Рауль-Габриэль Урма, Марио Фуско, Алан Майкрофт

Современный язык Java

Лямбда-выражения, потоки и функциональное программирование

MANNING

Modern Java in Action

LAMBDAS, STREAMS, FUNCTIONAL AND REACTIVE PROGRAMMING

RAOUL-GABRIEL URMA, MARIO FUSCO, AND ALAN MYCROFT

2nd Edition

MANNING
SHELTER ISLAND

Рауль-Габриэль Урма,
Марио Фуско,
Алан Майкрофт

Современный язык Java

Лямбда-выражения, потоки и функциональное программирование

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2020

Рауль-Габриэль Урма, Марио Фуско, Алан Майкрофт

Современный язык Java. Лямбда-выражения, потоки и функциональное программирование

Серия «Для профессионалов»

Перевел на русский И. Пальти

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*М. Матвеев*
Художники	*А. Барцевич, С. Заматевская*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*О. Богданович*

ББК 32.973.2-018.1
УДК 004.43

Урма Рауль-Габриэль, Фуско Марио, Майкрофт Алан

У69 Современный язык Java. Лямбда-выражения, потоки и функциональное программирование. — СПб.: Питер, 2020. — 592 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0997-5

Преимущество современных приложений — в передовых решениях, включающих микросервисы, реактивные архитектуры и потоковую обработку данных. Лямбда-выражения, потоки данных и долгожданная система модулей платформы Java значительно упрощают их реализацию. Пришло время повысить свою квалификацию и встретить любой вызов во всеоружии!

Книга поможет вам овладеть новыми возможностями современных дополнений, таких как API Streams и система модулей платформы Java. Откройте для себя новые подходы к конкурентности и узнайте, как концепции функциональности улучшают работу с кодом.

В этой книге:

- Новые возможности Java.
- Потоковые данные и реактивное программирование.
- Система модулей платформы Java.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1617293566 англ.
978-5-4461-0997-5

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 09.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 28.08.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 47,730. Тираж 1500. Заказ 0000.

Краткое содержание

Оглавление

Часть II. Функциональное программирование с помощью потоков

Часть III. Эффективное программирование с помощью потоков и лямбда-выражений

Часть IV. Java на каждый день

Часть V. Расширенная конкурентность в языке Java

Часть VI. Функциональное программирование и эволюция языка Java

Приложения

Предисловие

В 1998 году, когда мне было всего восемь лет, я открыл свою первую книгу по программированию — она была посвящена JavaScript и HTML. Я и не подозревал, что это простое действие перевернет всю мою жизнь — я открыл для себя мир языков программирования и всех тех потрясающих вещей, которые можно сделать с их помощью. Я попался на крючок. Время от времени я по-прежнему узнаю о новых возможностях языков программирования, которые воскрешают во мне то чувство восхищения, поскольку позволяют писать более аккуратный и лаконичный код, причем вдвое быстрее. Надеюсь, обсуждаемые в данной книге нововведения Java версий 8, 9 и 10, привнесенные из функционального программирования, будут так же вдохновлять вас.

Наверное, вам интересно, как появилась эта книга вообще и второе ее издание в частности?

Что ж, в 2011 году Брайан Гётц (Brian Goetz) — архитектор языка Java в Oracle — публиковал различные предложения по добавлению лямбда-выражений в Java, желая вовлечь в этот процесс сообщество разработчиков. Они вновь разожгли мой интерес, и я начал продвигать эти идеи, организуя семинары по Java 8 на различных конференциях разработчиков, а также читать лекции студентам Кембриджского университета.

К апрелю 2013 года мой издатель из Manning прислал мне по электронной почте письмо, где спросил, не хочу ли я написать книгу о лямбда-выражениях в Java 8. На тот момент я был всего лишь скромным соискателем степени PhD на втором году обучения, и это предложение показалось мне несвоевременным, поскольку могло негативно отразиться на моем графике подачи диссертации. С другой стороны, *carpe diem*[1]. Мне казалось, что написание небольшой книги требует не так уж

[1] В переводе с лат. «лови день». — *Примеч. пер.*

много времени (и только позднее я понял, как сильно ошибался!). В итоге я обратился за советом к своему научному руководителю профессору Алану Майкрофту (Alan Mycroft), который, как оказалось, был не против поддержать меня в такой авантюре (даже предложил помочь с этой не относящейся к моей диссертации работой, за что я навеки его должник). А несколько дней спустя мы встретили знакомого евангелиста Java 8 Марио Фуско (Mario Fusco), обладавшего богатым опытом и хорошо известного своими докладами по функциональному программированию на крупных конференциях разработчиков.

Мы быстро поняли, что, объединив наши усилия и столь разнородный опыт, можем написать не просто небольшую книгу по лямбда-выражениям Java 8, а книгу, которая, надеемся, будет полезной сообществу Java-разработчиков и через пять и даже десять лет. У нас появилась уникальная возможность подробно обсудить множество вопросов, важных для Java-разработчиков, и распахнуть двери в совершенно новую вселенную: функциональное программирование.

Наступил 2018 год, и оказалось, что было продано 20 000 экземпляров первого издания, версия Java 9 только появилась, вот-вот должна была появиться версия Java 10, и время притупило воспоминания о множестве долгих ночей редактирования книги. Итак, вот оно — второе издание *«Современного языка Java»*, охватывающее Java 8, Java 9 и Java 10! Надеемся, оно вам понравится!

Рауль-Габриэль Урма (Raoul-Gabriel Urma),
Cambridge Spark

Благодарности

Эта книга не появилась бы на свет без поддержки множества замечательных людей, в числе которых:

- наши близкие друзья и просто люди, на добровольных началах дававшие ценные отзывы и предложения: Ричард Уолкер (Richard Walker), Ян Сагановски (Jan Saganowski), Брайан Гётц (Brian Goetz), Стюарт Маркс (Stuart Marks), Джем Редиф (Cem Redif), Пол Сандоз (Paul Sandoz), Стивен Колбурн (Stephen Colebourne), Иньиго Медиавилла (Íñigo Mediavilla), Аллабакш Асадулла (Allahbaksh Asadullah), Томаш Нуркевич (Tomasz Nurkiewicz) и Михаэль Мюллер (Michael Müller);
- участники программы раннего доступа от издательства Manning (MEAP), публиковавшие свои комментарии на онлайн-форуме автора;
- рецензенты, дававшие полезные отзывы во время написания книги: Антонио Маньяни (Antonio Magnaghi), Брент Стэйнз (Brent Stains), Франциска Мейер (Franziska Meyer), Фуркан Камачи (Furkan Kamachi), Джейсон Ли (Jason Lee), Джорн Динкла (Jörn Dinkla), Лочана Меникараччи (Lochana Menikarachchi), Маюр Патил (Mayur Patil), Николаос Кайнтанцис (Nikolaos Kaintantzis), Симоне Борде (Simone Bordet), Стив Роджерс (Steve Rogers), Уилл Хейворт (Will Hayworth) и Уильям Уилер (William Wheeler);
- Кевин Харрельд (Kevin Harreld) — наш редактор-консультант из издательства Manning, терпеливо отвечавший на все наши вопросы и решавший наши проблемы. Он подробно консультировал нас при написании книги и вообще поддерживал, как только мог;
- Дэннис Селинджер (Dennis Selinger) и Жан-Франсуа Морен (Jean-François Morin), вычитавшие рукопись перед сдачей в верстку, а также Эл Шерер (Al Scherer), консультировавший нас по техническим вопросам во время работы над книгой.

Рауль-Габриэль Урма

Прежде всего я хотел бы поблагодарить родителей за их бесконечную любовь и поддержку на протяжении всей моей жизни. Моя маленькая мечта о написании книги стала реальностью! Кроме того, хотелось бы выразить бесконечную признательность Алану Майкрофту, моему соавтору и научному руководителю, за его доверие и поддержку. Хотел бы также поблагодарить моего соавтора Марио Фуско, разделившего со мной это приключение. Наконец, спасибо моим друзьям и наставникам за их полезные советы и моральную поддержку: Софии Дроссопулу (Sophia Drossopoulou), Эйдену Рошу (Aidan Roche), Алексу Бакли (Alex Buckley), Хаади Джабадо (Haadi Jabado) и Гаспару Робертсону (Jaspar Robertson). Вы крутые!

Марио Фуско

Я хотел бы в первую очередь поблагодарить свою жену Марилену, благодаря безграничному терпению которой мне удалось сосредоточиться на написании этой книги, и нашу дочь Софию, поскольку создаваемый ею бесконечный хаос помогал мне творчески отвлекаться от книги. Как вы узнаете во время чтения, София преподала нам один урок — о различиях между внутренней и внешней итерацией, — как это может сделать только двухлетняя девочка. Хотел бы также поблагодарить Рауля-Габриэля Урму и Алана Майкрофта, с которыми я разделил (большую) радость и (маленькие) огорчения от написания этой книги.

Алан Майкрофт

Я хотел бы выразить благодарность моей жене Хилари и остальным членам моей семьи, терпеливо выносившим мою постоянную занятость. Я хотел бы также поблагодарить своих коллег и студентов, которые на протяжении многих лет учили меня, как нужно учить, Марио и Рауля за эффективное соавторство и, в частности, Рауля — за умение столь обходительно требовать «следующий кусочек текста к пятнице».

Об этой книге

Грубо говоря, появление новых возможностей Java 8 вместе с менее явными изменениями в Java 9 — самые масштабные изменения в языке Java за 21 год, прошедший с момента выпуска Java 1.0. Ничего не было удалено, так что весь существующий код на Java сохранит работоспособность, но новые возможности предлагают мощные новые идиомы и паттерны проектирования, благодаря которым код становится чище и лаконичнее. Сначала у вас может возникнуть мысль: «Ну зачем они опять меняют мой язык?» Но потом, после начала работы с новыми функциями, вы обнаружите, что благодаря им пишете более краткий и чистый код вдвое быстрее, чем раньше, — и осознаете, что уже не хотите возвращаться к «старому Java».

Второе издание этой книги, «Современный язык Java. Лямбда-выражения, потоки и функциональное программирование», рассчитано на то, чтобы помочь вам преодолеть этап «в принципе, звучит неплохо, но все такое новое и непривычное» и начать ориентироваться в новых функциях как рыба в воде.

«Может, и так, — наверное, думаете вы, — но разве лямбды и функциональное программирование не из того разряда вещей, о которых бородатые теоретики рассуждают в своих башнях из слоновой кости?» Возможно, но в Java 8 предусмотрен ровно такой баланс идей, чтобы извлечь выгоду способами, понятными обычному Java-программисту. И эта книга как раз описывает Java с точки зрения обычного программиста, лишь с редкими отступлениями в стиле «Откуда это взялось?».

«“Лямбда-выражения” — звучит как какая-то тарабарщина!» Да, но они позволяют писать лаконичные программы на языке Java. Многие из вас сталкивались с обработчиками событий и функциями обратного вызова: когда регистрируется объект, содержащий метод, который должен вызываться в случае какого-либо события. Лямбда-выражения значительно расширяют сферу использования подобных

функций в Java. Попросту говоря, лямбда-выражения и их спутники — ссылки на методы — обеспечивают возможность максимально лаконичной их передачи в качестве аргументов для выполнения кода или методов без отрыва от какой-либо совершенно иной деятельности. В книге вы увидите, что эта идея встречается гораздо чаще, чем вы могли себе представить: от простой параметризации метода сортировки кодом для сравнения и до сложных запросов к коллекциям данных с помощью новых Stream API (API обработки потоков данных).

«Потоки данных — что это такое?» Это замечательная новая возможность Java 8. Потоки ведут себя подобно коллекциям, но обладают несколькими преимуществами перед последними, позволяя использовать новые стили программирования. Во-первых, если вам доводилось программировать на языках запросов баз данных, например SQL, то вы знаете, что они позволяют описывать запросы несколькими строками вместо множества строк на Java. Поддержка потоков данных в Java 8 дает возможность использовать подобный сжатый код в стиле запросов баз данных — но с синтаксисом Java и без необходимости каких-либо знаний о базах данных! Во-вторых, потоки устроены так, что их данные не обязательно должны находиться в памяти (или даже обрабатываться) одновременно. А значит, появляется возможность обработки потоков данных настолько больших, что они не помещаются в памяти компьютера. Кроме того, Java 8 умеет оптимизировать операции над потоками данных лучше, чем операции над коллекциями: например, может группировать несколько операций над одним потоком так, что данные требуется обходить только один раз, а не несколько. Более того, Java может автоматически параллелизовать потоковые операции (в отличие от операций над коллекциями).

«А функциональное программирование? Что это такое?» Это еще один стиль программирования, как и объектно-ориентированное программирование, но ориентированный на применение функций в качестве значений, как мы упоминали выше, говоря про лямбда-выражения.

Версия Java 8 хороша тем, что включает в привычный синтаксис Java множество замечательных идей из функционального программирования. Благодаря удачным проектным решениям можно рассматривать функциональное программирование в Java 8 как дополнительный набор паттернов проектирования и идиом, позволяющих быстрее писать более чистый и лаконичный код. Можете рассматривать его как расширение своего арсенала разработчика.

И да, помимо этих функциональных возможностей, основанных на глобальных концептуальных новшествах в Java, мы расскажем о множестве других удобных функций и новинок Java 8, например о методах с реализацией по умолчанию, новых классах `Optional` и `CompletableFuture`, а также о новом API для работы с датой и временем (Date and Time API).

Вдобавок Java 9 привносит еще несколько новшеств: новую систему модулей, поддержку реактивного программирования с помощью Flow API и другие разнообразные усовершенствования.

Но погодите, это же только введение, дальнейшие подробности вы узнаете из самой книги.

Структура издания

Книга делится на шесть частей: «Основы», «Функциональное программирование с помощью потоков», «Эффективное программирование с помощью потоков и лямбда-выражений», «Java на каждый день», «Расширенная конкурентность в языке Java» и «Функциональное программирование и эволюция языка Java». Мы настоятельно советуем вам прочитать сначала главы из первых двух частей (причем по порядку, поскольку многие из обсуждаемых понятий основываются на материале предыдущих глав), оставшиеся же четыре части можно читать относительно независимо друг от друга. В большинстве глав есть несколько контрольных заданий для лучшего усвоения вами материала.

В первой части книги приводятся основные сведения, необходимые для знакомства с новыми идеями, появившимися в Java 8. К концу первой части вы будете знать, что такое лямбда-выражения, и научитесь писать код, лаконичный и в то же время достаточно гибкий для адаптации к меняющимся требованиям.

- В главе 1 мы резюмируем основные изменения в Java (лямбда-выражения, ссылки на методы, потоки данных и методы с реализацией по умолчанию) и подготовим фундамент для остальной части книги.
- Из главы 2 вы узнаете о параметризации поведения — важном для Java 8 паттерне разработки программного обеспечения, лежащем в основе лямбда-выражений.
- Глава 3 подробно объясняет понятия лямбда-выражений и ссылок на методы, здесь приводятся примеры кода и контрольные задания.

Вторая часть книги представляет собой углубленное исследование нового Stream API, позволяющего писать мощный код для декларативной обработки коллекций данных. К концу второй части вы полностью разберетесь, что такое потоки данных и как их использовать в своем коде для лаконичной и эффективной обработки коллекций данных.

- В главе 4 вы познакомитесь с понятием потока данных и увидите, чем он отличается от коллекции.
- Глава 5 подробно описывает потоковые операции, с помощью которых можно выражать сложные запросы для обработки данных. Вы встретите в ней множество паттернов, касающихся фильтрации, срезов, сопоставления с шаблоном, отображения и свертки.
- Глава 6 посвящена сборщикам данных — функциональной возможности Stream API, с помощью которой можно выражать еще более сложные запросы для обработки данных.
- Из главы 7 вы узнаете, как организовать автоматическое распараллеливание потоков данных и в полной мере использовать многоядерную архитектуру своей машины. Кроме того, мы расскажем вам, как избежать многочисленных подводных камней и применять параллельные потоки правильно и эффективно.

В третьей части книги рассмотрены различные функции Java 8 и Java 9, которые помогут вам эффективнее использовать язык и обогатить вашу базу кода

современными идиомами. Поскольку мы нацелены на использование более продвинутых концепций программирования, то для понимания изложенного далее в книге материала не требуется знаний этих методик.

- Глава 8 написана специально для второго издания, в ней мы изучаем коллекцию API дополнений (Collection API Enhancements) Java 8 и Java 9. Глава охватывает вопросы использования фабрик коллекции и новых идиоматических паттернов для работы со списками и множествами (коллекциями `List` и `Set`), а также идиоматических паттернов для ассоциативных массивов (коллекций `Map`).
- Глава 9 посвящена вопросам усовершенствования существующего кода с помощью новых возможностей Java 8 и включает несколько полезных примеров. Кроме того, в ней мы исследуем такие жизненно важные методики разработки программного обеспечения, как применение паттернов проектирования, рефакторинг, тестирование и отладка.
- Глава 10 также написана специально для второго издания. В ней обсуждается использование API на основе предметно-ориентированного языка (DSL). Это многообещающий метод проектирования API, популярность которого неуклонно растет. Его уже можно встретить в таких интерфейсах Java, как `Comparator`, `Stream` и `Collector`.

Четвертая часть книги посвящена различным новым возможностям Java 8 и Java 9, касающимся упрощения написания кода и повышения его надежности. Мы начнем с двух API, появившихся в Java 8.

- В главе 11 описывается класс `java.util.Optional`, позволяющий проектировать более совершенные API, а заодно и снизить количество исключений, связанных с указателями на `null`.
- Глава 12 посвящена изучению Date and Time API (API даты и времени), который существенно лучше предыдущих API для работы с датами/временем, подверженных частым ошибкам.
- В главе 13 вы прочитаете, что такое методы с реализацией по умолчанию, как развивать с их помощью API с сохранением совместимости, а также познакомитесь с некоторыми паттернами и узнаете правила эффективного использования методов с реализацией по умолчанию.
- Глава 14 написана специально для второго издания, в ней изучается система модулей Java — масштабное нововведение Java 9, позволяющее разбивать огромные системы на хорошо документированные модули вместо «беспорядочного набора пакетов».

В пятой части книги исследуются способы структурирования параллельных программ на языке Java — более продвинутые, чем простая параллельная обработка потоков данных, с которой вы к тому времени уже познакомитесь в главах 6 и 7.

- Глава 15 появилась во втором издании и демонстрирует идею асинхронных API «с высоты птичьего полета» — включая понятие будущего (Future) и протокол публикации-подписки (Publish-Subscribe), лежащий в основе реактивного программирования и включенный в Flow API Java 9.

- Глава 16 посвящена классу `CompletableFuture`, с помощью которого можно выражать сложные асинхронные вычисления декларативным образом — аналогично Stream API.
- Глава 17 тоже появилась только во втором издании, в ней подробно изучается Flow API Java 9 с упором на пригодный для практического использования реактивный код.

В шестой, заключительной, части этой книги мы немного отступим от темы, чтобы предложить вам руководство по написанию эффективных программ в стиле функционального программирования на языке Java, а также сравнить возможности Java 8 с возможностями языка Scala.

- Глава 18 предлагает полное руководство по функциональному программированию, знакомит с его терминологией и объясняет, как писать программы в функциональном стиле на языке Java.
- Глава 19 охватывает более продвинутые методики функционального программирования, включая каррирующие неизменяемые структуры данных, ленивые списки и сопоставление с шаблоном. Материал этой главы представляет собой смесь практических методик, которые можно использовать в своем коде, и теоретической информации, направленной на расширение ваших познаний как программиста.
- Глава 20 продолжает тему сравнения возможностей Java 8 с возможностями языка Scala — языка, который, как и Java, основан на использовании JVM. Scala быстро развивается, угрожая занять часть ниши Java в экосистеме языков программирования.
- В главе 21 мы еще раз обсудим наш путь изучения Java 8 и свою аргументацию в пользу функционального программирования. В дополнение мы поговорим о возможных будущих усовершенствованиях и замечательных новых возможностях Java, которые могут появиться после Java 8, Java 9 и небольших добавлений в Java 10.

Наконец, в книге есть четыре приложения, охватывающие несколько дополнительных тем, связанных с Java 8. В приложении А рассматриваются несколько небольших функций языка Java 8, которые не обсуждаются в основной части книги. В приложении Б приведен обзор остальных важных дополнений библиотеки Java, которые могут быть вам полезны. Приложение В продолжает часть II и посвящено продвинутым возможностям использования потоков данных. Приложение Г изучает вопросы реализации лямбда-выражений внутри компилятора Java.

О коде

Весь исходный код в этой книге, как в листингах, так и в представленных фрагментах, набран `таким моноширинным шрифтом`, позволяющим отличить его от остального текста. Большинство листингов снабжено пояснениями, подчеркивающими важные понятия.

Все примеры из книги и инструкции по их запуску доступны в репозитории GitHub (https://github.com/java-manning/modern-java), а также на сайте издателя по адресу http://www.manning.com/books/modern-java-in-action.

Форум для обсуждения книги

На странице https://forums.manning.com/forums/modern-java-in-action вы найдете информацию о том, как попасть на форум издательства Manning после регистрации. Там можно оставлять свои комментарии по поводу данной книги, задавать технические вопросы и получать помощь от авторов и других пользователей. Узнать больше о форумах издательства Manning и правилах поведения на них можно на странице https://forums.manning.com/forums/about.

Издательство Manning взяло на себя обязательство по предоставлению места, где читатели могут конструктивно пообщаться как друг с другом, так и с авторами книги. Но оно не может гарантировать присутствия на форуме авторов, участие которых в обсуждениях остается добровольным (и неоплачиваемым). Мы советуем вам задавать авторам интересные и трудные вопросы, чтобы их интерес не угас! Как форум, так и архивы предыдущих обсуждений будут доступны на сайте издательства до тех пор, пока книга находится в продаже.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Об авторах

Рауль-Габриэль Урма — генеральный директор и один из основателей компании Cambridge Spark (Великобритания), ведущего образовательного сообщества для исследователей данных и разработчиков. Рауль был избран одним из участников программы Java Champions в 2017 году. Он работал в компаниях Google, eBay, Oracle и Goldman Sachs. Защитил диссертацию по вычислительной технике в Кембриджском университете. Кроме того, он имеет диплом магистра технических наук Имперского колледжа Лондона, окончил его с отличием и получил несколько премий за рационализаторские предложения. Рауль представил более 100 технических докладов на международных конференциях.

Марио Фуско — старший инженер-разработчик программного обеспечения в компании Red Hat, участвующий в разработке ядра Drools — механизма обработки правил JBoss. Обладает богатым опытом разработки на языке Java, участвовал (зачастую в качестве ведущего разработчика) во множестве корпоративных проектов в различных отраслях промышленности, начиная от СМИ и заканчивая финансовым сектором. В числе его интересов функциональное программирование и предметно-ориентированные языки. На основе этих двух увлечений он создал библиотеку lambdaj с открытым исходным кодом, желая разработать внутренний DSL Java для работы с коллекциями и сделать возможным применение некоторых элементов функционального программирования в Java.

Алан Майкрофт — профессор на факультете компьютерных наук Кембриджского университета, где он преподает с 1984 года. Он также сотрудник колледжа Робинсона, один из основателей Европейской ассоциации языков и систем программирования, а также один из основателей и попечителей Raspberry Pi Foundation. Имеет ученые степени в области математики (Кембридж) и компьютерных наук (Эдинбург). Алан — автор более 100 научных статей. Был научным руководителем для более 20 кандидатов на соискание PhD. Его исследования в основном касаются сферы языков программирования и их семантики, оптимизации и реализации. Какое-то время он работал в AT&T Laboratories и научно-исследовательском отделе Intel, а также участвовал в основании компании Codemist Ltd., выпустившей компилятор языка C для архитектуры ARM под названием Norcroft.

Иллюстрация на обложке

Рисунок на обложке называется «Одеяния военного мандарина в Китайской Тартарии, 1700 г.». Одеяния мандарина причудливо разукрашены, он носит меч, а также лук и колчан со стрелами. Если присмотреться к его поясу, можно заметить там греческую букву «лямбда» (намек на один из рассматриваемых в этой книге вопросов), аккуратно добавленную нашим дизайнером. Эта иллюстрация позаимствована из четырехтомника Томаса Джефферис (Thomas Jefferys) *A Collection of the Dresses of Different Nations, Ancient and Modern* («Коллекция платья разных народов, старинного и современного»), изданного в Лондоне в 1757–1772 годах. На титульном листе говорится, что это гравюра, раскрашенная вручную с применением гуммиарабика.

Томаса Джефферис (1719–1771) называли географом при короле Георге III. Он был английским картографом и ведущим поставщиком карт своего времени. Он чертил и печатал карты для правительства и других государственных органов. На его счету целый ряд коммерческих карт и атласов, в основном Северной Америки. Занимаясь картографией в разных уголках мира, он интересовался местными традиционными нарядами, которые впоследствии были блестяще переданы в данной коллекции. В конце XVIII века заморские путешествия для собственного удовольствия были относительно новым явлением, поэтому подобные коллекции пользовались популярностью, позволяя получить представление о жителях других стран как настоящим туристам, так и «диванным» путешественникам.

Разнообразие иллюстраций в книгах Джефферис — яркое свидетельство уникальности и оригинальности народов мира в то время. С тех пор тенденции в одежде сильно изменились, а региональные и национальные различия, такие значимые 200 лет назад, постепенно сошли на нет. В наши дни бывает сложно отличить друг от друга жителей разных континентов. Если взглянуть на это с оптимистической точки

зрения, мы пожертвовали культурным и внешним разнообразием в угоду более насыщенной личной жизни или более разнообразной и интересной интеллектуальной и технической деятельности.

В то время как большинство компьютерных изданий мало чем отличаются друг от друга, компания Manning выбирает для своих книг обложки, основанные на богатом региональном разнообразии, которое Джефферис воплотил в иллюстрациях два столетия назад. Это ода находчивости и инициативности современной компьютерной индустрии.

Часть I

Основы

В первой части книги приводятся основы, необходимые для понимания новых идей, появившихся в Java 8. К концу первой части вы будете знать, что такое лямбда-выражения, и научитесь писать код, лаконичный и в то же время достаточно гибкий для адаптации к меняющимся требованиям.

- В главе 1 мы резюмируем основные изменения в Java (лямбда-выражения, ссылки на методы, потоки данных и методы с реализацией по умолчанию) и подготовим фундамент для материала, изложенного в остальной части книги.
- Из главы 2 вы узнаете о параметризации поведения — важном для Java 8 паттерне разработки программного обеспечения, лежащем в основе лямбда-выражений.
- Глава 3 подробно рассказывает про понятия лямбда-выражений и ссылок на методы, приводятся прмеры кода и контрольные задания.

1 Java 8, 9, 10 и 11: что происходит?

В этой главе

- Почему язык Java продолжает меняться.
- Изменение идеологии вычислений.
- Обстоятельства, требующие эволюции Java.
- Новые базовые возможности Java 8 и Java 9.

С тех пор как в 1996 году вышла версия 1.0 набора инструментов для разработки приложений на языке Java (Java Development Kit, JDK), у Java появилось множество приверженцев и активных пользователей среди студентов, менеджеров по проектам и программистов. Java — весьма выразительный язык, который продолжает использоваться как для больших, так и для маленьких проектов. Его развитие (в виде добавления новых возможностей) от Java 1.1 (1997) до Java 7 (2011) носило очень продуманный характер. Версия Java 8 вышла в марте 2014 года, Java 9 — в сентябре 2017-го, Java 10 — в марте 2018-го, а Java 11 — в сентябре 2018-го. Вопрос вот в чем: как эти изменения касаются вас?

1.1. Итак, о чем вообще речь?

