преобразование при фиксации может быть выполнено только над истинным типом, и переменные вывода в $A_1,...,A_n$ могут еще не быть разрешены.

Граница вида `throws` α чисто информационная: она допускает оптимизацию инстанцирования α так, чтобы по возможности это не был тип проверяемого исключения.

Важным промежуточным результатом вывода является *множество границ* (bound set). Иногда оказывается удобно ссылаться на *пустое* множество границ как на символ *true*; это делается просто для удобства (обобщения), и эти объекты вполне взаимозаменяемы.

Вот некоторые примеры множеств границ.

- { α = `String` } содержит единственную границу, инстанцируя α как `String`.
- { `Integer` <: α, `Double` <: α, α <: `Object` } описывает две истинные нижние границы и одну истинную верхнюю границу для α.
- { α <: `Iterable<?>`, β <: `Object`, α <: `List`<β> } описывает истинную верхнюю границу как для α, так и для β, вместе с зависимостью между ними.
- { } не содержит ни границ, ни зависимостей, и к нему можно обращаться как к *true*.
- { *false* } выражает тот факт, что удовлетворяющего инстанцирования не существует.

Когда начинается вывод, множество границ обычно генерируется из списка объявлений параметров типа $P_1,...,P_p$ и связанных переменных вывода $\alpha_1,...,\alpha_p$. Такое множество границ строится следующим образом. Для каждого l ($1 \le l \le p$):

- если P_l не имеет *TypeBound*, в множество добавляется граница α_l <: `Object`;
- в противном случае для каждого типа T, отделенного `&` в *TypeBound*, в множество добавляется граница $\alpha_l <: T[P_1:=\alpha_1,...,P_p:=\alpha_p]$; если этот результат не входит в истинные верхние границы для α_l (только зависимости), то в множество вносится также граница α_l <: `Object`.

§18.2. Приведение

Приведение представляет собой процесс, с помощью которого множество формул ограничения (§18.1.2) упрощается до набора границ (§18.1.3).

По очереди рассматриваются все формулы ограничений. Правила в этом разделе определяют, как формула приводится к одному из двух видов.

- К границе (или множеству границ), которая затем объединяется с "текущим" множеством границ. Изначально текущее множество границ является пустым.
- К другим формулам ограничений, которые затем приводятся рекурсивно.

Приведение завершается, когда не остается приводимых формул ограничений.

> Результат шага приведения всегда является *сохраняющим надежность* (soundness-preserving): если инстанцирование переменной вывода удовлетворяет приведенным ограничениям и границам, то оно также удовлетворяет исходному ограничению. С другой стороны, приведение не является *сохраняющим полноту* (completeness-preserving): может существовать переменная вывода, которая удовлетворяет исходному ограничению, но *не* удовлетворяет приведенному ограничению или границе. Это связано с ограничениями алгоритма приведения наряду с желанием избежать неоправданных сложностей. В результате существуют выражения, для которых вывод аргументов типа не находит решения, но которые могут быть корректно типизированы, если программист явно укажет соответствующие типы.

§18.2.1. Ограничения совместимости выражений

Формула ограничений вида ‹*Expression* → *T*› приводится следующим образом.

- Если *T* представляет собой истинный тип, ограничение приводится к *true*, если выражение совместимо в контексте нестрогого присваивания с *T* (§5.3), и к *false* в противном случае.
- В противном случае, если выражение является автономным выражением (§15.2) типа *S*, ограничение приводится к ‹*S* → *T*›.
- В противном случае выражение является поливыражением (§15.2). Результат зависит от вида выражения.
  - ✦ Если выражение представляет собой выражение в скобках вида (*Expression'*), то ограничение приводится к ‹*Expression'* → *T*›.
  - ✦ Если выражение представляет собой выражение создания экземпляра класса или выражение вызова метода, ограничение приводится ко множеству границ B_3, которое будет использоваться для определения типа вызова выражения при целевом типе *T*, как определено в §18.5.2. (В случае выражения создания экземпляра класса соответствующий используемый для вывода "метод" определяется в §15.9.3.)

    Это множество границ может содержать новые переменные вывода, а также зависимости между этими новыми переменными и переменными вывода в *T*.
  - ✦ Если выражение представляет собой условное выражение вида e_1 ? e_2 : e_3, то ограничение приводится к двум формулам ограничений, ‹e_2 → *T*› и ‹e_3 → *T*›.

✦ Если выражение представляет собой лямбда-выражение или выражение ссылки на метод, результат описывается ниже.

Рассматривая вложенные вызовы обобщенных методов как поливыражения, мы улучшаем поведение вывода для вложенных вызовов. Например, приведенный далее фрагмент некорректен в Java SE 7, но вполне законен в Java SE 8.

```
ProcessBuilder b = new ProcessBuilder(Collections.emptyList());
// Конструктор ProcessBuilder ожидает List<String>
```

Когда *оба* вызова — и внешний, и вложенный — требуют вывода, задача становится более сложной. Например, рассмотрим такой код.

```
List<String> ls = new ArrayList<>(Collections.emptyList());
```

Наш подход состоит в том, чтобы "поднять" границы, выведенные для вложенного вызова (просто { α <: `Object` } в случае `emptyList`) в процесс вывода для внешнего вызова (в данном случае пытаясь вывести β, где конструктор создает `ArrayList`<β>). Мы также выводим зависимости между вложенными переменными вывода и внешними переменными вывода (ограничение ‹`List`<α> → `Collection`<β>› приводится к зависимости α = β). При таком способе разрешение переменных вывода во вложенном вызове может подождать до тех пор, пока дополнительная информация не сможет быть выведена из внешнего вызова (на основе цели присваивания β = `String`).

Формула ограничения вида ‹*LambdaExpression* → *T*›, где *T* упоминает как минимум одну переменную вывода, приводится следующим образом.

- Если *T* не является типом функционального интерфейса (§9.8), ограничение приводится к *false*.
- В противном случае пусть *T'* представляет собой базовый целевой тип, производный от *T*, как указано в §15.27.3. Если для вывода параметризованного типа функционального интерфейса используется §18.5.3, то проверка того, что $F\langle A'_1,...,A'_m\rangle$ является подтипом типа *F*<*A1*,...,*Am*>, не выполняется (вместо этого данный факт подтверждается с помощью приведенной ниже формулы ограничения). Пусть тип целевой функции лямбда-выражения представляет собой тип функции *T'*. Тогда справедливо следующее.
  - ✦ Если корректный тип функции не может быть найден, ограничение приводится к *false*.
  - ✦ В противном случае конгруэнтность *LambdaExpression* с целевым типом функции подтверждается следующим образом.
    - — Если количество параметров лямбда-выражения отличается от количества типов параметров типа функции, ограничение приводится к *false*.
    - — Если лямбда-выражение является неявно типизированным и один или несколько типов параметров типа функции не являются истинными типами, ограничение сводится к *false*.
    - — Если результат типа функции представляет собой `void`, а тело лямбда-выражения не является ни выражением инструкции, ни блоком, совместимым с `void`, ограничение приводится к *false*.

— Если результат типа функции представляет собой не `void`, а тело лямбда-выражения является блоком, который не совместим со значением, ограничение приводится к *false*.

— В противном случае ограничение приводится ко всем следующим формулам ограничений.

» Если параметры лямбда-выражения имеют явно объявленные типы $F_1,...,F_n$, а тип функции имеет типы параметров $G_1,...,G_n$, то приведение дает 1) для всех i $(1 \le i \le n)$ $\langle F_i = G_i \rangle$ и 2) $\langle T' <: T \rangle$.

» Если возвращаемый тип типа функции является (не-`void`) типом R, будем считать, что типы параметров лямбда-выражения те же, что и типы параметров типа функции. Тогда справедливо следующее.

* Если R является истинным типом и если тело лямбда-выражения или некоторое выражение результата в теле лямбда-выражения не совместимо в контексте присваивания с R, то ограничение приводится к *false*.

* В противном случае, если R не является истинным типом, то, если тело лямбда-выражения имеет вид *Expression*, ограничение приводится к $\langle Expression \rightarrow R \rangle$; если же тело лямбда-выражения является блоком с выражениями результата $e_1,...,e_m$, то приведение дает для всех i $(1 \le i \le m)$ $\langle e_i \rightarrow R \rangle$.

Ключевой частью информации для выполнения приведения ограничения совместимости, включающего лямбда-выражение, является множество границ переменных вывода, находящихся в возвращаемом типе целевого типа функции. Это важно, потому что функциональные интерфейсы часто являются обобщенными, как и многие методы, работающие с этими типами.

В простейшем случае лямбда-выражение может просто предоставлять нижнюю границу для переменной вывода.

```
<T> List<T> makeThree(Factory<T> factory) { ... }
String s = makeThree(() -> "abc").get(2);
```

В более сложных случаях выражение результата может быть поливыражением — возможно, даже другим лямбда-выражением, — так что переменная вывода может быть передана через несколько формул ограничений с различными целевыми типами перед тем, как будет получена граница.

Большей части работы, описанной в этом разделе, предшествуют утверждения (правила) о выражениях результата; их цель — в выводе типа функции лямбда-выражения и проверке выражений, которые явно не поддерживают совместимость.

Мы *не* пытаемся получить границы переменных вывода, которые находятся в конструкции `throws` целевого типа функции. Это связано с тем, что содержимое исключения не является частью совместимости (§15.27.3); в частности, оно не должно влиять на применимость метода (§18.5.1). Однако мы *получаем* границы для этих переменных позже, поскольку вывод типа вызова (§18.5.2) дает формулы (правила) ограничений содержимого исключений (§18.2.5).

Обратите внимание, что если целевой тип является переменной вывода или если типы параметров целевого типа содержат переменные вывода, мы получаем *false*.

> Во время вывода типа вызова (§18.5.2) для того, чтобы инстанцировать эти переменные вывода, тем самым избегая описанного сценария, выполняются дополнительные замены. (Другими словами, на практике приведение никогда не будет "вызвано" с целевым типом одного из этих видов.)
>
> Наконец, обратите внимание, что выражения результата лямбда-выражения согласно §15.27.3 должны быть совместимы в контексте присваивания с возвращаемым типом R целевого типа. Если R является истинным типом, таким как `Byte`, полученным из `Function<`α`, Byte>`, то присваиваемость достаточно легко протестировать, и приведение работает, как описано выше. Если R не является истинным типом, таким как α, полученное из `Function<String,`α`>`, то мы делаем упомянутое выше упрощающее предположение о том, что совместимости с нестрогим вызовом будет достаточно. Различие между совместимостью по присваиванию и совместимостью с нестрогим вызовом заключается только в том, что присваивание разрешает сужение константных выражений, таких, как `Byte b = 100;`. Следовательно, наше упрощающее предположение не является сохраняющим полноту: для заданного целевого типа возврата α и целочисленного литерального выражения возврата `100` возможно, что α может быть инстанцировано как `Byte`, но на самом деле приведение не даст такую границу.

