Содержание

Предисловие

Несомненно, новые возможности, появившиеся в Java 8, а особенно лямбда-выражения и Streams API, – гигантский шаг вперед. Вот уже несколько лет я пользуюсь версией Java 8 и рассказываю о новшествах разработчикам на конференциях, семинарах и в своем блоге. И мне совершенно ясно, что хотя лямбда-выражения и потоки привносят в Java функциональный стиль программирования (а заодно позволяют органично использовать возможности распараллеливания), не это привлекает разработчиков, впервые начинающих работать с ними, а то, насколько проще с их помощью становится решать некоторые типы задач и как эти идиомы повышают производительность труда программиста.

Мне как разработчику, лектору и писателю особенно интересно не просто рассказывать об эволюции языка Java, а демонстрировать, как эта эволюция упрощает нашу жизнь – как вновь появившиеся средства позволяют проще решать не только известные, но и совсем новые задачи. И в работе Кена меня подкупает именно это – стремление научить новому, не мусоля детали, которые вам уже известны или не нужны, а сосредоточившись на тех аспектах технологии, которые ценны для программистов-практиков.

Впервые я познакомился с подходом Кена, когда он презентовал свою книгу «Making Java Groovy» на конференции JavaOne. В то время наша команда как раз пыталась написать понятные и полезные тесты, и одним из рассматриваемых вариантов было применение Groovy. Будучи ветераном программирования на Java, я не очень хотел изучать совсем новый язык только для того, чтобы написать тесты, тем более что мне казалось, что уж, как писать тесты, я знаю. Но, послушав, как Кен рассказывает о Groovy для Java-программистов, я узнал много нового и полезного, не отвлекаясь на вещи, которые и так хорошо понимал. И понял, что если подойти к изучению материала правильно, то не придется продираться сквозь дебри языка только для того, чтобы узнать то немногое, что мне действительно необходимо. Я немедленно купил его книгу.

И эта книга о рецептах программирования на современном Java написана с тех же позиций – нам, опытным программистам, нет нужды изучать все возможности Java 8 и 9 так, будто это какой-то новый для нас язык, да и времени на это не хватит. Что нам нужно, так что быстро понять, что появилось интересного, и увидеть примеры реального кода, которые можно было бы применить в своих программах. Именно так и построена книга. На примерах рецептов решения повседневных задач с использованием новых средств Java 8 и 9 автор знакомит нас с языковыми нововведениями самым естественным для нас способом, так чтобы мы могли обогатить свой арсенал.

Прочитав раздел об операторах редукции, я действительно «врубился» в функциональный стиль программирования, не пытаясь перепрограммировать собственные мозги. Рассмотренные возможности Java 9 – это именно то, что полезно нам, разработчикам, и (пока) не очень хорошо известно. Это прекрасный способ познакомиться с последней версией Java быстро и эффективно. Любой пишущий на Java программист найдет в этой книге что-то такое, что позволит повысить свой уровень.

Триша Джи,
Java Champion и & Java Developer Advocate
в компании JetBrains,
июль 2017 г.

Вступление

Современный Java

Иногда трудно поверить, что язык, имеющий 20-летнюю историю обратной совместимости, мог так радикально измениться. До выхода версии Java SE 8 в марте 2014-го[1] Java, несмотря на все успехи в качестве языка программирования серверов, пользовался репутацией «COBOL'а XXI века». Стабильный, вездесущий, ориентированный на производительность. Изменения происходили медленно, если вообще происходили. Компании не особенно торопились переходить на новые версии, когда они наконец появлялись.

Все изменилось с выходом Java SE 8. В эту версию был включен «проект лямбда», главное нововведение, которое привносило идеи функционального программирования в то, что многие считали самым распространенным в мире языком объектно-ориентированного программирования. Лямбда-выражения, ссылки на методы и потоки принципиально изменили идиомы языка, и с тех пор разработчики стремятся не отстать от времени.

В этой книге я не пытался судить, хорошо ли то, что случилось, или плохо, и что можно было бы сделать по-другому. Принят другой подход: «вот что мы имеем и вот как этим можно воспользоваться во благо». Поэтому книга построена как сборник рецептов. Она о том, какую задачу требуется решить и как этому могут помочь новые средства Java.

Но нельзя не сказать о том, что у новой модели программирования немало достоинств, нужно к ним только привыкнуть. Функциональный код часто оказывается проще писать и читать. Функциональный подход поощряет неизменяемость, благодаря чему конкурентный код становится чище и с большей вероятностью правильнее. Когда язык Java только создавался, мы еще могли полагаться на закон Мура, гласящий, что быстродействие процессоров удваивается примерно каждые 18 месяцев. Но в наши дни повышение производительности обусловлено тем фактом, что даже большинство современных смартфонов оснащено несколькими процессорами.

Поскольку разработчики Java всегда уделяли первостепенное внимание обратной совместимости, многие компании и программисты перешли на Java SE 8, не дав себе труда освоить новые идиомы. Платформа-то в любом случае стала более эффективной, так что перейти на нее стоило, даже если забыть о том, что корпорация Oracle формально объявила апрель 2015-го датой кончины Java 7.

---

[1] Да, с тех пор прошло уже больше трех лет, просто не верится.

Понадобилось два года, но сейчас большинство Java-разработчиков использует Java 8 JDK, и пришло время разобраться в том, что же это означает и какие последствия несет для будущих проектов. Книга, которую вы держите в руках, поможет в этом.

Кому стоит прочитать эту книгу?

Приведенные в книге рецепты рассчитаны на читателя, который хорошо знаком с версиями Java, предшествующими Java SE 8. Экспертом быть необязательно, и о некоторых старых концепциях мы напомним, но книга определенно не является руководством по Java или объектно-ориентированному программированию для начинающих. Если вы уже писали проекты на Java и знакомы со стандартной библиотекой, то все нормально.

В книге охвачено почти все, что есть в Java SE 8, а одна глава целиком посвящена нововведениям в Java 9. Если вам интересно, как новые функциональные идиомы, добавленные в язык, изменяют подход к написанию кода, то книга поможет разобраться в этом на конкретных примерах.

Java повсеместно используется на стороне сервера и располагает богатой поддержкой в виде библиотек и инструментов с открытым исходным кодом. Каркасы Spring Framework и Hibernate относятся к числу наиболее популярных, и оба уже требуют или будут требовать в ближайшем будущем как минимум Java 8. Если вы планируете работать в этой экосистеме, то эта книга для вас.

О структуре книги

Книга представляет собой набор рецептов, но вряд ли возможно изложить рецепты, относящиеся к лямбда-выражениям, ссылкам на методы и потокам, так чтобы они не ссылались друг на друга. На самом деле в первых шести главах обсуждаются взаимосвязанные концепции, хотя читать их можно в любом порядке.

В главе 1 «Основные понятия» рассмотрены лямбда-выражения и ссылки на методы, а также новые возможности интерфейсов: методы по умолчанию и статические методы. Здесь же определен термин «функциональный интерфейс» и объяснено, почему он так важен для понимания лямбда-выражений.

В главе 2 «Пакет java.util.function» представлен новый пакет `java.util.function`, добавленный в Java 8. Интерфейсы, входящие в этот пакет, распределены по четырем категориям (потребители, поставщики, предикаты и функции) и используются во всей стандартной библиотеке.