Мы полагаем, что версия Java 8 подверглась в определенных смыслах наиболее серьезным изменениям за всю историю языка Java (внесенные в Java 9 изменения, касающиеся производительности, важны, но не столь серьезны, как вы увидите далее в этой главе, а Java 10 привнесла лишь небольшие усовершенствования правил вывода типов). Хорошая новость: благодаря этим изменениям написание программ упрощается. Например, вместо написания вот такого многословного кода (для сортировки по весу списка яблок из объекта `inventory`):

```
Collections.sort(inventory, new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

в Java 8 можно написать более лаконичный код, намного яснее отражающий формулировку задачи:

```
inventory.sort(comparing(Apple::getWeight)); ◄── Первый код на Java 8 в этой книге!
```

Его можно прочитать как «отсортировать список, сравнивая яблоки по весу». Пока не задумывайтесь насчет этого кода. Далее в книге мы объясним, что он делает и как подобный код писать.

Оказало свое влияние и аппаратное обеспечение: компьютеры стали многоядерными — процессор вашего ноутбука или стационарного компьютера наверняка содержит четыре или более ядра. Но большинство существующих программ на языке Java используют только одно из этих ядер, в то время как остальные три простаивают (или тратят лишь малую толику своих вычислительных возможностей на поддержание работы операционной системы или антивируса).

До Java 8 эксперты обычно говорили, что для применения этих ядер придется воспользоваться многопоточностью. Проблема в том, что работать с потоками сложно и чревато ошибками. Развитие языка Java всегда шло в направлении упрощения использования параллельного программирования и снижения количества ошибок. В Java 1.0 были потоки, блокировки и даже модель памяти — рекомендуемая практика на тот момент — но гарантировать надежность использования этих примитивов в проектных командах неспециалистов оказалось слишком сложной задачей. В Java 5 появились стандартные блоки промышленного уровня, такие как пулы потоков и потокобезопасные коллекции. В Java 7 появился фреймворк fork/join (ветвления-объединения), благодаря чему параллелизм стал более достижимой, но все еще трудной задачей. Параллелизм в Java 8 еще больше упрощается, но все равно необходимо следовать определенным правилам, о которых вы узнаете из этой книги.

В Java 9 появляется возможность дальнейшего структурирования параллельного выполнения программ — реактивное программирование. Хотя и ориентированное скорее на профессионалов, оно стандартизирует использование таких наборов инструментов для работы с реактивными потоками данных, как RxJava и Akka, все чаще применяемых в системах с высокой степенью параллелизации.

На предыдущих двух пожеланиях (более лаконичный код и упрощение использования многоядерных процессоров) выросла вся согласованная система Java 8. Мы начнем с того, что рассмотрим ее концепции «с высоты птичьего полета» (очень кратко, но, надеемся, достаточно полно, чтобы вас заинтриговать):

- Stream API;
- способы передачи кода в методы;
- методы с реализацией по умолчанию в интерфейсах.

Java 8 предоставляет новый API (Stream API), поддерживающий множество параллельных операций обработки данных и чем-то напоминающий языки запросов баз данных — нужные действия выражаются в высокоуровневом виде, а реализация (в данном случае библиотека Streams) выбирает оптимальный низкоуровневый механизм выполнения. В результате вам не нужно использовать в коде ключевое слово `synchronized`, которое не только часто приводит к возникновению ошибок, но и расходует на многоядерных CPU больше ресурсов, чем можно предположить[1].

Со слегка ревизионистской точки зрения добавление потоков в Java 8 можно считать непосредственной причиной двух других новшеств этой версии: *методики лаконичной передачи кода в методы* (ссылки на методы, лямбда-выражения) и *методов с реализацией по умолчанию* в интерфейсах.

Но если рассматривать передачу кода в методы просто как следствие появления Streams, то сфера возможного ее применения в Java 8 сужается. Эта возможность означает новый лаконичный способ выражения *параметризации поведения* (behavior parameterization). Допустим, вам нужно написать два метода, различающихся лишь несколькими строками кода. Теперь можно просто передать код отличающихся частей в качестве аргумента (такой способ программирования лаконичнее, понятнее и менее подвержен ошибкам, по сравнению с традиционным методом копирования и вставки). Специалист может отметить, что до Java 8 параметризацию поведения реально было реализовать с помощью анонимных классов, но пусть приведенный в начале этой главы пример, демонстрирующий большую лаконичность кода в случае Java 8, скажет сам за себя в смысле ясности кода.

Возможность Java 8 передавать код в методы (а также возвращать его и включать в структуры данных) позволяет использовать целый диапазон дополнительных приемов, которые обычно объединяют под названием *функционального программирования* (functional programming). В двух словах, подобный код, который в мире функционального программирования называют *функциями*, можно передавать куда-либо и сочетать между собой, создавая мощные идиомы программирования, которые будут встречаться вам в виде Java-кода на протяжении всей этой книги.

В следующем разделе мы начнем с обсуждения (краткого) причин эволюции языков программирования, а затем рассмотрим базовые возможности Java 8. Далее мы познакомим вас с функциональным программированием, пользоваться которым стало проще благодаря этим новым возможностям и которое поддерживает новые архитектуры компьютеров. Если вкратце, то в разделе 1.2 обсуждается процесс эволюции языков и ранее отсутствовавшие в языке Java концепции, облегчающие использование многоядерного параллелизма. Раздел 1.3 рассказывает, чем замечательна новая идиома программирования — передача кода в методы в Java 8,

[1] У многоядерных CPU есть отдельный кэш (быстрая память) для каждого ядра процессора. Для блокировки нужна синхронизация этих кэшей, требующая относительно медленного взаимодействия между ядрами по протоколу, сохраняющему когерентность кэша.

а раздел 1.4 делает то же самое для потоков данных — нового способа представления в Java 8 последовательных данных и указания на возможность их параллельной обработки. Раздел 1.5 объясняет, как появившиеся в Java 8 методы с реализацией по умолчанию делают эволюцию интерфейсов и их библиотек задачей менее хлопотной и требующей меньшего количества перекомпиляций; также в нем рассказывается о появившихся в Java 9 *модулях*, с помощью которых можно описывать компоненты больших Java-систем более понятным образом, чем «JAR-файл с пакетами». Наконец, раздел 1.6 заглядывает в будущее функционального программирования на Java и других Java-подобных языках, использующих JVM. Таким образом, в этой главе мы познакомим вас с концепциями, которые будут подробно описаны в оставшейся части книги. В добрый путь!

1.2. Почему Java все еще меняется

В 1960-х годах начался поиск идеального языка программирования. Питер Лэндин (Peter Landin), известный программист того времени, в 1966 году в своей исторической статье[1] отметил, что на тот момент *уже* существовало 700 языков программирования, и привел рассуждения на тему, какими будут следующие 700, включая доводы в пользу функционального программирования в стиле, аналогичном стилю Java 8.

Спустя многие тысячи языков программирования теоретики пришли к заключению, что языки программирования подобны экосистемам: новые языки возникают и вытесняют старые, если только последние не эволюционируют. Мы все мечтаем об идеальном универсальном языке, но на практике для конкретных сфер лучше подходят определенные языки. Например, C и C++ остаются популярными языками создания операционных систем и различных встраиваемых систем благодаря малым требованиям к ресурсам, хотя в них нет типобезопасности. Отсутствие типобезопасности может приводить к внезапным сбоям программ и возникновению открытых для вирусов уязвимостей; ничего удивительного, что Java и C# вытеснили C и C++ во многих типах приложений, где допустим дополнительный расход ресурсов.

Занятость ниши обычно расхолаживает соперников. Переход на новый язык программирования и набор программных средств зачастую оказывается слишком затратной вещью ради какой-то одной возможности, но новинки постепенно заменяют существующие языки, если те, конечно, не эволюционируют достаточно быстро. Читатели постарше, наверное, могут вспомнить целый список подобных канувших в Лету языков, на которых когда-то приходилось писать программы: Ada, Algol, COBOL, Pascal, Delphi и SNOBOL — и это лишь неполный список.

Вы программист на языке Java, успешно захватившем (вытесняя соперничавшие с ним языки) большую нишу вычислительных задач на период более 20 лет. Выясним, какие причины к этому привели.

---

[1] *Landin P. J.* The Next 700 Programming Languages. CACM 9(3): 157–165, март 1966 [Электронный ресурс]. — Режим доступа: http://www.math.bas.bg/bantchev/place/iswim/next700.pdf.

1.2.1. Место языка Java в экосистеме языков программирования

Java весьма успешно стартовал. С самого начала он был хорошо продуманным объектно-ориентированным языком программирования с множеством удобных библиотек. В нем уже с первого дня была ограниченная поддержка конкурентности и встроенная поддержка потоков выполнения и блокировок (и пророческое признание — в виде аппаратно независимой модели памяти — того факта, что конкурентные потоки на многоядерных процессорах могут вести себя не так, как на одноядерных). Кроме того, решение компилировать код Java в байт-код (код для виртуальной машины, который скоро будут поддерживать все браузеры) привело к тому, что Java стал оптимальным языком для интернет-апплетов (помните, что такое апплеты?). Конечно, существует опасность, что виртуальную машину Java (JVM) и ее байт-код сочтут более важными, чем сам язык Java, и в определенных приложениях Java будет заменен одним из его языков-соперников, например Scala, Groovy или Kotlin, также работающими в JVM. Многие недавние усовершенствования JVM (например, новая инструкция байт-кода `invokedynamic` в JDK7) направлены на обеспечение беспроблемной работы подобных языков-соперников Java на JVM и их взаимодействия с Java. Java также успешно проявил себя в различных задачах встроенных вычислений в устройствах (от смарт-карт, тостеров и ТВ-приставок до тормозных систем автомобилей).

Как Java вошел в нишу общего программирования

Популярность объектно-ориентированного программирования (ООП) в 1990-х возросла по двум причинам: его строгие правила инкапсуляции снизили количество проблем разработки программного обеспечения (ПО) по сравнению с языком C; а в качестве ментальной модели оно с легкостью отражало модель программирования WIMP (https://ru.wikipedia.org/wiki/WIMP) Windows 95 и более поздних версий. Если вкратце, то любая сущность представляет собой объект; щелчок кнопкой мыши приводит к отправке обработчику сообщения о событии (вызову метода `clicked` объекта `Mouse`). Модель «напиши один раз, запускай сколько угодно раз» языка Java и умение ранних версий браузеров безопасно выполнять Java-апплеты обеспечили ему свою нишу в вузах, выпускники которых затем стали разработчиками в промышленных компаниях. Хотя сначала многие сопротивлялись из-за дополнительных затрат ресурсов на выполнение кода Java по сравнению с C/C++, но машины становились все быстрее, а ценность рабочего времени программистов увеличивалась. Язык C# от Microsoft еще более упрочнил положение объектно-ориентированной модели в стиле Java.

Но климат в экосистеме языков программирования меняется, программисты все чаще имеют дело с так называемыми *большими данными* (наборы данных, чей размер исчисляется в терабайтах и более) и хотят эффективно использовать для их обработки многоядерные компьютеры и кластеры компьютеров. А это означает необходимость параллельной обработки, которая ранее не была сильной стороной Java. Возможно, вы уже слышали о разработках из других областей программирования (например, о созданной Google технологии Map-Reduce или относительном

упрощении манипулирования данными с помощью языков запросов баз данных вроде SQL), позволяющих справляться с большими объемами данных и многоядерными CPU.

Рисунок 1.1 демонстрирует всю экосистему языков в графическом виде: можете считать ландшафт пространством вычислительных задач, а преобладающую на конкретном участке почвы растительность — наиболее популярным языком для решения конкретного типа задач. Изменение климата соответствует представлению о том, что новое аппаратное обеспечение и новые тенденции программирования (например, закрадывающаяся в голову программиста мысль: «Почему я не могу программировать в стиле SQL?») постепенно делают различные языки предпочтительными для новых проектов, подобно тому как благодаря глобальному потеплению виноград теперь растет в более высоких широтах. Но отмечается некоторое запаздывание — многие старые фермеры продолжают выращивать традиционные культуры. Подводя итог вышесказанному: новые языки программирования появляются и становятся все популярнее потому, что быстро приспособились к изменениям климата.

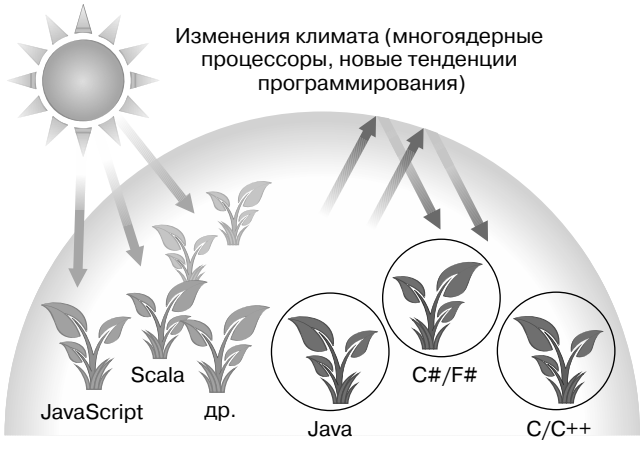


Рис. 1.1. Экосистема языков программирования и изменения климата

Главная польза от новшеств Java 8 для программиста: дополнительные утилиты и концепции для более быстрого (или, что важнее, с применением более лаконичного и лучше сопровождаемого способа) решения новых и существующих вычислительных задач. Хотя эти концепции для Java в новинку, они уже доказали свою пригодность в нишевых языках, предназначенных скорее для научных исследований. В следующих разделах мы проиллюстрируем и разовьем идеи, лежащие в основе трех подобных концепций, приведших к разработке таких возможностей Java 8, которые способствовали использованию параллелизма и повышению лаконичности кода. Мы познакомим вас с ними несколько в иной очередности, чем они будут приводиться в остальной части книги, чтобы провести аналогию с Unix-системами и продемонстрировать зависимости типа «нужно *это*, потому что *то*», характерные для многоядерного параллелизма Java 8.

Еще один фактор, меняющий климат для Java

Архитектура больших систем — один из факторов, меняющих климат. Сегодня большие системы обычно включают взятые откуда-либо крупные подсистемы, которые, в свою очередь, часто создаются на основе компонентов от других поставщиков. Хуже всего, что эти компоненты и их интерфейсы также развиваются. Для решения этой проблемы Java 8 и Java 9 предоставляют методы с реализацией по умолчанию и модули, способствующие такому стилю архитектуры.

В следующих подразделах мы рассмотрим три концепции, обусловившие архитектуру Java 8.

1.2.2. Потоковая обработка

Первая из этих концепций — *потоковая обработка* (stream processing). Если не вдаваться в подробности, то *поток данных* (stream) — это последовательность элементов данных, генерируемых в общем случае по одному. Программа может, например, читать данные из входного потока по одному элементу и аналогично записывать элементы в выходной поток. Выходной поток одной программы вполне может служить входным потоком другой.

В качестве примера из практики можно привести Unix или Linux, в которых многие программы читают данные из стандартного потока ввода (*stdin* в Unix и C, `System.in` в языке Java), производят над ними какие-либо действия и записывают результаты этих действий в стандартный поток вывода (*stdout* в Unix и C, `System.out` в языке Java). Небольшое пояснение: команда `cat` операционной системы Unix создает поток данных путем конкатенации двух файлов, `tr` преобразует символы из потока, `sort` сортирует строки потока, а команда `tail-3` возвращает три последние строки из потока. Командная строка Unix предоставляет возможность связывать подобные программы цепочкой с помощью символов конвейера (`|`), например:

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```

Эта команда (при условии, что файлы `file1` и `file2` содержат по одному слову на строку) выводит из указанных файлов три последних (в лексикографическом порядке) слова, предварительно преобразовав их в нижний регистр. При этом говорится, что `sort` получает на входе *поток* строк[1] и генерирует на выходе другой поток (отсортированный), как показано на рис. 1.2. Отметим, что в операционной системе Unix эти команды (`cat`, `tr`, `sort` и `tail`) выполняются конкурентным образом, так что команда `sort` может приступить к обработке нескольких первых строк, хотя `cat` и `tr` еще не завершили работу. Можно привести аналогию из области машиностроения. Представьте конвейер сборки автомобилей с потоком машин, выстроенных в очередь между пунктами сборки, каждый из которых получает на входе машину,

[1] Приверженцы строгости формулировок сказали бы «поток символов», но для понятности проще считать, что команда sort упорядочивает строки.

вносит в нее изменения и передает на следующий пункт для дальнейшей обработки; обработка в отдельных пунктах обычно выполняется конкурентно, несмотря на то что конвейер сборки физически последователен.

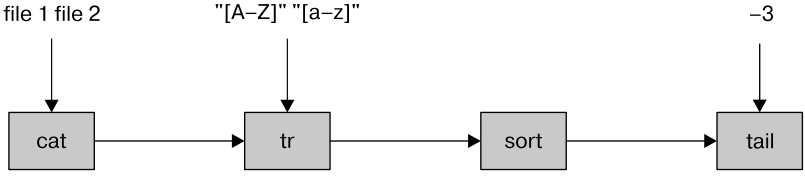


Рис. 1.2. Обработка потока данных командами Unix

В Java 8 в пакете `java.util.stream` появляется основанный на этой идее Stream API; объект `Stream<T>` представляет собой последовательность элементов типа `T`. Можете пока считать его необычным итератором. В Stream API есть множество методов, допускающих создание сложных конвейеров путем связывания их цепочкой, аналогично тому, как в предыдущем примере связывались команды Unix.

Главный вывод из этого: теперь появляется возможность создавать программы на Java 8 на более высоком уровне абстракции и мыслить в терминах преобразования потока одних элементов в поток других (аналогично написанию запросов баз данных), а не элементов поодиночке. Еще одно преимущество Java 8: возможность прозрачного выполнения конвейера потоковых операций несколькими ядрами CPU, работающими над непересекающимися подмножествами входных данных — параллелизм *практически даром* вместо кропотливой работы с использованием потоков выполнения. Мы рассмотрим Stream API Java 8 подробнее в главах 4–7.

1.2.3. Передача кода в методы и параметризация поведения

Вторая из добавленных в Java 8 концепций — возможность передачи в API фрагментов кода. Это звучит очень абстрактно. Если говорить в терминах примера с Unix-командами, то речь может идти, скажем, о передаче команде `sort` информации о пользовательском способе сортировки. Хотя команда `sort` поддерживает параметры командной строки, позволяющие выполнять сортировки различных видов (например, сортировку в обратном порядке), эти возможности ограничены.

Допустим, у вас имеется коллекция идентификаторов счетов-фактур в таком формате: 2013UK0001, 2014US0002 и т. д. Первые четыре цифры означают год, следующие две — код страны, а последние четыре — идентификатор клиента. Вы хотели бы отсортировать эти идентификаторы счетов-фактур по году или, например, по идентификатору покупателя или даже коду страны. По существу, вам хотелось бы передать команде `sort` в качестве аргумента информацию о задаваемом пользователем порядке сортировки в виде отдельного фрагмента кода.

Если провести прямую параллель с Java, то требуется сообщить методу `sort` о необходимости выполнять сравнение с учетом заданного пользователем порядка. Вы могли написать метод `compareUsingCustomerId` для сравнения двух идентифи-

каторов счетов-фактур, но в предшествующих Java 8 версиях не было возможности передать его другому методу! Можно создать объект типа `Comparator` для передачи методу `sort`, как мы показывали в начале этой главы, но это потребовало бы большого количества кода и только затуманивало бы простую идею переиспользования существующего поведения. В Java 8 появляется возможность передавать методы (ваш код) в качестве аргумента другим методам. Эта идея проиллюстрирована на рис. 1.3, в основе которого лежит рис. 1.2. С теоретической точки зрения это *параметризация поведения* (behavior parameterization). Почему она так важна? Потому, что в основе Stream API лежит идея передачи кода для параметризации поведения операций, подобно вышеприведенной передаче метода `compareUsingCustomerId` для параметризации поведения метода `sort`.

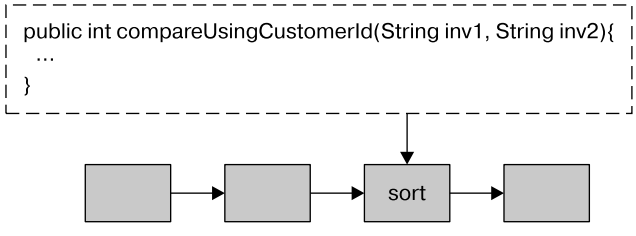


Рис. 1.3. Передача метода compareUsingCustomerId в виде аргумента методу sort

В разделе 1.3 мы вкратце опишем, как это работает, но придержим подробности до глав 2 и 3. Главы 18 и 19 посвящены более продвинутому применению этой возможности — с помощью методик *функционального программирования*.

1.2.4. Параллелизм и разделяемые изменяемые данные

Третья концепция носит менее явный характер и обусловлена фразой «параллелизм практически даром», упомянутой в нашем обсуждении потоковой обработки. Чем же приходится пожертвовать? Вам придется несколько изменить способ написания кода поведения, передаваемого методам потоков. Сначала эти изменения могут доставить небольшие неудобства, но они вам понравятся, как только вы к ним привыкнете. Вы должны обеспечить *безопасность конкурентного выполнения* передаваемого поведения в различных частях входных данных. Обычно это означает написание кода, который не обращается к разделяемым изменяемым данным. Иногда такие функции называют чистыми (pure functions), или функциями без побочных эффектов (side-effect-free functions), или функциями без сохранения состояния (stateless functions). Мы обсудим их подробнее в главах 18 и 19.

Вышеупомянутый параллелизм возможен, только если несколько копий фрагмента кода могут работать независимо друг от друга. Если производится запись в разделяемую переменную или объект, то работать ничего не будет. Ведь что случится, если двум процессам понадобится изменить эту разделяемую переменную одновременно? В разделе 1.4 приведено более подробное пояснение со схемой, и далее в книге мы еще поговорим о таком стиле написания программ.

Потоки данных Java 8 позволяют удобнее использовать параллелизм, чем существующий Thread API, поскольку *можно* нарушить правило отсутствия разделяемых изменяемых данных с помощью ключевого слова `synchronized`, хотя это приведет к неправильному использованию абстракции, оптимизированной в расчете на это правило. Применение `synchronized` в многоядерной системе требует гораздо больше ресурсов, чем можно предположить, поскольку синхронизация приводит к обязательному выполнению кода последовательно, что противоречит самой идее параллелизма.

Две из этих концепций (отсутствие разделяемых изменяемых данных и возможность передачи методов и функций, то есть кода, другим методам) — краеугольные камни парадигмы *функционального программирования* (functional programming), которую мы обсудим подробно в главах 18 и 19. Напротив, в парадигме *императивного программирования* (imperative programming) программа обычно описывается на языке последовательности операторов, изменяющих состояние. Требование отсутствия разделяемых изменяемых данных означает, что метод прекрасно описывается тем, как он преобразует аргументы в результаты; другими словами, этот метод ведет себя как математическая функция, у него нет (видимых) побочных эффектов.

1.2.5. Язык Java должен развиваться

Вы уже встречались с эволюцией языка Java. Например, появление обобщенных типов данных и использование `List<String>` вместо `List` поначалу могло вас раздражать. Но теперь вы уже знакомы с подобным стилем программирования и его преимуществами (перехват большего количества ошибок во время компиляции и повышение удобочитаемости кода, поскольку известно, из чего состоит данный список).

Благодаря другим изменениям стало проще выражать часто встречающиеся сущности (например, цикл `for-each` вместо шаблонного кода для `Iterator`). Основные изменения в Java 8 отражают переход от ориентации на традиционные объекты, часто связанной с изменением существующих значений, к многообразию функционального программирования (ФП). При ФП главное — *что* вы хотели бы сделать в общих чертах (например, *создать значение*, отражающее все маршруты транспорта из пункта А в пункт Б со стоимостью, не превышающей заданной), оно отделяется от способа *достижения цели* (например, *просмотра* структуры данных с *модификацией* определенных компонентов). Может показаться, что традиционное объектно-ориентированное и функциональное программирование, будучи противоположностями, конфликтуют. Но идея заключается именно в том, чтобы взять от обеих парадигм программирования лучшее и получить оптимальный инструмент для своей задачи. Мы обсудим это подробнее в разделах 1.3 и 1.4.

Из этого можно сделать следующий вывод: языки программирования должны эволюционировать, чтобы не отставать от изменений аппаратного обеспечения и ожиданий программистов (если вы все еще сомневаетесь в этом, задумайтесь, что COBOL когда-то считался одним из важнейших с коммерческой точки зрения языков программирования). Чтобы выжить, Java должен эволюционировать путем добавления новых возможностей. Такая эволюция бесцельна, если эти новые воз-

можности не применяются, так что, используя Java 8, вы тем самым защищаете свой образ жизни как Java-программиста. Кроме того, нам кажется, что вам понравятся новые возможности Java 8. Спросите кого угодно из тех, кто уже использовал Java 8, хотели бы они вернуться к старым версиям? Вдобавок новые возможности Java 8 могут, говоря в терминах аналогии с экосистемой, помочь Java завоевать территории вычислительных задач, занятых ныне другими языками программирования, так что спрос на программистов со знанием Java 8 еще возрастет.

Далее мы по очереди познакомим вас с новыми возможностями Java 8, указывая, в каких главах соответствующие концепции будут описаны подробнее.

1.3. Функции в Java

Слово *«функция»* (function) в языках программирования часто используется как синоним *метода* (method), особенно статического метода; помимо использования в качестве *математической функции* — функции без побочных эффектов. К счастью, как вы можете видеть, эти виды использования функций в случае Java 8 практически совпадают.

В Java 8 функции становятся новой разновидностью значений, что дает возможность использования потоков данных (см. раздел 1.4), которые служат для параллельного программирования на многоядерных процессорах. Мы начнем с того, что продемонстрируем полезность функций как значений самих по себе.

Программы на языке Java могут оперировать различными видами значений. Во-первых, существуют значения примитивных типов, например `42` (типа `int`) и `3.14` (типа `double`). Во-вторых, значения могут быть объектами (точнее, ссылками на объекты). Единственный способ получить такое значение — использовать ключевое слово `new`, возможно, посредством фабричного метода или библиотечной функции; ссылки на объекты указывают на *экземпляры* (instances) классов. В качестве примеров можно привести `"abc"` (типа `String`), `new Integer(1111)` (типа `Integer`) и результат явного вызова конструктора для `HashMap`: `new HashMap<Integer, String>(100)`. Даже массивы являются объектами. Так в чем же проблема?

Чтобы ответить на этот вопрос, нужно отметить, что главная задача любого языка программирования — работа со значениями, которые, следуя исторической традиции языков программирования, называются полноправными значениями (или полноправными гражданами в терминологии движения в защиту гражданских прав в 1960-х годах в США). Другие структуры языков программирования, которые, может, и помогают выражать структуру значений, но которые нельзя передавать во время выполнения программы, — граждане «второго сорта». Перечисленные выше значения — полноправные граждане Java, а различные другие понятия Java, например методы и классы, — примеры граждан «второго сорта». Методы прекрасно подходят для описания классов, из которых можно, в свою очередь, получить значения путем создания экземпляров, но ни те ни другие сами по себе значениями не являются. Важно ли это? Да, оказывается, что возможность передавать методы во время выполнения программы (которая делает их полноправными гражданами) полезна при программировании, так что создатели Java 8 добавили в язык возможность явного выражения этого. Кстати, вы можете задаться вопросом: хорошая ли

вообще идея превращать граждан «второго сорта», например классы, в полноправные значения? Эта дорога уже проторена различными языками программирования, в частности Smalltalk и JavaScript.

1.3.1. Методы и лямбда-выражения как полноправные граждане

Проведенные в других языках программирования, например Scala и Groovy, эксперименты показали, что возможность использования таких сущностей, как методы, в качестве полноправных значений упрощает программирование, расширяя доступный разработчикам набор инструментов. А когда программисты привыкают к этой замечательной возможности, то просто отказываются использовать языки, в которых она отсутствует! Создатели Java 8 решили позволить методам быть значениями — чтобы вам было проще писать программы. Более того, методы-значения из Java 8 закладывают фундамент для различных других возможностей Java 8 (например, потоков данных).

Первая из новых возможностей Java 8, с которой мы вас познакомим, — *ссылки на методы* (method references). Представьте себе, что вам нужно отфильтровать все скрытые файлы в каталоге. Для начала необходимо написать метод, который бы получал на входе объект `File` и возвращал информацию о том, скрытый ли этот файл. К счастью, в классе `File` такой метод уже есть — он называется `isHidden`. Его можно рассматривать как функцию, принимающую на входе объект `File` и возвращающую `boolean`. Но для выполнения фильтрации необходимо обернуть ее в объект `FileFilter` и передать его в метод `File.listFiles`:

```
File[] hiddenFiles = new File(".").listFiles(new FileFilter() {
    public boolean accept(File file) {
        return file.isHidden();      <-- Фильтрация скрытых файлов!
    }
});
```

Фу! Просто ужасно. Хотя этот фрагмент кода содержит только три значащие строки, все они трудны для понимания. Ведь каждый из нас помнит, как говорил при первом знакомстве с таким кодом: «Неужели это нужно делать именно так?» У вас уже есть метод `isHidden`, так зачем писать лишний код, обертывать его в класс `FileFilter`, создавать экземпляр этого класса?.. А затем, что до Java 8 это был единственный выход.

С появлением Java 8 можно переписать вышеприведенный код так:

```
File[] hiddenFiles = new File(".").listFiles(File::isHidden);
```

Ух ты! Правда, круто? Мы просто передали функцию `isHidden` в метод `listFiles` с помощью появившегося в Java 8 синтаксиса *ссылки на метод* `::` (означающего «использовать данный метод как значение»). Отметим также, что мы воспользовались для методов словом *«функция»*. Мы объясним позднее, как это работает. Теперь наш код — и это плюс — стал намного ближе к формулировке нашей задачи.

Приоткроем вам завесу тайны: методы больше не значения «второго сорта». Аналогично созданию *ссылки на объект* (object references) для его передачи (а ссылки на объекты создаются с помощью ключевого слова `new`), при написании `File::isHidden`

в Java 8 создается *ссылка на метод* (method reference), которую тоже можно передавать. Эту концепцию мы подробно обсудим в главе 3. А поскольку метод содержит код (исполняемое тело метода), то с помощью ссылок на методы можно передавать код, как показано на рис. 1.3. Рисунок 1.4 иллюстрирует эту концепцию. В следующем разделе мы покажем вам конкретный пример — выбор яблок из объекта `inventory`.

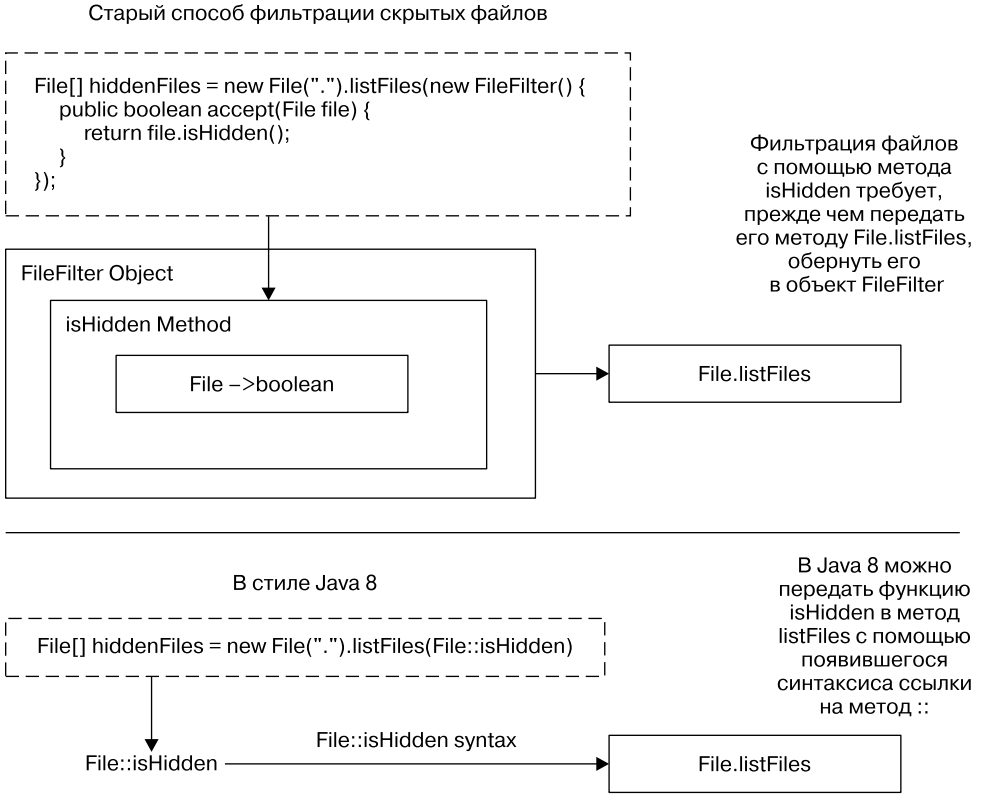


Рис. 1.4. Передача ссылки на метод File::isHidden в метод listFiles

Лямбда-выражения: анонимные функции. В Java 8 не только поименованные методы являются полноправными значениями, но и предлагается расширенное представление о *функциях как значениях*, включая лямбда-выражения (lambdas, анонимные функции)[1]. Например, теперь можно написать `(int x) -> x + 1`, что будет значить: «функция, которая, будучи вызванной с аргументом `x`, возвращает значение `x + 1`». Вы можете задаться вопросом, зачем это нужно, ведь можно определить метод `add1` в классе `MyMathsUtils` и написать `MyMathsUtils::add1`! Да, так можно поступить, но

[1] Названы в честь греческой буквы λ. Хотя сам этот символ в Java не используется, название осталось.

новый синтаксис лямбда-выражений оказывается намного лаконичнее в тех случаях, когда подходящего метода и класса нет. Лямбда-выражения подробно обсуждаются в главе 3. Об использующих их программах говорят, что они написаны в стиле функционального программирования; эта фраза означает «программы, в которых функции передаются как полноправные значения».

1.3.2. Передача кода: пример

Рассмотрим пример того, как эта возможность упрощает написание программ (обсуждается подробнее в главе 2). Весь код примеров можно найти в репозитории GitHub, а также скачать с сайта книги. Обе ссылки вы найдете по адресу http://www.manning.com/books/modern-java-in-action. Пускай у вас есть класс `Apple` с методом `getColor` и переменной `inventory`, содержащей список `Apples`. Допустим, вам нужно отобрать все зеленые яблоки (в данном случае с помощью перечисляемого типа `Color`, включающего значения `GREEN` (Зеленый) и `RED` (Красный)) и вернуть их в виде списка. Для описания этой концепции часто используется слово *«фильтр»* (filter). До появления Java 8 вы могли бы написать, скажем, следующий метод `filterGreenApples`:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (GREEN.equals(apple.getColor())) {
            result.add(apple);
        }
    }
    return result;
}
```

Итоговый список для накопления результатов; сначала он пуст, а затем в него по одному добавляются зеленые яблоки

Выделенный код отбирает только зеленые яблоки

Но затем, возможно, кому-нибудь понадобится список тяжелых яблок (скажем, тяжелее 150 г), так что вам скрепя сердце придется написать следующий метод (вероятно, даже путем копирования и вставки):

```
public static List<Apple> filterHeavyApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (apple.getWeight() > 150) {
            result.add(apple);
        }
    }
    return result;
}
```

Здесь отбираются только тяжелые яблоки

Всем известно про опасности, которые несет метод копирования и вставки применительно к разработке программного обеспечения (выполненные в одном месте обновления и исправления ошибок не переносятся в другое место), и, в конце концов, эти два метода различаются только одной строкой: условным оператором внутри конструкции `if`, выделенным жирным шрифтом. Если бы различие между двумя вызовами методов в выделенном коде заключалось только в диапазоне допустимого

веса яблока, то можно было бы передавать нижнюю и верхнюю границы диапазона в виде аргументов метода `filter` — скажем, `(150,1000)`, чтобы выбрать тяжелые яблоки (тяжелее 150 г), и `(0,80)`, чтобы выбрать легкие (легче 80 г).