Формула ограничения вида ‹*MethodReference* → T›, где T упоминает как минимум одну переменную вывода, приводится следующим образом.

- Если T не является типом функционального интерфейса или если T является типом функционального интерфейса, который не имеет типа функции (§9.9), ограничение приводится к *false*.
- В противном случае, если не существует потенциально применимого метода для выражения ссылки на метод с целевым типом T, ограничение приводится к *false*.
- В противном случае, если ссылка на метод точная (§15.13.1), пусть $P_1,...,P_n$ являются типами параметров типа функции T и пусть $F_1,...,F_k$ являются типами параметров потенциально применимого метода. Тогда ограничение приводится к новому множеству ограничений следующим образом.
  - ✦ В частном случае, когда $n = k + 1$, параметр типа P_1 действует как целевая ссылка вызова. Выражение ссылки на метод в обязательном порядке имеет вид *ReferenceType* `::` *[TypeArguments] Identifier*. Ограничение приводится к ‹$P1$ `<:` *ReferenceType*› и для всех i $(2 \le i \le n)$ — к ‹$P_i \rightarrow F_{i-1}$›.

    Во всех остальных случаях $n = k$, и ограничение для всех i $(1 \le i \le n)$ приводится к ‹$P_i \rightarrow F_i$›.
  - ✦ Если результат типа функции — не `void`, то обозначим тип результата как R. Затем, если результат потенциально применимого объявления времени компиляции — `void`, ограничение приводится к *false*. В противном случае ограничение приводится к ‹$R' \rightarrow R$›, где R' представляет собой результат применения преобразования при фиксации (§5.1.10) к возвращаемому типу потенциально применимого объявления времени компиляции.
- В противном случае ссылка на метод неточная, и выполняется следующее.

- ✦ Если один (или несколько) тип параметров типа функции не является истинным типом, ограничение приводится к *false*.
- ✦ В противном случае выполняется поиск объявления времени компиляции, определенный в §15.13.1. Если объявления времени компиляции для ссылки на метод нет, ограничение приводится к *false*. В противном случае имеется объявление времени компиляции, и выполняется следующее.

  — Если результат типа функции — `void`, ограничение приводится к *true*.

  — В противном случае, если выражение ссылки на метод скрывает *TypeArguments*, объявление времени компиляции представляет собой обобщенный метод, а возвращаемый тип объявления времени компиляции упоминает по меньшей мере один из параметров типа метода, то ограничение приводится к множеству границ B_3, которое будет использоваться для определения типа вызова ссылки на метод при целевом типе типа функции, как определено в §18.5.2. B_3 может содержать новые переменные вывода, а также зависимости между этими новыми переменными и переменными вывода в T.

  — В противном случае пусть возвращаемый тип типа функции — R и пусть R' представляет собой результат применения преобразования при фиксации (§5.1.10) к возвращаемому типу типа вызова (§15.12.2.6) объявления времени компиляции. Если R' представляет собой `void`, ограничение приводится к *false*; в противном случае ограничение приводится к $\langle R' \rightarrow R \rangle$.

> Стратегия, использованная для определения возвращаемого типа для обобщенного метода, на который имеется выражение ссылки, следует той же схеме, что и для вызовов обобщенных методов (§18.2.1). Она может включать "подъем" границ во внешний контекст и вывод зависимостей между двумя множествами переменных вывода.

§18.2.2. Ограничения совместимости типов

Формула ограничения вида $\langle S \rightarrow T \rangle$ приводится следующим образом.

- Если S и T являются истинными типами, ограничение приводится к *true*, если S совместимо в контексте нестрогого присваивания с T (§5.3), и к *false* в противном случае.
- В противном случае, если S является примитивным типом, пусть S' представляет собой результат применения преобразования упаковки (§5.1.7) к S.Тогда ограничение приводится к $\langle S' \rightarrow T \rangle$.
- В противном случае, если T является примитивным типом, пусть T' представляет собой результат применения преобразования упаковки (§5.1.7) к T. Тогда ограничение приводится к $\langle S = T' \rangle$.
- В противном случае, если T является параметризованным типом вида $G\langle T_1,...,T_n \rangle$ и не существует типа вида $G\langle ... \rangle$, который является супертипом типа S, но несформированный тип G является супертипом типа S, ограничение приводится к *true*.
- В противном случае, если T представляет собой тип массива вида $G\langle T_1,...,T_n \rangle [\,]^k$ и не существует типа вида $G\langle ... \rangle [\,]^k$, являющегося супертипом S, но несформированный

тип $G[]^k$ является супертипом S, ограничение приводится к *true*. (Запись $[]^k$ указывает на тип массива с k размерностями.)

- В противном случае ограничение приводится к $\langle S <: T \rangle$.

Четвертый и пятый случаи представляют собой неявное использование непроверяемого преобразования (§5.1.9). Они, вместе с любым использованием непроверяемого преобразования в первом случае, могут давать предупреждения времени компиляции о непроверенных типах и влиять на тип вызова метода (§15.12.2.6).

Упаковка T в T' не является сохраняющей полноту; например, если T представляет собой `long`, то S может быть инстанцировано в `Integer`, который не является подтипом `Long`, но может быть распакован, а затем расширен до `long`. В большинстве случаев мы избегаем этой проблемы, особым образом рассматривая возвращаемые типы переменных вывода, о которых нам известно, что они уже ограничены определенными упакованными примитивными типами. Смотрите §18.5.2.

Аналогично обработка непроверяемых преобразований жертвует полнотой в случаях, когда T не является параметризованным типом (например, если T представляет собой переменную вывода). В таких ситуациях не всегда ясно, является ли непроверяемое преобразование необходимым. Поскольку непроверяемые преобразования приводят к соответствующим предупреждениям, вывод предпочитает их избегать, если только они не являются очевидно необходимыми.

§18.2.3. Ограничения субтипирования

Формула ограничения вида $\langle S <: T \rangle$ приводится следующим образом.

- Если S и T являются истинными типами, ограничение приводится к *true*, если S является подтипом типа T (§4.10), и к *false* — в противном случае.
- В противном случае, если S является типом null, ограничение приводится к *true*.
- В противном случае, если T является типом null, ограничение приводится к *false*.
- В противном случае, если S является переменной вывода, α, ограничение приводится к границе $\alpha <: T$.
- В противном случае, если T является переменной вывода α, ограничение приводится к границе $S <: \alpha$.
- В противном случае ограничение приводится в соответствии с видом T.
  - ✦ Если T является параметризованным типом класса или интерфейса или типом внутреннего класса параметризованного типа класса или интерфейса (непосредственно или косвенно), то пусть $A_1,...,A_n$ представляют собой аргументы типа T. Среди супертипов S идентифицируется соответствующий тип класса или интерфейса с аргументами типа $B_1,...,B_n$. Если такой тип не существует, ограничение приводится к *false*. В противном случае ограничение приводится к следующим новым ограничениям: для всех i ($1 \le i \le n$) $\langle B_i <= A_i \rangle$.
  - ✦ Если T представляет собой тип любого другого класса или интерфейса, то ограничение приводится к *true*, если T — среди супертипов S, и к *false* в противном случае.

- ✦ Если T представляет собой тип массива, T'`[]`, то среди супертипов S, которые являются типами массивов, идентифицируется наиболее подходящий, S'`[]` (это может быть сам S). Если такой тип массива не существует, ограничение приводится к *false*. В противном случае:
  - — если ни S', ни T' не является примитивным типом, ограничение приводится к ‹S' `<:` T'›;
  - — в противном случае ограничение приводится к *true*, если S' и T' представляют собой один и тот же примитивный тип, и *false* в противном случае.
- ✦ Если T является переменной типа, есть три подслучая.
  - — Если S является типом пересечения, элементом которого является тип T, ограничение приводится к *true*.
  - — В противном случае, если T имеет нижнюю границу B, ограничение приводится к ‹S `<:` B›.
  - — В противном случае ограничение приводится к *false*.
- ✦ Если T является типом пересечения I_1 `&` ... `&` I_n, ограничение приводится к следующим новым ограничениям: для всех i ($1 \le i \le n$) ‹S `<:` I_i›.

Формула ограничения вида ‹S `<=` T›, где S и T являются аргументами типа (§4.5.1), приводится следующим образом.

- Если T является типом:
  - ✦ если S является типом, ограничение приводится к ‹S `=` T›;
  - ✦ если S является символом подстановки, ограничение приводится к *false*.
- Если T является символом подстановки вида `?`, ограничение приводится к *true*.
- Если T является символом подстановки вида `? extends` T':
  - ✦ если S является типом, ограничение приводится к ‹S `<:` T'›;
  - ✦ если S является символом подстановки вида `?`, ограничение приводится к ‹`Object <:` T'›;
  - ✦ если S является символом подстановки вида `? extends` S', ограничение приводится к ‹S' `<:` T'›;
  - ✦ если S является символом подстановки вида `? super` S', ограничение приводится к ‹`Object =` T'›.
- Если T является символом подстановки вида `? super` T':
  - ✦ если S является типом, ограничение приводится к ‹T' `<:` S›;
  - ✦ если S является символом подстановки вида `? super` S', ограничение приводится к ‹T' `<:` S'›;
  - ✦ в противном случае ограничение приводится к *false*.

§18.2.4. Ограничения эквивалентности типов

Формула ограничения вида ‹S = T›, где S и T являются типами, приводится следующим образом.

- Если S и T являются истинными типами, ограничение приводится к *true*, если S совпадает с T (§4.3.4), и к *false* в противном случае.
- В противном случае, если S является переменной вывода α, ограничение приводится к границе $\alpha = T$.
- В противном случае, если T является переменной вывода α, ограничение приводится к границе $S = \alpha$.
- В противном случае, если S и T являются типами класса или интерфейса с одним и тем же затиранием, где S имеет аргументы типа $B_1,...,B_n$, а T имеет аргументы типа $A_1,...,A_n$, ограничение приводится к следующим новым ограничениям: для всех i $(1 \le i \le n)$ ‹$B_i = A_i$›.
- В противном случае, если S и T являются типами массивов S'`[]` и T'`[]`, ограничение приводится к ‹$S' = T'$›.
- В противном случае ограничение приводится к *false*.

> Обратите внимание, что мы не рассматриваем описанные выше типы пересечения, поскольку приведение не может столкнуться с типом пересечения, который не является истинным типом.