В главе 3 «Потоки» вводятся понятие потока и та абстракция, которая позволяет использовать потоки для преобразования и фильтрации данных вместо итеративной обработки. В рецептах из этой главы рассматриваются связанные с потоками концепции «отображения», «фильтрации» и «редукции», которые ведут к идеям параллелизма и конкурентности, составляющим содержание главы 9.

В главе 4 «Компараторы и коллекторы» рассматриваются сортировка потоковых данных и преобразование их в коллекции. Сюда же включены операции разбиения и группировки, которые обычно считаются операциями базы данных, но представлены в виде простых библиотечных вызовов.

Глава 5 «Применение потоков, лямбда-выражений и ссылок на методы» – сборная солянка. Идея в том, что раз вы уже знаете, как использовать лямбда-выражения, ссылки на методы и потоки, то не худо бы посмотреть, как их комбинирование позволяет решать интересные задачи. Также рассматриваются отложенное (ленивое) выполнение, композиция замыканий и набившая оскомину тема обработки исключений.

В главе 6 «Тип Optional» обсуждается одно из самых спорных добавлений в язык – тип `Optional`. Описано, как предполагается использовать этот тип и как создавать экземпляры этого типа и получать хранящиеся в них значения. Мы также вернемся к функциональным операциям map и flat-map в применении к объектам типа `Optional` и обсудим, чем они отличаются от аналогичных операций для потоков.

В главе 7 «Файловый ввод-вывод» мы перейдем к практическому вопросу о потоках ввода-вывода (в отличие от функциональных потоков) и тем добавлениям в стандартную библиотеку, которые привносят новые функциональные идеи в работу с файлами и каталогами.

В главе 8 «Пакет java.time» описаны основы нового API для работы с датами и временем и рассказано, как он (наконец-то) заменяет унаследованные классы `Date` и `Calendar`. Новый API основан на библиотеке Joda-Time, за которой стоят многие человеко-годы разработки и использования и которая теперь переписана в виде пакета `java.time`. Откровенно говоря, даже если бы это было единственное добавление Java 8, оно уже оправдало бы переход на эту версию.

Глава 9 «Параллелизм и конкурентность» посвящена одному из неявных обещаний потоковой модели: что мы можем одним вызовом метода превратить последовательный поток в параллельный и воспользоваться всеми имеющимися процессорами. Конкурентность – обширная тема, но в этой главе описаны те добавления в библиотеку Java, которые упрощают экспериментирование и позволяют оценить, стоит ли игра свеч.

В главе 10 «Нововведения в Java 9» рассматриваются многие изменения, включенные в версию Java 9, выход которой был намечен на 21 сентября 2017 г. Детали проекта Jigsaw заслуживают отдельной книги[1], но основные элементы понятны и описаны в этой главе. Здесь же рассмотрены закрытые методы в интерфейсах, новые методы потоков, коллекторов и типа `Optional`, а также вопрос о создании потока дат[2].

---

[1] И такая книга есть: *Кишори Шаран*. Java 9. Полный обзор нововведений. М.: ДМК, 2017. – *Прим. перев.*

[2] Да, я тоже хотел бы, чтобы версии Java 9 была посвящена глава 9, но изменять логичный порядок глав ради этой, не столь существенной симметрии было бы неправильно. Достаточно и этой сноски.

Приложение А «Универсальные типы и Java 8» посвящено механизмам универсальности в Java. Хотя универсальные типы как технология были включены еще в версию 1.5, большинство разработчиков изучило только тот минимум, который необходим для работы с ними. Но даже беглое знакомство с официальной документацией по Java 8 и 9 убеждает, что эти дни давно миновали. Цель этого приложения – показать, как читать и интерпретировать API, чтобы разобраться в ставших гораздо более сложными сигнатурах методов.

Главы и, конечно, сами рецепты необязательно читать в каком-то определенном порядке. Да, они дополняют друг друга, и в конце каждого рецепта даются ссылки на другие рецепты, но начинать можно с любого места. Деление на главы позволило собрать схожие рецепты в одном месте, но предполагается, что вы будете перескакивать из одного места в другое, чтобы найти решение стоящей в данный момент задачи.

Условные обозначения и соглашения, принятые в книге

В книге используются следующие типографские соглашения:

Курсив
: Используется для смыслового выделения важных положений, новых терминов, имен команд и утилит, а также имен и расширений файлов и каталогов.

`Моноширинный шрифт`
: Используется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт`
: Используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

`Моноширинный курсив`
: Используется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.


Такая пиктограмма обозначает совет или рекомендацию.


Такая пиктограмма обозначает указание или примечание общего характера.


Эта пиктограмма обозначает предупреждение или предостережение.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Благодарности

Эта книга стала неожиданным результатом моего разговора с Джеем Циммерманном в июле 2015 года. Я принимал (и до сих пор принимаю) участие в конференциях No Fluff, Just Stuff, а в тот год с несколькими докладами по Java 8 выступал Венкат Субраманиам. Джей сказал мне, что Венкат решил снизить активность в будущем году, и поинтересовался, не хотел бы я сменить его в новом сезоне, стартующем в начале 2016 года. Я программировал на Java с середины 1990-х годов и в любом случае собирался изучить новые API, поэтому согласился.

С тех пор я вот уже два года провожу презентации по новым функциональным возможностям Java. Осенью 2016-го я закончил свою последнюю книгу[1] и, поскольку собирался писать еще один сборник рецептов для того же самого издательства, сгоряча решил, что проект будет легким.

Известный писатель-фантаст Нил Гейман как-то сказал, что ему казалось, будто после завершения «Американских богов» он знает, как писать романы. Но один приятель поправил его, заметив, что теперь он знает, как писать *этот* роман. Сейчас я понимаю, что он имел в виду. Первоначально предполагалось, что книга будет содержать 25–30 рецептов и насчитывать примерно 150 страниц. В итоге рецептов оказалось больше 70 и занимают они почти 300 страниц, но благодаря увеличению охвата материала и количества деталей книга получилась куда более ценной, чем я ожидал.

Конечно, все это стало возможным, потому что я был не один. Уже упомянутый Венкат Субраманиам очень помог мне и докладами, и своими книгами, и частными беседами. Он также любезно согласился выступить в роли технического редактора, так что если остались какие-то ошибки, то только по его вине. (Шучу, конечно, все ошибки мои, но не говорите ему, что я это признал.)

Я также весьма ценю помощь, которую мне часто оказывал Тим Йейтс (Tim Yates), один из лучших известных мне кодировщиков. Я знаю его по работе в сообществе Groovy, но его интересы гораздо шире, о чем красноречиво свидетельствует его рейтинг на сайте Stack Overflow. Род Хилтон, с которым я повстречался, когда делал доклад о Java 8 на одной из конференций NFJS, также любезно предложил отрецензировать рукопись. Рекомендации того и другого были просто бесценны.

Мне повезло работать с отличными редакторами и персоналом издательства O'Reilly, где вышли две мои книги, с дюжины видеокурсов и много онлайновых уроков, размещенных на разработанной издательством платформе Safari. Брайан Фостер неизменно поддерживал меня и к тому же обладает фантастической способностью устранять бюрократические препоны. Я познакомился с ним во время работы над предыдущей книгой, и, хотя не он был редактором

[1] «Gradle Recipes for Android», также вышедшую в O'Reilly Media и посвященную применению инструмента сборки Gradle к проектам для Android.