Но, как мы уже упоминали, в Java 8 можно передавать в виде аргумента код условия, что позволяет избежать дублирования кода в методе `filter`. Теперь можно написать следующее:

```
public static boolean isGreenApple(Apple apple) {
    return GREEN.equals(apple.getColor());
}
public static boolean isHeavyApple(Apple apple) {
    return apple.getWeight() > 150;
}
public interface Predicate<T>{
    boolean test(T t);
}
static List<Apple> filterApples(List<Apple> inventory,
                                Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory){
        if (p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Включено сюда для большей понятности (обычно импортируется из пакета java.util.function)

Метод передается в виде параметра типа Predicate с именем p (см. врезку «Что представляет собой интерфейс Predicate» далее)

Удовлетворяет ли яблоко условию из предиката p?

Воспользоваться этим методом можно, выполнив либо вызов:

```
filterApples(inventory, Apple::isGreenApple);
```

либо вызов:

```
filterApples(inventory, Apple::isHeavyApple);
```

Мы поясним, как это работает, в следующих двух главах. Главный вывод из изложенного: в Java 8 можно передавать методы как значения.

Что представляет собой интерфейс Predicate

В вышеприведенном коде метод `Apple::isGreenApple` (принимающий в качестве параметра объект типа `Apple` и возвращающий значение типа `boolean`) передается в функцию `filterApples`, которая ожидает на входе параметр типа `Predicate<Apple>`. Слово *«предикат»* (predicate) часто используется в математике для обозначения сущностей, напоминающих функции, которые принимают значение в качестве аргумента и возвращают `true` или `false`. Как вы увидите далее, в Java 8 можно было бы написать и `Function<Apple, Boolean>` — это более привычная форма для тех из наших читателей, кому в школе рассказывали про функции, а не о предикатах, но форма `Predicate<Apple>` — общепринятая (и немного более эффективная, поскольку не требует преобразования примитивного типа `boolean` в объект типа `Boolean`).

1.3.3. От передачи методов к лямбда-выражениям

Передавать методы как значения, безусловно, удобно, хотя весьма досаждает необходимость описывать даже короткие методы, такие как `isHeavyApple` и `isGreenApple`, которые используются только один-два раза. Но Java 8 решает и эту проблему. В нем появилась новая нотация (анонимные функции, называемые также лямбда-выражениями), благодаря которой можно написать просто:

```
filterApples(inventory, (Apple a) -> GREEN.equals(a.getColor()) );
```

или:

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

или даже:

```
filterApples(inventory, (Apple a) -> a.getWeight() < 80 ||
                                     RED.equals(a.getColor()) );
```

Используемый однократно метод не требуется описывать; код становится четче и понятнее, поскольку не нужно тратить время на поиски по исходному коду, чтобы понять, какой именно код передается.

Но если размер подобного лямбда-выражения превышает несколько строк (и его поведение сразу не понятно), лучше воспользоваться вместо анонимной лямбды ссылкой на метод с наглядным именем. Понятность кода — превыше всего.

Создатели Java 8 могли на этом практически завершить свою работу и, наверное, так бы и поступили, если бы не многоядерные процессоры. Как вы увидите, функциональное программирование в представленном нами объеме обладает весьма широкими возможностями. Язык Java мог бы в качестве вишенки на торте добавить обобщенный библиотечный метод `filter` и несколько родственных ему, например:

```
static <T> Collection<T> filter(Collection<T> c, Predicate<T> p);
```

Вам не пришлось бы даже писать самим такие методы, как `filterApples`, поскольку даже предыдущий вызов:

```
filterApples(inventory, (Apple a) -> a.getWeight() > 150 );
```

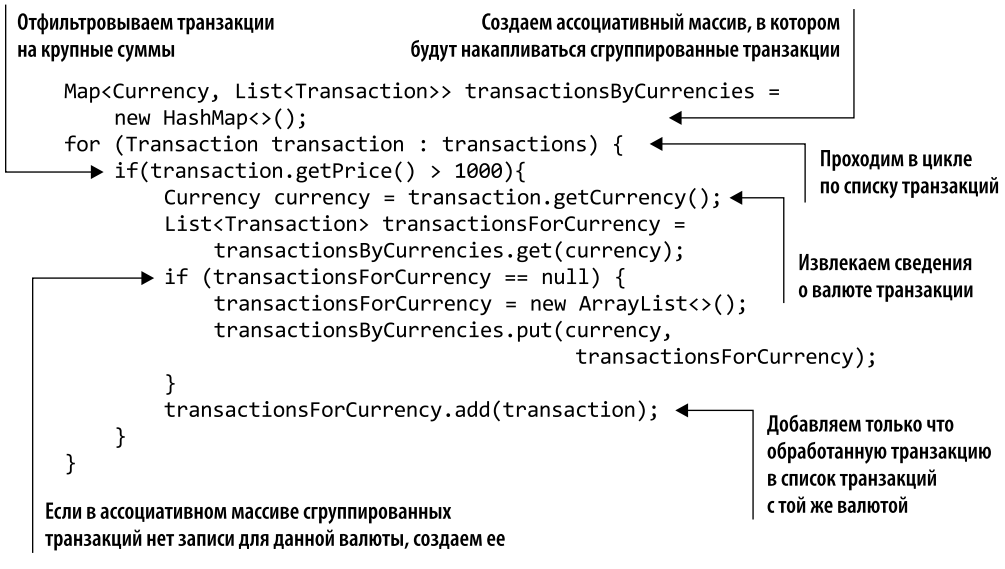
можно было бы переписать в виде обращения к библиотечному методу `filter`:

```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

Но из соображений оптимального использования параллелизма создатели Java не пошли по этому пути. Вместо этого Java 8 включает новый Stream API (потоков данных, сходных с коллекциями), содержащий обширный набор подобных `filter`-операций, привычных функциональным программистам (например, `map` и `reduce`), а также методы для преобразования коллекций в потоки данных и наоборот. Этот API мы сейчас и обсудим.

1.4. Потоки данных

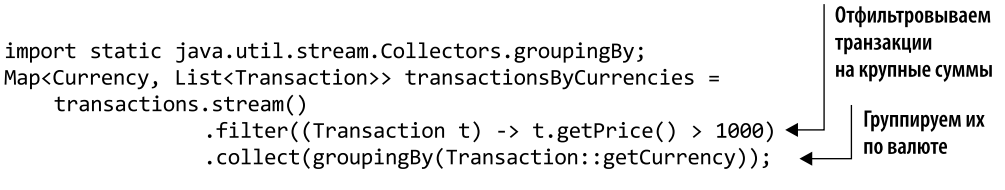
Практически любое приложение Java *создает* и *обрабатывает* коллекции. Однако работать с коллекциями не всегда удобно. Например, представьте себе, что вам нужно отфильтровать из списка транзакции на крупные суммы, а затем сгруппировать их по валюте. Для реализации подобного запроса по обработке данных вам пришлось бы написать огромное количество стереотипного кода, как показано ниже:




```
Map<Currency, List<Transaction>> transactionsByCurrencies =
    new HashMap<>();
for (Transaction transaction : transactions) {
    if(transaction.getPrice() > 1000){
        Currency currency = transaction.getCurrency();
        List<Transaction> transactionsForCurrency =
            transactionsByCurrencies.get(currency);
        if (transactionsForCurrency == null) {
            transactionsForCurrency = new ArrayList<>();
            transactionsByCurrencies.put(currency,
                                         transactionsForCurrency);
        }
        transactionsForCurrency.add(transaction);
    }
}
```

Помимо прочего, весьма непросто понять с первого взгляда, что этот код делает, из-за многочисленных вложенных операторов управления потоком выполнения.

С помощью Stream API можно решить ту же задачу следующим образом:




```
import static java.util.stream.Collectors.groupingBy;
Map<Currency, List<Transaction>> transactionsByCurrencies =
    transactions.stream()
                .filter((Transaction t) -> t.getPrice() > 1000)
                .collect(groupingBy(Transaction::getCurrency));
```

Не стоит переживать, если этот код пока кажется вам своего рода магией. Главы 4–7 помогут вам разобраться со Stream API. Пока стоит отметить лишь, что этот API позволяет обрабатывать данные несколько иначе, чем Collection API. При использовании коллекции вам приходится организовывать цикл самостоятельно: пройтись по всем элементам, обрабатывая их по очереди в цикле `for-each`. Подобная итерация по данным называется *внешней итерацией* (external iteration). При использовании же Stream API думать о циклах не нужно. Вся обработка данных происходит внутри библиотеки. Мы будем называть это *внутренней итерацией* (internal iteration).

Вторая основная проблема при работе с коллекциями: как обработать список транзакций, если их очень много? Одноядерный процессор не сможет обработать такой большой объем данных, но, вероятно, процессор вашего компьютера — многоядерный. Оптимально было бы разделить объем работ между ядрами процессора, чтобы уменьшить время обработки. Теоретически при наличии восьми ядер обработка данных должна занять в восемь раз меньше времени, поскольку они работают параллельно.

Многоядерные компьютеры

Все новые настольные компьютеры и ноутбуки — многоядерные. Они содержат не один, а несколько процессоров (обычно называемых ядрами). Проблема в том, что обычная программа на языке Java использует лишь одно из этих ядер, а остальные простаивают. Аналогично во многих компаниях для эффективной обработки больших объемов данных применяются *вычислительные кластеры* (computing clusters) — компьютеры, соединенные быстрыми сетями. Java 8 продвигает новые стили программирования, чтобы лучше использовать подобные компьютеры.

Поисковый движок Google — пример кода, который слишком велик для работы на отдельной машине. Он читает все страницы в Интернете и создает индекс соответствия всех встречающихся на каждой странице слов адресу (URL) этой страницы. Далее при поиске в Google по нескольким словам ПО может воспользоваться этим индексом для быстрой выдачи набора страниц, содержащих эти слова. Попробуйте представить себе реализацию этого алгоритма на языке Java (даже в случае гораздо меньшего индекса, чем у Google, вам придется использовать все процессорные ядра вашего компьютера).

1.4.1. Многопоточность — трудная задача

Проблема в том, что реализовать параллелизм, прибегнув к *многопоточному* (multithreaded) коду (с помощью Thread API из предыдущих версий Java), — довольно непростая задача. Лучше воспользоваться другим способом, ведь потоки выполнения могут одновременно обращаться к разделяемым переменным и изменять их. В результате данные могут меняться самым неожиданным образом, если их использование не согласовано, как полагается[1]. Такая модель сложнее для понимания[2], чем последовательная, пошаговая модель. Например, одна из возможных проблем с двумя (не синхронизированными должным образом) потоками выполнения, которые пытаются прибавить число к разделяемой переменной `sum`, приведена на рис. 1.5.

---

[1] Обычно это делают с помощью ключевого слова synchronized, но если указать его не там, где нужно, то может возникнуть множество ошибок, которые трудно обнаружить. Параллелизм на основе потоков данных Java 8 поощряет программирование в функциональном стиле, при котором synchronized используется редко; упор делается на секционировании данных, а не на координации доступа к ним.

[2] А вот и одна из причин, побуждающих язык эволюционировать!

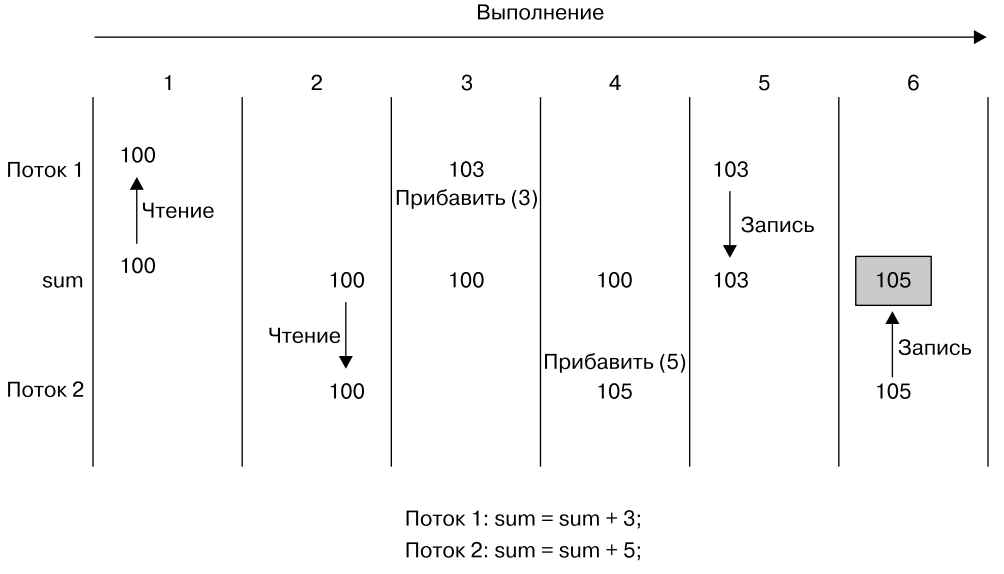


Рис. 1.5. Возможная проблема с двумя потоками выполнения, пытающимися прибавить число к разделяемой переменной sum. Результат равен 105 вместо ожидаемых 108

Java 8 решает обе проблемы (стереотипность и малопонятность кода обработки коллекций, а также трудности использования многоядерности) с помощью Stream API (`java.util.stream`). Первый фактор, определивший архитектуру этого API, — существование множества часто встречающихся паттернов обработки данных (таких как `filterApples` из предыдущего раздела или операции, знакомые вам по языкам запросов вроде SQL), которые разумно было бы поместить в библиотеку: *фильтрация* данных по какому-либо критерию (например, выбор тяжелых яблок), *извлечение* (например, извлечение значения поля «вес» из всех яблок в списке) или *группировка* данных (например, группировка списка чисел по отдельным спискам четных и нечетных чисел) и т. д. Второй фактор — часто встречающаяся возможность распараллеливания подобных операций. Например, как показано на рис. 1.6, фильтрацию списка на двух процессорах можно осуществить путем его разбиения на две половины, одну из которых будет обрабатывать один процессор, а вторую — другой. Это называется *шагом ветвления* (forking step) ❶. Далее процессоры фильтруют свои половины списка ❷. И наконец ❸, один из процессоров объединяет результаты (это очень похоже на алгоритм, благодаря которому поиск в Google так быстро работает, хотя там используется гораздо больше двух процессоров).

Пока мы скажем только, что новый Stream API ведет себя аналогично Collection API: оба обеспечивают доступ к последовательностям элементов данных. Но запомните: Collection API в основном связан с хранением и доступом к данным, а Stream API — с описанием производимых над данными вычислений. Важнее всего то, что Stream API позволяет и поощряет параллельную обработку элементов потока. Хотя

на первый взгляд это может показаться странным, но часто самый быстрый способ фильтрации коллекции (например, использовать `filterApples` из предыдущего раздела для списка) — преобразовать ее в поток данных, обработать параллельно и затем преобразовать обратно в список. Мы снова скажем «параллелизм практически даром» и покажем вам, как отфильтровать тяжелые яблоки из списка последовательно и как — параллельно, с помощью потоков данных и лямбда-выражения.

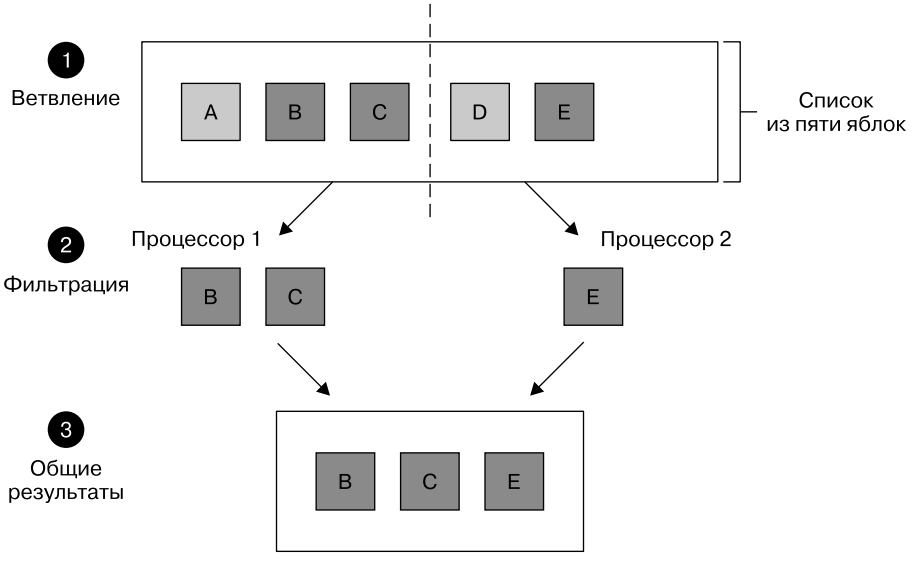


Рис. 1.6. Ветвление filter на два процессора и объединение результатов

Вот пример последовательной обработки:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
```

А вот пример параллельной обработки:

```
import static java.util.stream.Collectors.toList;
List<Apple> heavyApples =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

Параллелизм в языке Java и отсутствие разделяемого изменяемого состояния

Параллелизм в Java всегда считался непростой задачей, а ключевое слово `synchronized` — источником потенциальных ошибок. Так что же за чудодейственное средство появилось в Java 8?

На самом деле чудодейственных средств два. Во-первых, библиотека, обеспечивающая секционирование — разбиение больших потоков данных на несколько маленьких, для параллельной обработки. Во-вторых, параллелизм «практически даром» с помощью потоков возможен только в том случае, если методы, передаваемые библиотечным методам, таким как `filter`, не взаимодействуют между собой (например, через изменяемые разделяемые объекты). Но оказывается, что это ограничение представляется программистам вполне естественным (см. в качестве примера наш `Apple::isGreenApple`). Хотя основной смысл слова *«функциональный»* во фразе *«функциональное программирование»* — «использующий функции в качестве полноправных значений», у него часто есть оттенок «без взаимодействия между компонентами во время выполнения».

В главе 7 вы найдете более подробное обсуждение параллельной обработки данных в Java 8 и вопросов ее производительности. Одна из проблем, с которыми столкнулись на практике разработчики Java, несмотря на все ее замечательные новинки, касалась эволюции существующих интерфейсов. Например, метод `Collections.sort` относится к интерфейсу `List`, но так и не был в него включен. В идеале хотелось бы иметь возможность написать `list.sort(comparator)` вместо `Collections.sort(list, comparator)`. Это может показаться тривиальным, но до Java 8 для обновления интерфейса необходимо было обновить все реализующие его классы — просто логистический кошмар! Данная проблема решена в Java 8 с помощью *методов с реализацией по умолчанию*.

1.5. Методы с реализацией по умолчанию и модули Java

Как мы уже упоминали ранее, современные системы обычно строятся из компонентов — возможно, приобретенных на стороне. Исторически в языке Java не было поддержки этой возможности, за исключением JAR-файлов, хранящих набор Java-пакетов без четкой структуры. Более того, развивать интерфейсы, содержащиеся в таких пакетах, было непросто — изменение Java-интерфейса означало необходимость изменения каждого реализующего его класса. Java 8 и 9 вступили на путь решения этой проблемы.

Во-первых, в Java 9 есть система модулей и синтаксис для описания *модулей*, содержащих наборы пакетов, а контроль видимости и пространств имен значительно усовершенствован. Благодаря модулям у простого компонента типа JAR появляется структура, как для пользовательской документации, так и для автоматической проверки (мы расскажем об этом подробнее в главе 14). Во-вторых, в Java 8 появились методы с реализацией по умолчанию для поддержки *изменяемых интерфейсов*. Мы рассмотрим их подробнее в главе 13. Знать про них важно, ведь они часто будут встречаться вам в интерфейсах, но, поскольку относительно немногим программистам приходится самим писать методы с реализацией по умолчанию, а также поскольку они скорее способствуют эволюции программ, а не

помогают написанию какой-либо конкретной программы, мы расскажем здесь о них кратко, на примере.

В разделе 1.4 приводился следующий пример кода Java 8:

```
List<Apple> heavyApples1 =
    inventory.stream().filter((Apple a) -> a.getWeight() > 150)
                      .collect(toList());
List<Apple> heavyApples2 =
    inventory.parallelStream().filter((Apple a) -> a.getWeight() > 150)
                              .collect(toList());
```

С этим кодом есть проблема: до Java 8 в интерфейсе `List<T>` не было методов `stream` или `parallelStream` — как и в интерфейсе `Collection<T>`, который он реализует, — поскольку эти методы еще не придумали. А без них такой код не скомпилируется. Простейшее решение для создателей Java 8, какое вы могли бы использовать для своих интерфейсов, — добавить метод `stream` в интерфейс `Collection` и его реализацию в класс `ArrayList`.

Но это стало бы настоящим кошмаром для пользователей, ведь множество разных фреймворков коллекций реализуют интерфейсы из Collection API. Добавление нового метода в интерфейс означает необходимость реализации его во всех конкретных классах. У создателей языка нет контроля над существующими реализациями интерфейса `Collection`, так что возникает дилемма: как развивать опубликованные интерфейсы, не нарушая при этом существующие реализации?

Решение, предлагаемое Java 8, заключается в разрыве последнего звена этой цепочки: отныне интерфейсы могут содержать сигнатуры методов, которые не реализуются в классе-реализации. Но кто же тогда их реализует? Отсутствующие тела методов описываются в интерфейсе (поэтому и называются реализациями по умолчанию), а не в реализующем классе.

Благодаря этому у разработчика интерфейса появляется способ увеличить интерфейс, выйдя за пределы изначально планировавшихся методов и не нарушая работу существующего кода. Для этого в Java 8 в спецификациях интерфейсов можно использовать уже существующее ключевое слово `default`.

Например, в Java 8 можно вызвать метод `sort` непосредственно для списка. Это возможно благодаря следующему методу с реализацией по умолчанию из интерфейса `List`, вызывающему статический метод `Collections.sort`:

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

Это значит, что конкретные классы интерфейса `List` не обязаны явным образом реализовывать метод `sort`, в то время как в предыдущих версиях Java без такой реализации они бы не скомпилировались.

Но подождите секундочку. Один класс может реализовывать несколько интерфейсов, так? Не будут ли несколько реализаций по умолчанию в нескольких интерфейсах означать возможность своего рода множественного наследования в Java? Да, до некоторой степени. В главе 13 мы обсудим правила, предотвращающие такие проблемы, как печально известная *проблема ромбовидного наследования* (diamond inheritance problem) в языке C++.

1.6. Другие удачные идеи, заимствованные из функционального программирования

В предыдущих разделах мы познакомили вас с двумя основными идеями функционального программирования, которые ныне стали частью Java: с использованием методов и лямбда-выражений как полноправных значений, а также с идеей эффективного и безопасного параллельного выполнения функций и методов при условии отсутствия изменяемого разделяемого состояния. Обе эти идеи были воплощены в описанном выше новом Stream API.

В распространенных функциональных языках (SML, OCaml, Haskell) есть и другие конструкции, помогающие программистам в их работе. В их числе возможность обойтись без `null` за счет более явных типов данных. Тони Хоар (Tony Hoare), один из корифеев теории вычислительной техники, сказал на презентации во время конференции QCon в Лондоне в 2009 году:

> *«Я считаю это своей ошибкой на миллиард долларов: изобретение ссылки на null в 1965-м... Я поддался искушению включить в язык ссылку на null просто потому, что ее реализация была так проста».*

В Java 8 появился класс `Optional<T>`, который при систематическом использовании помогает избегать исключений, связанных с указателем на `null`. Это объект-контейнер, который может содержать или не содержать значение. В `Optional<T>` есть методы для обработки явным образом варианта отсутствия значения, в результате чего можно избежать исключений, связанных с указателем на `null`. С помощью системы типов он дает возможность отметить, что у переменной может потенциально отсутствовать значение. Мы обсудим класс `Optional<T>` подробнее в главе 11.

Еще одна идея — *(структурное) сопоставление с шаблоном*[1], используемое в математике. Например:

```
f(0) = 1
в противном случае f(n) = n*f(n-1)
```

В Java для этого можно написать оператор `if-then-else` или `switch`. Другие языки продемонстрировали, что в случае более сложных типов данных сопоставление с шаблоном позволяет выражать идеи программирования лаконичнее по сравнению с оператором `if-then-else`. Для таких типов данных в качестве альтернативы `if-then-else` можно также использовать полиморфизм и переопределение методов, но дискуссия относительно того, что уместнее, все еще продолжается[2]. Мы полагаем,

---

[1] Эту фразу можно толковать двояко. Одно из толкований знакомо нам из математики и функционального программирования, в соответствии с ним функция определяется операторами case, а не с помощью конструкции if-then-else. Второе толкование относится к фразам типа «найти все файлы вида 'IMG*.JPG' в заданном каталоге» и связано с так называемыми регулярными выражениями.

[2] Введение в эту дискуссию можно найти в статье «Википедии», посвященной «проблеме выражения» (expression problem — термин, придуманный Филом Уэдлером (Phil Wadler)).