Формула ограничения вида ‹S = T›, где S и T являются аргументами типов (§4.5.1), приводится следующим образом.

- Если S и T являются типами, ограничение приводится так, как описано выше.
- Если S имеет вид `?`, а T имеет вид `?`, ограничение приводится к *true*.
- Если S имеет вид `?`, а T имеет вид `? extends` T', ограничение приводится к ‹`Object` = T'›.
- Если S имеет вид `? extends` S', а T имеет вид `?`, ограничение приводится к ‹S' = `Object`›.
- Если S имеет вид `? extends` S', а T имеет вид `? extends` T', ограничение приводится к ‹$S' = T'$›.
- Если S имеет вид `? super` S', а T имеет вид `? super` T', ограничение приводится к ‹$S' = T'$›.
- В противном случае ограничение приводится к *false*.

§18.2.5. Ограничения проверяемых исключений

Формула ограничения вида ‹$LambdaExpression \rightarrow_{throws} T$› приводится следующим образом.

- Если T не является типом функционального интерфейса (§9.8), ограничение приводится к *false*.

- В противном случае пусть целевой тип функции для лямбда-выражения определяется так, как описано в §15.27.3. Если корректный тип функции не может быть найден, ограничение приводится к *false*.
- В противном случае, если лямбда-выражение неявно типизировано и один или несколько типов параметров типа функции не являются истинными типами, ограничение приводится к *false*.
- В противном случае, если тип возврата типа функции не является ни `void`, ни истинным типом, ограничение приводится к *false*.
- В противном случае пусть $E_1,...,E_n$ представляют собой типы в конструкции `throws` типа функции, которые *не* являются истинными типами. Если лямбда-выражение неявно типизировано, пусть типами его параметров являются типы параметров типа функции. Если тело лямбда-выражения является поливыражением или блоком, содержащим выражение результата, являющееся поливыражением, то пусть целевой возвращаемый тип является возвращаемым типом типа функции. Пусть $X_1,...,X_m$ являются типами проверяемых исключений, которые может генерировать тело лямбда-выражения (§11.2). Тогда следует рассмотреть два случая.
  - ✦ Если $n = 0$ (конструкция `throws` типа функции состоит только из истинных типов), то, если существует некоторое i ($1 \le i \le m$), такое, что X_i не является подтипом любого истинного типа в конструкции `throws`, ограничение приводится к *false*; в противном случае ограничение приводится к *true*.
  - ✦ Если $n > 0$, ограничение приводится к множеству ограничений субтипирования: для всех i ($1 \le i \le m$), если X_i не является подтипом любого истинного типа в конструкции `throws`, ограничения включают для всех j ($1 \le j \le n$) ‹X_i <: E_j›. Кроме того, для всех j ($1 \le j \le n$) ограничение приводится к границе `throws` E_j.

Формула ограничения вида ‹*MethodReference* $\rightarrow_{throws}$ T› приводится следующим образом.

- Если T не является типом функционального интерфейса или если T является типом функционального интерфейса, но не имеет типа функции (§9.9), ограничение приводится к *false*.
- В противном случае пусть целевой тип функции выражения ссылки на метод является типом функции типа T. Если ссылка на метод является неточной (§15.13.1) и один или несколько типов параметров типа функции не является истинным типом, ограничение приводится к *false*.
- В противном случае, если ссылка на метод является неточной и результат типа функции не является ни `void`, ни истинным типом, ограничение приводится к *false*.
- В противном случае пусть $E_1,...,E_n$ представляют собой типы в конструкции `throws` типа функции, которые *не* являются истинными типами. Пусть $X_1,...,X_m$ — проверяемые исключения в конструкции `throws` типа вызова объявления времени компиляции ссылки на метод (§15.13.2) (полученного из типов параметров и возвращаемого типа типа функции). Тогда имеются два случая, которые следует рассмотреть.

- Если $n = 0$ (конструкция `throws` типа функции состоит только из истинных типов), то, если существует некоторое i ($1 \le i \le m$), такое, что X_i не является подтипом любого истинного типа в конструкции `throws`, ограничение приводится к *false*; в противном случае ограничение приводится к *true*.
- Если $n > 0$, ограничение приводится к множеству ограничений субтипирования: для всех i ($1 \le i \le m$), если X_i не является подтипом любого истинного типа в конструкции `throws`, ограничения включают для всех j ($1 \le j \le n$) $\langle X_i <: E_j \rangle$. Кроме того, для всех j ($1 \le j \le n$) ограничение приводится к границе `throws` E_j.

Ограничения на проверяемые исключения обрабатываются отдельно от ограничений на возвращаемые типы, поскольку совместимость возвращаемого типа влияет на применимость методов (§18.5.1), в то время как исключения влияют только на тип вызова после разрешения перегрузки (§18.5.2). Это можно упростить, включив совместимость исключений в определение совместимости лямбда-выражения (§15.27.3), но это могло бы привести к, возможно, удивительным случаям, в которых исключения, могущие быть сгенерированными явно типизированным телом лямбда-выражения, изменяют разрешение перегрузки.

Исключения, генерируемые телом лямбда-выражения, не могут быть определены до тех пор, пока 1) не будут известны типы параметров лямбда-выражения, и 2) не будет известен целевой тип выражений возврата в теле лямбда-выражения. (Второе требование заключается в учете вызовов обобщенных методов, в которых, например, один и тот же тип параметра находится и в возврате выражения, и в конструкции `throws`.) Следовательно, нужны оба требования, чтобы правильно определить производную от целевого типа T.

Одним из следствий является то, что лямбда-выражения, возвращенные из *других* лямбда-выражений, не могут генерировать ограничения из генерируемых ими исключений. Данные ограничения могут формироваться только из лямбда-выражений верхнего уровня.

Обратите внимание, что обработка случая, в котором в конструкции `throws` появляется более одной переменной вывода, не является сохраняющей полноту. Каждая переменная может сама по себе удовлетворять ограничению, которое объявляет каждое проверяемое исключение, но мы не можем знать точно, какая из переменных подразумевается. Поэтому для предсказуемости мы ограничиваем их все.

§18.3. Объединение

По мере создания и увеличения множеств границ во время вывода возможно, что новые границы могут выводиться на основании утверждений об исходных границах. Процесс *объединения* (incorporation) идентифицирует эти новые границы и добавляет их в множество границ.

Объединение может происходить в двух сценариях. Один из сценариев — множество границ содержит комплементарные пары границ; это приводит к новым формулам ограничений, как указано в §18.3.1. Второй сценарий — множество границ содержит границу, включающую преобразование при фиксации; это приводит к новым границам и может

привести к новым формулам ограничений, как указано в §18.3.2. В обоих сценариях выполняется приведение любых новых формул ограничений, а все новые границы добавляются в множество границ. Это может привести к новому объединению; в конечном итоге множество достигнет фиксированной точки, после чего вывод новых границ прекратится.

Если объединение множества границ достигает фиксированной точки и множество не содержит границу *false*, то множество границ обладает следующими свойствами.

- Для каждой комбинации истинной нижней границы L и истинной верхней границы U переменной вывода $L <: U$.
- Если каждая переменная вывода, упомянутая границей, инстанцирована, граница удовлетворяется соответствующей подстановкой.
- Для данной зависимости $\alpha = \beta$ каждая граница α соответствует границе β и наоборот.
- Для данной зависимости $\alpha <: \beta$ каждая нижняя граница α является нижней границей β, а каждая верхняя граница β является верхней границей α.

> Утверждение, что объединение достигает фиксированной точки, немного упрощает дело. Опираясь на работу Кеннеди (Kennedy) и Пирса (Pierce), *On Decidability of Nominal Subtyping with Variance* ("О разрешимости номинального субтипирования с вариациями"), это свойство можно доказать с помощью аргумента, что множество типов, которое может находиться в множестве границ, конечно. Этот аргумент опирается на два предположения:
>
> - новые переменные фиксации при приведении ограничений субтипирования не генерируются (§18.2.3);
> - пути экспансивного наследования не рассматриваются (т.е. например, иерархия классов ограничена).
>
> Данная спецификация в настоящий момент не гарантирует указанные свойства (это неточно в случае обработки символов подстановки при приведении ограничений субтипирования и не обнаруживает пути экспансивного наследования (неограниченной иерархии)), но в будущих версиях ситуация может измениться. (Эта проблема не нова: у алгоритма субтипирования в Java также имеется риск зацикливания.)

§18.3.1. Комплементарные пары границ

(В этом разделе S и T представляют собой переменные вывода или типы, а U является истинным типом. Для краткости граница вида $\alpha = T$ может также быть границей вида $T = \alpha$.)

Когда множество границ содержит пару границ, которые отвечают одному из приведенных далее правил, из них вытекает новая формула ограничений.

- Из $\alpha = S$ и $\alpha = T$ следует $\langle S = T \rangle$.
- Из $\alpha = S$ и $\alpha <: T$ следует $\langle S <: T \rangle$.
- Из $\alpha = S$ и $T <: \alpha$ следует $\langle T <: S \rangle$.
- Из $S <: \alpha$ и $\alpha <: T$ следует $\langle S <: T \rangle$.
- Из $\alpha = U$ и $S = T$ следует $\langle S[\alpha:=U] = T[\alpha:=U] \rangle$.

- Из $\alpha = U$ и $S <: T$ следует $\langle S[\alpha:=U] <: T[\alpha:=U] \rangle$.

Когда множество границ содержит пару границ $\alpha <: S$ и $\alpha <: T$ и имеются супертип S вида $G<S1,...,Sn>$ и супертип T вида $G<T1,...,Tn>$ (для некоторого обобщенного класса или интерфейса G), для всех i ($1 \le i \le n$), если Si и Ti являются типами (не символами подстановки), вытекает формула ограничения $\langle Si = Ti \rangle$.

§18.3.2. Границы, включающие преобразование при фиксации

Когда множество границ содержит границу вида $G<\alpha_1,...,\alpha_n> = \text{capture}(G<A_1,...,A_n>)$, отсюда вытекают новые границы, и могут получаться новые формулы ограничений, как показано ниже.

Пусть $P_1,...,P_n$ представляют параметры типов G и пусть $B_1,...,B_n$ представляют границы этих параметров типов. Пусть θ представляет подстановку $[P_1:=\alpha_1,...,P_n:=\alpha_n]$. Пусть R представляет собой тип, *не* являющийся переменной вывода (но не обязательно являющийся истинным типом).

Мы получаем множество границ $\alpha_1,...,\alpha_n$, построенное из объявленных границ $P_1,...,P_n$, как указано в §18.1.3.

Кроме того, для всех i ($1 \le i \le n$) справедливо следующее.