этой, его помощь и дружеское участие на протяжении всего процесса были очень важны для меня.

Мой редактор Джефф Блейел выказал полное понимание, видя, как книга растет в объеме, и обеспечил всю организационную структуру, необходимую для продолжения работы. Я очень доволен нашей совместной работой и надеюсь продолжить ее в будущем.

Хочу выразить признательность другим докладчикам на конференциях NFJS: Нейту Шутта (Nate Schutta), Майклу Кардуччи (Michael Carducci), Мэтту Стайну (Matt Stine), Брайану Слеттену (Brian Sletten), Марку Ричардсу (Mark Richards), Пратику Пателю (Pratik Patel), Нилу Форду (Neal Ford), Крейгу Уоллсу (Craig Walls), Раджу Ганди (Raju Gandhi), Кирку Кноерншильду (Kirk Knoernschild), Дэну «the Man» Инойоза (Dan «the Man» Hinojosa) и Джанелл Клейн (Janelle Klein) – за постоянную целеустремленность и поддержку. Написание книг и преподавание на курсах (моя повседневная деятельность) – работа, которую выполняешь в одиночестве, и тем важнее иметь друзей и единомышленников, с которыми можно поделиться своими мыслями, получить совет и вместе поразвлечься.

И наконец, выражаю бесконечную любовь своей жене Джинджер и сыну Ксандеру. Без поддержки и сердечного отношения семьи я не был бы тем, кем стал, и с каждым годом это становится для меня все более очевидно. Не могу выразить словами, как много вы значите для меня.

Глава 1

Основные понятия

Самое важное изменение в Java 8 – включение в язык ряда концепций функционального программирования: лямбда-выражений, ссылок на методы и потоков.

Если вы еще не пользовались функциональными средствами, то, вероятно, будете удивлены тем, как сильно ваш код станет отличаться от того, что вы писали в прошлом. Java 8 знаменует самое сильное изменение языка за всю его историю. Иногда складывается впечатление, что изучаешь совсем новый язык.

Возникает вопрос: а зачем это нужно? К чему такие радикальные изменения в языке, которому уже исполнилось двадцать лет и который планирует обеспечивать обратную совместимость? Зачем переходить на функциональную парадигму в языке, который считается одним из самых успешных когда-либо созданных объектно-ориентированных языков?

Все дело в том, что мир разработки программного обеспечения меняется, и если язык хочет остаться успешным, то должен адаптироваться к новому. В середине 1990-х годов, когда Java еще сверкал новизной, действовал закон Мура[1]. Надо было подождать каких-то два года, чтобы быстродействие вашего компьютера возросло в два раза.

Но сегодня рост быстродействия оборудования зависит не от плотности упаковки на кристалле. Ныне даже телефоны оборудованы несколькими процессорными ядрами, а значит, программы нужно писать с учетом того, что они будут исполняться в многопроцессорной среде. Функциональное программирование с его вниманием к «чистым» функциям (которые возвращают одинаковый результат при одних и тех же входных данных и не имеют побочных эффектов) и неизменяемости упрощает создание программ, допускающих распараллеливание. Если отсутствует разделяемое изменяемое состояние и программу можно разложить на несколько простых функций, то ее поведение проще понять и предсказать.

---

[1] Сформулированный Гордоном Муром, одним из основателей компании Fairchild Semiconductor, этот закон основан на том наблюдении, что количество транзисторов в интегральной схеме удваивается примерно каждые 18 месяцев. Детали см. в статье Википедии о законе Мура.

Но это не книга о Haskell, Erlang, Frege или еще каком-то функциональном языке программирования. Это книга о Java и о тех изменениях, которые позволили включить функциональные концепции в язык, остающийся в основе своей объектно-ориентированным.

Теперь Java поддерживает лямбда-выражения, т. е., по существу, методы, которые рассматриваются как полноправные объекты. В языке появились также ссылки на методы, позволяющие использовать существующий метод там, где ожидается лямбда-выражение. Чтобы в полной мере ощутить преимущества лямбда-выражений и ссылок на методы, в язык добавлена также потоковая модель, которая порождает элементы и передает их по конвейеру преобразований и фильтров, не изменяя источник.

В рецептах из этой главы описывается базовый синтаксис лямбда-выражений, ссылок на методы и функциональных интерфейсов, а также поддержка статических методов и методов по умолчанию в интерфейсах. Потоки подробно обсуждаются в главе 3.

1.1. Лямбда-выражения

Проблема

Вы хотите использовать в своем коде лямбда-выражения.

Решение

Воспользуйтесь одной из синтаксических разновидностей лямбда-выражений и присвойте результат ссылке, имеющей тип функционального интерфейса.

Обсуждение

Функциональный интерфейс – это интерфейс, имеющий единственный абстрактный метод. Класс реализует любой интерфейс, предоставляя реализации всех его методов. Это может быть класс верхнего уровня, внутренний класс и даже анонимный внутренний класс.

Рассмотрим, к примеру, интерфейс `Runnable`, существующий со времен версии Java 1.0. В нем имеется единственный абстрактный метод `run`, который не принимает аргументов и возвращает `void`. Конструктор класса `Thread` принимает экземпляр `Runnable` в качестве аргумента, в примере 1.1 показана реализация анонимного внутреннего класса.

Пример 1.1 ❖ Анонимный внутренний класс, реализующий интерфейс Runnable

```
public class RunnableDemo {
  public static void main(String[] args) {
    new Thread(new Runnable() { ❶
      @Override
      public void run() {
        System.out.println(
```

```
            "внутри Runnable в анонимном внутреннем классе");
      }
    }).start();
  }
}
```

❶ Анонимный внутренний класс

Синтаксически анонимный внутренний класс начинается словом `new`, за которым следуют имя интерфейса `Runnable` и скобки, означающие, что определяется класс без явно указанного имени, который реализует интерфейс. Код внутри фигурных скобок – это переопределенный метод `run`, который просто выводит строку на консоль.

В примере 1.2 показано, как то же самое реализуется с помощью лямбда-выражения.

Пример 1.2 ❖ Использование лямбда-выражения в конструкторе Thread

```
new Thread(() -> System.out.println(
  "внутри конструктора Thread с использованием лямбды")).start();
```

Синтаксически здесь используется стрелка, отделяющая аргументы (в данном случае аргументов нет, так что мы видим только пустую пару скобок) от тела. В этом примере тело содержит всего одну строку, поэтому фигурные скобки не нужны. Такая конструкция называется лямбда-выражением. Вычисленное значение выражения автоматически возвращается. В данном случае `println` возвращает `void`, поэтому и выражение имеет тип `void`, что соответствует сигнатуре метода `run`.

Типы аргументов и возвращаемого значения лямбда-выражения должны соответствовать сигнатуре единственного абстрактного метода интерфейса. Это называется *совместимостью* с сигнатурой метода. Таким образом, лямбда-выражение является реализацией метода интерфейса и может быть при желании присвоено ссылке, имеющей тип интерфейса.

Для демонстрации в примере 1.2 показано присваивание лямбда-выражения переменной.