что и то и другое — полезные инструменты, которые не будут лишними в вашем арсенале. К сожалению, Java 8 не полностью поддерживает сопоставление с шаблоном, хотя в главе 19 мы покажем, как его можно выразить на этом языке. В настоящий момент обсуждается предложение по расширению Java — добавление в будущие версии языка поддержки сопоставления с шаблоном (см. http://openjdk.java.net/jeps/305). Тем временем в качестве иллюстрации приведем пример на языке программирования Scala (еще один Java-подобный язык, использующий JVM и послуживший источником вдохновения для некоторых аспектов эволюции Java; см. главу 20). Допустим, вы хотите написать программу для элементарных упрощений дерева, представляющего арифметическое выражение. В Scala можно написать следующий код для декомпозиции объекта типа `Expr`, служащего для представления подобных выражений, с последующим возвратом другого объекта `Expr`:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    case BinOp("-", e, Number(0)) => e
    case BinOp("*", e, Number(1)) => e
    case BinOp("/", e, Number(1)) => e
    case _ => expr
}
```

Прибавляем 0

Вычитаем 0

Умножаем на 1

Делим на 1

С помощью вышеприведенных операторов case это выражение не упрощается, так что не трогаем его

Синтаксис `expr match` языка Scala соответствует `switch (expr)` в языке Java. Пока не задумывайтесь над этим кодом — мы расскажем о сопоставлении с шаблоном подробнее в главе 19. Сейчас можете считать сопоставление с шаблоном просто расширенным вариантом `switch`, способным в то же время разбивать тип данных на компоненты.

Но почему нужно ограничивать оператор `switch` в Java простыми типами данных и строками? Функциональные языки программирования обычно позволяют использовать `switch` для множества других типов данных, включая возможность сопоставления с шаблоном (в коде на языке Scala это делается с помощью операции `match`). Для прохода по объектам семейства классов (например, различных компонентов автомобиля: колес (`Wheel`), двигателя (`Engine`), шасси (`Chassis`) и т. д.) с применением какой-либо операции к каждому из них в объектно-ориентированной архитектуре часто применяется паттерн проектирования «Посетитель» (Visitor). Одно из преимуществ сопоставления с шаблоном — возможность для компилятора выявлять типичные ошибки вида: «Класс `Brakes` — часть семейства классов, используемых для представления компонентов класса `Car`. Вы забыли обработать его явным образом».

Главы 18 и 19 представляют собой полное введение в функциональное программирование и написание программ в функциональном стиле на языке Java 8, включая описание набора имеющихся в его библиотеке функций. Глава 20 развивает эту тему сравнением возможностей Java 8 с возможностями языка Scala — языка, в основе которого также лежит использование JVM и который эволюционировал настолько быстро, что уже претендует на часть ниши Java в экосистеме языков программирования. Мы разместили этот материал ближе к концу книги, чтобы вам было понятнее, почему были добавлены те или иные новые возможности Java 8 и Java 9.

Возможности Java 8, 9, 10 и 11: с чего начать?

В Java 8, как и в Java 9, были внесены значительные изменения. Но повседневную практику Java-программистов в масштабе мелких фрагментов кода, вероятно, сильнее всего затронут дополнения, внесенные в восьмую версию: идея передачи метода или лямбда-выражения быстро становится жизненно важным знанием в Java. Напротив, усовершенствования Java 9 расширяют возможности описания и использования крупных компонентов, идет ли речь о модульном структурировании системы или импорте набора инструментов для реактивного программирования. Наконец, Java 10 вносит намного меньшие изменения, по сравнению с предыдущими двумя версиями, — они состоят из правил вывода типов для локальных переменных, которые мы вкратце обсудим в главе 21, где также упомянем связанный с ними расширенный синтаксис для аргументов лямбда-выражений, который введен в Java 11.

Java 11 был выпущен в сентябре 2018 года и включает обновленную асинхронную клиентскую библиотеку HTTP (http://openjdk.java.net/jeps/321), использующую новые возможности Java 8 и 9 (подробности см. в главах 15–17) — `CompletableFuture` и реактивное программирование.

Резюме

- Всегда учитывайте идею экосистемы языков программирования и следующее из нее бремя необходимости для языков эволюционировать или увядать. Хотя на текущий момент Java более чем процветающий язык, можно вспомнить и другие процветавшие языки, такие как COBOL, которые не сумели эволюционировать.
- Основные новшества Java 8 включают новые концепции и функциональность, облегчающую написание лаконичных и эффективных программ.
- Возможности многоядерных процессоров плохо использовались в Java до версии 8.
- Функции стали полноправными значениями; методы теперь можно передавать в виде функциональных значений. Обратите также внимание на возможность написания анонимных функций (лямбда-выражений).
- Концепция потоков данных Java 8 во многом обобщает понятие коллекций, но зачастую позволяет получить более удобочитаемый код и обрабатывать элементы потока параллельно.
- Программирование с использованием крупномасштабных компонентов, а также эволюция интерфейсов системы исторически плохо поддерживались языком Java. Методы с реализацией по умолчанию появились в Java 8. Они позволяют расширять интерфейс без изменения всех реализующих его классов.
- В числе других интересных идей из области функционального программирования — обработка `null` и сопоставление с шаблоном.

2 Передача кода и параметризация поведения

В этой главе

- Адаптация к меняющимся требованиям.
- Параметризация поведения.
- Анонимные классы.
- Первое знакомство с лямбда-выражениями.
- Примеры из практики: `Comparator`, `Runnable` и GUI.

Одна из широко известных проблем в сфере разработки программного обеспечения заключается в том, что, независимо от ваших действий, требования пользователя к ПО все равно поменяются. Представьте себе приложение для фермера, который хотел бы лучше ориентироваться в своих запасах. Например, он может захотеть иметь возможность искать в своих запасах все зеленые яблоки. Но на следующий же день передумать и сказать: «Знаешь, на самом деле мне хотелось бы найти и все яблоки тяжелее 150 г». А еще через два дня он вернется и добавит: «Было бы также неплохо найти все яблоки, которые зеленые *и* тяжелее 150 г». Как справиться со всеми этими постоянно меняющимися требованиями? В идеале желательно минимизировать выполняемую работу. Кроме того, хотелось бы простоты реализации и сопровождения новой функциональности в долгосрочной перспективе.

Параметризация поведения (behavior parameterization) — паттерн разработки программного обеспечения, нацеленный на решение проблемы частых изменений требований. Если вкратце, параметризация поведения означает обеспечение доступности блока кода без его выполнения. Этот блок кода может вызываться далее другими частями приложения, то есть его выполнение можно отложить. В качестве

примера можно привести передачу блока кода в виде аргумента методу, который выполнит этот блок позднее. В результате поведение метода параметризуется этим блоком кода. Например, при обработке коллекции может понадобиться написать метод, который:

- выполняет какое-либо действие над каждым элементом списка;
- выполняет какое-либо другое действие по окончании обработки списка;
- выполняет какое-нибудь еще действие в случае ошибки.

Вот это и есть *параметризация поведения*. Проведем аналогию: ваш сосед по комнате знает дорогу к супермаркету и обратно. Вы можете попросить его купить некоторые продукты: хлеб, сыр и вино, что эквивалентно вызову метода `goAndBuy` с передачей списка продуктов в качестве аргумента. Но в один прекрасный день вы, будучи еще в офисе, хотите попросить его выполнить нечто, что он никогда раньше не делал, — забрать посылку на почте. Вам нужно передать ему список указаний: пойти на почту, поговорить с менеджером, назвать номер почтового отправления и забрать посылку. Вы можете отправить ему список указаний, которым ему нужно будет следовать, по электронной почте. Это несколько сложнее, чем покупка продуктов, и эквивалентно передаче методу `goAndDo` нового варианта поведения в виде аргумента.

Мы начнем эту главу с демонстрации примера повышения приспособляемости кода к меняющимся требованиям. А затем на основе этих знаний покажем, как использовать параметризацию поведения, на нескольких примерах из реальной практики. Возможно, вы уже применяли паттерн параметризации поведения при использовании существующих классов и API Java для сортировки списка, фильтрации названий файлов, указания экземпляру `Thread` выполнить определенный блок кода или даже для обработки событий GUI. Как вы скоро поймете, исторически этот паттерн требует в Java написания объемного кода. Но отныне, в Java 8, эта проблема решена благодаря лямбда-выражениям. В главе 3 мы покажем, как создавать лямбда-выражения, где их использовать и как повысить с их помощью лаконичность своего кода.

2.1. Как адаптироваться к меняющимся требованиям

Писать код, способный справляться с меняющимися требованиями, — непростая задача. Рассмотрим пример, который мы будем постепенно совершенствовать, попутно демонстрируя рекомендуемые практики для повышения гибкости кода. В контексте приложения для учета запросов на ферме нам нужно реализовать функциональность фильтрации *зеленых* яблок из списка. Проще не бывает, правда?

2.1.1. Первая попытка: фильтрация зеленых яблок

Пусть, как и в главе 1, различные цвета яблок представлены у нас с помощью перечисляемого типа `Color`:

```
enum Color { RED, GREEN }
```

Первое приходящее на ум решение может быть таким:

```
public static List<Apple> filterGreenApples(List<Apple> inventory) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if( GREEN.equals(apple.getColor()) ) {
            result.add(apple);
        }
    }
    return result;
}
```

Список для накопления яблок

Выбираем только зеленые яблоки

Выделенный жирным шрифтом фрагмент кода содержит условие для выбора зеленых яблок. Можно считать, что перечисляемый тип `Color` с набором цветов, например `GREEN`, у нас есть. Но фермер вдруг передумал и хочет теперь фильтровать и *красные* яблоки. Что делать? Простейшим решением было бы скопировать ваш метод, переименовав копию в `filterRedApples`, и изменить условный оператор `if` так, чтобы выбирать красные яблоки. Однако такой подход окажется неудачным, если требования снова изменятся и фермер захочет выбирать яблоки нескольких цветов. Рекомендуемая практика такова: старайтесь использовать абстрагирование везде, где замечаете, что пишете практически идентичный код.

2.1.2. Вторая попытка: параметризация цвета

Как же избежать дублирования большей части кода метода `filterGreenApples` при создании `filterRedApples`? Вы можете добавить в свой метод еще один параметр, чтобы параметризовать цвет яблок и повысить приспособляемость кода к подобным изменениям требований:

```
public static List<Apple> filterApplesByColor(List<Apple> inventory,
Color color) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( apple.getColor().equals(color) ) {
            result.add(apple);
        }
    }
    return result;
}
```

Теперь вы можете полностью удовлетворить пожелания вашего фермера, вызывая метод следующим образом:

```
List<Apple> greenApples = filterApplesByColor(inventory, GREEN);
List<Apple> redApples = filterApplesByColor(inventory, RED);
...
```

Слишком просто, да? Немного усложним пример. Фермер вернулся и сказал вам: «Было бы просто чудесно различать тяжелые и легкие яблоки. Тяжелые яблоки обычно весят больше 150 г».

Но вы, как опытный разработчик, заранее предположили, что фермер может захотеть различать яблоки по весу. И создали следующий метод с дополнительным параметром для учета веса яблок:

```
public static List<Apple> filterApplesByWeight(List<Apple> inventory,
int weight) {
    List<Apple> result = new ArrayList<>();
    For (Apple apple: inventory){
        if ( apple.getWeight() > weight ) {
            result.add(apple);
        }
    }
    return result;
}
```

Это неплохое решение, но обратите внимание, что вам пришлось продублировать бóльшую часть реализации обхода списка и применения критерия фильтрации. Это плохо, поскольку нарушает принцип DRY (Don't Repeat Yourself — «не повторяйся») разработки программного обеспечения. Ведь если вам нужно будет изменить обход фильтра для повышения производительности, придется модифицировать реализации *всех* методов вместо одного. Слишком большие затраты, если говорить об объеме работы.

Конечно, можно объединить обработку цвета и веса в один метод `filter`. Но как тогда отличить, по какому атрибуту необходимо фильтровать? Можно добавить флаг, чтобы различать запросы на фильтрацию по цвету и по весу.

Никогда так не делайте! Вскоре мы объясним почему.

2.1.3. Третья попытка: фильтрация по всем возможным атрибутам

Неуклюжая попытка слияния всех атрибутов могла бы выглядеть вот так:

```
public static List<Apple> filterApples(List<Apple> inventory, Color color,
                                       int weight, boolean flag) {
    List<Apple> result = new ArrayList<>();
    for (Apple apple: inventory) {
        if ( (flag && apple.getColor().equals(color)) ||
             (!flag && apple.getWeight() > weight) ){
            result.add(apple);
        }
    }
    return result;
}
```

Неудачный способ выбора цвета или веса

А использовать этот метод можно было бы следующим, весьма неудачным образом:

```
List<Apple> greenApples = filterApples(inventory, GREEN, 0, true);
List<Apple> heavyApples = filterApples(inventory, null, 150, false);
...
```

Это исключительно неудачное решение. Во-первых, клиентский код выглядит ужасно. Что означают `true` и `false`? Кроме того, это решение плохо адаптируется

к смене требований. Что, если фермеру понадобится фильтрация по различным атрибутам яблок, например по размеру, форме, месту происхождения и т. д.? Кроме того, фермеру могут понадобиться более сложные запросы с комбинацией атрибутов, например тяжелые зеленые яблоки. Вам придется написать или несколько дублирующихся методов `filter`, или один огромный запутанный метод. Пока мы параметризовали метод `filterApples` *значениями* типов `String`, `Integer`, перечисляемым типом и `boolean`. Для некоторых четко сформулированных задач этого достаточно. Но в данном случае нам нужен более продвинутый способ передачи методу `filterApples` критериев выбора яблок. В следующем разделе мы расскажем, как достичь необходимой адаптивности с помощью *параметризации поведения*.

2.2. Параметризация поведения

В предыдущем разделе вы видели, что для адаптации к меняющимся требованиям нужен способ более продвинутый, чем добавление множества параметров метода. Отступим на шаг назад и найдем оптимальный уровень абстракции. Одно из возможных решений состоит в моделировании критериев отбора — это работа с объектами-яблоками и возврат `boolean` в зависимости от каких-либо атрибутов объекта `Apple`. Например, зеленое ли яблоко? Весит ли яблоко более 150 г? Это называется *предикатом*, или *условием* (функцией, возвращающей булево значение). Опишем интерфейс для моделирования критериев отбора:

```
public interface ApplePredicate{
    boolean test (Apple apple);
}
```

Теперь мы можем объявить несколько реализаций интерфейса `ApplePredicate`, которые будут соответствовать различным критериям отбора, как показано в следующем фрагменте кода (и проиллюстрировано на рис. 2.1):

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
```

Выбираем только тяжелые яблоки

Выбираем только зеленые яблоки

Эти критерии можно рассматривать как различные варианты поведения метода `filter`. Фактически речь идет о реализации паттерна проектирования «Стратегия» (см. https://ru.wikipedia.org/wiki/Стратегия_(шаблон_проектирования)), который позволяет описать семейство алгоритмов, инкапсулируя каждый из них (называемый стратегией), и выбирать нужный во время выполнения. В данном случае семейство алгоритмов — `ApplePredicate` со стратегиями `AppleHeavyWeightPredicate` и `AppleGreenColorPredicate`.

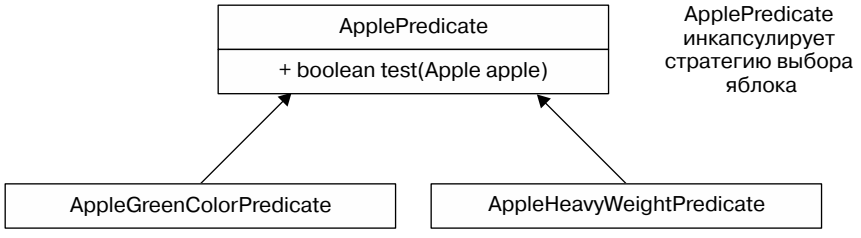


Рис. 2.1. Различные стратегии выбора объектов Apple

Но как воспользоваться различными реализациями `ApplePredicate`? Для этого нужно, чтобы метод `filterApples` принимал на входе объекты `ApplePredicate` для последующей проверки того, соответствует ли конкретный объект `Apple` условию. *Параметризация поведения* — это способность метода принимать в качестве параметров множество вариантов поведения, а затем, опираясь на них, воплощать различные последовательности операций.

Чтобы добиться такого эффекта в текущем примере, мы добавим в метод `filterApples` параметр для объекта `ApplePredicate`. Это действие дает вам как разработчику программного обеспечения огромное преимущество: теперь вы можете отделить логику организации цикла по коллекции внутри метода `filterApples` от варианта поведения, используемого для каждого элемента этой коллекции (в данном случае предиката).

2.2.1. Четвертая попытка: фильтрация по абстрактному критерию

Наш модифицированный метод `filter`, использующий `ApplePredicate`, выглядит следующим образом:

```
public static List<Apple> filterApples(List<Apple> inventory,
                                       ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory) {
        if(p.test(apple)) {
            result.add(apple);
        }
    }
    return result;
}
```

Предикат p инкапсулирует условие, проверяемое для каждого яблока

Передача кода/поведения

Стоит остановиться на минуту и отпраздновать наши достижения. Гибкость кода намного выше, чем при нашей первой попытке, вдобавок его намного легче читать и использовать! Вы можете теперь создавать различные объекты `ApplePredicate` и передавать их в метод `filterApples`. Абсолютная гибкость! Например, если фермер попросит вас найти все красные яблоки тяжелее 150 г, вам достаточно будет написать класс, реализующий `ApplePredicate` соответствующим образом. Наш код уже достаточно гибок для любой смены требований, касающейся атрибутов `Apple`:

```
public class AppleRedAndHeavyPredicate implements ApplePredicate {
        public boolean test(Apple apple){
                return RED.equals(apple.getColor())
                        && apple.getWeight() > 150;
        }
}
List<Apple> redAndHeavyApples =
    filterApples(inventory, new AppleRedAndHeavyPredicate());
```

Это по-настоящему круто: поведение метода `filterApples` зависит теперь от *передаваемого вами* в него через объект `ApplePredicate` кода. Мы параметризовали поведение метода `filterApples`!

Обратите внимание, что в предыдущем примере важен только код реализации метода `test`, как показано на рис. 2.2; именно он определяет поведение метода `filterApples`. К сожалению, в силу того, что метод `filterApples` может принимать в качестве параметра только объекты, приходится оборачивать данный код в объект `ApplePredicate`. Это схоже с передачей встраиваемого кода, поскольку мы передаем булево выражение через объект, реализующий метод `test`. В разделе 2.3 (а более подробно — в главе 3) вы увидите, что благодаря лямбда-выражениям можно непосредственно передать методу `filterApples` выражение `RED.equals(apple.getColor()) && apple.getWeight() > 150` без необходимости описывать несколько классов `ApplePredicate`. Это повышает лаконичность кода.

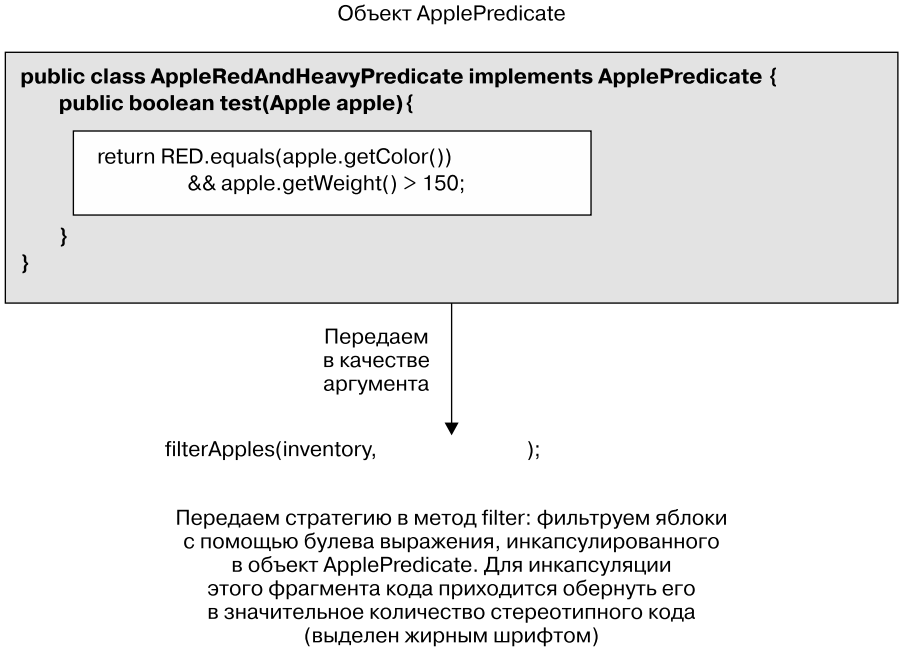


Рис. 2.2. Параметризация поведения метода filterApples и передача ему различных стратегий фильтрации

Различные варианты поведения, один параметр

Как мы уже поясняли ранее, параметризация поведения хороша тем, что позволяет отделять логику организации цикла по коллекции от используемого для каждого из элементов коллекции варианта поведения. В результате появляется возможность переиспользования метода с различными вариантами поведения для решения разных задач, как показано на рис. 2.3. Именно поэтому параметризация поведения — полезный инструмент, который пригодится вам для создания гибких API.

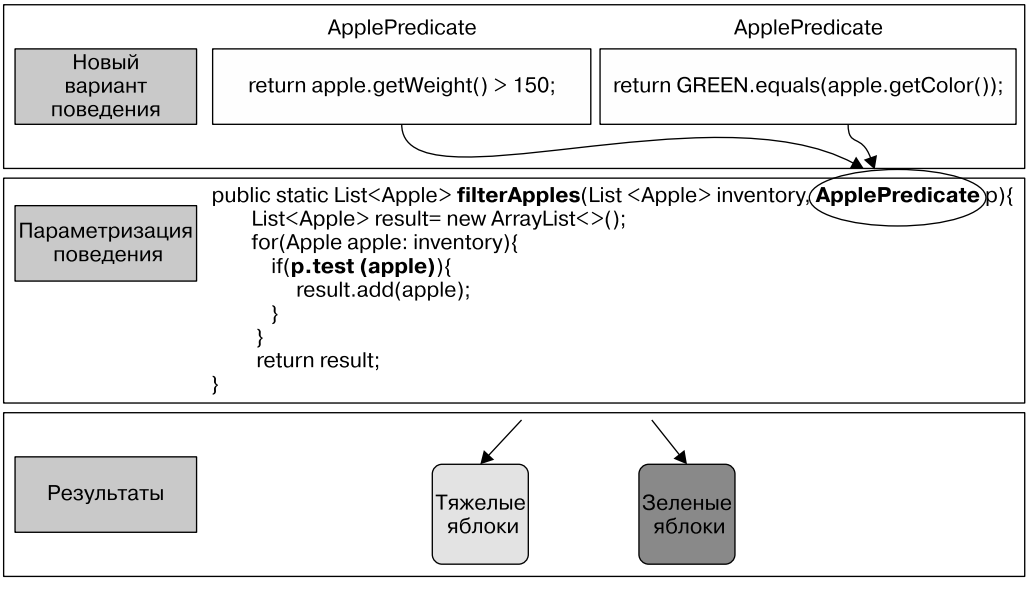


Рис. 2.3. Параметризация поведения метода filterApples и передача ему различных стратегий фильтрации

Чтобы проверить свое умение использовать параметризацию поведения, попробуйте выполнить контрольное задание 2.1!

Контрольное задание 2.1. Напишите гибкий метод prettyPrintApple

Напишите метод `prettyPrintApple`, который принимает на входе список `List` объектов типа `Apple` и который можно параметризовать несколькими способами, для генерации на основе объекта `apple` результата в виде `String` (что-то вроде множества специализированных методов `toString`). Например, от метода `prettyPrintApple` можно потребовать вывести только вес каждого яблока, а можно выводить информацию по каждому яблоку отдельно, указывая, тяжелое оно или легкое. Решение аналогично вышеприведенным примерам фильтрации. Чтобы упростить вам задачу, приведем черновик метода `prettyPrintApple`:

```
public static void prettyPrintApple(List<Apple> inventory, ???) {
   for(Apple apple: inventory) {
```

```
        String output = ???.???(apple);
        System.out.println(output);
    }
}
```

Ответ: во-первых, вам понадобится способ выразить такое поведение, при котором метод получает на входе объект `Apple` и возвращает результат в виде отформатированной строки. Вы уже делали нечто подобное, когда создавали интерфейс `ApplePredicate`:

```
public interface AppleFormatter {
    String accept(Apple a);
}
```

Теперь вы можете выражать поведение для различных видов форматирования посредством реализаций интерфейса `AppleFormatter`:

```
public class AppleFancyFormatter implements AppleFormatter {
    public String accept(Apple apple) {
        String characteristic = apple.getWeight() > 150 ? "heavy" : "light";
        return "A " + characteristic +
               " " + apple.getColor() +" apple";
    }
}
public class AppleSimpleFormatter implements AppleFormatter {
    public String accept(Apple apple) {
        return "An apple of " + apple.getWeight() + "g";
    }
}
```

Наконец, вам нужно, чтобы объекты типа `AppleFormatter` принимались методом `prettyPrintApple` и использовались внутри него. Для этого необходимо добавить в метод `prettyPrintApple` соответствующий параметр:

```
public static void prettyPrintApple(List<Apple> inventory,
                                    AppleFormatter formatter) {
  for(Apple apple: inventory) {
    String output = formatter.accept(apple);
    System.out.println(output);
  }
}
```

Готово! Теперь можно передавать различные варианты поведения в метод `prettyPrintApple`. А добились мы этого, создав экземпляры реализаций интерфейса `AppleFormatter` и передав их в качестве аргументов методу `prettyPrintApple`:

```
prettyPrintApple(inventory, new AppleFancyFormatter());
```

В результате выполнения этого кода будет выведен результат наподобие следующего:

```
A light green apple
A heavy red apple
...
```

Или попробуйте вот такое:

```
prettyPrintApple(inventory, new AppleSimpleFormatter());
```

В результате выполнения этого кода будет выведен примерно такой результат:

```
An apple of 80g
An apple of 155g
...
```

Как видите, существует возможность абстрагировать поведение и обеспечить адаптацию кода к изменению требований, но при этом приходится писать слишком много кода для объявления различных классов, экземпляры которых создаются лишь один раз. Посмотрим, как исправить эту ситуацию.

2.3. Делаем код лаконичнее

Всем известно, что громоздких конструкций следует избегать. На текущий момент при необходимости передать новое поведение методу `filterApples` нам приходится объявлять несколько классов, реализующих интерфейс `ApplePredicate`, а затем создавать несколько объектов `ApplePredicate`, используемых лишь один раз, как показано в итоговом листинге 2.1. Код получается очень громоздким, а его написание занимает много времени!

Листинг 2.1. Параметризация поведения: фильтрация яблок с помощью предикатов

```
public class AppleHeavyWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return apple.getWeight() > 150;
    }
}
public class AppleGreenColorPredicate implements ApplePredicate {
    public boolean test(Apple apple) {
        return GREEN.equals(apple.getColor());
    }
}
public class FilteringApples {
    public static void main(String...args) {
        List<Apple> inventory = Arrays.asList(new Apple(80, GREEN),
                                              new Apple(155, GREEN),
                                              new Apple(120, RED));

        List<Apple> heavyApples =
            filterApples(inventory, new AppleHeavyWeightPredicate());
        List<Apple> greenApples =
            filterApples(inventory, new AppleGreenColorPredicate());
    }
    public static List<Apple> filterApples(List<Apple> inventory,
                                           ApplePredicate p) {
        List<Apple> result = new ArrayList<>();
        for (Apple apple : inventory) {
            if (p.test(apple)){
                result.add(apple);
```

Выбирает тяжелые яблоки

Выбирает зеленые яблоки

Результаты в объекте List, содержащем одно яблоко весом 155 г

Результаты в объекте List, содержащем два зеленых яблока

```
            }
        }
        return result;
    }
}
```

Это совершенно излишние накладные расходы. Можно ли как-то исправить ситуацию? В Java есть механизм под названием *«анонимные классы»* (anonymous classes), позволяющий одновременно объявлять класс и создавать его экземпляр. Благодаря этому можно еще улучшить наш код, сделав его немного более лаконичным. Но этот вариант тоже неидеален. В подразделе 2.3.3 приведен небольшой анонс следующей главы, посвященной повышению удобочитаемости кода с помощью лямбда-выражений.

2.3.1. Анонимные классы

Анонимные классы (anonymous classes) напоминают локальные классы в Java (классы, описанные внутри блока кода), с которыми вы уже знакомы. Но у анонимных классов нет названий. Они позволяют одновременно объявлять класс и создавать его экземпляр. Короче говоря, они позволяют создавать реализации специально для конкретного случая.

2.3.2. Пятая попытка: использование анонимного класса

В следующем коде показано, как можно переписать наш пример фильтрации, создав объект, реализующий интерфейс `ApplePredicate`, с помощью анонимного класса:

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple apple){
        return RED.equals(apple.getColor());
    }
});
```

Параметризация поведения метода filterApples с помощью анонимного класса

Анонимные классы часто применяются в контексте приложений с GUI для создания объектов обработчиков событий. Нам не хотелось бы воскрешать горькие воспоминания о Swing, но следующий код представляет собой распространенный паттерн, который часто встречается на практике (в данном случае с использованием API JavaFX — современной платформы UI для Java):

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Whoooo a click!!");
    }
});
```

При этом анонимные классы все же недостаточно хороши. Во-первых, они несколько громоздки и занимают довольно много места, как видно из выделенного жирным шрифтом (это те же два примера, что и выше):

```
List<Apple> redApples = filterApples(inventory, new ApplePredicate() {
    public boolean test(Apple a){
        return RED.equals(a.getColor());
    }
});
button.setOnAction(new EventHandler<ActionEvent>() {
```

Большое количество стереотипного кода

```
public void handle(ActionEvent event) {
    System.out.println("Whoooo a click!!");
}
```

Во-вторых, многие программисты путаются при их использовании. Например, в контрольном задании 2.2 приведена классическая Java-головоломка, которая ставит в тупик большинство программистов. Попробуйте свои силы и вы.