- Если A_i не является символом подстановки, то отсюда следует граница $\alpha_i = A_i$.
- Если A_i является символом подстановки вида `?`:
  - из $\alpha_i = R$ вытекает граница *false*;
  - из $\alpha_i <: R$ вытекает формула ограничения $\langle B_i\ \theta <: R \rangle$;
  - из $R <: \alpha_i$ вытекает граница *false*.
- Если A_i является символом подстановки вида `? extends` T:
  - из $\alpha_i = R$ вытекает граница *false*;
  - если B_i представляет собой `Object`, то из $\alpha_i <: R$ вытекает формула ограничений $\langle T <: R \rangle$;
  - если T представляет собой `Object`, то из $\alpha_i <: R$ вытекает формула ограничений $\langle B_i\ \theta <: R \rangle$;
  - из $R <: \alpha_i$ вытекает граница *false*.
- Если A_i является символом подстановки вида `? super` T:
  - из $\alpha_i = R$ вытекает граница *false*;
  - из $\alpha_i <: R$ вытекает формула ограничений $\langle B_i\ \theta <: R \rangle$;
  - из $R <: \alpha_i$ вытекает формула ограничений $\langle R <: T \rangle$.

§18.4. Разрешение

Для данного множества границ, которое не содержит границу *false*, подмножество переменных вывода, упоминаемое множеством границ, может быть *разрешено* (resolved). Это означает, что для каждой переменной вывода к множеству может добавляться удов-

летворяющее инстанцирование, пока все интересующие переменные не будут иметь инстанцирований.

Зависимости в множестве границ могут потребовать разрешения переменных в определенном порядке или разрешения дополнительных переменных. Зависимости определяются следующим образом.

- Для данной границы одного из приведенных далее видов, где T является либо переменной вывода β, либо типом, упоминающим β:
  - ✦ $\alpha = T$;
  - ✦ $\alpha <: T$;
  - ✦ $T = \alpha$;
  - ✦ $T <: \alpha$.

  Если α находится в левой части другой границы вида $G<...,\alpha,...>$ = capture($G<...>$), то β зависит от разрешения α. В противном случае α зависит от разрешения β.
- Переменная вывода α, находящаяся в левой части границы вида $G<...,\alpha,...>$ = captu re($G<...>$), зависит от разрешения каждой другой переменной вывода, упомянутой в этой границе (с обеих сторон от знака равенства).
- Переменная вывода α зависит от разрешения переменной вывода β, если существует переменная вывода χ, такая, что α зависит от разрешения χ, а χ зависит от разрешения β.
- Переменная вывода α зависит от разрешения самой себя.

Для заданного множества разрешаемых переменных вывода обозначим через V объединение этого множества и всех переменных, от которых зависит разрешение по крайней мере одной переменной из этого множества.

Если каждая переменная в V имеет инстанцирование, то разрешение успешно и процедура завершается.

В противном случае пусть $\{\alpha_1,...,\alpha_n\}$ является непустым подмножеством неинстанцированных переменных в V, такое, что 1) для всех i ($1 \le i \le n$), если α_i зависит от разрешения переменной β, то либо β имеет инстанцирование, либо имеется некоторое j, такое, что $\beta = \alpha_j$; и 2) не существует непустого истинного подмножества множества $\{\alpha_1,...,\alpha_n\}$, обладающего этим свойством. Разрешение выполняется путем генерации инстанцирований для каждой из $\alpha_1,...,\alpha_n$ на основе границ в множество границ.

- Если множество границ не содержит границу вида $G<...,\alpha_i,...>$ = capture($G<...>$) для всех i ($1 \le i \le n$), то для каждой α_i определен кандидат для инстанцирования T_i:
  - ✦ если α_i имеет одну или более *истинных* нижних границ $L_1,...,L_k$, то T_i = lub($L_1,...,L_k$) (§4.10.4);
  - ✦ в противном случае, если множество границ содержит `throws` α_i, а истинными верхними границами α_i являются, в пределах `Exception`, `Throwable` и `Object`, имеем T_i = `RuntimeException`;
  - ✦ в противном случае, когда α_i имеет *истинные* верхние границы $U_1,...,U_k$, T_i = glb × × ($U_1,...,U_k$) (§5.1.10).

Границы $\alpha_1 = T_1$, ..., $\alpha_n = T_n$ объединяются с текущим множеством границ.

Если результат не содержит границу *false*, то он становится новым множеством границ, и разрешение выполняется путем выбора (при необходимости) нового множества инстанцируемых переменных, как описано выше.

В противном случае результат содержит границу *false*, так что предпринимается вторая попытка инстанцировать $\{\alpha_1,...,\alpha_n\}$ с помощью описанных ниже действий.

- Если множество границ содержит границу вида $G<...,\alpha_i,...>$ = capture($G<...>$) для некоторого i ($1 \le i \le n$);

или если множество границ, полученные на описанном выше шаге содержит границу *false*,

то пусть $Y_1,...,Y_n$ представляют собой новые переменные типа, границы которых определяются следующим образом.

- ✦ Для всех i ($1 \le i \le n$), если α_i имеет одну или более *истинных* нижних границ $L_1,...,L_k$, нижней границей Y_i является lub($L_1,...,L_k$); если нет, то Y_i нижней границы не имеет.
- ✦ Для всех i ($1 \le i \le n$), если α_i имеет верхние границы $U_1,...,U_k$, верхней границей Y_i является glb($U_1\,\theta$, ..., $U_k\,\theta$), где θ представляет собой подстановку $[\alpha_1:=Y_1,...,\alpha_n:=Y_n]$.

Если переменные типов $Y_1,...,Y_n$ не имеют корректно сформированных границ (т.е. нижняя граница не является подтипом верхней границы или тип пересечения является несовместимым), то разрешение завершается неудачно.

В противном случае для всех i ($1 \le i \le n$) все границы вида $G<...,\alpha_i,...>$ = capture($G<...>$) удаляются из текущего множества границ, а границы $a_1 = Y_1,...,a_n = Y_n$ объединяются с ним.

Если результат не содержит границу *false*, то результатом становится новое множество границ, и разрешение выполняется путем выбора (при необходимости) нового множества инстанцируемых переменных, как описано выше.

В противном случае результат содержит границу *false*, и разрешение завершается неудачно.

> Первый способ инстанцирования переменной вывода порождает инстанцирование из границ этой переменной. Однако иногда сложные зависимости означают, что результат не будет находиться в пределах границ переменной. В таком случае используется другой метод инстанцирования, аналогичный преобразованию при фиксации (§5.1.10): вводятся новые переменные вывода с границами, полученными из границ переменных вывода. Обратите внимание, что нижние границы этих "фиксированных" переменных вычисляются с использованием только истинных типов: это важно для того, чтобы избежать попыток выполнения вычисления типов над неинстанцированными переменными типов.

§18.5. Использование вывода

Во время компиляции выполняется описанный далее анализ с использованием процесса вывода, изложенного выше.

§18.5.1. Вывод применимости вызова

Для данного вызова метода, не предоставляющего явные аргументы типа, процесс определения потенциальной применимости обобщенного метода m осуществляется следующим образом.

- Пусть $P_1,...,P_p$ ($p \geq 1$) представляют собой параметры типов m; $a_1,...,a_p$ являются переменными вывода, а θ — подстановка $[P_1:=a_1,...,P_p:=a_p]$.
- Начальное множество границ B_0 строится из объявленных границ $P_1,...,P_p$, как описано в §18.1.3.
- Для всех i ($1 \leq i \leq p$), если P_i находится в конструкции `throws` метода m, получается граница `throws` a_i. Эти границы, если таковые имеются, объединяются с B_0 и дают новое множество границ B_1.
- Множество формул ограничения C строится следующим образом.

  Пусть $F_1,...,F_n$ являются типами формальных параметров m и пусть $e_1,...,e_k$ являются фактическими выражениями аргументов вызова. Тогда выполняется следующее.

  ✦ Для проверки на *применимость для строгого вызова.*

  Если $k \neq n$, или если существует i ($1 \leq i \leq n$), такое, что e_i подходит для применимости (§15.12.2.2) и либо 1) e_i является автономным выражением примитивного типа, но F_i является типом ссылки, либо 2) F_i является примитивным типом, но e_i не является автономным выражением примитивного типа, то метод не применим и в выводе нет необходимости.

  В противном случае C для всех i ($1 \leq i \leq k$), где e_i подходит для применимости, включает $\langle e_i \rightarrow F_i\ \theta \rangle$.

  ✦ Для проверки на *применимость для нестрогого вызова.*

  Если $k \neq n$, метод не применим и в выводе нет необходимости.

  В противном случае C для всех i ($1 \leq i \leq k$), где e_i подходит для применимости, включает $\langle e_i \rightarrow F_i\ \theta \rangle$.

  ✦ Для проверки на *применимость для вызова переменной арности.*

  Пусть $F_1',...,F_k'$ представляют собой первые k типов параметров переменной арности метода m (§15.12.2.4). C для всех i ($1 \leq i \leq k$), где e_i подходит для применимости, включает $\langle e_i \rightarrow F_i'\ \theta \rangle$.
- C приводится (§18.2), и получающиеся в результате границы объединяются с B_1 и дают новое множество границ B_2.
- Наконец метод m примен*и*м, если B_2 не содержит границу *false* и разрешение всех переменных вывода в B_2 успешно (§18.4).

> Рассмотрим следующий вызов метода и присваивание.
>
> ```
> List<Number> ln = Arrays.asList(1, 2.0);
> ```
>
> Наиболее подходящий применимый метод для вызова должен быть идентифицирован так, как описано в §15.12. Единственный потенциально применимый метод (§15.12.2.1) объявлен следующим образом.

```
public static <T> List<T> asList(T... a)
```

Тривиально (в силу его арности) этот метод не применим ни строгим вызовом (§15.12.2.2), ни нестрогим вызовом (§15.12.2.3). Но поскольку других кандидатов нет, в третьей фазе метод проверяется на применимость вызовом переменной арности.

Исходное множество границ B является тривиальной верхней границей для единственной переменной вывода a.

$\{ a <: \texttt{Object} \}$

Исходное множество формул ограничений имеет следующий вид.

$\{ \langle 1 \rightarrow a \rangle, \langle 2.0 \rightarrow a \rangle \}$

Они приводятся к новому множеству границ B_1.

$\{ a <: \texttt{Object}, \texttt{Integer} <: a, \texttt{Double} <: a \}$

Затем для проверки применимости метода мы пытаемся разрешить эти границы. Это успешно выполняется, давая более сложное инстанцирование.

```
a = Number & Comparable<? extends Number & Comparable<?>>
```

Таким образом, мы показали, что метод примен*и*м; поскольку других кандидатов нет, это наиболее подходящий применимый метод. Тем не менее тип вызова метода и его совместимость с целевым типом в присваивании не определены до тех пор, пока не будет выполнен дальнейший вывод, как описано в следующем разделе.