Пример 1.3 ❖ Присваивание лямбда-выражения переменной

```
Runnable r = () -> System.out.println(
  "лямбда-выражение, реализующее метод run");
new Thread(r).start();
```

В библиотеке Java нет класса с именем `Lambda`. Лямбда-выражения можно присваивать только ссылкам типа функционального интерфейса.

Присвоить лямбда-выражение переменной типа функционального интерфейса – все равно, что сказать, что это лямбда-выражение является реализацией его единственного абстрактного метода. Мы можем рассматривать лямбда-выражение как тело анонимного внутреннего класса, реализующего интерфейс. Именно поэтому лямбда-выражение должно быть совместимо

с абстрактным методом, т. е. типы его аргументов и возвращаемого значения должны соответствовать сигнатуре метода. Отметим, однако, что имя реализуемого метода несущественно. Оно вообще не фигурирует в синтаксисе лямбда-выражения.

Это очень простой пример, поскольку метод `run` не принимает аргументов и возвращает `void`. Рассмотрим вместо этого функциональный интерфейс `java.io.FilenameFilter`, который также является частью стандартной библиотеки Java, начиная с версии 1.0. Аргументом метода `File.list` должен быть экземпляр интерфейса `FilenameFilter`, а сам метод возвращает список имен файлов, удовлетворяющих условию фильтрации.

Согласно документации Java, интерфейс `FilenameFilter` содержит единственный абстрактный метод `accept` с такой сигнатурой:

```
boolean accept(File dir, String name)
```

Аргумент `dir` – каталог, в котором находится файл, а аргумент `name` – имя файла.

В примере 1.4 интерфейс `FilenameFilter` реализован с помощью анонимного внутреннего класса, так что возвращаются только файлы, содержащие исходный код на Java.

Пример 1.4 ❖ Реализация FilenameFilter с помощью анонимного внутреннего класса

```
File directory = new File("./src/main/java");

String[] names = directory.list(new FilenameFilter() { ❶
  @Override
  public boolean accept(File dir, String name) {
    return name.endsWith(".java");
  }
});
System.out.println(Arrays.asList(names));
```

❶ Анонимный внутренний класс

Здесь метод `accept` возвращает `true`, если имя файла заканчивается строкой *.java*, и `false` в противном случае.

В примере 1.5 приведена версия с лямбда-выражением.

Пример 1.5 ❖ Лямбда-выражение, реализующее интерфейс FilenameFilter

```
File directory = new File("./src/main/java");

String[] names = directory.list((dir, name) -> name.endsWith(".java")); ❶
  System.out.println(Arrays.asList(names));
}
```

❶ Лямбда-выражение

Этот код намного проще. На этот раз в скобках указаны аргументы, но без типов. Компилятор знает, что метод `list` принимает аргумент типа `FilenameFilter`, и, следовательно, ему известна сигнатура единственного абстрактного

метода accept. А раз так, то он знает, что accept принимает аргументы типа File и String, так что совместимое лямбда-выражение должно принимать аргументы таких же типов. Метод accept возвращает значение типа boolean, значение такого же типа должно возвращать выражение справа от стрелки.

При желании можно задать типы данных явно, как показано в примере 1.6.

Пример 1.6 ❖ Лямбда-выражение с явно заданными типами данных

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) -> ❶
    name.endsWith(".java"));
```

❶ Явные типы данных

Наконец, если реализация лямбда-выражения занимает несколько строчек, то необходимо использовать фигурные скобки и включать явное предложение return, как показано в примере 1.7.

Пример 1.7 ❖ Блочное лямбда-выражение

```
File directory = new File("./src/main/java");

String[] names = directory.list((File dir, String name) -> { ❶
  return name.endsWith(".java");
});
System.out.println(Arrays.asList(names));
```

❶ Блочный синтаксис

Такое лямбда-выражение называется блочным. В данном случае тело содержит всего одну строчку, но благодаря наличию фигурных скобок строчек могло бы быть несколько. Ключевое слово return в этом варианте обязательно.

Лямбда-выражение никогда не существует само по себе. Всегда имеется *контекст*, который определяет, объекту какого функционального интерфейса присваивается выражение. Лямбда-выражение может быть аргументом метода, значением, возвращаемым методом, или значением, присваиваемым ссылке. В любом случае, соответствующий объект должен иметь тип функционального интерфейса.

1.2. Ссылки на методы

Проблема

Требуется использовать ссылку на метод, чтобы получить доступ к существующему методу и рассматривать его как лямбда-выражение.

Решение

Воспользуйтесь нотацией с двойным двоеточием, чтобы отделить имя ссылки или класса от имени метода.

Обсуждение

Если лямбда-выражение – это, по существу, способ обращаться с методом, как с объектом, то ссылка на метод – способ обращаться с существующим методом, как с лямбда-выражением.

Например, метод `forEach` интерфейса `Iterable` принимает в качестве аргумента объект типа `Consumer`. В примере 1.8 показано, что `Consumer` можно реализовать как с помощью лямбда-выражения, так и с помощью ссылки на метод.

Пример 1.8 ❖ Использование ссылки на метод для доступа к println

```
Stream.of(3, 1, 4, 1, 5, 9)
    .forEach(x -> System.out.println(x));        ❶
Stream.of(3, 1, 4, 1, 5, 9)
    .forEach(System.out::println);               ❷

Consumer<Integer> printer = System.out::println; ❸
Stream.of(3, 1, 4, 1, 5, 9)
    .forEach(printer);
```

❶ С помощью лямбда-выражения
❷ С помощью ссылки на метод
❸ Присваивание ссылки на метод переменной типа функционального интерфейса

Нотация с двойным двоеточием дает ссылку на метод `println` объекта `System.out`, т. е. экземпляра типа `PrintStream`. В конце ссылки на метод скобки не ставятся. В примере выше все элементы потока выводятся в стандартный вывод[1].

Если написанное вами лямбда-выражение состоит из одной строки, в которой вызывается метод, попробуйте заменить его эквивалентной ссылкой на метод.

Ссылка на метод обладает двумя (не очень существенными) преимуществами, по сравнению с лямбда-выражением. Во-первых, она немного короче, а во-вторых, часто включает имя класса, содержащего метод. То и другое упрощает чтение кода.

Ссылки на методы применимы и к статическим методам, как показано в примере 1.9.

Пример 1.9 ❖ Ссылка на статический метод

```
Stream.generate(Math::random)        ❶
    .limit(10)
    .forEach(System.out::println);   ❷
```

❶ Статический метод
❷ Метод экземпляра

[1] Довольно трудно обсуждать лямбда-выражения и ссылки на методы, не затрагивая потоков, которым будет посвящена отдельная глава. Пока скажем лишь, что поток порождает последовательность элементов, но нигде не хранит их и не модифицирует источник.

Метод `generate` интерфейса `Stream` принимает в качестве аргумента экземпляр функционального интерфейса `Supplier`, единственный абстрактный метод которого не имеет аргументов и порождает один результат. Метод `random` класса `Math` совместим с этой сигнатурой, потому что тоже не имеет аргументов и возвращает одно псевдослучайное число типа `double` с равномерным распределением в интервале от 0 до 1. Выражение `Math::random` ссылается на этот метод в качестве реализации интерфейса `Supplier`.

Поскольку метод `Stream.generate` порождает бесконечный поток, мы используем метод `limit` – он оставляет только 10 значений, которые выводятся в стандартный вывод с помощью ссылки на метод `System.out::println`, играющей роль реализации `Consumer`.

Синтаксис

Есть три синтаксических варианта ссылки на метод, один из которых может сбить с толку:

`object::instanceMethod`
: Ссылка на метод с помощью имеющейся ссылки на объект, например `System.out::println`.

`Class::staticMethod`
: Ссылка на статический метод, например `Math::max`.

`Class::instanceMethod`
: Вызов метода экземпляра от имени ссылки на объект, предоставляемой контекстом, например `String::length`.

Именно последний пример приводит в замешательство, поскольку Java-разработчики привыкли, что от имени класса вызываются только статические методы. Напомним, что лямбда-выражения и ссылки на методы никогда не обитают в вакууме – всегда существует контекст. В случае ссылки на объект контекст определяет аргументы метода. Если говорить о печати, то эквивалентным лямбда-выражением (в контексте оно показано в примере 1.8) будет:

```
// эквивалентно System.out::println
x -> System.out.println(x)
```

Контекст предоставляет значение `x`, которое используется в качестве аргумента метода. Для статического метода `max` ситуация аналогична:

```
// эквивалентно Math::max
(x,y) -> Math.max(x,y)
```

Теперь контекст предоставляет два аргумента, и лямбда-выражение возвращает больший из них.

Синтаксическая конструкция «метод экземпляра через имя класса» интерпретируется иначе. Эквивалентное лямбда-выражение выглядит так:

```
// эквивалентно String::length
x -> x.length()
```

На этот раз ссылка x, предоставляемая контекстом, используется как объект, от имени которого вызывается метод, а не как аргумент метода.

Если сослаться на метод, принимающий несколько аргументов, через имя класса, то первый предоставляемый контекстом элемент становится объектом, от имени которого вызывается метод, а все остальные – аргументами метода.

Пример 1.10 ❖ Вызов метода экземпляра с несколькими аргументами через имя класса

```
List<String> strings =
  Arrays.asList("this", "is", "a", "list", "of", "strings");
List<String> sorted = strings.stream()
  .sorted((s1, s2) -> s1.compareTo(s2))    ❶
  .collect(Collectors.toList());

List<String> sorted = strings.stream()
  .sorted(String::compareTo)               ❶
  .collect(Collectors.toList());
```

❶ Ссылка на метод и эквивалентное лямбда-выражение

Метод `sorted` класса `Stream` принимает в качестве аргумента объект типа `Comparator<T>`, в котором имеется единственный абстрактный метод `int compare(String other)`. Метод `sorted` передает пары строк компаратору и сортирует их в зависимости от знака возвращенного целого числа. В данном случае контекстом является пара строк. Поскольку указана ссылка на метод с использованием имени класса `String`, метод `compareTo` вызывается от имени первого элемента (`s1` в лямбда-выражении), а второй элемент `s2` передается методу в качестве аргумента.

В процессе потоковой обработки последовательности входных данных мы часто обращаемся к методу экземпляра, используя ссылку на метод через имя класса. В примере 1.11 показано, как метод `length` вызывается для каждого элемента потока типа `String`.

Пример 1.11 ❖ Вызов метода length объекта типа String с помощью ссылки на метод

```
Stream.of("this", "is", "a", "stream", "of", "strings")
    .map(String::length)              ❶
    .forEach(System.out::println);    ❷
```

❶ Метод экземпляра через метод класса
❷ Метод экземпляра через ссылку на объект

Здесь каждая строка преобразуется в целое число путем вызова метода `length`, а затем результат выводится на консоль.

Ссылка на метод – это, по существу, сокращенный синтаксический вариант лямбда-выражения. Лямбда-выражения – более общая конструкция в том смысле, что для любой ссылки на метод существует эквивалентное лямбда-выражение, но обратное неверно. В примере 1.12 показаны лямбда-выражения, эквивалентные ссылкам на методы в примере 1.11.

Пример 1.12 ❖ Лямбда-выражения, эквивалентные ссылкам на методы

```
Stream.of("this", "is", "a", "stream", "of", "strings")
    .map(s -> s.length())
    .forEach(x -> System.out.println(x));
```

Как всегда для лямбда-выражений, контекст имеет значение. Если возникает неоднозначность, то в левой части ссылки на метод можно использовать ключевое слово `this` или `super`.

См. также

С помощью синтаксиса ссылки на метод можно вызывать также конструкторы. Ссылки на конструктор описываются в рецепте 1.3. Пакет функциональных интерфейсов, включающий, в частности, упомянутый в этом рецепте интерфейс `Supplier`, обсуждается в главе 2.

1.3. Ссылки на конструкторы

Проблема

Требуется с помощью ссылки на метод создать объект в потоковом конвейере.

Решение

Использовать в ссылке на метод ключевое слово `new`.

Обсуждение

Говоря о новых синтаксических конструкциях, добавленных в Java 8, обычно упоминают лямбда-выражения, ссылки на методы и потоки. Пусть, к примеру, требуется преобразовать список людей в список имен. В примере 1.13 показан один из возможных способов.

Пример 1.13 ❖ Преобразование списка людей в список имен

```
List<String> names = people.stream()
  .map(person -> person.getName()) ❶
  .collect(Collectors.toList());

// или
List<String> names = people.stream()
  .map(Person::getName)            ❷
  .collect(Collectors.toList());
```

❶ Лямбда-выражение
❷ Ссылка на метод

Но что, если нужно сделать прямо противоположное: имея список строк, создать список объектов `Person`? В таком случае можно воспользоваться ссылкой на метод, указав в качестве имени метода ключевое слово `new`. Эта синтаксическая конструкция называется *ссылкой на конструктор*.

Чтобы показать, как она используется, начнем с простого класса `Person` (Plain Old Java Object – POJO). Он всего лишь обертывает строковый атрибут с именем `name`.

Пример 1.14 ❖ Класс Person

```
public class Person {
  private String name;

  public Person() {}

  public Person(String name) {
    this.name = name;
  }

  // методы чтения и установки ...
  // методы equals, hashCode и toString methods ...
}
```

Имея коллекцию строк, мы можем отобразить каждую из них на объект `Person`, применив либо лямбда-выражение, либо ссылку на конструктор.

Пример 1.15 ❖ Преобразование строк в экземпляры класса Person

```
List<String> names =
  Arrays.asList("Grace Hopper", "Barbara Liskov", "Ada Lovelace",
    "Karen Spärck Jones");

List<Person> people = names.stream()
  .map(name -> new Person(name))    ❶
  .collect(Collectors.toList());

// или

List<Person> people = names.stream()
  .map(Person::new)                 ❷
  .collect(Collectors.toList());
```

❶ Вызов конструктора с помощью лямбда-выражения
❷ Создание объекта `Person` с помощью ссылки на конструктор

Конструкция `Person::new` ссылается на конструктор класса `Person`. Как всегда в лямбда-выражениях, какой конструктор вызывать, определяется контекстом. Поскольку контекст предоставляет строку, вызывается конструктор с одним аргументом типа `String`.

Копирующий конструктор

Копирующий конструктор принимает аргумент типа `Person` и возвращает новый объект `Person` с такими же атрибутами (пример 1.16).

Пример 1.16 ❖ Копирующий конструктор класса Person