Контрольное задание 2.2. Головоломка на тему анонимных классов

Какие результаты вернет следующий код при выполнении: 4, 5, 6 или 42?

```
public class MeaningOfThis {
    public final int value = 4;
    public void doIt() {
        int value = 6;
        Runnable r = new Runnable() {
                public final int value = 5;
                public void run(){
                    int value = 10;
                    System.out.println(this.value);
                }
            };
            r.run();
        }
        public static void main(String...args) {
            MeaningOfThis m = new MeaningOfThis();
            m.doIt();
        }
}
```

Что выводится в этой строке?

Ответ: будет выведено значение 5, поскольку `this` относится к объемлющему этот фрагмент кода анонимному объекту `Runnable`, а не внешнему объемлющему классу `MeaningOfThis`.

«Многословность» кода — черта в целом отрицательная, препятствующая использованию возможностей языка, поскольку для написания и сопровождения такого кода нужно много времени, да и читать его неудобно! Хороший код должен быть понятен с первого взгляда. И хотя анонимные классы иногда решают проблему «многословности» кода, связанную с объявлением нескольких конкретных классов для интерфейса, это все равно неприемлемое решение. Если говорить о передаче простого фрагмента кода (например, булева выражения, отражающего критерии отбора), для описания поведения все равно приходится создавать объект и явным образом реализовывать метод (например, метод `test` для `Predicate` или метод `handle` для `EventHandler`).

Оптимально было бы поощрять использование программистами паттерна параметризации поведения, поскольку, как вы только что видели, он повышает приспособляемость кода к изменениям требований. В главе 3 вы увидите, как создатели языка Java 8 решили эту проблему с помощью лямбда-выражений — более лаконичного способа передачи кода. Хватит держать вас в неизвестности: вот краткий

предварительный рассказ о том, как лямбда-выражения могут помочь в стремлении к более чистому коду.

2.3.3. Шестая попытка: использование лямбда-выражения

Предыдущий код можно переписать на Java 8 следующим образом с использованием лямбда-выражения:

```
List<Apple> result =
  filterApples(inventory, (Apple apple) -> RED.equals(apple.getColor()));
```

Согласитесь, этот код выглядит гораздо аккуратнее, чем предыдущие варианты! Его преимущество в том, что он намного яснее отражает формулировку задачи. Мы наконец решили проблему «многословности» кода. Рисунок 2.4 подытоживает пройденный нами путь.

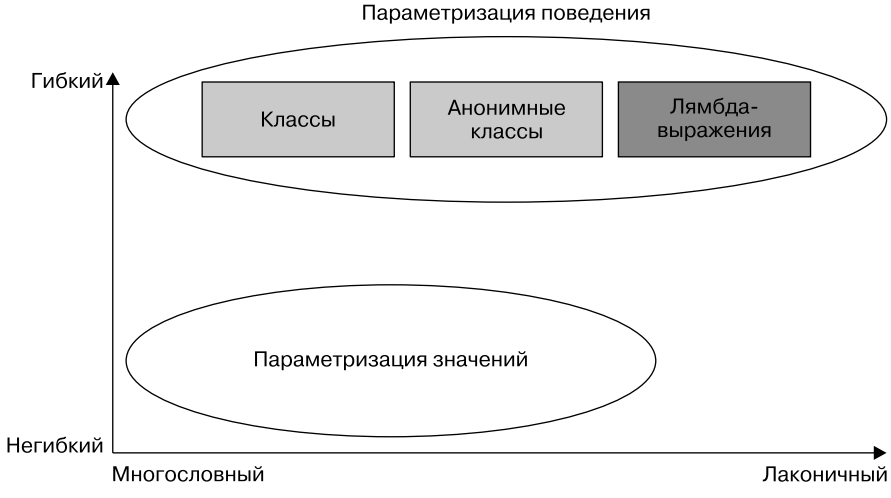


Рис. 2.4. Параметризация поведения по сравнению с параметризацией значений

2.3.4. Седьмая попытка: абстрагирование по типу значений списка

Мы можем продвинуться еще на один шаг в сторону повышения уровня абстракции. В настоящий момент метод `filterApples` работает только для типа `Apple`. Но можно также абстрагировать тип значений `List` и выйти за пределы предметной области текущей задачи, вот так:

```
public interface Predicate<T> {
    boolean test(T t);
}
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> result = new ArrayList<>();
    for(T e: list) {
        if(p.test(e)) {
            result.add(e);
```

Вводим параметр типа T

```
        }
    }
    return result;
}
```

Теперь метод `filter` можно использовать для фильтрации бананов, апельсинов, значений типа `Integer` или `String`! Вот пример применения лямбда-выражений:

```
List<Apple> redApples =
  filter(inventory, (Apple apple) -> RED.equals(apple.getColor()));
List<Integer> evenNumbers =
  filter(numbers, (Integer i) -> i % 2 == 0);
```

Потрясающе, правда? Нам удалось найти золотую середину между гибкостью и лаконичностью, что было невозможно до появления Java 8!

2.4. Примеры из практики

Как вы уже видели, параметризация поведения — паттерн, позволяющий легко адаптироваться к меняющимся требованиям. С помощью этого паттерна можно инкапсулировать какие-либо виды поведения (фрагменты кода, например различные предикаты для `Apple`) и параметризовать путем их передачи и использования поведение методов. Мы уже упоминали ранее, что этот подход аналогичен применению паттерна проектирования «Стратегия», который, возможно, вам уже доводилось использовать на практике. Множество методов API Java можно параметризовать различными видами поведения. Эти методы часто используются вместе с анонимными классами. Мы приведем четыре примера, которые сделают идею передачи кода более реальной для вас: сортировку с помощью интерфейса `Comparator`, выполнение блока кода с помощью интерфейса `Runnable`, возвращение результатов с помощью интерфейса `Callable` и обработку событий GUI.

2.4.1. Сортировка с помощью интерфейса Comparator

Сортировка коллекции — часто встречающаяся в программировании задача. Например, фермер может попросить нас отсортировать яблоки по весу. Или, скажем, он может передумать и захочет, чтобы вы отсортировали их по цвету. Знакомая ситуация? Да, вам нужен способ представления и использования различных поведений сортировки, чтобы быстро адаптироваться к меняющимся требованиям.

Начиная с Java 8, в интерфейсе `List` имеется метод `sort` (можно также воспользоваться методом `Collections.sort`). Параметризовать поведение метода `sort` можно с помощью объекта `java.util.Comparator`, интерфейс которого выглядит следующим образом:

```
// java.util.Comparator
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Следовательно, можно создавать различные варианты поведения для метода `sort`, создавая реализации `Comparator` специально для конкретного случая. Например, его можно использовать для сортировки наших яблок в порядке возрастания веса с помощью анонимного класса:

```
inventory.sort(new Comparator<Apple>() {
  public int compare(Apple a1, Apple a2) {
    return a1.getWeight().compareTo(a2.getWeight());
  }
});
```

Если же фермер передумает и захочет сортировать яблоки иначе, вы сможете создать специальный `Comparator` для этих новых требований и передать его в метод `sort`. Внутренние детали сортировки абстрагируются. При использовании лямбда-выражения этот код выглядит вот так:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

Опять же пока не задумывайтесь об этом новом для вас синтаксисе. В следующей главе мы подробно рассмотрим, как писать и использовать лямбда-выражения.

2.4.2. Выполнение блока кода с помощью интерфейса Runnable

С помощью *потоков выполнения* (threads) Java можно выполнить блок кода конкурентно с остальной частью программы. Но как указать потоку выполнения, какой блок кода ему нужно выполнить? Каждый из нескольких потоков может выполнять различный код. До появления Java 8 конструктору класса `Thread` можно было передавать только объекты, так что обычно применяли довольно неуклюжий паттерн передачи анонимного класса, содержащего метод `run`, возвращающий `void`. Подобные анонимные классы реализовывали интерфейс `Runnable`.

В Java интерфейс `Runnable` можно использовать для представления блока кода, который необходимо исполнить; обратите внимание, что этот код возвращает `void` (то есть не возвращает результата):

```
// java.lang.Runnable
public interface Runnable {
    void run();
}
```

С помощью этого интерфейса можно создавать потоки выполнения с нужным вам поведением, например:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello world");
    }
});
```

Но начиная с Java 8 можно воспользоваться лямбда-выражением, так что обращение к `Thread` будет выглядеть следующим образом:

```
Thread t = new Thread(() -> System.out.println("Hello world"));
```

2.4.3. Возвращение результатов выполнения задачи с помощью интерфейса Callable

Возможно, вы уже знакомы с абстракцией `ExecutorService`, появившейся в Java 5. Интерфейс `ExecutorService` отделяет отправку задачи на выполнение от непосредственно самого выполнения. По сравнению с использованием потоков и интерфейсом `Runnable` удобно то, что с помощью `ExecutorService` можно отправить задачу в пул потоков выполнения с сохранением ее результата в объекте `Future`. Не волнуйтесь, если эти понятия вам незнакомы, мы вернемся к данному вопросу в последующих главах, когда будем подробнее обсуждать конкурентность. Пока вам достаточно знать, что интерфейс `Callable` применяется для моделирования задачи, которая возвращает какой-либо результат. Можете рассматривать его как улучшенный вариант интерфейса `Runnable`:

```
// java.util.concurrent.Callable
public interface Callable<V> {
    V call();
}
```

Его можно применять, как показано ниже, путем отправки задачи сервису выполнения. В данном случае мы возвращаем название потока, отвечающего за выполнение данной задачи:

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<String> threadName = executorService.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return Thread.currentThread().getName();
    }
});
```

С помощью лямбда-выражения этот код можно упростить до следующего:

```
Future<String> threadName = executorService.submit(
                    () -> Thread.currentThread().getName());
```

2.4.4. Обработка событий графического интерфейса

Типичный паттерн в программировании графических интерфейсов — выполнять действия в ответ на определенные события, например щелчок кнопкой мыши или зависание указателя над участком текста. Допустим, если пользователь нажмет кнопку Отправить, то может понадобиться отобразить всплывающее окно или записать информацию о действии в файл. Опять же необходим какой-то способ адаптации к изменению требований; должна быть возможность выполнения любого ответного действия. В JavaFX можно воспользоваться объектом `EventHandler`, передавая его в метод `setOnAction` в качестве поведения ответа:

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

В этом коде поведение метода `setOnAction` параметризуется с помощью объектов `EventHandler`. С помощью лямбда-выражения код можно переписать так:

```
button.setOnAction((ActionEvent event) -> label.setText("Sent!!"));
```

Резюме

- Параметризация поведения — это способность метода *принимать* в качестве параметров множество вариантов поведения, а затем, опираясь на них, *воплощать* различные последовательности операций.
- Параметризация поведения позволяет повышать приспособляемость кода к изменению требований и экономить в будущем время и силы.
- Передача кода — способ передавать методу новые виды поведения в качестве аргументов. Но до появления Java 8 этот способ не отличался лаконичностью кода. С помощью анонимных классов до Java 8 можно было частично решить проблему «многословности» кода, связанную с объявлением нескольких конкретных классов для интерфейса, которые используются только один раз.
- API языка Java содержит множество методов, допускающих параметризацию различными видами поведения, в том числе относящихся к сортировке, потокам выполнения и обработке событий GUI.

3 Лямбда-выражения

В этой главе

- Главное о лямбда-выражениях.
- Где и как использовать лямбда-выражения.
- Паттерн охватывающего выполнения.
- Функциональные интерфейсы, вывод типов.
- Ссылки на методы.
- Создание лямбда-выражений.

В предыдущей главе было показано, что передача кода при параметризации поведения позволяет адаптироваться к частым сменам требований. Благодаря этому можно определить блок кода, отражающий некий вариант поведения, и затем передать его куда-либо. Такой блок кода можно выполнять в случае определенного события (например, щелчка кнопкой мыши) или в каких-либо заданных местах алгоритма (например, при условии «только яблоки, которые тяжелее 150 г» — в алгоритме фильтрации или в пользовательской операции сравнения при сортировке). В целом эта концепция позволяет писать более гибкий и более удобный для повторного использования код.

Однако, как вы видели, анонимные классы плохо подходят для отражения различных вариантов поведения. Подобный код не отличается лаконичностью, что отнюдь не стимулирует программистов использовать параметризацию поведения на практике. В этой главе мы расскажем вам про новую возможность Java 8, которая решает данную проблему: лямбда-выражения. С их помощью можно лаконично выражать поведение или передавать код. Пока вы можете считать лямбда-выражения анонимными функциями, методами без объявленных названий, которые можно передавать в качестве аргументов в метод, аналогично тому, как мы поступали с анонимными классами.

Мы продемонстрируем, как их конструировать, где использовать и как повысить за счет этого лаконичность кода. Кроме того, расскажем о некоторых новых приятных возможностях, например о выводе типов и о новых важных интерфейсах, появившихся в API Java 8. Наконец, мы познакомим вас со ссылками на методы — удобной новой возможностью, часто используемой там же, где и лямбда-выражения.

Эта глава организована таким образом, чтобы поэтапно учить вас писать более лаконичный и гибкий код. В конце главы мы сведем воедино все изложенное на конкретном примере: поэтапно усовершенствуем пример сортировки из главы 2, сделав его более лаконичным и удобочитаемым. Эта глава важна не только сама по себе, но и потому, что далее вам предстоит использовать лямбда-выражения на протяжении всей книги.

3.1. Главное о лямбда-выражениях

Лямбда-выражение (lambda expression, lambda) можно рассматривать как краткую форму анонимной функции, которую разрешено куда-либо передавать. У нее отсутствует название, но имеется список параметров, тело, тип возвращаемого значения, а также, возможно, список потенциально генерируемых исключений. Это довольно большое определение, поэтому разобьем его на составные части.

- *Анонимное* — анонимным лямбда-выражение является потому, что у него отсутствует явное название, как у метода. Меньше писать и меньше обдумывать!
- *Функция* — лямбда-выражение является функцией потому, что не связано с конкретным классом, подобно методу. Но, как и у метода, у лямбда-выражения есть список параметров, тело, возвращаемый тип, а также, возможно, список потенциально генерируемых исключений.
- *Возможность передачи* — лямбда-выражение можно передать в качестве аргумента в метод или сохранить в переменной.
- *Лаконичное* — нет необходимости писать длинный стереотипный (иногда называют «шаблонный») код, как в случае анонимных классов.

Если вам интересно, то слово *«лямбда»* ведет свое начало от разработанной в научной среде системы под названием *лямбда-исчисление* (lambda calculus), которая использовалась для описания вычислений.

Чем интересны лямбда-выражения? Как вы видели в предыдущей главе, передача кода в настоящий момент утомительное занятие, требующее написания большого объема стереотипного кода. Что ж, у нас есть хорошая новость! Лямбда-выражения решают эту проблему, позволяя передавать код намного более лаконичным образом. Формально с их помощью нельзя делать ничего, что не было бы возможно до Java 8, но теперь больше не нужно писать неуклюжий код с анонимными классами, чтобы воспользоваться параметризацией поведения. Лямбда-выражения вдохновляют программистов на использование стиля параметризации поведения, описанного в предыдущей главе. В итоге код становится более чистым и гибким. Например, с помощью лямбда-выражения можно создать пользовательский объект `Comparator` более лаконичным образом.

Раньше:

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

Теперь (при использовании лямбда-выражений):

```
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Признайте, что этот код намного аккуратнее! Не беспокойтесь, если некоторые части этого лямбда-выражения вам пока непонятны, — вскоре мы все поясним. Пока отметим, что передается фактически только код, необходимый для сравнения двух яблок по весу. Выглядит так, как будто передается тело метода `compare`. Скоро вы узнаете, что можно еще больше упростить код. В следующем разделе мы объясним, где и как можно использовать лямбда-выражения.

Приведенное выше лямбда-выражение состоит из трех частей, как показано на рис. 3.1.

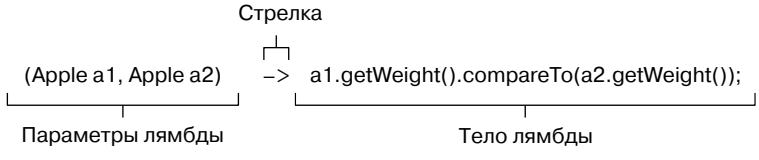


Рис. 3.1. Лямбда-выражение состоит из параметров, стрелки и тела

- *Список параметров* в данном случае отражает параметры метода `compare` интерфейса `Comparator` — два объекта `Apple`.
- *Стрелка* (arrow) `->` отделяет список параметров от тела лямбда-выражения.
- *Тело лямбды* — сравнение двух объектов `Apple` по весу. Это выражение считается возвращаемым значением лямбды.

Для дальнейшей иллюстрации приведем в листинге 3.1 пять примеров допустимых лямбда-выражений на языке Java 8.

Такой синтаксис был выбран при проектировании языка Java в силу позитивных отзывов о нем в других языках, например C# и Scala. Аналогичный синтаксис есть и в JavaScript. Базовый синтаксис лямбда-выражения имеет вид (обычно называется лямбдой *в форме выражения* (expression-style lambda)):

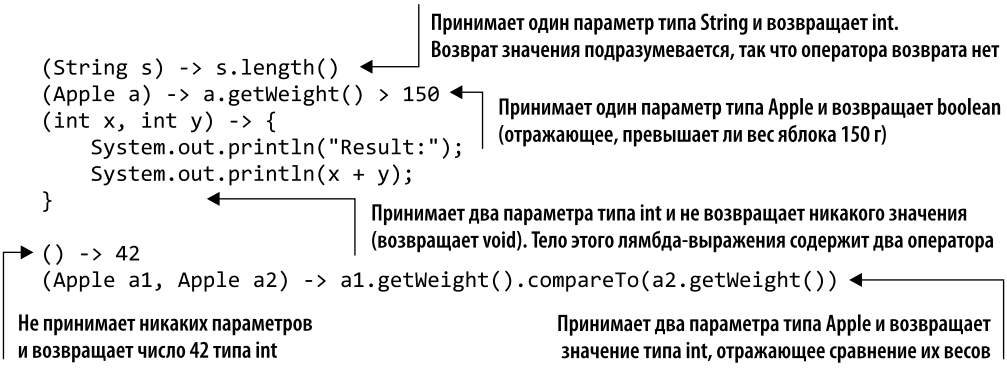
```
(параметры) -> выражение
```

или вид (обратите внимание на фигурные скобки вокруг операторов — такие лямбды часто называют *лямбдами в форме блока* (block-style lambdas))[1]:

```
(параметры) -> { операторы; }
```

[1] Для простоты мы будем далее называть и те и другие лямбда-выражениями или просто лямбдами. — *Примеч. пер.*

Листинг 3.1. Допустимые лямбда-выражения на языке Java 8



Как вы можете видеть, синтаксис лямбда-выражений довольно прост. Контрольное задание 3.1 поможет вам проверить, хорошо ли вы понимаете этот паттерн.

Контрольное задание 3.1. Синтаксис лямбда-выражений

Основываясь на только что приведенных правилах, ответьте, какие из следующих лямбда-выражений не являются допустимыми.

1. `() -> {}`.
2. `() -> "Raoul"`.
3. `() -> { return "Mario"; }`.
4. `(Integer i) -> return "Alan" + i;`.
5. `(String s) -> { "Iron Man"; }`.

Ответ: 4 и 5 — недопустимые лямбда-выражения, остальные допустимы. Далее приведены подробные объяснения.

1. У этого лямбда-выражения нет параметров, оно возвращает `void`. Оно напоминает метод с пустым телом: `public void run() { }`. Любопытный факт: обычно подобное называют лямбдой-гамбургером. Взгляните на выражение сбоку, и вы увидите, что оно похоже на гамбургер с двумя булочками.
2. У этого лямбда-выражения нет параметров, и оно возвращает объект `String` в виде выражения.
3. У этого лямбда-выражения нет параметров, и оно возвращает объект `String` (с помощью явного оператора `return` внутри блока).
4. `return` — оператор управления потоком выполнения. Чтобы синтаксис этой лямбды был корректным, необходимы фигурные скобки: `(Integer i) -> { return "Alan" + i; }`.
5. `"Iron Man"` — выражение, а не оператор. Чтобы синтаксис этой лямбды был корректным, можно убрать фигурные скобки и точку с запятой: `(String s) -> "Iron Man"`. Или же, если вам так больше нравится, можете воспользоваться явным оператором `return`: `(String s) -> { return "Iron Man"; }`.

В табл. 3.1 приведен перечень примеров лямбда-выражений и сценариев их использования.

Таблица 3.1. Примеры лямбда-выражений

Сценарий использования	Примеры лямбда-выражений
Булево выражение	(List<String> list) -> list.isEmpty()
Создание объектов	() -> new Apple(10)

Потребление данных из объекта	(Apple a) -> {
	System.out.println(a.getWeight());
	}

Выбор/извлечение данных из объекта	(String s) -> s.length()
Сочетание двух значений	(int a, int b) -> a * b
Сравнение двух объектов	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())

3.2. Где и как использовать лямбда-выражения

Наверное, теперь вам интересно, где можно использовать лямбда-выражения. В предыдущем примере мы присвоили лямбда-выражение переменной типа `Comparator<Apple>`. Можно также воспользоваться еще одним лямбда-выражением с методом `filter`, которое мы реализовали в предыдущей главе:

```
List<Apple> greenApples =
        filter(inventory, (Apple a) -> GREEN.equals(a.getColor()));
```

Где именно можно использовать лямбда-выражения? Например, в контексте функционального интерфейса. В показанном выше коде лямбда-выражение можно передать в качестве второго аргумента в метод `filter`, поскольку этот метод ожидает на входе объект типа `Predicate<T>`, который является функциональным интерфейсом. Не беспокойтесь из-за малопонятности этой фразы — сейчас мы объясним, что она означает и что такое функциональный интерфейс.

3.2.1. Функциональный интерфейс

Помните интерфейс `Predicate<T>`, который мы создали в главе 2 для параметризации поведения метода `filter`? Это функциональный интерфейс! Почему? Потому что в интерфейсе `Predicate` описан только один абстрактный метод:

```
public interface Predicate<T> {
    boolean test (T t);
}
```

В двух словах, *функциональный интерфейс* (functional interface) — интерфейс, в котором описывается ровно один абстрактный метод. Вы уже знакомы с несколькими функциональными интерфейсами из API Java, такими как `Comparator` и `Runnable`, о которых мы говорили в главе 2:

```
public interface Comparator<T> {  <-- java.util.Comparator
    int compare(T o1, T o2);
}
```

```
public interface Runnable {  ◄——— java.lang.Runnable
    void run();
}
public interface ActionListener extends EventListener { ◄——— java.awt.ActionListener
    void actionPerformed(ActionEvent e);
}
public interface Callable<V> {  ◄——— java.util.Concurrent.Callable
    V call() throws Exception;
}
public interface PrivilegedAction<T> { ◄——— java.security.PrivilegedAction
    T run();
}
```

ПРИМЕЧАНИЕ

В главе 13 мы увидим, что в интерфейсах могут быть также методы с реализацией по умолчанию (метод, тело которого описывает какую-либо реализацию по умолчанию для метода на случай, если он не будет реализован в классе). Интерфейс остается функциональным даже при наличии множества методов с реализацией по умолчанию, если в нем задан только один абстрактный метод.

Проверьте, насколько хорошо вы поняли концепцию функционального интерфейса, выполнив контрольное задание 3.2.

Контрольное задание 3.2. Функциональный интерфейс

Какой из следующих интерфейсов является функциональным?

```
public interface Adder {
    int add(int a, int b);
}
public interface SmartAdder extends Adder {
    int add(double a, double b);
}
public interface Nothing {
}
```

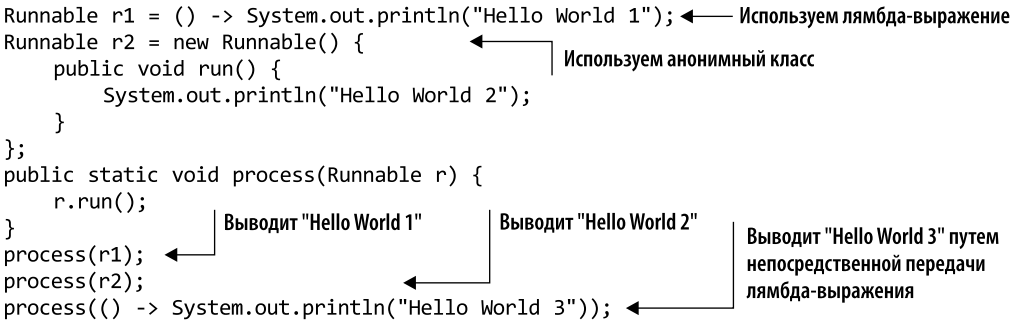
Ответ: функциональным интерфейсом является только `Adder`.

Интерфейс `SmartAdder` не функциональный, поскольку в нем определены два абстрактных метода с названием `add` (один из которых унаследован от `Adder`).

`Nothing` — не функциональный интерфейс, поскольку в нем вообще не объявлено абстрактных методов.

Чем могут быть полезны функциональные интерфейсы? С помощью лямбда-выражений можно задавать реализацию абстрактных методов функционального интерфейса на месте и *работать с выражением в целом как с экземпляром функционального интерфейса* (точнее говоря, экземпляром *конкретной реализации* функционального интерфейса). То же самое можно сделать с помощью анонимного внутреннего класса,

хотя это более неуклюжее решение: при этом на месте приводится реализация и сразу же создается экземпляр класса. Следующий код допустим, поскольку `Runnable` — функциональный интерфейс, в котором описан только один абстрактный метод, `run`:



3.2.2. Функциональные дескрипторы

Сигнатура абстрактного метода функционального интерфейса описывает сигнатуру лямбда-выражения. Мы будем называть такой абстрактный метод *функциональным дескриптором* (function descriptor). Например, интерфейс `Runnable` можно рассматривать как сигнатуру функции, не принимающей параметров и ничего не возвращающей (то есть возвращающей `void`), поскольку в нем описан только один абстрактный метод `run`, не принимающий параметров и ничего не возвращающий (возвращающий `void`)[1].

Для описания сигнатур лямбда-выражений и функциональных интерфейсов в этой главе мы пользовались специальной нотацией. Нотация `() -> void` соответствует функции с пустым списком параметров, которая возвращает `void`. Это как раз соответствует интерфейсу `Runnable`. Другой пример, `(Apple, Apple) -> int`, описывает функцию, принимающую в качестве параметров два объекта `Apple` и возвращающую значение типа `int`. Мы расскажем подробнее о функциональных дескрипторах далее в этой главе, в разделе 3.4 и табл. 3.2.

Наверное, вам уже стало интересно, как происходит проверка типов для лямбда-выражений. В разделе 3.5 мы подробно опишем, как компилятор проверяет, допустимо ли лямбда-выражение в данном контексте. Пока вам достаточно знать, что лямбда-выражение можно присвоить переменной или передать в метод, ожидающий в качестве аргумента функциональный интерфейс, при условии, что сигнатура лямбда-выражения совпадает с сигнатурой абстрактного метода этого функционального интерфейса. В частности, в ранее приведенном примере можно передать лямбда-выражение непосредственно методу `process` следующим образом:

```
public void process(Runnable r) {
    r.run();
}
process(() -> System.out.println("Потрясающе!!"));
```

[1] В системе типов некоторых языков программирования, например Scala, есть явные аннотации типов, предназначенные для описания типов функций (они называются функциональными типами). Язык Java переиспользует уже существующие типы, предоставляемые функциональными интерфейсами, и неявно отображает их в форму функциональных типов.

При выполнении этого кода будет выведено `Потрясающе!!`. Параметров в лямбда-выражении `process(() -> System.out.println("Потрясающе!!"))` нет, оно возвращает `void`. Это в точности соответствует сигнатуре метода `run` из интерфейса `Runnable`.

Лямбда-выражения и вызов методов, возвращающих void

Как ни странно, следующее лямбда-выражение вполне допустимо:

```
process(() -> System.out.println("Потрясающе"));
```

В конце концов, `System.out.println` возвращает `void`, так что это явно не выражение! Почему же можно не заключать тело лямбды в фигурные скобки, как здесь?

```
process(() -> { System.out.println("Потрясающе"); });
```

Оказывается, что в спецификации языка Java есть специальное правило относительно вызова методов, возвращающих `void`. Заключать отдельный вызов возвращающего `void` метода в фигурные скобки не обязательно.

Наверное, вы недоумеваете: «Почему передавать лямбда-выражения можно только там, где ожидается функциональный интерфейс?» Создатели языка рассматривали и другие подходы, например добавление в Java функциональных типов (подобных специальной нотации, использовавшейся нами для описания сигнатур лямбда-выражений, — мы вернемся к этому вопросу в главах 20 и 21). Но они остановились на этом варианте как наиболее естественном и не повышающем сложность языка. Кроме того, большинство Java-программистов уже знакомы с идеей интерфейса с одним абстрактным методом (например, для обработки событий). Однако основная причина — широкое применение функциональных интерфейсов и до Java 8, так что они представляют собой удобный путь перехода к лямбда-выражениям. На самом деле если вы используете функциональные интерфейсы, например `Comparator` и `Runnable` или даже ваши собственные интерфейсы, в которых — так случилось — описан только один абстрактный метод, то можете использовать лямбда-выражения без изменений своих API. Попробуйте выполнить контрольное задание 3.3, чтобы проверить, насколько хорошо вы понимаете, где можно использовать лямбда-выражения.

Контрольное задание 3.3. Где допустимы лямбда-выражения

Какие из следующих трех примеров — допустимые варианты использования лямбда-выражений?

1.
```
execute(() -> {});
public void execute(Runnable r) {
  r.run();
}
```

2.
```
public Callable<String> fetch() {
  return () -> "Хитрый пример ;-)";
}
```

3. `Predicate<Apple> p = (Apple a) -> a.getWeight();`

Ответ: допустимы только 1 и 2.