§18.5.2. Вывод типа вызова

Для данного вызова метода, который не предоставляет явных аргументов типа, и соответствующего наиболее подходящего применимого обобщенного метода m процесс вывода типа вызова (§15.12.2.6) выбранного метода выглядит следующим образом.

- Пусть θ представляет собой подстановку $[P_1:=a_1,...,P_p:=a_p]$, определенную в §18.5.1 для замены параметров типов m переменными вывода.
- Пусть B_2 представляет собой множество границ, полученное приведением для того, чтобы продемонстрировать, что m применимо в §18.5.1. (Хотя в §18.5.1 для установления применимости было необходимо демонстрировать, что переменные вывода в B_2 могут быть разрешены, инстанцирования, полученные на этом шаге разрешения, не рассматриваются как часть B_2.)
- Если вызов не является поливыражением, множество границ B_3 совпадает с B_2.

  Если вызов является поливыражением, множество границ B_3 получается из B_2 следующим образом. Пусть R — возвращаемый тип m, а T — целевой тип вызова.

  - Если для того, чтобы метод был применим, в процессе приведения множества ограничений в §18.5.1 было необходимо непроверяемое преобразование, приводится и объединяется с B_2 формула ограничения $\langle |R| \rightarrow T \rangle$.
  - В противном случае, если R θ является параметризованным типом $G<A_1,...,A_n>$ и один из аргументов $A_1,...,A_n$ является символом подстановки, для новых переменных вывода $\beta_1,...,\beta_n$ приводится и объединяется с B_2 формула ограничения $\langle G<\beta_1,...,\beta_n> \rightarrow T \rangle$ вместе с границей $G<\beta_1,...,\beta_n> = \text{capture}(G<A_1,...,A_n>)$.

- ✦ В противном случае, если $R\,\theta$ представляет собой переменную вывода α и истинно одно из перечисленных ниже условий, α разрешается в B_2 и, если фиксация результирующего инстанцирования α представляет собой U, приводится и объединяется с B_2 формула ограничения $\langle U \rightarrow T \rangle$.
  - — T является типом ссылки, но не типом, параметризованным символами подстановки, и либо 1) B_2 содержит границу вида $\alpha = S$ или $S <: \alpha$, где S является параметризованным символом подстановки типом, или 2) B_2 содержит две границы вида $S_1 <: \alpha$ и $S_2 <: \alpha$, где S_1 и S_2 имеют супертипы, которые представляют собой две различные параметризации одного и того же обобщенного класса или интерфейса.
  - — T является параметризацией обобщенного класса или интерфейса G, B_2 содержит границу вида $\alpha = S$ или $S <: \alpha$ и при этом не имеется типа вида $G<...>$, который является супертипом S, но несформированный тип $|G<...>|$ является супертипом S.
  - — T является примитивным типом, и один из примитивных классов-оболочек, упоминающихся в §5.1.7, является инстанцированием, верхней границей или нижней границей для α в B_2.
- ✦ В противном случае приводится и объединяется с B_2 формула ограничения $\langle R\,\theta \rightarrow T \rangle$.

- • Множество формул ограничений C строится следующим образом.

Пусть $e_1,...,e_k$ представляют собой фактические выражения аргументов вызова. Если m примен*и*м строгим или нестрогим вызовом, пусть $F_1,...,F_k$ являются типами формальных параметров m; если же m применим вызовом переменной арности, пусть $F_1,...,F_k$ представляют собой k типов параметров переменной арности m (§15.12.2.4).

- ✦ Для всех i $(1 \le i \le k)$, если e_i не имеет отношения к применимости, C содержит $\langle e_i \rightarrow F_i\,\theta \rangle$.
- ✦ Для всех i $(1 \le i \le k)$ могут быть включены дополнительные ограничения в зависимости от вида e_i.
  - — Если e_i представляет собой *LambdaExpression*, C содержит $\langle$*LambdaExpression* $\rightarrow_{throws} Fi\,\theta \rangle$.
  - — Если e_i представляет собой *MethodReference*, C содержит $\langle$*MethodReference* $\rightarrow_{throws} F_i\,\theta \rangle$.
  - — Если e_i является поливыражением создания экземпляра класса (§15.9) или поливыражением вызова метода (§15.12), C содержит все формулы ограничений, которые появляются в множестве C, сгенерированном согласно §18.5.2 при выводе типа вызова поливыражения.
  - — Если e_i представляет собой параметризованное выражение, эти правила применяются рекурсивно к содержащемуся в нем выражению.
  - — Если e_i представляет собой условное выражение, эти правила применяются рекурсивно ко второму и третьему операндам.

- Пока C не станет пустым, приведенный ниже процесс повторяется, начиная с множества границ B_3 и накапливая новые границы в "текущем" множестве границ, в конечном итоге приводя к новому множеству границ B_4.

  1. В C выбирается подмножество ограничений, удовлетворяющих тому свойству, что для каждого ограничения никакая входная переменная не зависит от разрешения (§18.4) выходной переменной другого ограничения в C. (*Входная переменная* и *выходная переменная* определены ниже.)

     Если это подмножество пустое, то имеется цикл (или циклы) в графе зависимостей между ограничениями. В этом случае все ограничения рассматриваются как участвующие в цикле (или циклах) и не зависящие от каких-либо ограничений вне цикла (или циклов). Единственное ограничение выбирается из рассматриваемых ограничений следующим образом.

     - Если любое из рассматриваемых ограничений имеет вид ‹*Expression* → T›, то выбранное ограничение является рассматриваемым ограничением данного вида, которое содержит выражение слева (§3.5) от выражения каждого другого рассматриваемого ограничения этого вида.
     - Если рассматриваемого выражения вида ‹*Expression* → T› нет, то выбранное ограничение является рассматриваемым ограничением, которое содержит выражение слева от выражения каждого другого рассматриваемого ограничения.

  2. Выбранное ограничение (или ограничения) удаляется из C.
  3. Выполняется разрешение входных переменных $\alpha_1,...,\alpha_m$ всех выбранных ограничений.
  4. Если $T_1,...,T_m$ являются инстанцированиями $\alpha_1,...,\alpha_m$, к каждому ограничению применяется подстановка $[\alpha_1:=T_1,...,\alpha_m:=T_m]$.
  5. Ограничение или ограничения, получающиеся из подстановки, приводятся и объединяются с текущим множеством границ.

- Наконец, если B_4 не содержит границу *false*, разрешаются переменные вывода в B_4.

  Если разрешение с инстанцированиями $T_1,...,T_p$ для переменных вывода $\alpha_1,...,\alpha_p$ успешно, то путь q' представляет собой подстановку $[P_1:=T_1,...,P_p:=T_p]$.

  - Если для того, чтобы метод был применим, в процессе приведения множества ограничений в §18.5.1 необходимо непроверяемое преобразование, то типы параметров типа вызова m получаются с помощью применения q' к типам параметров типа m, а возвращаемый тип и типы исключений типа вызова m получаются путем затирания возвращаемого типа и типов исключений типа метода m.
  - Если для того, чтобы метод был применим, непроверяемое преобразование не является необходимым, то тип вызова m получается путем применения q' к типу метода m.

  Если B_4 содержит границу *false* или если разрешение завершается неудачно, то генерируется ошибка времени компиляции.

Вывод типа вызова может потребовать тщательного упорядочения приведений формул ограничений вида ‹*Expression* → *T*›, ‹*LambdaExpression* →$_{throws}$ *T*› и ‹*MethodReference* →$_{throws}$ *T*›. Для облегчения этого упорядочения *входные переменные* этих ограничений определяются следующим образом.

- Для ‹*LambdaExpression* → *T*›
  - ✦ Если *T* является переменной вывода, это (единственная) входная переменная.
  - ✦ Если *T* является типом функционального интерфейса и если тип функции может быть порожден из *T* (§15.27.3), то входные переменные включают: 1) если лямбда-выражение неявно типизировано — переменные вывода, упоминаемые типами параметров типа функции; и 2) если возвращаемый тип *R* типа функции не является `void`, то для каждого выражения возврата `e` в теле лямбда-выражения (или для самого тела, если оно является выражением) — входные переменные ограничения ‹`e` → *R*›.
  - ✦ В противном случае входных переменных нет.
- Для ‹*LambdaExpression* →$_{throws}$ *T*›
  - ✦ Если *T* является переменной вывода, это (единственная) входная переменная.
  - ✦ Если *T* является типом функционального интерфейса и тип функции может быть выведен, как описано в §15.27.3, то входные переменные включают: 1) если лямбда-выражение неявно типизировано, переменные вывода, упоминаемые типами параметров типа функции; и 2) переменные вывода, упоминаемые в возвращаемом типе типа функции.
  - ✦ В противном случае входных переменных нет.
- Для ‹*MethodReference* → *T*›
  - ✦ Если *T* является переменной вывода, это (единственная) входная переменная.
  - ✦ Если *T* является типом функционального интерфейса с типом функции и если ссылка на метод является неточной (§15.13.1), входные переменные представляют собой переменные вывода, упоминаемые типами параметров типа функции.
  - ✦ В противном случае входных переменных нет.
- Для ‹*MethodReference* →$_{throws}$ *T*›
  - ✦ Если *T* является переменной вывода, это (единственная) входная переменная.
  - ✦ Если *T* является типом функционального интерфейса с типом функции и если ссылка на метод является неточной (§15.13.1), входные переменные представляют собой переменные вывода, упоминаемые типами параметров типа функции и возвращаемым типом типа функции.
  - ✦ В противном случае входных переменных нет.
- Для ‹*Expression* → *T*›, если *Expression* является выражением в скобках

  Если выражение, содержащееся в *Expression*, представляет собой *Expression'*, то входные переменные являются входными переменными ‹*Expression'* → *T*›.
- Для ‹*ConditionalExpression* → *T*›

Если условное выражение имеет вид e_1 ? e_2 : e_3, входные переменные представляют собой входные переменные ‹$e_2 \rightarrow T$› и ‹$e_3 \rightarrow T$›.

- Для всех прочих формул ограничений входных переменных нет

Выходные переменные этих ограничений представляют собой все переменные вывода, упоминаемые типом правой части ограничения, T, которые не являются входными переменными.

Важно отметить, что поиск типа вызова метода выполняется в два “раунда”. Это необходимо для того, чтобы позволить целевому типу влиять на тип вызова, не влияя на выбор применимого метода. В первом раунде генерируется множество границ и выполняется проверка существования разрешения, но результат не фиксируется. Во втором раунде выполняется приведение дополнительных ограничений, а затем — второе разрешение, на этот раз “реальное”.