```
public Person(Person p) {
  this.name = p.name;
}
```

Это полезно, если требуется изолировать потоковый код от исходных экземпляров. Например, если преобразовать список людей в поток, а затем обратно в список, то мы получим те же самые ссылки (см. пример 1.17).

Пример 1.17 ❖ Преобразование списка в поток и обратно

```
Person before = new Person("Grace Hopper");

List<Person> people = Stream.of(before)
  .collect(Collectors.toList());
Person after = people.get(0);

assertTrue(before == after);                          ❶

before.setName("Grace Murray Hopper");                ❷
assertEquals("Grace Murray Hopper", after.getName()); ❸
```

❶ Тот же объект
❷ Изменить имя с помощью ссылки `before`
❸ Имя изменилось и в ссылке `after`

Копирующий конструктор позволяет разорвать эту связь.

Пример 1.18 ❖ Применение копирующего конструктора

```
people = Stream.of(before)
    .map(Person::new)          ❶
    .collect(Collectors.toList());

after = people.get(0);
assertFalse(before == after);  ❷
assertEquals(before, after);   ❸

before.setName("Rear Admiral Dr. Grace Murray Hopper");
assertFalse(before.equals(after));
```

❶ Используется копирующий конструктор
❷ Объекты разные
❸ Но эквивалентные

Теперь при вызове метода `map` контекстом является поток объектов `Person`. Поэтому `Person::new` вызывает конструктор, который принимает `Person` и возвращает новый, но эквивалентный экземпляр. Тем самым связь между ссылками *before* и *after* разрывается[1].

Конструктор с переменным числом аргументов

Добавим в класс `Person` конструктор с переменным числом аргументов, показанный в листинге 1.19.

---

[1] Я не хотел выказать неуважение, рассматривая адмирала Хоппер как объект. Не сомневаюсь, что она могла бы надрать мне задницу, но она ушла от нас в 1992 году.

Пример 1.19 ❖ Конструктор Person, принимающий переменное число аргументов типа String

```
public Person(String... names) {
  this.name = Arrays.stream(names)
                    .collect(Collectors.joining(" "));
}
```

Этот конструктор принимает нуль или более строковых аргументов и конкатенирует их через пробел.

Как вызвать такой конструктор? Это может сделать любой клиент, который передаст нуль или более строковых аргументов, разделенных запятыми. Например, можно воспользоваться методом `split` класса `String`, который принимает разделитель и возвращает массив объектов типа `String`:

```
String[] split(String delimiter)
```

Поэтому код в примере 1.20 разбивает каждую строку из списка на слова и вызывает конструктор с переменным числом аргументов.

Пример 1.20. ❖ Использование конструктора с переменным числом аргументов

```
names.stream()                  ❶
  .map(name -> name.split(" ")) ❷
  .map(Person::new)             ❸
  .collect(Collectors.toList()); ❹
```

❶ Создать поток строк
❷ Отобразить на поток массивов строк
❸ Отобразить на поток объектов `Person`
❹ Собрать в список объектов `Person`

На этот раз контекстом метода `map`, содержащего ссылку на конструктор `Person::new`, является поток массивов строк, поэтому вызывается конструктор с переменным числом аргументов. Если включить в этот конструктор простую печать:

```
System.out.println("Varargs ctor, names=" + Arrays.toList(names));
```

то получится такой результат:

```
Varargs ctor, names=[Grace, Hopper]
Varargs ctor, names=[Barbara, Liskov]
Varargs ctor, names=[Ada, Lovelace]
Varargs ctor, names=[Karen, Spdrck, Jones]
```

Массивы

Ссылки на конструктор можно использовать и вместе с массивами. Если требуется массив объектов `Person`, т. е. `Person[]` вместо списка, то можно воспользоваться методом `toArray` класса `Stream` с такой сигнатурой:

```
<A> A[] toArray(IntFunction<A[]> generator)
```

Здесь A представляет универсальный тип возвращенного массива, который содержит элементы потока и создается с помощью переданной порождающей функции. Интересно, что и в этой ситуации можно использовать ссылку на конструктор.

Пример 1.21 ❖ Создание массива объектов Person

```
Person[] people = names.stream()
  .map(Person::new)          ❶
  .toArray(Person[]::new);   ❷
```

❶ Ссылка на конструктор Person
❷ Ссылка на конструктор массива Person

Аргумент метода toArray создает массив объектов Person нужного размера и заполняет его созданными экземплярами Person.

Ссылка на конструктор – это просто ссылка на метод, в которой в качестве имени метода указано ключевое слово new. Какой именно конструктор будет вызван, как обычно, определяется контекстом. Эта техника находит широкое применение при обработке потоков.

См. также

Ссылки на методы обсуждаются в рецепте 1.2.

1.4. Функциональные интерфейсы

Проблема

Требуется использовать уже имеющийся функциональный интерфейс или написать свой собственный.

Решение

Создать интерфейс с единственным абстрактным методом, снабдив его аннотацией @FunctionalInterface.

Обсуждение

Функциональным интерфейсом в Java 8 называется интерфейс с единственным абстрактным методом. Благодаря этому свойству переменным такого типа можно присваивать лямбда-выражение или ссылку на метод.

Слово «абстрактный» здесь важно. До Java 8 все методы интерфейсов по умолчанию считались абстрактными, даже ключевое слово abstract не нужно было добавлять. Так, в примере 1.22 приведено определение интерфейса PalindromeChecker.

Пример 1.22 ❖ Интерфейс PalindromeChecker

```
@FunctionalInterface
public interface PalindromeChecker {
  boolean isPalidrome(String s);
}
```

Все методы интерфейса являются открытыми[1], поэтому модификатор доступа можно опускать, точно так же как ключевое слово `abstract`.

Поскольку в этом интерфейсе объявлен единственный абстрактный метод, он является функциональным. В Java 8 имеется аннотация `@FunctionalInterface`, применимая к этому интерфейсу (она находится в пакете `java.lang`).

Задавать эту аннотацию необязательно, но имеет смысл по двум причинам. Во-первых, если она присутствует, то компилятор проверяет, что интерфейс удовлетворяет требованиям к функциональному интерфейсу. Если в интерфейсе нет абстрактных методов или их больше одного, то будет выдано сообщение об ошибке.

Во-вторых, при наличии аннотации `@FunctionalInterface` в документацию включается такой текст:

```
Functional Interface:
This is a functional interface and can therefore be used as the assignment
target for a lambda expression or method reference.

Функциональный интерфейс:
Это функциональный интерфейс, поэтому переменным этого типа можно присваивать лямбда-выражение
или ссылку на метод.
```

В функциональных интерфейсах также могут быть методы, объявленные с ключевыми словами `default` и `static`. Методы по умолчанию и статические методы имеют реализации, поэтому не нарушают требования о существовании единственного абстрактного метода.

Пример 1.23 ❖ MyInterface – функциональный интерфейс, содержащий статический метод и метод по умолчанию

```
@FunctionalInterface
public interface MyInterface {
  int myMethod();          ❶
  // int myOtherMethod();  ❷

  default String sayHello() {
    return "Hello, World!";
  }

  static void myStaticMethod() {
```

[1] По крайней мере, так было до выхода версии Java 9, в которой в интерфейсах разрешены также закрытые (`private`) методы. Подробнее см. рецепт 10.2.