Первый пример допустим, поскольку сигнатура `() -> {}` представляет собой `() -> void`, то есть соответствует сигнатуре абстрактного метода `run` из интерфейса `Runnable`. Обратите внимание, что при выполнении этого кода ничего не произойдет, поскольку тело лямбда-выражения пусто!

Второй пример допустим. Действительно, возвращаемый тип метода `fetch` — `Callable<String>`. При замене `T` на `String` получается, что `Callable<String>` описывает метод с сигнатурой `() -> String`. А поскольку у лямбда-выражения `() -> "Хитрый пример ;-)"` сигнатура `() -> String`, то такое выражение можно использовать в данном контексте.

Третий пример недопустим, поскольку у лямбда-выражения `(Apple a) -> a.getWeight()` сигнатура `(Apple) -> Integer`, которая отличается от сигнатуры метода `test`, описанного в `Predicate<Apple>`, а именно: `(Apple) -> boolean`.

Аннотация @FunctionalInterface

При изучении нового API Java можно заметить, что у многих функциональных интерфейсов присутствует аннотация `@FunctionalInterface` (в разделе 3.4 мы приведем обширный список таких аннотаций). Эта аннотация указывает, что интерфейс задумывался как функциональный, а значит, будет полезен для документации. Кроме того, компилятор вернет осмысленную ошибку, если попытаться описать не функциональный интерфейс с аннотацией `@FunctionalInterface`. Сообщение об ошибке, например, может выглядеть так: `Multiple non-overriding abstract methods found in interface Foo` («В интерфейсе Foo обнаружено несколько непереопределяемых абстрактных методов»). Оно будет указывать на наличие в интерфейсе нескольких абстрактных методов. Обратите внимание, что аннотация `@FunctionalInterface` необязательна, но рекомендуется к использованию при проектировании функциональных интерфейсов. Считайте, что это аналог аннотации `@Override`, указывающей на переопределяемый метод.

3.3. Практическое использование лямбда-выражений: паттерн охватывающего выполнения

Рассмотрим пример использования на практике лямбда-выражений совместно с параметризацией поведения для повышения гибкости и лаконичности кода. Часто встречающийся при работе с ресурсами (например, с файлами и базами данных) паттерн: открыть ресурс, выполнить обработку его данных, после чего закрыть его. Фазы начальных настроек и очистки всегда одинаковы и окружают важный код, отвечающий за собственно обработку. Этот паттерн носит название *паттерна охватывающего выполнения* (execute-around pattern) и продемонстрирован на рис. 3.2.

Рис. 3.2. Задачи A и B окружены стереотипным кодом подготовки/очистки

Например, выделенные строки в следующем коде представляют собой стереотипный код, необходимый для чтения из файла одной строки (заметим, что здесь используется оператор `try` с ресурсами Java 7, который сам по себе упрощает код, поскольку не нужно закрывать ресурс явным образом):

```
public String processFile() throws IOException {
try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

Это единственная строка, которая делает что-то полезное

3.3.1. Шаг 1: вспоминаем параметризацию поведения

Возможности текущего кода ограничены: можно прочитать только первую строку файла. А что делать, если нужно вернуть первые две строки вместо одной или, допустим, чаще всего используемое слово? Оптимально было бы переиспользовать код для начальных настроек и очистки, а также указать методу `processFile`, какие (различные) действия необходимо совершить с файлом. Звучит знакомо, не правда ли? Да, вам необходимо параметризовать поведение метода `processFile`. Требуется какой-то способ передачи поведения в метод `processFile`, чтобы он мог выполнять различные последовательности операций с помощью `BufferedReader`.

Лямбда-выражения предназначены как раз для передачи поведения. Как должен выглядеть новый метод `processFile` для случая чтения сразу двух строк? Необходимо лямбда-выражение, принимающее на входе объект `BufferedReader` и возвращающее объект `String`. Например, вывести две строки `BufferedReader` можно следующим образом:

```
String result
    = processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

3.3.2. Шаг 2: используем функциональный интерфейс для передачи поведения

Мы уже объясняли ранее, что лямбда-выражения можно использовать только в контексте функционального интерфейса. В данном случае нам нужно создать функциональный интерфейс, соответствующий сигнатуре `BufferedReader ->`

`String` и способный генерировать исключение `IOException`. Назовем его `BufferedReaderProcessor`:

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

Теперь можно указать его в качестве параметра нашего нового метода `processFile`:

```
public String processFile(BufferedReaderProcessor p) throws IOException {
   ...
}
```

3.3.3. Шаг 3: выполняем нужный вариант поведения!

В качестве аргументов нашему методу можно передавать любые лямбда-выражения вида `BufferedReader -> String`, поскольку они соответствуют сигнатуре метода `process`, описанного в интерфейсе `BufferedReaderProcessor`. Необходимо только найти способ выполнить код, представленный в виде лямбда-выражения, внутри тела метода `processFile`. Запомните: благодаря лямбда-выражениям можно непосредственно на месте задать реализацию абстрактного метода функционального интерфейса и *работать с выражением в целом как с экземпляром функционального интерфейса*. Следовательно, в данном случае можно вызвать для обработки метод `process` итогового объекта `BufferedReaderProcessor` внутри тела метода `processFile`:

```
public String processFile(BufferedReaderProcessor p) throws IOException {
    try (BufferedReader br =
                  new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);
    }
}
```

Обрабатывает объект BufferedReaderProcessor

3.3.4. Шаг 4: передаем лямбда-выражения

Теперь можно повторно использовать метод `processFile` и обрабатывать файлы различными способами, передавая различные лямбда-выражения.

Следующий код демонстрирует обработку одной строки файла:

```
String oneLine =
    processFile((BufferedReader br) -> br.readLine());
```

А этот код — обработку двух строк файла:

```
String twoLines =
    processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

На рис. 3.3 вкратце подведены итоги четырех шагов, благодаря которым мы сделали метод `processFile` более гибким.

Мы продемонстрировали применение функциональных интерфейсов для передачи лямбда-выражений. Но для этого пришлось описывать свои собственные интерфейсы. В следующем разделе мы покажем новые, добавленные в Java 8 интерфейсы, которыми можно воспользоваться для передачи множества различных лямбда-выражений.

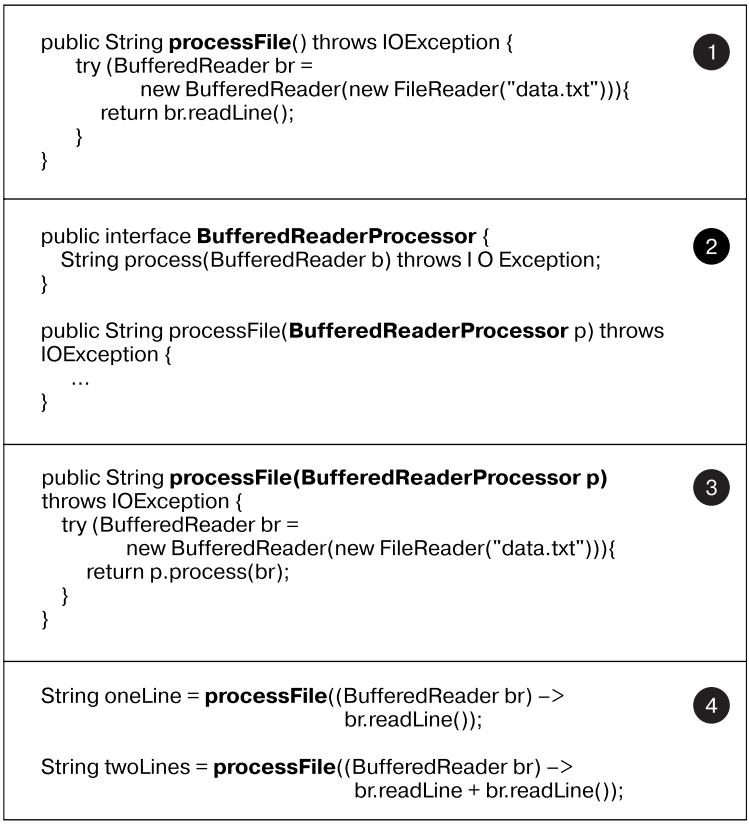


Рис. 3.3. Четырехшаговый процесс применения паттерна охватывающего выполнения

3.4. Использование функциональных интерфейсов

Как вы узнали из подраздела 3.2.1, функциональный интерфейс определяет ровно один абстрактный метод. Функциональные интерфейсы удобны, поскольку сигнатура абстрактного метода может описывать сигнатуру лямбда-выражения. Сигнатура абстрактного метода функционального интерфейса называется *функциональным дескриптором* (function descriptor). Для использования различных лямбда-выражений необходим набор функциональных интерфейсов, описывающих часто встречающиеся функциональные дескрипторы. В API Java уже есть несколько готовых функциональных интерфейсов, например `Comparable`, `Runnable` и `Callable`, о которых мы говорили в разделе 3.2.

Создатели библиотек Java для версии Java 8 облегчили задачу разработчикам, включив в пакет `java.util.function` несколько новых функциональных интерфейсов. Далее мы опишем интерфейсы `Predicate`, `Consumer` и `Function`. Более полный список можно найти в табл. 3.2 в конце данного раздела.

3.4.1. Интерфейс Predicate

В интерфейсе `java.util.function.Predicate<T>` описан абстрактный метод `test`, принимающий на входе объект обобщенного типа `T` и возвращающий `boolean`. Точно такой же, как мы создали ранее, но уже готовый! Этот интерфейс может пригодиться для представления булева выражения, в котором используется объект типа `T`. Например, можно описать лямбда-выражение, принимающее на входе объекты типа `String`, как показано в листинге 3.2.

Листинг 3.2. Использование интерфейса Predicate

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
public <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T t: list) {
        if(p.test(t)) {
            results.add(t);
        }
    }
    return results;
}
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

Если заглянуть в спецификацию интерфейса `Predicate` из Javadoc, можно увидеть дополнительные методы, например `and` и `or`. Пока не задумывайтесь о них, мы вернемся к ним в разделе 3.8.

3.4.2. Интерфейс Consumer

В интерфейсе `java.util.function.Consumer<T>` описан абстрактный метод `accept`, принимающий на входе объект обобщенного типа `T` и не возвращающий ничего (возвращающий `void`). Он обычно применяется для доступа к объекту типа `T` и выполнению над ним каких-либо операций. Например, им можно воспользоваться для создания метода `forEach`, принимающего на входе список объектов `Integer` и производящего какую-то операцию над каждым из элементов списка. В листинге 3.3 мы воспользуемся методом `forEach` в сочетании с лямбда-выражением для вывода в системную консоль всех элементов списка.

Листинг 3.3. Использование интерфейса Consumer

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
public <T> void forEach(List<T> list, Consumer<T> c) {
    for(T t: list) {
        c.accept(t);
    }
}
```

```
forEach(
         Arrays.asList(1,2,3,4,5),
        (Integer i) -> System.out.println(i)
       );
```

Это лямбда-выражение является реализацией метода accept из интерфейса Consumer

3.4.3. Интерфейс Function

В интерфейсе `java.util.function.Function<T>` описан абстрактный метод `apply`, принимающий на входе объект обобщенного типа `T` и возвращающий объект обобщенного типа `R`. Он обычно применяется для описания лямбда-выражения, отображающего информацию входного объекта на выходной (например, извлекающего данные о весе яблок или ставящего в соответствие строке ее длину). В листинге 3.4 мы покажем, как создать с его помощью метод `map` для преобразования списка объектов `String` в список `Integer`, содержащих длину каждого объекта из списка `String`.

Листинг 3.4. Использование интерфейса Function

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
public <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T t: list) {
        result.add(f.apply(t));
    }
    return result;
}

// [7, 2, 6]
List<Integer> l = map(
                        Arrays.asList("lambdas", "in", "action"),
                        (String s) -> s.length()
                  );
```

Реализация метода apply интерфейса Function

Версии для примитивных типов данных. Мы описали выше три обобщенных функциональных интерфейса: `Predicate<T>`, `Consumer<T>` и `Function<T, R>`. Существуют также функциональные интерфейсы, специально предназначенные для определенных типов данных.

Напомним: все типы данных Java делятся на ссылочные (например, `Byte`, `Integer`, `Object`, `List`) и примитивные (простые) типы (например, `int`, `double`, `byte`, `char`). Но обобщенные параметры (например, `T` в описании типа `Function<T>`) допустимы только для ссылочных типов. Причина этого лежит во внутренней реализации обобщенных типов[1]. В результате в Java существует механизм преобразования примитивных типов данных в соответствующие ссылочные типы. Этот механизм называется *упаковкой*

[1] В части других языков программирования, например C#, такого ограничения нет. А в некоторых языках, таких как Scala, вообще есть только ссылочные типы данных. Мы вернемся к этому вопросу в главе 20.

(boxing). Обратный процесс (преобразование ссылочного типа данных в соответствующий примитивный тип) называется *распаковкой* (unboxing). В Java существует также механизм *автоупаковки* (autoboxing) для упрощения работы программистов: упаковка и распаковка производятся автоматически. Именно благодаря этому механизму допустим следующий код (значение типа `int` преобразуется в объект типа `Integer`):

```
List<Integer> list = new ArrayList<>();
for (int i = 300; i < 400; i++){
    list.add(i);
}
```

Но эти возможности сопряжены с определенными потерями производительности. Упакованные значения представляют собой фактически адаптеры для примитивных типов данных и хранятся в куче. Следовательно, упакованные значения требуют большего количества памяти и дополнительных операций поиска для извлечения обернутых простых значений.

Во избежание операций автоупаковки (в случае, когда входные или выходные значения относятся к примитивным типам данных) в Java 8 также были добавлены узкоспециализированные версии вышеописанных функциональных интерфейсов. Например, в следующем коде благодаря использованию `IntPredicate` удается избежать операции упаковки значения `1000`, в то время как применение `Predicate<Integer>` привело бы к упаковке аргумента `1000` в объект типа `Integer`:

```
public interface IntPredicate {
    boolean test(int t);
}
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);
Predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
oddNumbers.test(1000);
```

true (не упаковывается)

false (упаковывается)

В целом соответствующий простой тип данных указывается в начале названия функциональных интерфейсов, ориентированных на работу с определенным типом входного параметра (например, `DoublePredicate`, `IntConsumer`, `LongBinaryOperator`, `IntFunction` и т. д.). У интерфейса `Function` есть также и версии с различным типом параметра-результата: `ToIntFunction<T>`, `IntToDoubleFunction` и т. д.

В табл. 3.2 перечислены наиболее часто используемые функциональные интерфейсы из API Java и их функциональные дескрипторы, а также их варианты для примитивных типов данных. Помните, что это лишь начальный набор и при необходимости всегда можно создать собственные (например, как в контрольном задании 3.7, где вам нужно будет создать интерфейс `TriFunction`). Создание собственных интерфейсов удобно еще и тем, что предметно-ориентированное название повышает понятность и сопровождаемость программы. Напомним, что нотация вида `(T, U) -> R` наглядно описывает функциональный дескриптор. Слева от стрелки — список типов аргументов, а справа — типы результатов. В данном случае она отражает функцию с двумя аргументами обобщенных типов `T` и `U`, а также с типом возвращаемого значения `R`.

Вы уже видели немало функциональных интерфейсов, подходящих для описания сигнатур разнообразных лямбда-выражений. Чтобы проверить свои знания, попробуйте выполнить контрольное задание 3.4.

Таблица 3.2. Распространенные функциональные интерфейсы из числа добавленных в Java 8

Функциональный интерфейс	Функциональный дескриптор	Варианты для примитивных типов данных
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	—
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

Контрольное задание 3.4. Функциональные интерфейсы

Какими функциональными интерфейсами вы бы воспользовались для следующих функциональных дескрипторов (сигнатур лямбда-выражений)? Большую часть ответов вы найдете в табл. 3.2. В качестве дальнейшего упражнения придумайте корректные лямбда-выражения для применения с этими функциональными интерфейсами.

1. `T -> R`.
2. `(int, int) -> int`.
3. `T -> void`.
4. `() -> T`.
5. `(T, U) -> R`.

Ответы

1. Неплохой кандидат на эту роль — `Function<T, R>`. Обычно он используется для преобразования объекта типа `T` в объект типа `R` (например, `Function<Apple, Integer>` для получения веса яблока).
2. В интерфейсе `IntBinaryOperator` описан один абстрактный метод `applyAsInt`, отражающий функциональный дескриптор `(int, int) -> int`.
3. В интерфейсе `Consumer<T>` описан один абстрактный метод `accept`, отражающий функциональный дескриптор `T -> void`.
4. В интерфейсе `Supplier<T>` описан один абстрактный метод `get`, отражающий функциональный дескриптор `() -> T`.
5. В интерфейсе `BiFunction<T, U, R>` описан один абстрактный метод `apply`, отражающий функциональный дескриптор `(T, U) -> R`.

Подытожим обсуждение функциональных интерфейсов и лямбда-выражений в табл. 3.3, содержащей сводку сценариев использования, примеры лямбда-выражений и соответствующих функциональных интерфейсов.

Таблица 3.3. Примеры лямбда-выражений с соответствующими функциональными интерфейсами

Сценарий использования	Пример лямбда-выражения	Соответствующий функциональный интерфейс
Булево выражение	(List<String> list) -> list.isEmpty()	Predicate<List<String>>
Создание объектов	() -> new Apple(10)	Supplier<Apple>
Потребление данных из объекта	(Apple a) -> System.out.println(a.getWeight())	Consumer<Apple>
Выборка/извлечение данных из объекта	(String s) -> s.length()	Function<String, Integer> или ToIntFunction<String>
Комбинация двух значений	(int a, int b) -> a * b	IntBinaryOperator
Сравнение двух объектов	(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())	Comparator<Apple> или BiFunction<Apple, Apple, Integer> или ToIntBiFunction<Apple, Apple>

Исключения, лямбда-выражения и функциональные интерфейсы

Обратите внимание, что ни один из функциональных интерфейсов не позволяет генерировать проверяемые исключения. Если вам нужно, чтобы в теле лямбда-выражения генерировалось исключение, есть два варианта: описать собственный функциональный интерфейс, объявляющий проверяемое исключение, или обернуть тело лямбда-выражения в блок `try/catch`.

Например, в разделе 3.3 мы создали новый функциональный интерфейс `BufferedReaderProcessor`, явным образом объявляющий исключение `IOException`:

```
@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
BufferedReaderProcessor p = (BufferedReader br) -> br.readLine();
```

Но, возможно, вы используете API, ожидающий на входе функциональный интерфейс вроде `Function<T, R>`, и нет возможности создать собственный. Вы увидите в следующей главе, что Stream API активно использует функциональные интерфейсы из табл. 3.2. В данном случае можно явным образом перехватить проверяемое исключение:

```
Function<BufferedReader, String> f =
  (BufferedReader b) -> {
    try {
      return b.readLine();
    }
    catch(IOException e) {
      throw new RuntimeException(e);
    }
  };
```

Теперь вы уже знаете, как создавать лямбда-выражения, а также где и как их использовать. Далее мы приведем дополнительные подробности: о проверке типов лямбда-выражений компилятором и правилах, о которых не помешает знать, например, относительно лямбда-выражений со ссылками на локальные переменные внутри тела, а также `void`-совместимых лямбда-выражений. Нет необходимости понимать сейчас весь изложенный в следующем разделе материал — вы можете вернуться к нему позднее, а пока перейти к разделу 3.6, посвященному ссылкам на методы.

3.5. Проверка типов, вывод типов и ограничения

Когда мы впервые упоминали лямбда-выражения, то сказали, что они дают возможность генерировать экземпляры функциональных интерфейсов. Тем не менее само по себе лямбда-выражение не содержит информации о том, реализацией какого функционального интерфейса оно является. Для формализации представления о лямбда-выражениях необходимо рассказать, что такое тип лямбда-выражения.

3.5.1. Проверка типов

Тип лямбда-выражения выводится исходя из контекста, в котором оно используется. Ожидаемый внутри контекста тип лямбда-выражения (например, тип параметра метода, в который оно передается, или локальной переменной, которой оно присваивается) называется *целевым типом* (target type). Взглянем на пример, чтобы разобраться, что происходит «под капотом» при использовании

лямбда-выражения. Рисунок 3.4 вкратце описывает процесс проверки типа для следующего кода:

```
List<Apple> heavierThan150g =
        filter(inventory, (Apple apple) -> apple.getWeight() > 150);
```

Процесс проверки типа состоит из следующих этапов.

1. Выполняется поиск объявления метода `filter`.
2. Этот метод в качестве второго формального параметра ожидает объект типа `Predicate<Apple>` (целевой тип).
3. `Predicate<Apple>` — функциональный интерфейс, описывающий один абстрактный метод `test`.
4. Метод `test` описывает функциональный дескриптор, принимающий `Apple` и возвращающий `boolean`.
5. Любой аргумент метода `filter` должен соответствовать этому требованию.

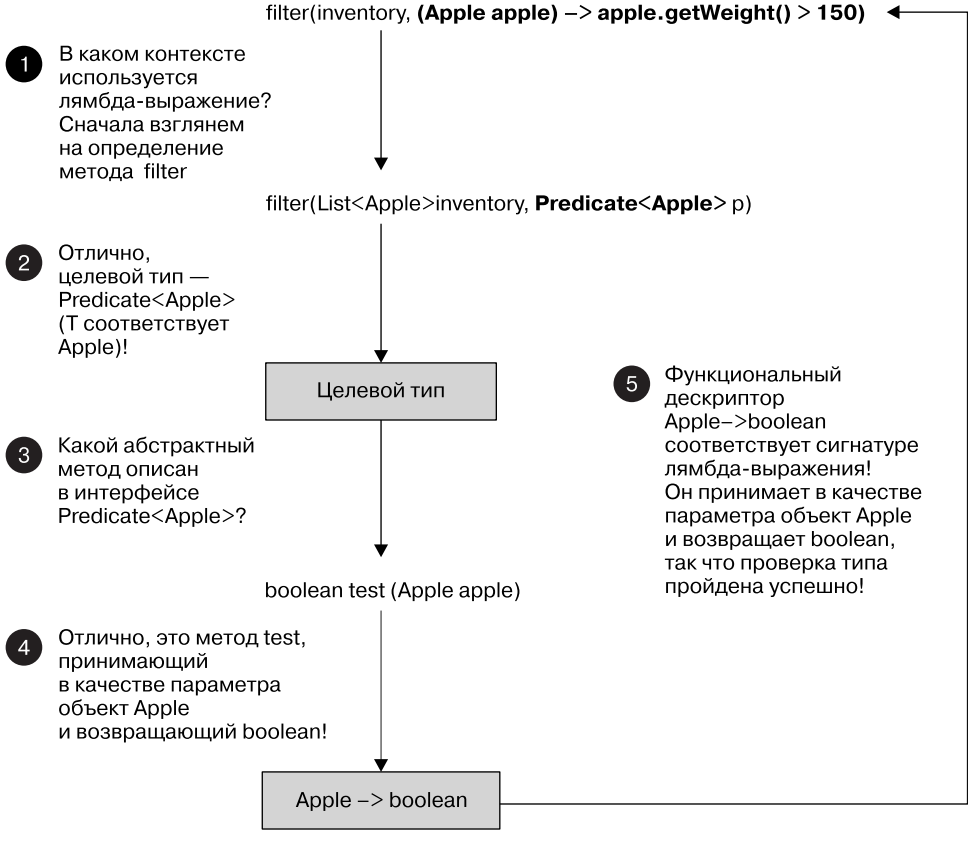


Рис. 3.4. Пошаговый разбор процесса проверки типа лямбда-выражения

Этот код корректен, поскольку передаваемое лямбда-выражение тоже принимает в качестве параметра объект `Apple` и возвращает `boolean`. Обратите внимание, что если лямбда-выражение генерирует исключение, то должен совпадать и пункт `throws` из объявления абстрактного метода.

3.5.2. То же лямбда-выражение, другие функциональные интерфейсы

Благодаря концепции *ожидаемого в данном контексте типа* (target typing) одно и то же лямбда-выражение может относиться к различным функциональным интерфейсам, если сигнатуры их абстрактных методов совместимы. Например, оба вышеописанных интерфейса `Callable` и `PrivilegedAction` не принимают параметров и возвращают обобщенный тип `T`. То есть допустимы обе приведенные ниже операции присвоения:

```
Callable<Integer> c = () -> 42;
PrivilegedAction<Integer> p = () -> 42;
```

В данном случае целевой тип первой операции присвоения — `Callable<Integer>`, а второй — `PrivilegedAction<Integer>`.

В табл. 3.3 мы демонстрировали похожий пример: одно лямбда-выражение может использоваться с несколькими разными функциональными интерфейсами:

```
Comparator<Apple> c1 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
ToIntBiFunction<Apple, Apple> c2 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
BiFunction<Apple, Apple, Integer> c3 =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Ромбовидный оператор

Хорошо знакомые с историей эволюции языка Java вспомнят, что идея вывода конкретного типа для обобщенного из контекста с помощью ромбовидного оператора (`<>`) появилась уже в Java 7 (для обобщенных методов даже еще раньше). Одно и то же выражение создания экземпляра класса может встретиться в двух или более контекстах с выводом соответствующего типа аргумента, как показано в следующем коде:

```
List<String> listOfStrings = new ArrayList<>();
List<Integer> listOfIntegers = new ArrayList<>();
```

Специальное правило void-совместимости

Если тело лямбда-выражения представляет собой выражение-оператор, то оно совместимо с возвращающим `void` функциональным дескриптором (при условии, что список параметров тоже совместим). Например, обе следующие строки кода допустимы, несмотря на то что метод `add` интерфейса `List` возвращает `boolean`, а не `void`, как ожидается в контексте интерфейса `Consumer` (`T -> void`):

```
// В Predicate возвращается boolean
Predicate<String> p = (String s) -> list.add(s);
```

```
// В Consumer возвращается void
Consumer<String> b = (String s) -> list.add(s);
```

К настоящему моменту, вероятно, вы уже хорошо понимаете, когда и где можно использовать лямбда-выражения. Целевой тип может выводиться из контекста присваивания, контекста вызова метода (параметры и возвращаемый тип), а также контекста приведения типов. Попробуйте проверить свои знания на контрольном задании 3.5.

Контрольное задание 3.5. Проверка типов — почему следующий код не скомпилируется?

Как решить данную проблему?

```
Object o = () -> { System.out.println("Tricky example"); };
```

Ответ: контекст лямбда-выражения — `Object` (целевой тип). Но `Object` — это не функциональный интерфейс. Для исправления ситуации можно поменять целевой тип на интерфейс `Runnable`, который соответствует функциональному дескриптору `() -> void`:

```
Runnable r = () -> { System.out.println("Tricky example"); };
```

Можно также исправить ситуацию путем приведения типа лямбда-выражения к `Runnable`, чтобы явным образом указать целевой тип.

```
Object o = (Runnable) () -> { System.out.println("Tricky example"); };
```

Такая методика может пригодиться и в контексте перегрузки, когда метод принимает в качестве параметров два различных функциональных интерфейса с одинаковым функциональным дескриптором. С помощью приведения типа лямбда-выражения можно явным образом устранить неоднозначность относительно того, какая сигнатура метода должна быть выбрана.

Например, неоднозначным может оказаться вызов `execute(() -> {})`, как показано в следующем коде, поскольку функциональные дескрипторы интерфейсов `Runnable` и `Action` совпадают:

```
public void execute(Runnable runnable) {
    runnable.run();
}
public void execute(Action<T> action) {
    action.act();
}
@FunctionalInterface
interface Action {
    void act();
}
```

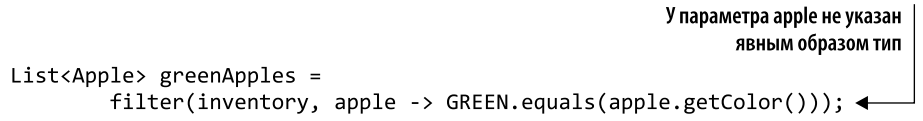
Но устранить неоднозначность этого вызова можно явным образом с помощью выражения приведения типа:

```
execute ((Action) () -> {});
```

Мы показали вам, как с применением целевого типа можно проверить, допустимо ли лямбда-выражение в данном контексте. С его помощью можно решить и несколько отличную задачу по выводу типов параметров лямбда-выражения.