Рассмотрим пример из предыдущего раздела.

```
List<Number> ln = Arrays.asList(1, 2.0);
```

Наиболее подходящий применимый метод идентифицируется как

```
public static <T> List<T> asList(T... a)
```

Для того чтобы завершить проверку типов вызова метода, мы должны определить, совместим ли он с целевым типом `List<Number>`.

Множество границ, использованное для демонстрации применимости в предыдущем разделе, B_2, имело следующий вид.

{ a <: `Object`, `Integer` <: a, `Double` <: a }

Новое множество формул ограничений имеет такой вид.

{ ‹`List<`a`>` → `List<Number>`› }

Это ограничение совместимости дает границу равенства для a, которая включается в новое множество границ B_3.

{ a <: `Object`, `Integer` <: a, `Double` <: a, a = `Number` }

Эти границы тривиально разрешаются.

a = `Number`

Наконец мы выполняем подстановку объявленного возвращаемого типа `asList` для определения того, что вызов метода имеет тип `List<Number>`; очевидно, он совместим с целевым типом.

Эта стратегия вывода отличается от стратегии вывода в версии Java SE 7 Edition, в которой инстанцирование a основывается на ее нижних границах (до рассмотрения целевого типа вызова), как мы поступали в предыдущем разделе. Это приводит к ошибке типа, так как результирующий тип не является подтипом `List<Number>`.

При различных особых обстоятельствах, основанных на границах из B_2, мы сразу разрешаем переменную вывода, которая появляется в качестве возвращаемого типа вызова. Это позволяет избежать неудачных ситуаций, в которых обычное ограничение ‹$R\ \theta \rightarrow T$› не является сохраняющим полноту. К сожалению, возможно, что при раннем разрешении переменной мы не сможем воспользоваться границами, которые будут выведены позже. Возможно также, что в некоторых случаях границы, которые позже будут выведены из аргументов вызова (как в случае неявно

типизированных лямбда-выражений), привели бы к другому результату, если бы они присутствовали в B_2. Несмотря на эти ограничения данная стратегия позволяет нам получить разумные результаты в типичных ситуациях, и при этом она обратно совместима с алгоритмом из Java SE 7 Edition.

§18.5.3. Вывод параметризации функционального интерфейса

Если для лямбда-выражения с явными типами параметров $P_1,...,P_n$ целевым является тип функционального интерфейса $F<A_1,...,A_m>$ как минимум с одним аргументом типа, являющимся символом подстановки, то параметризация F может быть получена как базовый целевой тип лямбда-выражения следующим образом.

Пусть $Q_1,...,Q_k$ представляют собой типы параметров типа функции типа $F<\alpha_1,...,\alpha_m>$, где $\alpha_1,...,\alpha_m$ — новые переменные вывода.

Если $n \neq k$, корректной параметризации не существует. В противном случае формируется множество формул ограничений из ограничений $\langle P_i = \Theta_i \rangle$ для всех i ($1 \leq i \leq n$). Это множество формул ограничений приводится к виду множества границ B.

Если B содержит границу *false*, корректной параметризации не существует. В противном случае строится новая параметризация типа функционального интерфейса, $F<A'_1,..., A'_m>$. Для этого для $1 \leq i \leq m$ имеем:

- если B содержит инстанцирование a_i, которое представляет собой T, то $A'_i = T$;
- в противном случае $A'_i = A_i$.

Если $F<A'_1,...,A'_m>$ не является корректно сформированным типом (т.е. аргументы типа не находятся в пределах своих границ) или если $F<A'_1,...,A'_m>$ не является подтипом $F<A_1,...,A_m>$, корректной параметризации не существует. В противном случае выведенная параметризация либо представляет собой $F<A'_1,...,A'_m>$, если все аргументы типа являются типами, либо является параметризацией без символов подстановки (§9.8) для $F<A'_1,..., A'_m>$, если один или более аргументов типа по-прежнему являются символами подстановки.

Для того чтобы определить тип функции функционального интерфейса, параметризованного символами подстановки, мы должны "инстанцировать" аргументы типа с символами подстановки конкретными (определенными) типами. Подход "по умолчанию" заключается в простой замене символов подстановки их границами, как описано в §9.8, но при этом получаются ложные ошибки в случаях, когда лямбда-выражение имеет явные типы параметров, которые *не* соответствуют границам символов подстановки, как, например, показано ниже.

```
Predicate<? super Integer> p = (Number n) -> n.equals(23);
```

Тип лямбда-выражения представляет собой `Predicate<Number>`, что является подтипом `Predicate<? super Integer>`, но не `Predicate<Integer>`. Применение анализа из данного раздела приводит к выводу, что `Number` является подходящим выбором для аргумента типа для `Predicate`.

Впрочем, проведенный здесь анализ, будучи описанным в терминах общего вывода типа, преднамеренно упрощен. Единственными ограничениями являются ограничения эквивалентности, а это означает, что приведение сводится к простому соответствию шаблонов. Более мощная стратегия также может вывести ограничения

> из тела лямбда-выражения. Но с учетом возможного взаимодействия с выводом окружающих и/или вложенных вызовов обобщенных методов это может привести к большому количеству излишних сложностей.

§18.5.4. Вывод более подходящего метода

При проверке того, что один применимый метод является *более подходящим*, чем другой (§15.12.2.5), где второй метод является обобщенным, необходимо проверить, может ли быть выведено некоторое инстанцирование параметров типа второго метода, которое делает первый метод более подходящим, чем второй.

Пусть m_1 представляет собой первый метод, а m_2 — второй. Если m_2 имеет параметры типов $P_1,...,P_p$, обозначим переменные вывода как $\alpha_1,...,\alpha_p$, а как θ — подстановку $[P_1:=\alpha_1,...,P_p:=\alpha_p]$.

Пусть $e_1,...,e_k$ представляют собой выражения аргумента соответствующего вызова.

- Если m_1 и m_2 являются применимыми строгим или нестрогим вызовом (§15.12.2.2, §15.12.2.3), то пусть $S_1,...,S_k$ являются типами формальных параметров m_1, а $T_1,...,T_k$ — результатом применения θ к типам формальных параметров m_2.
- Если m_1 и m_2 являются применимыми вызовом переменной арности (§15.12.2.4), то пусть $S_1,...,S_k$ являются первыми k типами параметров переменной арности m_1, а $T_1,...,T_k$ — результатом применения θ к первым k типам параметров переменной арности m_2.

> Обратите внимание, что к $S_1,...,S_k$ не применяется никакая подстановка; даже если m_1 является обобщенным, параметры типа m_1 рассматриваются как переменные типа, а не переменные вывода.

Процесс определения того, что m_1 является более подходящим, чем m_2, выглядит следующим образом.

- Во-первых, из объявленных границ $P_1,...,P_p$ строится начальное множество границ B, как указано в §18.1.3.
- Во-вторых, для всех i ($1 \le i \le k$) генерируется множество формул ограничения.

  Если T_i является истинным типом, то результат представляет собой *true*, если S_i более подходящий, чем T_i, для e_i (§15.12.2.5), и *false* в противном случае. (Обратите внимание, что S_i всегда является истинным типом.)

  В противном случае, если T_i не является типом функционального интерфейса, генерируется формула ограничения $\langle S_i <: T_i \rangle$.

  В противном случае T_i является параметризацией функционального интерфейса I. Следует определить, удовлетворяет ли S_i следующим пяти ограничениям.

  - S_i является типом функционального интерфейса.
  - S_i не является ни суперинтерфейсом I, ни параметризацией суперинтерфейса I.
  - S_i не является ни подынтерфейсом I, ни параметризацией подынтерфейса I.
  - Если S_i является типом пересечения, как минимум один элемент пересечения не является ни суперинтерфейсом I, ни параметризацией суперинтерфейса I.

- ✦ Если S_i является типом пересечения, ни один элемент пересечения не является ни подынтерфейсом I, ни параметризацией подынтерфейса I.

Если все перечисленное выше истинно, то генерируются следующие формулы ограничений или границы (где U_1 ... U_k и R_1 являются типами параметров и типом возврата типа функции фиксации S_i, а V_1 ... V_k и R_2 — типами параметров и типом возврата типа функции T_i).

- ✦ Если e_i является явно типизированным лямбда-выражением
  - — Если R_2 является `void`, генерируется *true*.
  - — В противном случае, если R_1 и R_2 являются типами функционального интерфейса и если ни один интерфейс не является подынтерфейсом другого, эти правила рекурсивно применяются к R_1 и R_2, для каждого выражения результата в e_i.
  - — В противном случае, если R_1 является примитивным типом, а R_2 — нет и если каждое выражение результата e_i представляет собой автономное выражение (§15.2) примитивного типа, генерируется *true*.
  - — В противном случае, если R_2 является примитивным типом, а R_1 — нет и если каждое выражение результата e_i является либо автономным выражением ссылочного типа или поливыражением, генерируется *true*.
  - — В противном случае генерируется $\langle R_1 <: R_2 \rangle$.
- ✦ Если e_i является точной ссылкой на метод
  - — Для всех j $(1 \le j \le k)$ генерируется $\langle U_j = V_j \rangle$.
  - — Если R_2 является `void`, генерируется *true*.
  - — В противном случае, если R_1 является примитивным типом, а R_2 — нет и если объявление времени компиляции для e_i имеет примитивный тип возврата, генерируется *true*.
  - — В противном случае, если R_2 является примитивным типом, а R_1 — нет и если объявление времени компиляции для e_i имеет ссылочный тип возврата, генерируется *true*.
  - — В противном случае генерируется $\langle R_1 <: R_2 \rangle$.
- ✦ Если e_i является выражением в скобках, эти правила применяются рекурсивно к содержащемуся в скобках выражению.
- ✦ Если e_i является условным выражением, эти правила применяются рекурсивно ко второму и третьему операндам.
- ✦ В противном случае генерируется *false*.

Если пять ограничений на S_i не удовлетворяются, генерируется формула ограничения $\langle S_i <: T_i \rangle$.

- • В-третьих, если m_2 применим вызовом переменной арности и имеет $k+1$ параметров, то, если S_{k+1} является $k+1$-м типом параметра переменной арности m_1, а T_{k+1} является результатом применения θ к $k+1$-му типу параметра переменной арности m_2, генерируется ограничение $\langle S_{k+1} <: T_{k+1} \rangle$.

- В-четвертых, генерируемые границы и формулы ограничений приводятся и объединяются с B, образуя множество границ B'.

  Если B' не содержит границу *false*, а разрешение всех переменных вывода в B' успешно, то m_1 более подходящий, чем m_2.