```
    System.out.println("Это статический метод интерфейса");
  }
}
```

❶ Единственный абстрактный метод
❷ Если раскомментировать эту строку, интерфейс перестанет быть функциональным

Отметим, что если бы мы включили закомментированный метод `myOtherMethod`, то интерфейс перестал бы удовлетворять требованию к функциональному интерфейсу. И в этом случае аннотация выдала бы сообщение «multiple non-overriding abstract methods found» (обнаружено несколько непереопределяющих абстрактных методов).

Интерфейс может расширять другие интерфейсы и даже несколько. Аннотация проверяет текущий интерфейс. Поэтому если некоторый интерфейс расширяет функциональный интерфейс и добавляет еще один абстрактный метод, то он уже не считается функциональным интерфейсом (см. пример 1.24).

Пример 1.24 ❖ После расширения функциональный интерфейс перестает быть функциональным

```
public interface MyChildInterface extends MyInterface {
  int anotherMethod();   ❶
}
```

❶ Дополнительный абстрактный метод

Интерфейс `MyChildInterface` не является функциональным, потому что в нем два абстрактных метода: `myMethod`, унаследованный от `MyInterface`, и `anotherMethod`, объявленный в нем самом. Без аннотации `@FunctionalInterface` этот код компилируется, поскольку здесь определен стандартный интерфейс. Но присвоить переменной такого типа лямбда-выражение нельзя.

Следует отметить один любопытный случай. Интерфейс `Comparator` используется для сортировки и обсуждается в других рецептах. Если открыть документацию по этому интерфейсу и перейти на вкладку **Abstract Methods** (Абстрактные методы), то мы увидим методы, показанные на рис. 1.1.

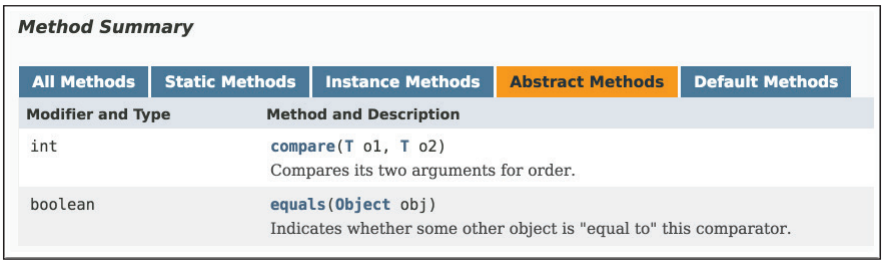

Рис. 1.1 ❖ Абстрактные методы интерфейса Comparator

Как же так? Как этот интерфейс может быть функциональным, если в нем два абстрактных метода, причем один из них фактически реализован в `java.lang.Object`?

Но это всегда было разрешено. Мы можем объявлять методы `Object` как абстрактные в интерфейсе, но это не делает их абстрактными. Обычно так делают для того, чтобы добавить в документацию пояснения, касающиеся контракта интерфейса. В случае `Comparator` контракт состоит в том, что если метод `equals` возвращает для двух элементов `true`, то метод `compare` должен вернуть 0. Добавление метода `equals` в `Comparator` позволяет включить в документацию соответствующее пояснение.

В требованиях к функциональным интерфейсам оговорено, что методы `Object` не учитываются при подсчете абстрактных методов, поэтому `Comparator` все же считается функциональным интерфейсом.

См. также

Методы по умолчанию в интерфейсах обсуждаются в рецепте 1.5, а статические методы – в рецепте 1.6.

1.5. Методы по умолчанию в интерфейсах

Проблема

Требуется предоставить реализацию метода в самом интерфейсе.

Решение

Включить в объявление метода интерфейса ключевое слово `default` и добавить реализацию, как обычно.

Обсуждение

Множественное наследование никогда не поддерживалось в Java из-за *проблемы ромбовидного наследования*. Допустим, что иерархия наследования имеет вид, показанный на рис. 1.2 (в нотации, напоминающей UML).

У класса `Animal` два дочерних класса, `Bird` и `Horse`, в каждом из которых переопределен метод `speak` из класса `Animal`: в классе `Horse` печатается «и-го-го», а в классе `Bird` – «чирик». Но что тогда должен напечатать этот метод в классе `Pegasus` (который наследует и `Horse`, и `Bird`)[1]? А если объекту типа `Animal` присвоена ссылка на экземпляр `Pegasus`? Что тогда должен сделать метод `speak`?

```
Animal animal = new Pegasus();
animal.speak(); // и-го-го, чирик или что-то другое?
```

[1] «Великолепная лошадь с мозгами птицы». (Так Пегас определен в диснеевском мультфильме «Геркулес», который забавно посмотреть, если предположить, что вы ничего не знаете о греческой мифологии и никогда не слышали о Геракле.)

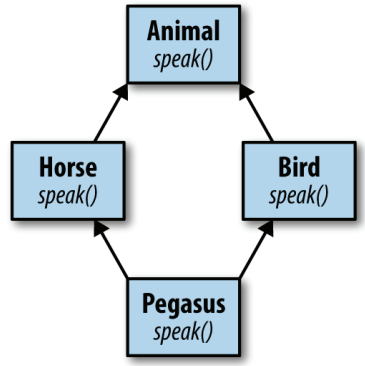

Рис. 1.2 ❖ Иерархия наследования класса `Animal`

В разных языках к этой проблеме подходят по-разному. Так, в C++ множественное наследование разрешено, но если класс наследует конфликтующие реализации, то программа не откомпилируется[1]. В Eiffel[2] компилятор позволяет выбрать нужную реализацию.

В Java было решено вообще запретить множественное наследование, а вместо него были введены интерфейсы, позволяющие выразить отношение «является разновидностью», когда типов более одного. Поскольку в интерфейсе можно было объявить только абстрактные методы, конфликта реализаций не возникало. Для интерфейсов множественное наследование разрешено, но работает это лишь потому, что наследуются только сигнатуры методов.

Проблема в том, что если реализовать метод в интерфейсе невозможно, то дизайн иногда принимает уродливые формы. Например, в интерфейсе `java.util.Collection` есть такие методы:

```
boolean isEmpty()
int size()
```

Метод `isEmpty` возвращает `true`, если в коллекции нет элементов, и `false` в противном случае. Метод `size` возвращает количество элементов в коллекции. При любой реализации `isEmpty` выражается в терминах `size`, как показано в примере 12.5.

Пример 1.25 ❖ Реализация isEmpty в терминах size

```
public boolean isEmpty() {
  return size() == 0;
}
```

---

[1] Эту проблему можно решить, используя виртуальное наследование, но тем не менее.

[2] Возможно, это упоминание для вас ничего не значит, но Eiffel был одним из основополагающих языков для объектно-ориентированного программирования. См.: *Bertrand Meyer*. Object-Oriented Software Construction. Second Edition (Prentice Hall, 1997).

Но поскольку Collection – интерфейс, мы не можем сделать это прямо в определении интерфейса. Вместо этого в стандартную библиотеку включен абстрактный класс java.util.AbstractCollection, который среди прочего содержит приведенную выше реализацию. Если вы создаете свою реализацию коллекции, для которой нет никакого суперкласса, то можете расширить AbstractCollection и получить метод isEmpty задаром. Если же суперкласс есть, то придется реализовывать интерфейс Collection самостоятельно, в т. ч. оба метода: isEmpty и size.

Все это хорошо знакомо опытным Java-разработчикам, но в Java 8 ситуация изменилась. Теперь методы интерфейса могут иметь реализации. Нужно лишь включить в определение метода ключевое слово default и предоставить реализацию. В примере 1.26 приведен интерфейс, имеющий абстрактные методы и метод по умолчанию.

Пример 1.26 ❖ Интерфейс Employee с методом по умолчанию