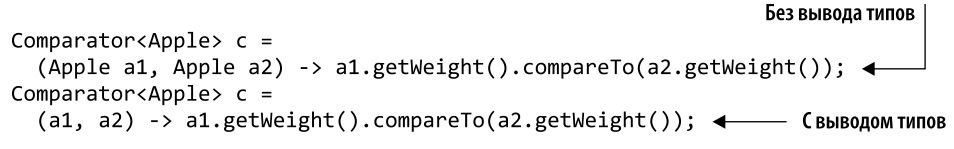
3.5.3. Вывод типов

Вы можете еще немного упростить свой код. Компилятор Java определяет по контексту (целевому типу), какой функциональный интерфейс нужно связать с лямбда-выражением, а значит, он способен определить и подходящую сигнатуру для этого лямбда-выражения, ведь функциональный дескриптор можно узнать по целевому типу. Выгода в том, что у компилятора есть доступ к информации о типах параметров лямбда-выражений, а значит, их можно опустить в синтаксисе лямбда-выражения. В показанном ниже примере компилятор Java выводит типы параметров[1]:




```
List<Apple> greenApples =
        filter(inventory, apple -> GREEN.equals(apple.getColor()));
```

Повышение удобочитаемости кода еще заметнее в случае лямбда-выражений с несколькими параметрами. Например, вот так можно создать объект `Comparator`:




```
Comparator<Apple> c =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

Отметим, что в одних случаях удобочитаемость кода оказывается выше с явным указанием типов, а в других — без него. Какой вариант лучше — зависит от обстоятельств, единого правила на этот счет нет; разработчики должны сами выбирать вариант, при котором их код окажется понятнее.

3.5.4. Использование локальных переменных

Внутри тела всех показанных до сих пор лямбда-выражений использовались только их аргументы. Но в лямбда-выражениях, как и в анонимных классах, могут применяться *свободные переменные* (free variables) — переменные, не являющиеся параметрами и описанные во внешней области видимости. Такие лямбда-выражения называются *лямбда-выражениями с захватом переменных* (capturing lambdas). Например, следующее лямбда-выражение захватывает переменную `portNumber`:

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

Тем не менее здесь есть нюанс в виде ограничений списка того, что можно делать с такими переменными. Лямбда-выражения могут без ограничений захватывать (чтобы ссылаться на них в своем теле) переменные экземпляра и статические переменные. Но захватываемые локальные переменные должны быть явным образом

[1] Отметим, что если у лямбда-выражения только один параметр, тип которого выводится компилятором, то скобки, окружающие имя параметра, тоже можно опустить.

объявлены как `final` или фактически являться таковыми. Лямбда-выражения могут захватывать лишь локальные переменные, значение которых задается однократно (примечание: захват переменной экземпляра класса можно рассматривать как захват локальной `final`-переменной `this`). Например, следующий код не скомпилируется, поскольку переменной `portNumber` значение присваивается дважды:

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

Ошибка: локальная переменная не объявлена final и фактически таковой не является

Ограничения, налагаемые на локальные переменные. Возможно, вы недоумеваете: почему на локальные переменные накладываются такие ограничения? Во-первых, существует ключевое различие во внутренней реализации переменных экземпляра класса и локальных переменных. Переменные экземпляра класса (объекта) хранятся в куче, в то время как локальные переменные располагаются в стеке. Если бы лямбда могла обратиться к локальной переменной напрямую, причем использовалась в отдельном потоке выполнения, то этот поток мог бы попытаться обратиться к данной переменной после того, как поток, выделивший под нее память, освободит ресурсы. Поэтому доступ к свободной локальной переменной в Java реализован в виде доступа к ее копии, а не к исходной переменной. Если присвоение значения переменной выполняется лишь однократно, то никакой разницы нет — отсюда и возникает вышеописанное ограничение.

Во-вторых, это ограничение также препятствует реализации типичных паттернов императивного программирования (которые, как мы поясним в дальнейших главах, затрудняют распараллеливание), связанных с изменением значения внешней переменной.

Замыкание

Возможно, вы слышали термин *«замыкание»* (closure, не путайте с языком программирования Clojure) и вам интересно, соответствуют ли его определению лямбда-выражения. На более формальном языке *замыкание* — экземпляр функции с возможностью без всяких ограничений обращаться к нелокальным по отношению к этой функции переменным. Например, замыкание можно передать как аргумент другой функции. Оно может также *обращаться* к переменным, объявленным вне его области видимости, и *изменять* их значения. Лямбда-выражения и анонимные классы Java 8 похожи на замыкания: их можно передавать в качестве аргументов методам, они могут обращаться к переменным вне своей области видимости. Но на них налагается ограничение: они не могут модифицировать содержимое локальных переменных метода, в котором описаны. Эти переменные фактически являются неизменяемыми (`final`). Удобно считать, что лямбда-выражения замыкают *значения*, а не *переменные*. Как уже объяснялось ранее, это ограничение связано с тем, что локальные переменные располагаются в стеке и неявным образом ограничены своим потоком выполнения. Если разрешить захват изменяемых локальных переменных, это приведет к открытию новых потоконебезопасных, а значит, нежелательных возможностей (переменные экземпляра в этом смысле допустимы, поскольку хранятся в куче, совместно используемой разными потоками выполнения).

Далее мы опишем еще одну замечательную возможность, появившуюся в Java 8: *ссылки на методы* (method references). Можете считать их сокращенным вариантом обозначения определенных лямбда-выражений.

3.6. Ссылки на методы

С помощью ссылок на методы можно переиспользовать существующие определения методов и передавать их подобно лямбда-выражениям. В некоторых случаях они оказываются более удобочитаемыми и выглядят естественнее, чем лямбда-выражения. Рассмотрим наш пример сортировки, переписанный с помощью ссылки на метод, а также обновленного API Java 8 (мы обсудим этот пример подробнее в разделе 3.7).

Раньше было:

```
inventory.sort((Apple a1, Apple a2) ->
   a1.getWeight().compareTo(a2.getWeight()));
```

Теперь (при использовании ссылки на метод и метода `java.util.Comparator.comparing`):

```
inventory.sort(comparing(Apple::getWeight));
```

Ваша первая ссылка на метод

Не волнуйтесь насчет нового синтаксиса и того, как что работает. В следующих нескольких разделах мы все вам расскажем!

3.6.1. В общих чертах

Чем интересны ссылки на методы? Их можно рассматривать как вариант сокращенного написания для лямбда-выражений, вызывающих только определенный метод. Основная идея: если лямбда-выражение говорит «вызовите этот метод напрямую», то лучше сослаться на метод по названию, а не по описанию способа его вызова. Конечно, благодаря ссылке на метод можно создать лямбда-выражение из существующей реализации метода. Но явная отсылка к названию метода *повышает удобочитаемость кода*. Как это работает? Для создания ссылки на метод необходимо указать целевую ссылку перед разделителем `::`, а название метода — после него. Например, `Apple::getWeight` представляет собой ссылку на метод `getWeight`, объявленный в классе `Apple` (помните, что скобки после `getWeight` не нужны, поскольку вы не вызываете этот метод сейчас, а просто цитируете его название). Данная ссылка на метод является сокращением для лямбда-выражения `(Apple apple) -> apple.getWeight()`. В табл. 3.4 приведено еще несколько примеров ссылок на методы в Java 8.

Таблица 3.4. Примеры лямбда-выражений и эквивалентных ссылок на методы

Лямбда-выражение	Эквивалентная ссылка на метод
(Apple apple) -> apple.getWeight()	Apple::getWeight
() -> Thread.currentThread().dumpStack()	Thread.currentThread()::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println
(String s) -> this.isValidName(s)	this::isValidName

Ссылки на методы можно рассматривать как «синтаксический сахар» для лямбда-выражений, поскольку с их помощью можно писать меньше кода для выражения того же самого.

Правила формирования ссылок на методы. Существует три основные разновидности ссылок на методы.

- Ссылки на *статические методы* (например, ссылка на метод `parseInt` класса `Integer` имеет вид `Integer::parseInt`).
- Ссылки на методы экземпляра произвольного типа (например, ссылка на метод `length` класса `String` имеет вид `String::length`).
- Ссылки на *методы экземпляра существующего объекта или выражения* (допустим, что у вас есть локальная переменная `expensiveTransaction`, содержащая объект типа `Transaction`, который поддерживает метод экземпляра `getValue`; в этом случае вы можете написать ссылку на метод `expensiveTransaction::getValue`).

Вторая и третья разновидности ссылок на методы могут сначала привести в легкое замешательство. Идея второго варианта ссылок (`String::length`) состоит в ссылке на метод объекта, который будет передан в качестве одного из параметров лямбда-выражения. Например, лямбда-выражение `(String s) -> s.toUpperCase()` можно переписать в виде `String::toUpperCase`. А третья разновидность ссылок относится к ситуации, когда в лямбда-выражении вызывается метод уже существующего внешнего объекта. В частности, лямбда-выражение `() -> expensiveTransaction.getValue()` можно переписать в виде `expensiveTransaction::getValue`. Эта разновидность ссылок на методы особенно удобна, когда нужно передать метод, который описан как приватный вспомогательный метод. Например, пусть есть определение вспомогательного метода `isValidName`:

```
private boolean isValidName(String string) {
    return Character.isUpperCase(string.charAt(0));
}
```

Теперь его можно передавать в контексте `Predicate<String>` с помощью следующей ссылки на метод:

```
filter(words, this::isValidName)
```

Чтобы помочь вам разобраться в этой новой информации, на рис. 3.5 приведены краткие инструкции по преобразованию лямбда-выражений в эквивалентные ссылки на методы.

Отметим также, что существуют специальные формы ссылок на методы для конструкторов, конструкторов массивов и вызовов методов суперкласса. Рассмотрим ссылки на методы на конкретном примере. Допустим, нужно отсортировать список строк без учета регистра. Метод `sort` интерфейса `List` ожидает в качестве параметра объект типа `Comparator`. Как вы видели ранее, объекту `Comparator` соответствует функциональный дескриптор с сигнатурой `(T, T) -> int`. Мы можем описать лямбда-выражение, использующее метод `compareToIgnoreCase` из класса `String`, следующим образом (обратите внимание, что метод `compareToIgnoreCase` уже описан в классе `String`):

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));
```

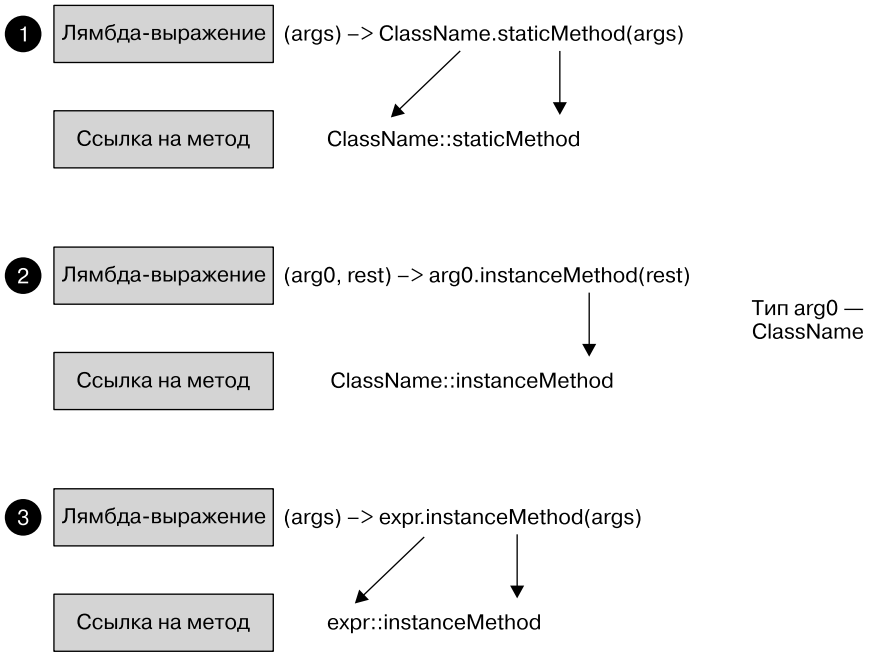


Рис. 3.5. Инструкции по конструированию ссылок на методы для трех различных типов лямбда-выражений

Сигнатура этого лямбда-выражения совместима с функциональным дескриптором интерфейса `Comparator`. Согласно приведенным выше инструкциям можно переписать этот пример с помощью ссылки на метод; получится более лаконичный код, вот такой:

```
List<String> str = Arrays.asList("a","b","A","B");
str.sort(String::compareToIgnoreCase);
```

Отметим, что компилятор осуществляет такой же процесс проверки типов, как и для лямбда-выражений, чтобы выяснить, подходит ли ссылка на метод для данного функционального интерфейса. Сигнатура ссылки на метод должна соответствовать типу контекста.

Чтобы проверить, насколько хорошо вы разбираетесь в ссылках на методы, попробуйте выполнить контрольное задание 3.6!

Контрольное задание 3.6. Ссылки на методы

Каковы эквивалентные ссылки на методы для следующих лямбда-выражений?

1.
```
ToIntFunction<String> stringToInt =
        (String s) -> Integer.parseInt(s);
```
2.
```
BiPredicate<List<String>, String> contains =
        (list, element) -> list.contains(element);
```

3. `Predicate<String> startsWithNumber =`
   `(String string) -> this.startsWithNumber(string);`

Ответы

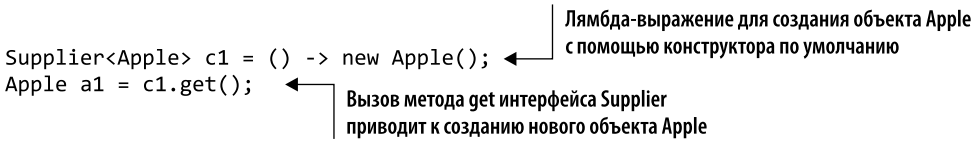
1. Это лямбда-выражение передает свой аргумент статическому методу `parseInt` класса `Integer`. Метод принимает в качестве параметра объект `String`, предназначенный для синтаксического разбора, и возвращает `int`. В результате лямбда-выражение можно переписать на основе инструкции ❶ из рис. 3.5 (лямбда-выражения, вызывающие статический метод):
   ```
   ToIntFunction<String> stringToInt = Integer::parseInt;
   ```
2. Метод `contains` вызывается для первого аргумента этого лямбда-выражения. А поскольку тип первого аргумента `List`, то можно воспользоваться инструкцией ❷ из рис. 3.5:
   ```
   BiPredicate<List<String>, String> contains = List::contains;
   ```
   Дело в том, что целевой тип описывает функциональный дескриптор `(List<String>, String) -> boolean`, до которого можно развернуть `List::contains`.
3. Эта лямбда в форме выражения вызывает приватный вспомогательный метод. Здесь можно воспользоваться инструкцией ❸ из рис. 3.5:
   ```
   Predicate<String> startsWithNumber = this::startsWithNumber
   ```

Мы продемонстрировали только переиспользование существующих реализаций методов и создание ссылок на методы. Но нечто подобное можно делать и с конструкторами классов.

3.6.2. Ссылки на конструкторы

Для создания ссылки на уже существующий конструктор достаточно его названия и ключевого слова `new`. Ссылка выглядит как `ClassName::new` и работает аналогично ссылке на статический метод. Например, пускай у нас есть конструктор без аргументов. Он соответствует сигнатуре `() -> Apple` интерфейса `Supplier`. Мы можем сделать так:

что эквивалентно следующему:



При наличии конструктора с сигнатурой `Apple(Integer weight)`, соответствующего сигнатуре интерфейса `Function`, можно сделать следующее:

```
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);
```

Ссылка на конструктор Apple (Integer weight)

Вызов метода apply интерфейса Function с заданным весом в качестве параметра приводит к генерации объекта Apple

что эквивалентно:

```
Function<Integer, Apple> c2 = (weight) -> new Apple(weight);
Apple a2 = c2.apply(110);
```

Лямбда-выражение для создания объекта Apple с заданным весом

Вызов метода apply интерфейса Function с заданным весом приводит к генерации нового объекта Apple

В коде далее каждый из элементов списка (`List`) объектов `Integer` передается конструктору класса `Apple` с помощью метода `map`, аналогичного описанному нами ранее, в результате чего получается список `List` яблок различного веса:

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);
public List<Apple> map(List<Integer> list, Function<Integer, Apple> f) {
    List<Apple> result = new ArrayList<>();
    for(Integer i: list) {
        result.add(f.apply(i));
    }
    return result;
}
```

Передача ссылки на конструктор в метод map

Конструктор с двумя аргументами, например `Apple (Color color, Integer weight)`, соответствует сигнатуре интерфейса `BiFunction`, так что можно сделать вот что:

```
BiFunction<Color, Integer, Apple> c3 = Apple::new;
Apple a3 = c3.apply(GREEN, 110);
```

Ссылка на конструктор Apple (Color color, Integer weight)

Вызов метода apply интерфейса BiFunction с заданными цветом и весом приводит к генерации нового объекта Apple

а это эквивалентно следующему:

```
BiFunction<String, Integer, Apple> c3 =
    (color, weight) -> new Apple(color, weight);
```

Лямбда-выражение для создания объекта Apple с заданными цветом и весом

```
Apple a3 = c3.apply(GREEN, 110);
```

Вызов метода apply интерфейса BiFunction с заданными цветом и весом приводит к генерации нового объекта Apple

Возможность ссылаться на конструктор без создания экземпляра очень интересна. Например, можно воспользоваться интерфейсом `Map`, чтобы поставить конструкторам в соответствие строковые значения. В результате этого можно создать метод `giveMeFruit`, который, получая в качестве аргументов `String` и `Integer`, будет создавать разные виды фруктов различного веса, вот так:

```
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
    map.put("apple", Apple::new);
    map.put("orange", Orange::new);
    // и т. д.
}
public static Fruit giveMeFruit(String fruit, Integer weight){
    return map.get(fruit.toLowerCase())
              .apply(weight);
}
```

Получаем из ассоциативного массива Function<Integer,Fruit>

Вызов метода apply интерфейса Function с параметром веса создает нужный объект Fruit

Для проверки своего понимания ссылок на методы и конструкторы попробуйте выполнить контрольное задание 3.7.

Контрольное задание 3.7. Ссылки на конструкторы

Вы уже видели, как преобразовывать конструкторы без аргументов, с одним аргументом и с двумя аргументами в ссылки на конструкторы. А что, по вашему мнению, следует сделать для создания ссылки на конструктор с тремя аргументами, например `RGB(int, int, int)`?

Ответ: синтаксис для ссылки на конструктор имеет вид `ClassName::new`, так что в данном случае получается `RGB::new`. Но нам нужен функциональный интерфейс, который бы соответствовал сигнатуре этой ссылки на конструктор. А поскольку такого в начальном комплекте функциональных интерфейсов Java нет, придется создать свой собственный:

```
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

Теперь можно использовать вышеприведенную ссылку на конструктор следующим образом:

```
TriFunction<Integer, Integer, Integer, RGB> colorFactory = RGB::new;
```

Мы рассмотрели в этой главе немало новой информации: лямбда-выражения, функциональные интерфейсы и ссылки на методы. В следующем разделе применим все это на практике.

3.7. Практическое использование лямбда-выражений и ссылок на методы

В завершение этой главы и обсуждения лямбда-выражений мы продолжим решение нашей исходной задачи сортировки списка объектов `Apple` в различном порядке. Мы покажем вам, как постепенно сделать это «наивное» решение лаконичным с помощью всех концепций и возможностей, описанных ранее в этой книге: параметризации поведения, анонимных классов, лямбда-выражений и ссылок на методы. Итоговое решение, к которому мы будем стремиться, имеет следующий вид (обратите внимание, что весь исходный код есть на сайте данной книги: http://www.manning.com/books/modern-java-in-action):

```
inventory.sort(comparing(Apple::getWeight));
```

3.7.1. Шаг 1: передаем код

Вам повезло: в API Java 8 для объектов `List` уже доступен метод `sort`, так что реализовывать его не придется. Самое сложное уже сделано! Но как передать методу `sort` информацию о требуемом порядке сортировки? Что ж, сигнатура метода `sort` выглядит следующим образом:

```
void sort(Comparator<? super E> c)
```

Он ожидает в качестве аргумента объект `Comparator` для сравнения двух `Apple`. Именно так в Java и передаются различные стратегии упорядочения: путем обертывания их в какой-либо объект. Это называется *параметризацией поведения* метода `sort`: его поведение меняется в зависимости от переданной в него стратегии упорядочения.

Наше первое решение выглядит следующим образом:

```
public class AppleComparator implements Comparator<Apple> {
        public int compare(Apple a1, Apple a2){
                return a1.getWeight().compareTo(a2.getWeight());
        }
}
inventory.sort(new AppleComparator());
```

3.7.2. Шаг 2: используем анонимный класс

Вместо того чтобы реализовывать `Comparator` ради создания его объекта один-единственный раз, можно усовершенствовать наше решение с помощью анонимного класса:

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

3.7.3. Шаг 3: используем лямбда-выражения

Наше текущее решение все еще недостаточно лаконично. В Java 8 появились лямбда-выражения, позволяющие решить ту же задачу — *передачи кода* — с помощью более простого синтаксиса. Вы уже знаете, что лямбда-выражение можно использовать везде, где ожидается *функциональный интерфейс*. Напомним, что функциональный интер-

фейс — это интерфейс, в котором описан ровно один абстрактный метод. Сигнатура такого абстрактного метода *(функциональный дескриптор)* может описывать сигнатуру лямбда-выражения. В данном случае интерфейсу `Comparator` соответствует функциональный дескриптор `(T, T) -> int`. А поскольку речь идет об объектах `Apple`, то, если точнее, `(Apple, Apple) -> int`. Наше новое, усовершенствованное решение выглядит вот так:

```
inventory.sort((Apple a1, Apple a2)
                -> a1.getWeight().compareTo(a2.getWeight())
);
```

Мы уже рассказывали, что компилятор Java может *определить (вывести) типы* параметров лямбда-выражения на основе контекста, в котором данное лямбда-выражение встречается. Следовательно, наше решение можно переписать следующим образом:

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

Можно ли сделать наш код еще более удобочитаемым? В интерфейсе `Comparator` есть статический вспомогательный метод `comparing`. Он принимает в качестве параметра функцию (объект `Function`), служащую для извлечения ключа типа `Comparable`, и возвращает объект `Comparator` (в главе 13 мы объясним, почему интерфейсы могут включать статические методы). Использовать его можно следующим образом (обратите внимание, что лямбда-выражение теперь передается только с одним аргументом; это лямбда-выражение задает способ извлечения ключа сравнения из объекта `Apple`):

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

С учетом этого можно переписать наше решение в чуть более сжатом виде:

```
import static java.util.Comparator.comparing;
inventory.sort(comparing(apple -> apple.getWeight()));
```

3.7.4. Шаг 4: используем ссылки на методы

Мы уже объясняли, что ссылки на методы — это фактически «синтаксический сахар» для передачи аргументов лямбда-выражений. С помощью ссылки на метод можно сделать наш код еще более лаконичным (при условии, что выполнен статический импорт `java.util.Comparator.comparing`):

```
inventory.sort(comparing(Apple::getWeight));
```

Поздравляем, это наше окончательное решение! Этот код лучше того, который был возможен до Java 8, и не только своей лаконичностью, но и понятностью. Его можно прочитать в полном соответствии с формулировкой задачи как «отсортировать запасы, сравнивая яблоки по весу».

3.8. Методы, удобные для композиции лямбда-выражений

Несколько функциональных интерфейсов из API Java 8 содержат удобные для этой цели методы, упрощающие работу. В частности, множество используемых для передачи лямбда-выражений функциональных интерфейсов, например `Comparator`,

`Function` и `Predicate`, предоставляют методы, допускающие композицию. Что это значит? На практике здесь подразумевается, что можно объединять несколько простых лямбда-выражений для создания более сложных. Например, можно объединить два предиката в один большой, выполняющий над этими двумя операцию `or`. Более того, можно также объединять функции таким образом, что результат выполнения одной станет входными данными для другой. Возможно, вы недоумеваете: откуда в функциональном интерфейсе могут взяться дополнительные методы (в конце концов, это противоречит определению функционального интерфейса!). Хитрость в том, что эти методы, с которыми мы вас познакомим, называются *методами с реализацией по умолчанию* (default methods). Мы расскажем о них подробнее в главе 13. Пока просто поверьте нам на слово, а главу 13 прочитайте потом, когда захотите больше узнать о методах с реализацией по умолчанию и о том, что с их помощью можно сделать.

3.8.1. Композиция объектов Comparator

Как вы видели, статическим методом `Comparator.comparing` можно воспользоваться для того, чтобы вернуть `Comparator`, основанный на объекте `Function`, извлекающем ключ сравнения:

```
Comparator<Apple> c = Comparator.comparing(Apple::getWeight);
```

Обратный порядок сортировки

Но что, если требуется отсортировать яблоки по убыванию веса? Создавать для этого отдельный экземпляр интерфейса `Comparator` не нужно. В данном интерфейсе существует метод с реализацией по умолчанию `reversed`, который меняет порядок сортировки заданного компаратора на обратный. Для сортировки яблок по убыванию веса можно модифицировать предыдущий пример и переиспользовать исходный объект `Comparator` следующим образом:

```
inventory.sort(comparing(Apple::getWeight).reversed());
```

Сортировка по убыванию веса

Организация объектов Comparator цепочкой

Все бы хорошо, но что, если вам попадутся два яблока одинакового веса? Какое из них должно идти первым в отсортированном списке? Имеет смысл добавить второй объект `Comparator` для дальнейшего уточнения сравнения. Например, после сравнения двух яблок по весу можно сравнить их по стране происхождения. Сделать это можно с помощью метода `thenComparing`. Он принимает функцию в качестве параметра (подобно методу `comparing`), позволяя указать второй объект `Comparator` на случай, если два объекта на основе исходного `Comparator` были сочтены равными. Задачу опять же можно решить довольно изящным способом:

```
inventory.sort(comparing(Apple::getWeight)
         .reversed()
         .thenComparing(Apple::getCountry));
```

Сортировка по убыванию веса

Дальнейшая сортировка по стране в случае двух яблок одинакового веса

3.8.2. Композиция предикатов

Интерфейс `Predicate` включает три метода, с помощью которых можно переиспользовать существующий объект `Predicate` для создания более сложных предикатов: `negate`, `and` и `or`. Например, можно вернуть отрицание предиката с помощью метода `negate`, скажем, найти все яблоки не красного цвета:

```
Predicate<Apple> notRedApple = redApple.negate();
```

Выполняет отрицание существующего объекта redApple типа Predicate

Можно также объединить два лямбда-выражения, чтобы найти красные *и* тяжелые яблоки с помощью метода `and`:

```
Predicate<Apple> redAndHeavyApple =
    redApple.and(apple -> apple.getWeight() > 150);
```

Объединяем цепочкой два предиката и получаем новый объект Predicate

Далее можно добавить в эту цепочку предикатов еще один, чтобы он соответствовал тяжелым красным (более 150 г) или только зеленым яблокам:

```
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(apple -> apple.getWeight() > 150)
            .or(apple -> GREEN.equals(a.getColor()));
```

Формирование более сложного объекта Predicate путем организации цепочкой трех предикатов

Что в этом такого замечательного? Из простых лямбда-выражений можно составлять более сложные, но по-прежнему ясно отражающие формулировку задачи! Заметим, что приоритет методов `and` и `or` в цепочке — слева направо, эквивалента скобок здесь не существует. Так что `a.or(b).and(c)` следует читать как `(a || b) && c`. Аналогично `a.and(b).or(c)` нужно читать как `(a && b) || c`.

3.8.3. Композиция функций

И наконец, можно выполнять композицию лямбда-выражений, представленных с помощью интерфейса `Function`. В интерфейсе `Function` для этой цели предусмотрено два метода — `andThen` и `compose`, и оба возвращают экземпляр `Function`.

Метод `andThen` возвращает функцию, которая сначала применяет заданную функцию к входным данным, а затем применяет другую заданную функцию к полученному результату. Например, если даны функция `f`, увеличивающая число на 1: `(x -> x + 1)`, и другая функция `g`, умножающая число на 2, то можно их объединить для создания функции `h`, которая сначала увеличивает число на 1, а затем умножает полученный результат на 2:

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);
int result = h.apply(1);
```

На математическом языке это можно записать как g(f(x)) или (g∘f)(x)

Результат равен 4

Можно также воспользоваться методом `compose` и сначала применить заданную в качестве аргумента `compose` функцию, а затем применить вторую функцию

к результату. Например, в предыдущем примере при использовании `compose` получилось бы `f(g(x))` вместо `g(f(x))` при использовании `andThen`:

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);
```

На математическом языке это можно записать как f(g(x)) или (f ∘ g)(x)

Результат равен 3

Рисунок 3.6 иллюстрирует различие между методами `andThen` и `compose`.

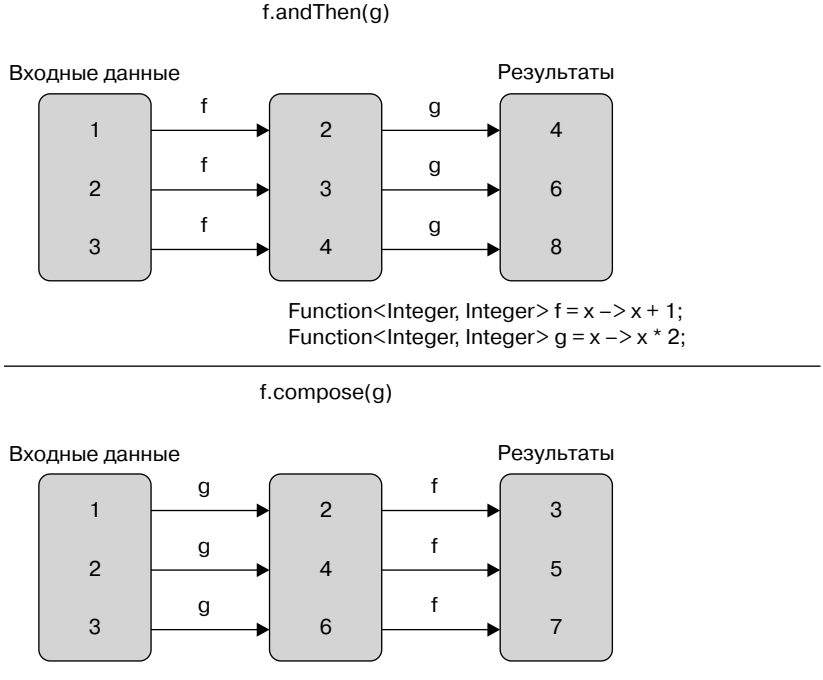


Рис. 3.6. andThen по сравнению с compose

Это звучит несколько отвлеченно. Как реализовать все это на практике? Допустим, у нас есть несколько вспомогательных методов для текстовых преобразований писем, представленных в виде объектов `String`:

```
public class Letter{
    public static String addHeader(String text) {
        return "From Raoul, Mario and Alan: " + text;
    }
    public static String addFooter(String text) {
        return text + " Kind regards";
    }
    public static String checkSpelling(String text) {
        return text.replaceAll("labda", "lambda");
    }
}
```

Путем композиции этих вспомогательных методов теперь можно создавать различные конвейеры преобразований. Например, можно создать конвейер, в котором сначала будет добавлен заголовок, затем выполнена проверка орфографии и, наконец, добавлена подпись, как показано в следующем коде (и проиллюстрировано на рис. 3.7):

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::checkSpelling)
             .andThen(Letter::addFooter);
```

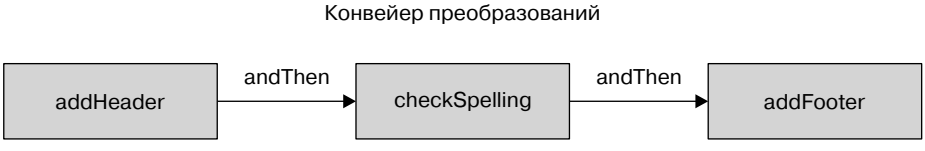


Рис. 3.7. Создание конвейера преобразований с помощью метода andThen

Еще в одном конвейере могут только добавляться заголовок и подпись, без проверки орфографии:

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::addFooter);
```

3.9. Схожие математические идеи

Те читатели, которые хорошо ориентируются в математике в объеме средней школы, смогут взглянуть в этом разделе на лямбда-выражения и передачу функций с другой точки зрения. Можете спокойно его пропустить; оставшаяся часть книги никоим образом не основана на этом материале. Но, возможно, вам интересно будет взглянуть на проблему с другой стороны.