  В противном случае m_1 не более подходящий, чем m_2.

ГЛАВА 19

Синтаксис

В этой главе повторяются синтаксическая грамматика языка программирования Java, приведенная в главах 4, 6–11, 14 и 15, а также ключевые части лексической грамматики из главы 3 с применением обозначений из §2.4.

Продукции из главы 3, “Лексическая структура”

Identifier:
IdentifierChars, но не *Keyword, BooleanLiteral* или *NullLiteral*

IdentifierChars:
JavaLetter {JavaLetterOrDigit}

JavaLetter:
любой символ Unicode, являющийся “буквой Java”

JavaLetterOrDigit:
любой символ Unicode, являющийся “буквой или цифрой Java”

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

Продукции из главы 4, “Типы, значения и переменные”

Type:
PrimitiveType
ReferenceType

PrimitiveType:
{Annotation} NumericType

{Annotation} `boolean`

NumericType:
IntegralType
FloatingPointType

IntegralType: одно из
`byte short int long char`

FloatingPointType: одно из
`float double`

ReferenceType:
ClassOrInterfaceType
TypeVariable
ArrayType

ClassOrInterfaceType:
ClassType
InterfaceType

ClassType:
{Annotation} Identifier [TypeArguments]
ClassOrInterfaceType `.` *{Annotation} Identifier [TypeArguments]*

InterfaceType:
ClassType

TypeVariable:
{Annotation} Identifier

ArrayType:
PrimitiveType Dims
ClassOrInterfaceType Dims
TypeVariable Dims

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

TypeParameter:
{TypeParameterModifier} Identifier [TypeBound]

TypeParameterModifier:
Annotation

TypeBound:
`extends` *TypeVariable*

`extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
`&` *InterfaceType*

TypeArguments:
`<` *TypeArgumentList* `>`

TypeArgumentList:
TypeArgument {`,` *TypeArgument*}

TypeArgument:
ReferenceType
Wildcard

Wildcard:
{*Annotation*} `?` [*WildcardBounds*]

WildcardBounds:
`extends` *ReferenceType*
`super` *ReferenceType*

Продукции из главы 6, “Имена”

TypeName:
Identifier
PackageOrTypeName `.` *Identifier*

PackageOrTypeName:
Identifier
PackageOrTypeName `.` *Identifier*

ExpressionName:
Identifier
AmbiguousName `.` *Identifier*

MethodName:
Identifier

PackageName:
Identifier
PackageName `.` *Identifier*

AmbiguousName:
Identifier
AmbiguousName `.` *Identifier*

Продукции из главы 7, “Пакеты”

CompilationUnit:
[PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

PackageDeclaration:
{PackageModifier} `package` *Identifier* `{ .` *Identifier}* `;`

PackageModifier:
Annotation

ImportDeclaration:
SingleTypeImportDeclaration
TypeImportOnDemandDeclaration
SingleStaticImportDeclaration
StaticImportOnDemandDeclaration

SingleTypeImportDeclaration:
`import` *TypeName* `;`

TypeImportOnDemandDeclaration:
`import` *PackageOrTypeName* `. * ;`

SingleStaticImportDeclaration:
`import static` *TypeName* `.` *Identifier* `;`

StaticImportOnDemandDeclaration:
`import static` *TypeName* `. * ;`

TypeDeclaration:
ClassDeclaration
InterfaceDeclaration
`;`

Продукции из главы 8, “Классы”

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration

NormalClassDeclaration:
{ClassModifier} `class` *Identifier [TypeParameters]*
[Superclass] [Superinterfaces] ClassBody

ClassModifier: одно из
Annotation `public protected private`

```
abstract static final strictfp
```

TypeParameters:
< *TypeParameterList* >

TypeParameterList:
TypeParameter {, *TypeParameter*}

Superclass:
`extends` *ClassType*

Superinterfaces:
`implements` *InterfaceTypeList*

InterfaceTypeList:
InterfaceType {, *InterfaceType*}

ClassBody:
{ {*ClassBodyDeclaration*} }

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

FieldDeclaration:
{*FieldModifier*} *UnannType VariableDeclaratorList* ;

FieldModifier: одно из
Annotation `public protected private`
`static final transient volatile`

VariableDeclaratorList:
VariableDeclarator {, *VariableDeclarator*}

VariableDeclarator:
VariableDeclaratorId [= *VariableInitializer*]

VariableDeclaratorId:

Identifier [Dims]

VariableInitializer:
Expression
ArrayInitializer

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
`boolean`

UnannReferenceType:
UnannClassOrInterfaceType
UnannTypeVariable
UnannArrayType

UnannClassOrInterfaceType:
UnannClassType
UnannInterfaceType

UnannClassType:
Identifier [TypeArguments]
UnannClassOrInterfaceType . {Annotation} Identifier [TypeArguments]

UnannInterfaceType:
UnannClassType

UnannTypeVariable:
Identifier

UnannArrayType:
UnannPrimitiveType Dims
UnannClassOrInterfaceType Dims
UnannTypeVariable Dims

MethodDeclaration:
{MethodModifier} MethodHeader MethodBody

MethodModifier: одно из
Annotation `public protected private`
`abstract static final synchronized native strictfp`

MethodHeader:

Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

Result:
UnannType
`void`

MethodDeclarator:
Identifier (*[FormalParameterList]*) *[Dims]*

FormalParameterList:
FormalParameters , *LastFormalParameter*
LastFormalParameter

FormalParameters:
FormalParameter {, *FormalParameter*}
ReceiverParameter {, *FormalParameter*}

FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId

VariableModifier: одно из
Annotation `final`

LastFormalParameter:
{VariableModifier} UnannType {Annotation} `...` *VariableDeclaratorId*
FormalParameter

ReceiverParameter:
{Annotation} UnannType [Identifier .] `this`

Throws:
`throws` *ExceptionTypeList*

ExceptionTypeList:
ExceptionType {, *ExceptionType*}

ExceptionType:
ClassType
TypeVariable

MethodBody:
Block
;

InstanceInitializer:
Block

StaticInitializer:
`static` *Block*

ConstructorDeclaration:
{ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

ConstructorModifier: одно из
Annotation `public protected private`

ConstructorDeclarator:
[TypeParameters] SimpleTypeName `(` *[FormalParameterList]* `)`

SimpleTypeName:
Identifier

ConstructorBody:
`{` *[ExplicitConstructorInvocation] [BlockStatements]* `}`

ExplicitConstructorInvocation:
[TypeArguments] `this (` *[ArgumentList]* `) ;`
[TypeArguments] `super (` *[ArgumentList]* `) ;`
ExpressionName `.` *[TypeArguments]* `super (` *[ArgumentList]* `) ;`
Primary `.` *[TypeArguments]* `super (` *[ArgumentList]* `) ;`

EnumDeclaration:
{ClassModifier} `enum` *Identifier [Superinterfaces] EnumBody*

EnumBody:
`{` *[EnumConstantList] [,] [EnumBodyDeclarations]* `}`

EnumConstantList:
EnumConstant {, *EnumConstant}*

EnumConstant:
{EnumConstantModifier} Identifier [(*[ArgumentList]* *)] [ClassBody]*

EnumConstantModifier:
Annotation

EnumBodyDeclarations:
`;` *{ClassBodyDeclaration}*

Продукции из главы 9, “Интерфейсы”

InterfaceDeclaration:
NormalInterfaceDeclaration

AnnotationTypeDeclaration

NormalInterfaceDeclaration:
{InterfaceModifier} `interface` *Identifier [TypeParameters]*
[ExtendsInterfaces] InterfaceBody

InterfaceModifier: одно из
Annotation `public protected private`
`abstract static strictfp`

ExtendsInterfaces:
`extends` *InterfaceTypeList*

InterfaceBody:
`{` *{InterfaceMemberDeclaration}* `}`

InterfaceMemberDeclaration:
ConstantDeclaration
InterfaceMethodDeclaration
ClassDeclaration
InterfaceDeclaration
`;`

ConstantDeclaration:
{ConstantModifier} UnannType VariableDeclaratorList `;`

ConstantModifier: одно из
Annotation `public`
`static final`

InterfaceMethodDeclaration:
{InterfaceMethodModifier} MethodHeader MethodBody

InterfaceMethodModifier: одно из
Annotation `public`
`abstract default static strictfp`

AnnotationTypeDeclaration:
{InterfaceModifier} `@ interface` *Identifier AnnotationTypeBody*

AnnotationTypeBody:
`{` *{AnnotationTypeMemberDeclaration}* `}`

AnnotationTypeMemberDeclaration:
AnnotationTypeElementDeclaration
ConstantDeclaration

ClassDeclaration
InterfaceDeclaration
;

AnnotationTypeElementDeclaration:
{AnnotationTypeElementModifier} UnannType Identifier ( ) *[Dims]*
[DefaultValue] ;

AnnotationTypeElementModifier: одно из
Annotation `public`
`abstract`

DefaultValue:
`default` *ElementValue*

Annotation:
NormalAnnotation
MarkerAnnotation
SingleElementAnnotation

NormalAnnotation:
@ *TypeName* (*[ElementValuePairList]*)

ElementValuePairList:
ElementValuePair {, *ElementValuePair}*

ElementValuePair:
Identifier = *ElementValue*

ElementValue:
ConditionalExpression
ElementValueArrayInitializer
Annotation

ElementValueArrayInitializer:
{ *[ElementValueList]* [,] }

ElementValueList:
ElementValue {, *ElementValue}*

MarkerAnnotation:
@ *TypeName*

SingleElementAnnotation:
@ *TypeName* (*ElementValue*)

Продукции из главы 10, "Массивы"

ArrayInitializer:
{ *[VariableInitializerList]* *[,]* }

VariableInitializerList:
VariableInitializer {, VariableInitializer}

Продукции из главы 14, "Блоки и инструкции"

Block:
{ *[BlockStatements]* }

BlockStatements:
BlockStatement {BlockStatement}

BlockStatement:
LocalVariableDeclarationStatement
ClassDeclaration
Statement

LocalVariableDeclarationStatement:
LocalVariableDeclaration ;

LocalVariableDeclaration:
{VariableModifier} UnannType VariableDeclaratorList

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement

AssertStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement

EmptyStatement:
;

LabeledStatement:
Identifier : *Statement*

LabeledStatementNoShortIf:
Identifier : *StatementNoShortIf*

ExpressionStatement:
StatementExpression ;

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

IfThenStatement:
`if` (*Expression*) *Statement*

IfThenElseStatement:
`if` (*Expression*) *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if` (*Expression*) *StatementNoShortIf* `else` *StatementNoShortIf*