```
public interface Employee {
  String getFirst();

  String getLast();

  void convertCaffeineToCodeForMoney();

  default String getName() {    ❶
    return String.format("%s %s", getFirst(), getLast());
  }
}
```

❶ Метод по умолчанию, имеющий реализацию

В определении метода getName имеется ключевое слово default, а реализован он в терминах других, абстрактных методов, getFirst и getLast.

Многие существующие интерфейсы Java были дополнены методами по умолчанию, чтобы сохранить обратную совместимость. Обычно после добавления нового метода в интерфейс все его существующие реализации перестают компилироваться. Но если новый метод является методом по умолчанию, то все существующие реализации наследуют его и продолжают работать. Благодаря этому в разные места JDK удалось добавить новые методы, не «поломав» существующих реализаций.

Например, интерфейс java.util.Collection теперь содержит такие методы по умолчанию:

```
default boolean removeIf(Predicate<? super E> filter)
default Stream<E> stream()
default Stream<E> parallelStream()
default Spliterator<E> spliterator()
```

Метод removeIf удаляет из коллекции все элементы, удовлетворяющие предикату Predicate[1], и возвращает true, если был удален хотя бы один элемент.

[1] Predicate – один из новых функциональных интерфейсов в пакете java.util.function, который подробно описан в рецепте 2.3.

Фабричные методы `stream` и `parallelStream` служат для создания потоков. Метод `spliterator` возвращает объект класса, реализующего интерфейс `Spliterator`, который предназначен для обхода и разбиения на группы элементов из источника.

Методы по умолчанию используются так же, как любые другие (см. пример 1.27).

Пример 1.27 ❖ Использование методов по умолчанию

```
List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9);
boolean removed = nums.removeIf(n -> n <= 0);  ❶
System.out.println("Элементы " + (removed ? "были" : "НЕ были") + " удалены");
nums.forEach(System.out::println);             ❷
```

❶ Использование метода по умолчанию `removeIf` интерфейса `Collection`
❷ Использование метода по умолчанию `forEach` интерфейса `Iterator`

Что произойдет, если класс реализует два интерфейса, содержащих одноименные методы по умолчанию? Это тема рецепта 5.5, но краткий ответ такой: если класс реализует этот метод самостоятельно, то все хорошо.

См. также

В рецепте 5.5 сформулированы правила, действующие тогда, когда класс реализует несколько интерфейсов, содержащих методы по умолчанию.

1.6. Статические методы в интерфейсах

Проблема

Требуется включить в интерфейс вспомогательный метод уровня класса вместе с реализацией.

Решение

Включить в определение метода ключевое слово `static` и предоставить реализацию, как обычно.

Обсуждение

Статические члены классов Java определены на уровне класса, т. е. ассоциированы с классом в целом, а не с его конкретным экземпляром. Из-за этого их использование в интерфейсах поднимает ряд вопросов.

- Что понимать под членом уровня класса, когда интерфейс реализуется несколькими классами?
- Должен ли класс реализовать интерфейс, чтобы воспользоваться его статическим методом?
- К статическим методам классов обращаются, указывая имя класса. А если класс реализует интерфейс, то следует ли указывать при обращении имя класса или имя интерфейса?

Проектировщики Java могли бы ответить на эти вопросы по-разному. До Java 8 статические члены в интерфейсах вообще были запрещены.

К сожалению, это привело к появлению *служебных* классов, не содержащих ничего, кроме статических методов. Типичный пример – класс `java.util.Collections`, который содержит методы для сортировки и поиска, обертывания коллекций синхронизированными или немодифицируемыми типами и т. д. Другой пример – класс `java.nio.file.Paths` из пакета NIO. Он содержит только статические методы для построения экземпляров `Path` из строк или URI-адресов.

Но в Java 8 мы наконец-то можем помещать статические методы в интерфейсы. При этом предъявляются следующие требования:

- добавить в определение метода ключевое слово `static`;
- предоставить реализацию (которую нельзя переопределить). В этом отношении статические методы похожи на методы по умолчанию и в документации по Java находятся на вкладке **Default Methods**;
- обращаться к методу, указывая имя интерфейса. Классы *не* обязаны реализовывать интерфейс, чтобы воспользоваться его статическими методами.

В качестве примера удобного статического метода интерфейса приведем метод `comparing` интерфейса `java.util.Comparator`, а также его варианты для примитивных типов: `comparingInt`, `comparingLong` и `comparingDouble`. В интерфейсе `Comparator` есть также статические методы `naturalOrder` и `reverseOrder`. В примере 1.28 показано, как они используются.

Пример 1.28 ❖ Сортировка строк

```
List<String> bonds = Arrays.asList("Connery", "Lazenby", "Moore",
  "Dalton", "Brosnan", "Craig");

List<String> sorted = bonds.stream()
  .sorted(Comparator.naturalOrder())                    ❶
  .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

sorted = bonds.stream()
  .sorted(Comparator.reverseOrder())                    ❷
  .collect(Collectors.toList());
// [Moore, Lazenby, Dalton, Craig, Connery, Brosnan]

sorted = bonds.stream()
  .sorted(Comparator.comparing(String::toLowerCase))    ❸
  .collect(Collectors.toList());
// [Brosnan, Connery, Craig, Dalton, Lazenby, Moore]

sorted = bonds.stream()
  .sorted(Comparator.comparingInt(String::length))      ❹
  .collect(Collectors.toList());
// [Moore, Craig, Dalton, Connery, Lazenby, Brosnan]

sorted = bonds.stream()
```

```
  .sorted(Comparator.comparingInt(String::length)          ❺
    .thenComparing(Comparator.naturalOrder()))
  .collect(Collectors.toList());
// [Craig, Moore, Dalton, Brosnan, Connery, Lazenby]
```

❶ Естественный порядок (лексикографический)
❷ Обратный лексикографический порядок
❸ Сортировать по имени в нижнем регистре
❹ Сортировать по длине имени
❺ Сортировать по длине, а при равной длине лексикографически

В этом примере продемонстрировано применение нескольких статических методов интерфейса `Comparator` для сортировки списка актеров, в разные годы игравших роль Джеймса Бонда[1]. Мы еще будем обсуждать компараторы в рецепте 4.1.

Возможность включать статические методы в интерфейсы устраняет необходимость в отдельных служебных классах, хотя никто не мешает создавать их, если это удобно с точки зрения проектирования.

Запомните следующие положения:

- статический метод должен иметь реализацию;
- статический метод нельзя переопределять;
- при вызове статического метода указывается имя интерфейса;
- чтобы воспользоваться статическими методами интерфейса, реализовывать его необязательно.

См. также

Статические методы интерфейсов встречаются на протяжении всей книги, а в рецепте 4.1 специально рассматриваются статические методы интерфейса `Comparator`.

[1] Очень хочется включить в этот список Идриса Эльбу, но пока время не настало.