AssertStatement:
`assert` *Expression* ;
`assert` *Expression* : *Expression* ;

SwitchStatement:
`switch` (*Expression*) *SwitchBlock*

SwitchBlock:
{ *{SwitchBlockStatementGroup} {SwitchLabel}* }

SwitchBlockStatementGroup:
SwitchLabels BlockStatements

SwitchLabels:
SwitchLabel {SwitchLabel}

SwitchLabel:
`case` *ConstantExpression* `:`
`case` *EnumConstantName* `:`
`default :`

EnumConstantName:
Identifier

WhileStatement:
`while (` *Expression* `)` *Statement*

WhileStatementNoShortIf:
`while (` *Expression* `)` *StatementNoShortIf*

DoStatement:
`do` *Statement* `while (` *Expression* `) ;`

ForStatement:
BasicForStatement
EnhancedForStatement

ForStatementNoShortIf:
BasicForStatementNoShortIf
EnhancedForStatementNoShortIf

BasicForStatement:
`for (` *[ForInit] ; [Expression] ; [ForUpdate]* `)` `Statement`

BasicForStatementNoShortIf:
`for (` *[ForInit] ; [Expression] ; [ForUpdate]* `)` *StatementNoShortIf*

ForInit:
StatementExpressionList
LocalVariableDeclaration

ForUpdate:
StatementExpressionList

StatementExpressionList:
    StatementExpression {`,` *StatementExpression*}

EnhancedForStatement:
    `for` `(` {*VariableModifier*} *UnannType VariableDeclaratorId*
        `:` *Expression* `)` *Statement*

EnhancedForStatementNoShortIf:
    for `(` {*VariableModifier*} *UnannType VariableDeclaratorId*
        `:` *Expression* `)` *StatementNoShortIf*

BreakStatement:
    `break` [*Identifier*] `;`

ContinueStatement:
    `continue` [*Identifier*] `;`

ReturnStatement:
    `return` [*Expression*] `;`

ThrowStatement:
    `throw` *Expression* `;`

SynchronizedStatement:
    `synchronized` `(` *Expression* `)` *Block*

TryStatement:
    `try` *Block Catches*
    `try` *Block* [*Catches*] *Finally*
    TryWithResourcesStatement

Catches:
    CatchClause {*CatchClause*}

CatchClause:
    `catch` `(` *CatchFormalParameter* `)` *Block*

CatchFormalParameter:
    {*VariableModifier*} *CatchType VariableDeclaratorId*

CatchType:
    UnannClassType {`|` *ClassType*}

Finally:
    `finally` *Block*

TryWithResourcesStatement:
`try` *ResourceSpecification Block [Catches] [Finally]*

ResourceSpecification:
`(` *ResourceList [;]* `)`

ResourceList:
Resource {; Resource}

Resource:
{VariableModifier} UnannType VariableDeclaratorId = Expression

Продукции из главы 15, “Выражения”

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
ClassLiteral
`this`
TypeName `.` `this`
`(` *Expression* `)`
ClassInstanceCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

ClassLiteral:
TypeName {`[ ]`*}* `.` `class`
NumericType {`[ ]`*}* `.` `class`
boolean {`[ ]`*}* `.` `class`
`void` `.` `class`

ClassInstanceCreationExpression:
UnqualifiedClassInstanceCreationExpression
ExpressionName `.` *UnqualifiedClassInstanceCreationExpression*
Primary `.` *UnqualifiedClassInstanceCreationExpression*

UnqualifiedClassInstanceCreationExpression:
`new` *[TypeArguments]*
ClassOrInterfaceTypeToInstantiate `(` *[ArgumentList]* `)` *[ClassBody]*

ClassOrInterfaceTypeToInstantiate:

{Annotation} Identifier { . {Annotation} Identifier}
[TypeArgumentsOrDiamond]

TypeArgumentsOrDiamond:
TypeArguments
<>

FieldAccess:
Primary . *Identifier*
`super` . *Identifier*
TypeName . `super` . *Identifier*

ArrayAccess:
ExpressionName [*Expression*]
PrimaryNoNewArray [*Expression*]

MethodInvocation:
MethodName (*[ArgumentList]*)
TypeName . *[TypeArguments] Identifier* (*[ArgumentList]*)
ExpressionName . *[TypeArguments] Identifier* (*[ArgumentList]*)
Primary . *[TypeArguments] Identifier* (*[ArgumentList]*)
`super` . *[TypeArguments] Identifier* (*[ArgumentList]*)
TypeName . `super` . *[TypeArguments] Identifier* (*[ArgumentList]*)

ArgumentList:
Expression {, Expression}

MethodReference:
ExpressionName :: *[TypeArguments] Identifier*
ReferenceType :: *[TypeArguments] Identifier*
Primary :: *[TypeArguments] Identifier*
super :: *[TypeArguments] Identifier*
TypeName . `super` :: *[TypeArguments] Identifier*
ClassType :: *[TypeArguments]* `new`
ArrayType :: `new`

ArrayCreationExpression:
`new` *PrimitiveType DimExprs [Dims]*
`new` *ClassOrInterfaceType DimExprs [Dims]*
`new` *PrimitiveType Dims ArrayInitializer*
`new` *ClassOrInterfaceType Dims ArrayInitializer*

DimExprs:
DimExpr {DimExpr}

DimExpr:

{Annotation} [*Expression*]

Expression:
LambdaExpression
AssignmentExpression

LambdaExpression:
LambdaParameters -> *LambdaBody*

LambdaParameters:
Identifier
(*[FormalParameterList]*)
(*InferredFormalParameterList*)

InferredFormalParameterList:
Identifier {, Identifier}

LambdaBody:
Expression
Block

AssignmentExpression:
ConditionalExpression
Assignment

Assignment:
LeftHandSide AssignmentOperator Expression

LeftHandSide:
ExpressionName
FieldAccess
ArrayAccess

AssignmentOperator: одно из
= *= /= %= += -= <<= >>= >>>= &= ^= |=

ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression ? *Expression* : *ConditionalExpression*
ConditionalOrExpression ? *Expression* : *LambdaExpression*

ConditionalOrExpression:
ConditionalAndExpression
ConditionalOrExpression || *ConditionalAndExpression*

ConditionalAndExpression:

InclusiveOrExpression
ConditionalAndExpression `&&` *InclusiveOrExpression*

InclusiveOrExpression:
ExclusiveOrExpression
InclusiveOrExpression `|` *ExclusiveOrExpression*

ExclusiveOrExpression:
AndExpression
ExclusiveOrExpression `^` *AndExpression*

AndExpression:
EqualityExpression
AndExpression `&` *EqualityExpression*

EqualityExpression:
RelationalExpression
EqualityExpression `==` *RelationalExpression*
EqualityExpression `!=` *RelationalExpression*

RelationalExpression:
ShiftExpression
RelationalExpression `<` *ShiftExpression*
RelationalExpression `>` *ShiftExpression*
RelationalExpression `<=` *ShiftExpression*
RelationalExpression `>=` *ShiftExpression*
RelationalExpression `instanceof` *ReferenceType*

ShiftExpression:
AdditiveExpression
ShiftExpression `<<` *AdditiveExpression*
ShiftExpression `>>` *AdditiveExpression*
ShiftExpression `>>>` *AdditiveExpression*

AdditiveExpression:
MultiplicativeExpression
AdditiveExpression `+` *MultiplicativeExpression*
AdditiveExpression `-` *MultiplicativeExpression*

MultiplicativeExpression:
UnaryExpression
MultiplicativeExpression `*` *UnaryExpression*
MultiplicativeExpression `/` *UnaryExpression*
MultiplicativeExpression `%` *UnaryExpression*

UnaryExpression:

PreIncrementExpression
PreDecrementExpression
\+ *UnaryExpression*
\- *UnaryExpression*
UnaryExpressionNotPlusMinus

PreIncrementExpression:
++ *UnaryExpression*

PreDecrementExpression:
-- *UnaryExpression*

UnaryExpressionNotPlusMinus:
PostfixExpression
~ *UnaryExpression*
! *UnaryExpression*
CastExpression

PostfixExpression:
Primary
ExpressionName
PostIncrementExpression
PostDecrementExpression

PostIncrementExpression:
PostfixExpression ++

PostDecrementExpression:
PostfixExpression --

CastExpression:
(*PrimitiveType*) *UnaryExpression*
(*ReferenceType {AdditionalBound}*) *UnaryExpressionNotPlusMinus*
(*ReferenceType {AdditionalBound}*) *LambdaExpression*

ConstantExpression:
Expression

Предметный указатель

Эта книга написана разработчиками языка Java и является полным техническим справочником по этому языку программирования. В ней полностью описаны новые возможности, добавленные в Java SE 8, включая лямбда-выражения, ссылки на методы, методы по умолчанию, аннотации типов и повторяющиеся аннотации. В книгу также включено множество поясняющих примечаний. В ней четко обозначены отличия формальных правил языка от практического поведения компиляторов.

Джеймс Гослинг — создатель языка программирования Java и бывший сотрудник Sun Microsystems. Он разработал исходный компилятор Java и виртуальную машину Java, был главой проекта Andrew в университете Карнеги-Меллон, где получил ученую степень в области информатики. С 2011 г. он занимает должность главного архитектора программного обеспечения в компании Liquid Robotics.

Билл Джой — один из основателей компании Sun Microsystems и главный архитектор версии Berkeley UNIX®, за которую и получил пожизненную награду от USENIX Association в 1993 г. Джой играл одну из центральных ролей в формировании языка программирования Java. Он присоединился к KPCB в качестве Greentech Partner в 2005 г.

Гай Л. Стил, мл. — архитектор программного обеспечения в Oracle Labs, где он занимается исследованиями в области стратегий проектирования и реализации языков программирования, параллельных алгоритмов и компьютерной арифметики. Стил является одним из авторов языка программирования Scheme, сотрудником ACM и IEEE, и членом Национальной инженерной академии.

Гилад Брача — создатель языка программирования Newspeak и бывший почетный инженер компании Sun Microsystems. До Sun он работал над языком Strongtalk в компании Animorphic Smalltalk System. Он имеет ученую степень в области информатики, полученную в университете штата Юта.

Алекс Бакли — руководитель группы спецификации языка программирования Java и виртуальной машины Java в Oracle. Он имеет ученую степень в области информатики, полученную в имперском колледже Лондона.

Издательский дом "Вильямс"
http://www.williamspublishing.com

Категория: программирование/Java

ISBN 978-5-8459-1875-8

13065

informit.com/aw
docs.oracle.com/javase/specs

Изображение на обложке: © ssguy/Shutterstock.com

Addison-Wesley

ALWAYS LEARNING

PEARSON