3.9.1. Интегрирование

Пусть у нас есть (математическая, не Java-) функция `f` со следующим определением:

$$f(x) = x + 10.$$

Часто встречающаяся задача (в школе, а также в научных и инженерных исследованиях): поиск площади под графиком функции (считая ось X линией нуля). Например, вычислить площадь области, показанной на рис. 3.8, можно следующим образом:

$$\int_3^7 f(x)dx \text{ или } \int_3^7 f(x+10)dx.$$

В этом примере функция f линейная, так что можно легко найти нужную площадь методом трапеций (разбивая область на треугольники и прямоугольники) и получить следующее решение:

$$1 / 2 \cdot ((3 + 10) + (7 + 10)) \cdot (7 - 3) = 60.$$

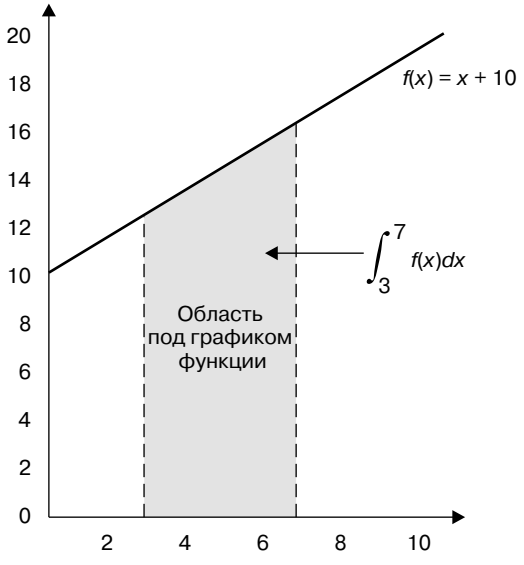


Рис. 3.8. Область под графиком функции f(x) = x + 10 для x от 3 до 7

А как же выразить это на языке Java? Первая задача — найти соответствие в привычной нотации языка программирования для такой странной нотации, как символ интеграла или `dy/dx`.

Конечно, исходя из базовых принципов программирования, необходимо создать метод (скажем, `integrate`), который бы принимал три аргумента: первый — `f`, а остальные два — пределы интегрирования (в данном случае 3,0 и 7,0). Иным словами, хотелось бы написать на языке Java что-то вроде следующего, где функция `f` передается в виде аргумента:

```
integrate(f, 3, 7)
```

Обратите внимание, что написать еще проще, вот так:

```
integrate(x + 10, 3, 7)
```

не получится по двум причинам. Во-первых, непонятна область определения `x`, а во-вторых, при этом будет интегрироваться значение `x + 10`, а не функция `f`.

И действительно, тайный смысл `dx` в математике звучит как: «функция, получающая аргумент `x`, значение которой равно `x + 10`».

3.9.2. Переводим на язык лямбда-выражений Java 8

Как мы уже упоминали, в Java 8 нотация вида `(double x) -> x + 10` используется как раз для этой цели; следовательно, можно написать:

```
integrate((double x) -> x + 10, 3, 7)
```

или

```
integrate((double x) -> f(x), 3, 7)
```

или, с помощью описанных выше ссылок на методы:

```
integrate(C::f, 3, 7)
```

где `C` — класс, в котором описан статический метод `f`. Идея заключается в передаче кода функции `f` методу `integrate`.

Наверное, вам интересно, как должен выглядеть сам метод `integrate`. В предположении, что `f` — линейная функция (то есть ее график представляет собой прямую линию), вы могли бы попробовать написать его в виде, подобном математическому:

```
public double integrate((double -> double) f, double a, double b) {
      return (f(a) + f(b)) * (b - a) / 2.0
}
```

Недопустимый на языке Java код!
(Нельзя писать функции так, как в математике)

Но поскольку лямбда-выражения можно использовать только в контексте, где ожидается функциональный интерфейс (в данном случае `DoubleFunction`[1]), то вам придется написать этот код вот так:

```
public double integrate(DoubleFunction<Double> f, double a, double b) {
      return (f.apply(a) + f.apply(b)) * (b - a) / 2.0;
}
```

или с помощью интерфейса `DoubleUnaryOperator`, при использовании которого также не происходит упаковки результата:

```
public double integrate(DoubleUnaryOperator f, double a, double b) {
      return (f.applyAsDouble(a) + f.applyAsDouble(b)) * (b - a) / 2.0;
}
```

Попутно заметим: немного жаль, что приходится писать `f.apply(a)` вместо простого `f(a)`, как в математике, но язык Java никак не может перейти от представления обо всех сущностях как объектах к идее о функциях как полностью независимых сущностях!

[1] С точки зрения производительности, DoubleFunction<Double> в данном случае уместнее, чем Function<Double,Double>, поскольку при этом не требуется упаковка результата.

Резюме

- *Лямбда-выражения* можно считать разновидностью анонимных функций: у них нет названия, но есть список параметров, тело, возвращаемый тип, а возможно, и список потенциально генерируемых исключений.
- С помощью лямбда-выражений можно лаконично выражать передачу кода.
- *Функциональный интерфейс* — интерфейс, в котором объявлен ровно один абстрактный метод.
- Лямбда-выражения можно использовать только в контексте, в котором ожидается функциональный интерфейс.
- С помощью лямбда-выражений можно задавать реализацию абстрактных методов функционального интерфейса на месте и *работать со всем выражением как с экземпляром функционального интерфейса*.
- В пакете Java 8 `java.util.function` можно найти целый список распространенных функциональных интерфейсов, в том числе `Predicate<T>`, `Function<T, R>`, `Supplier<T>`, `Consumer<T>` и `BinaryOperator<T>`, описанные в табл. 3.2.
- С помощью версий распространенных функциональных интерфейсов, например `Predicate<T>` и `Function<T, R>`, для простых типов данных можно избежать операций упаковки: `IntPredicate`, `IntToLongFunction` и т. д.
- Для дополнительной гибкости и возможностей переиспользования с лямбда-выражениями можно применять паттерн охватывающего выполнения (предназначенный для случаев, когда необходимо выполнить некую заданную последовательность операций, окруженную нужным для метода стереотипным кодом, например выделением ресурсов и очисткой).
- Ожидаемый тип лямбда-выражения называется *целевым* типом.
- С помощью ссылок на методы можно переиспользовать существующие реализации методов и передавать их напрямую.
- В функциональных интерфейсах, таких как `Comparator`, `Predicate` и `Function`, есть несколько методов с реализацией по умолчанию, пригодных для комбинирования лямбда-выражений.

Часть II

Функциональное программирование с помощью потоков

Вторая часть книги представляет собой углубленное исследование новых Stream API, позволяющих писать впечатляющий код для декларативной обработки коллекций данных. К концу второй части вы полностью разберетесь, что такое потоки данных и как их использовать в своем коде для лаконичной и эффективной обработки коллекций данных.

- В главе 4 вы познакомитесь с понятием потока данных и увидите, чем он отличается от коллекции.
- Глава 5 подробно описывает потоковые операции, с помощью которых можно выражать сложные запросы для обработки данных. Вы встретите в ней описание множества типовых способов, таких как фильтрация, срезы, сопоставление с шаблоном, отображение и свертка.
- Глава 6 посвящена сборщикам данных — функциональной возможности Stream API, с помощью которой можно выражать еще более сложные запросы для обработки данных.
- Из главы 7 вы узнаете, как организовать автоматическое распараллеливание потоков данных и в полной мере использовать многоядерную архитектуру своей машины. Кроме того, мы расскажем вам, как избежать многочисленных подводных камней и правильно и эффективно применять параллельные потоки.

4 Знакомство с потоками данных

В этой главе

- Что такое поток данных.
- Коллекции и потоки.
- Внутренняя и внешняя итерация.
- Промежуточные и завершающие операции.

Что бы вы делали без коллекций в языке Java? Практически любое приложение Java *создает* коллекции и *производит над ними* какие-либо *операции*. Коллекции необходимы для многих задач программирования, поскольку дают возможность группировать и обрабатывать данные. Чтобы проиллюстрировать коллекции в действии, предлагаем вам представить себе, что нужно создать коллекцию блюд в виде меню для выполнения различных запросов. Или нужно отфильтровать меню, выбрав только низкокалорийные блюда для специального диетического стола. Хотя коллекции необходимы практически для всех приложений Java, возможности работы с ними далеки от совершенства.

- Бизнес-логика часто требует операций в стиле баз данных, таких как *группировка* списка блюд по категориям (например, выбор всех вегетарианских блюд) или *поиск* самого дорогого блюда. Наверняка нередко случалось так, что вам приходилось заново реализовывать подобные операции с помощью итераторов. Большинство баз данных позволяют задавать их декларативно. Например, следующий SQL-запрос дает возможность выбрать (отфильтровать) названия низкокалорийных блюд: `SELECT name FROM dishes WHERE calorie < 400`. Как вы можете видеть, при работе с SQL не нужно описывать, *как* отфильтровать данные на основе атрибута `calorie` блюд (как при работе с коллекциями Java, например, с помощью итератора и накопителя). Вместо этого описывается *результат*, который вы хотели

бы получить. Эта простая идея означает, что не нужно заботиться о реализации подобных запросов — база данных все делает вместо вас! Почему бы не создать аналог SQL для коллекций?

- Как лучше обрабатывать коллекцию из большого количества элементов? Для повышения производительности ее нужно обрабатывать параллельно, с помощью многоядерных архитектур. Но написание параллельного кода — задача непростая, по сравнению с использованием итераторов. Кроме того, отладка параллельного кода — то еще развлечение!

Как же создатели языка Java помогают сэкономить время и облегчить жизнь программистов? Наверное, вы уже догадались: с помощью *потоков данных*.

4.1. Что такое потоки данных

Потоки данных (streams) — новая возможность Java API, позволяющая декларативно описывать операции над коллекциями данных (писать запрос, а не код импровизированной реализации). Можете пока считать их необычными итераторами для коллекций данных. Кроме того, потоки данных можно обрабатывать параллельно *явным образом*, без написания какого-либо многопоточного кода! В главе 7 мы расскажем подробнее, как происходит параллельная обработка с помощью потоков данных. Для демонстрации преимуществ использования потоков данных сравним следующий код, возвращающий названия низкокалорийных блюд, отсортированных по числу калорий, — сначала в Java 7, а потом в Java 8, с помощью потоков данных. Не задумывайтесь над кодом Java 8 слишком сильно — в следующих главах мы подробно его прокомментируем.

Ранее (Java 7):

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() < 400) {
        lowCaloricDishes.add(dish);
    }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {
    public int compare(Dish dish1, Dish dish2) {
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish dish: lowCaloricDishes) {
    lowCaloricDishesName.add(dish.getName());
}
```

Фильтрация элементов с помощью накопителя

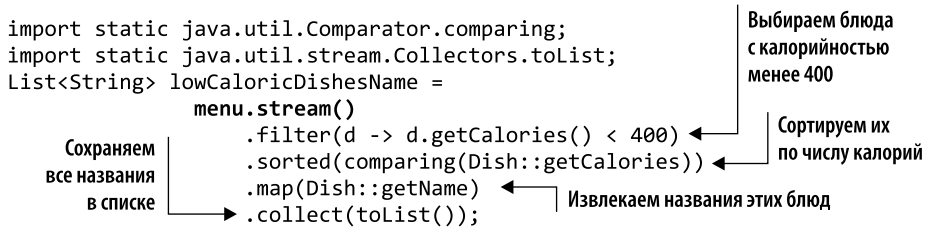
Сортировка блюд с помощью анонимного класса

Обработка отсортированного списка для выбора названий блюд

В этом коде используется «мусорная» переменная: `lowCaloricDishes`. Ее единственная роль — одноразовый промежуточный контейнер. В Java 8 эта деталь реализации перемещена в библиотеку, где ей самое место.

После (Java 8):

```
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
List<String> lowCaloricDishesName =
                menu.stream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dish::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```



Чтобы воспользоваться преимуществами многоядерной архитектуры, для параллельного выполнения кода достаточно поменять вызов `stream()` на `parallelStream()`:

```
List<String> lowCaloricDishesName =
                menu.parallelStream()
                    .filter(d -> d.getCalories() < 400)
                    .sorted(comparing(Dishes::getCalories))
                    .map(Dish::getName)
                    .collect(toList());
```

Наверное, вам интересно, что именно происходит при вызове метода `parallelStream()`. Сколько потоков выполнения используется? Насколько выше оказывается производительность? Имеет ли смысл вообще применять этот метод? На эти вопросы вы получите подробные ответы в главе 7. Пока, как видите, у нового подхода есть несколько очевидных выгод с точки зрения инженерии разработки программного обеспечения.

- Код написан в *декларативном* стиле: вы указываете, *что* хотели бы получить (*отфильтровать низко*калорийные блюда), а не *как* реализовать операцию (с помощью операторов управления потоком выполнения, например циклов и условных операторов `if`). Как вы видели в предыдущей главе, такой подход вместе с параметризацией поведения позволяет адаптироваться к изменению требований: можно с легкостью создать дополнительный вариант кода для фильтрации высококалорийных блюд с помощью лямбда-выражения, без копирования/вставки кода. Другая сторона этого преимущества: отделение модели многопоточного выполнения от самого запроса. Ведь благодаря тому, что описывается только «рецепт» запроса, его можно выполнять последовательно или параллельно. Вы узнаете об этом больше в главе 7.
- Для выражения сложного конвейера обработки данных несколько стандартных операций соединяются цепочкой (мы соединили `filter`, `sorted`, `map` и `collect`, как показано на рис. 4.1). При этом код сохраняется удобочитаемым, а его назначение — понятным. Результат выполнения операции `filter` передается операции `sorted`, результат которой, в свою очередь, передается методу `map`, а затем методу `collect`.

Благодаря тому что такие операции, как `filter` (а также `sorted`, `map` и `collect`), представлены в виде *высокоуровневых стандартных блоков*, не зависящих от конкретной модели выполнения в потоках, их внутренняя реализация может быть как однопоточной, так и — потенциально — максимально многопоточной в вашей архитектуре!

На практике это значит, что больше не нужно волноваться о потоках выполнения и блокировках при необходимости параллелизовать определенные задачи обработки данных: Stream API все сделает за вас!

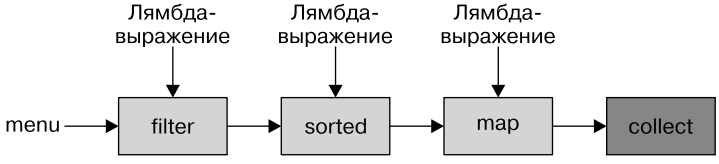


Рис. 4.1. Создание конвейера потоковой обработки с помощью соединения потоковых операций цепочкой

Новый Stream API впечатляет. Например, после прочтения этой главы, а также глав 5 и 6, вы сможете писать код следующего вида:

```
Map<Dish.Type, List<Dish>> dishesByType =
    menu.stream().collect(groupingBy(Dish::getType));
```

Этот конкретный пример подробно обсуждается в главе 6. Код группирует блюда по типам внутри объекта `Map`. Например, итоговые результаты в `Map` могут выглядеть следующим образом:

```
{FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

Представьте теперь себе, как бы вы реализовывали это в обычном императивном стиле программирования с помощью циклов. Но не тратьте на это слишком много времени, а лучше прочитайте о возможностях потоков данных в этой и следующих главах!

Другие библиотеки: Guava, Apache и lambdaj

Было предпринято немало попыток создать для Java-программистов наилучшие библиотеки для работы с коллекциями. Например, Guava — созданная компанией Google популярная библиотека с дополнительными классами контейнеров, такими как мультикарты и мультимножества. Библиотека коллекций Apache Commons предоставляет аналогичные возможности. Наконец, lambdaj, написанная Марио Фуско, одним из авторов данной книги, предоставляет множество утилит для работы с коллекциями в декларативном стиле.

В настоящее время в Java 8 входит своя собственная библиотека для работы с коллекциями в более декларативном стиле.

Подытожим. Stream API Java 8 позволяет писать код:

- *декларативный* — более лаконичный и удобочитаемый;
- *удобный для компоновки* — более гибкий;
- *параллелизуемый* — более производительный.

В оставшейся части данной главы и в следующей в примерах будет использоваться такая предметная область: объект `menu`, представляющий собой просто список блюд.

```
List<Dish> menu = Arrays.asList(
    new Dish("pork", false, 800, Dish.Type.MEAT),
    new Dish("beef", false, 700, Dish.Type.MEAT),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("season fruit", true, 120, Dish.Type.OTHER),
    new Dish("pizza", true, 550, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("salmon", false, 450, Dish.Type.FISH) );
```

Здесь `Dish` — неизменяемый класс со следующим определением:

```
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;
    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }
    public String getName() {
        return name;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
    public int getCalories() {
        return calories;
    }
    public Type getType() {
        return type;
    }
    @Override
    public String toString() {
        return name;
    }
    public enum Type { MEAT, FISH, OTHER }
}
```

А сейчас мы подробнее рассмотрим вопрос использования Stream API. Сравним потоки с коллекциями и напомним некоторые основные сведения. В следующей главе мы подробно исследуем доступные потоковые операции, пригодные для выражения сложных запросов обработки данных. Мы рассмотрим множество стандартных методов, таких как фильтрация, срезы, поиск, сопоставление с шаблоном, отображение и свертка. Вас ждет немало контрольных заданий и упражнений для закрепления своих знаний.

Далее мы обсудим создание числовых потоков данных и работу с ними (например, генерацию потока четных чисел или пифагоровых троек). Наконец, мы обсудим создание потоков на основе различных источников, например файлов. Мы также поговорим о генерации потоков данных с бесконечным количеством элементов — о возможности, которую коллекции явно не предоставляют!

4.2. Знакомимся с потоками данных

Разговор о потоках данных мы начнем с коллекций, поскольку так проще всего перейти к работе с потоками. Коллекции Java 8 поддерживают новый метод `stream`, который возвращает поток данных (описание его интерфейса можно найти в `java.util.stream.Stream`). Далее вы увидите, что получать потоки данных можно и другими способами (например, путем генерации элементов потока на основе числового диапазона или ресурсов ввода/вывода).

Во-первых, что такое *поток данных (stream)*? Краткое определение звучит так: «полученная из источника последовательность элементов, поддерживающая операции обработки данных». Разобьем это определение на составные части.

- *Последовательность элементов* — подобно коллекции, поток данных обеспечивает интерфейс для упорядоченного набора значений определенного типа. Коллекции, как структуры данных, в основном решают задачи хранения и доступа к элементам с конкретными нюансами временного или пространственного характера (например, сравните `ArrayList` и `LinkedList`). Потоки данных же выражают такие вычислительные операции, как вышеупомянутые `filter`, `sorted` и `map`. При работе с коллекциями упор делается на данных; а при работе с потоками данных — на их обработке. Мы разъясним эту идею подробнее в следующих разделах.
- *Источник* — потоки потребляют данные из таких источников, как коллекции, массивы или ресурсы ввода/вывода. Отметим, что при генерации потока данных на основе упорядоченной коллекции упорядоченность сохраняется. Порядок элементов потока, получаемых из списка, будет таким же, как и в списке.
- *Операции обработки данных* — потоки данных поддерживают операции в стиле баз данных и распространенные операции над данными из функциональных языков программирования, такие как `filter`, `map`, `reduce`, `find`, `match`, `sort` и т. д. Выполнять потоковые операции можно как последовательно, так и параллельно.

  Кроме того, у потоковых операций есть две важные характеристики.

- *Возможность конвейерной организации* — многие из потоковых операций сами возвращают поток данных, благодаря чему их можно соединять цепочкой в конвейер. Это делает возможными кое-какие оптимизации, о которых мы поговорим в следующей главе, в частности *отложенную обработку* (laziness) и *сокращенное вычисление* (short-circuiting). Конвейер операций можно рассматривать как аналог запроса (такого как в базах данных) к источнику данных.
- *Внутренняя итерация* — в отличие от коллекций, циклы по которым описываются явным образом, с помощью итераторов, при потоковых операциях итерация

производится незаметно для вас. Мы вкратце упоминали эту идею в главе 1 и вернемся к ней позднее, в следующем разделе.

Поясним все эти идеи на примере кода:

В этом примере мы сначала создаем поток данных на основе списка блюд, вызывая метод `stream` объекта `menu`. *Источником данных* является список блюд (меню), из которого поток получает *последовательность элементов*. Далее мы применяем к потоку ряд *операций обработки данных*: `filter`, `map`, `limit` и `collect`. Каждая из этих операций, за исключением `collect`, возвращает новый поток данных, так что их допускается соединить цепочкой и сформировать *конвейер*, который можно рассматривать в качестве запроса к источнику. Наконец, операция `collect` запускает обработку данных и возвращает результат (эта операция отличается от других тем, что возвращает не поток данных, а — в данном случае — объект `List`). До тех пор пока не будет вызван метод `collect`, никакого результата не возвращается и, конечно, не выбирается ни один элемент из списка `menu`. Можете считать, что вызовы методов в цепочке ожидают в очереди, пока не будет вызван `collect`. Рисунок 4.2 демонстрирует последовательность потоковых операций — `filter`, `map`, `limit` и `collect`, — кратко описанных ниже.

- `filter` — принимает в качестве параметра лямбда-выражение, служащее для исключения из потока определенных элементов. В данном случае мы выбираем блюда, содержащие более 300 калорий, с помощью передачи лямбда-выражения `dish -> dish.getCalories() > 300`.
- `map` — принимает в качестве параметра лямбда-выражение для преобразования элемента в другой или извлечения информации. В данном случае мы извлекаем названия блюд, передавая ссылку на метод `Dish::getName`, эквивалентную лямбда-выражению `d -> d.getName()`.
- `limit` — укорачивает список так, чтобы он содержал не более заданного количества элементов.
- `collect` — преобразует поток данных в другую форму. В этом случае мы преобразуем поток данных в список. Это выглядит какой-то магией! В главе 6 мы опишем, как работает операция `collect`. Пока вы можете рассматривать `collect`

как операцию, принимающую в качестве аргумента различные инструкции по накоплению элементов потока данных в итоговом результате. В данном случае метод `toList()` представляет собой инструкцию по преобразованию потока данных в список.

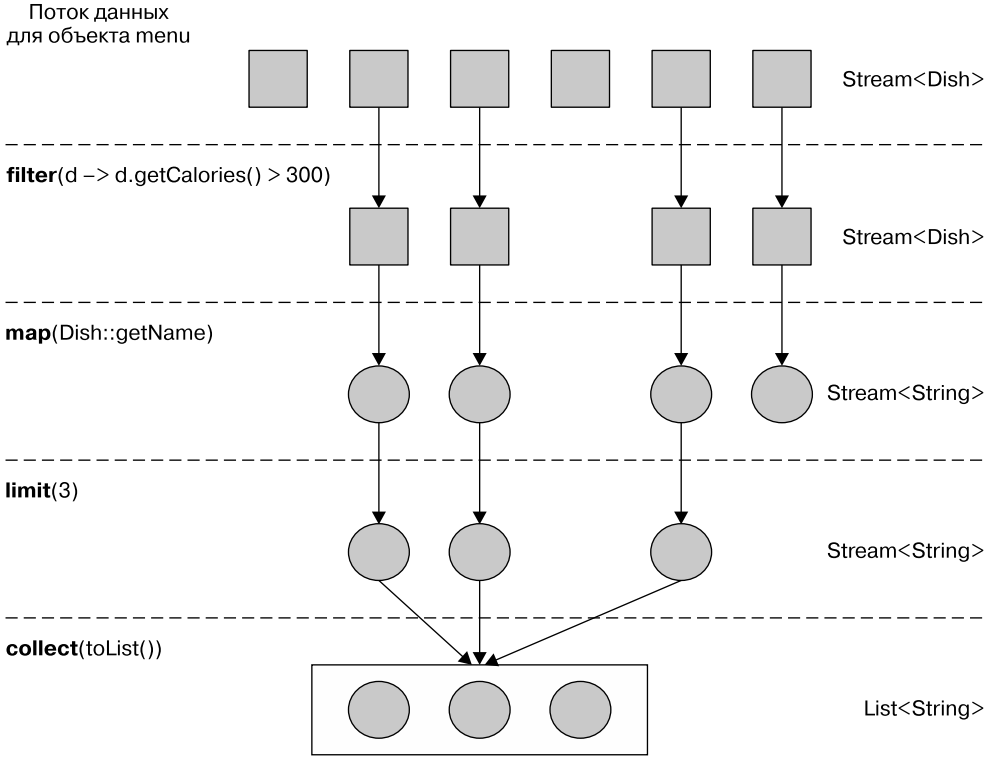


Рис. 4.2. Фильтрация меню с помощью потока данных для получения названий трех высококалорийных блюд

Обратите внимание, что описанный нами код отличается от кода для пошаговой обработки списка элементов меню. Во-первых, используется намного более декларативный стиль описания обработки данных из меню: мы сообщаем, *что* хотели бы получить в результате: «Найти названия трех высококалорийных блюд». Мы не реализуем функциональность операций фильтрации (`filter`), извлечения (`map`) или усечения (`limit`), поскольку она уже есть в готовом виде в библиотеке потоков. В результате Stream API получает больше свободы для принятия решений об оптимизации конвейера. Например, шаги фильтрации, извлечения и усечения можно выполнить за один проход и остановить обработку сразу же после обнаружения трех искомых блюд. Мы покажем пример этого в следующей главе.

Давайте вернемся на шаг назад и рассмотрим основные различия между Collection API и новым Stream API, прежде чем приступить к подробному рассказу о возможных операциях над потоками данных.

4.3. Потоки данных и коллекции

Как уже существовавшая в Java концепция коллекций, так и новая концепция потоков данных служат интерфейсами к структурам данных, отражающим последовательный набор значений определенного типа. Под *последовательным* мы понимаем обход этих значений по очереди, а не в произвольном порядке. В чем же различие?

Начнем с наглядной метафоры. Рассмотрим фильм, хранящийся на DVD. Он представляет собой коллекцию (байтов или кадров — в данном случае неважно), поскольку содержит полную структуру данных. Теперь рассмотрим тот же фильм при *потоковой* трансляции через Интернет. Теперь это поток (байтов или кадров). Проигрывателю потокового видео достаточно скачивать данные с опережением лишь в несколько кадров относительно просматриваемого места фильма, так что можно приступить к визуализации значений с начала потока даже до того, как будет вычислена бóльшая часть значений потока (например, при потоковой трансляции футбольного матча в прямом эфире). Отдельно отметим, что видеопроигрыватель может столкнуться с нехваткой оперативной памяти для буферизации всего потока данных в памяти в виде коллекции — а время запуска видео окажется неприемлемо большим, если для того, чтобы начать показ видео, придется ждать вычисления последнего кадра. Возможно, вы захотите *буферизовать* часть потока данных в виде коллекции вследствие специфики реализации видеопроигрывателя, но это не составляет принципиальной разницы.

Грубо говоря, различие между коллекциями и потоками данных состоит в том, *когда* производятся вычисления. Коллекция представляет собой структуру в оперативной памяти, в которой хранятся все имеющиеся в настоящий момент значения — перед добавлением любого элемента в коллекцию его значение должно быть вычислено. В коллекцию можно добавлять сущности и удалять их оттуда, но в любой заданный момент времени все элементы коллекции хранятся в оперативной памяти; прежде чем элемент попадет в коллекцию, его значение должно быть вычислено.

Напротив, поток данных — фиксированная структура данных (добавить в нее элементы или удалять их из нее нельзя), элементы которой вычисляются *по запросу*, что приводит к немалым выгодам. В главе 6 мы покажем, насколько легко сформировать поток данных, содержащий все простые числа (2, 3, 5, 7, 11...), несмотря на то что их бесконечно много. Идея состоит в том, что пользователи будут извлекать из потока данных только нужные им значения, и эти значения генерируются — незаметно для пользователя, — только *когда* в них возникает потребность. Это разновидность взаимосвязи «генератор — потребитель». Можно также рассматривать поток данных как коллекцию с отложенным выполнением: значения вычисляются тогда, когда их запрашивает потребитель (на управленческом языке это обусловленное спросом производство или даже производство по принципу «точно в срок»).

В отличие от потоков данных коллекции ориентированы на немедленное вычисление (ориентированное на поставщиков производство: заполнить склад, прежде чем открыть продажи, например, рождественских елок, срок продажи которых ограничен). Представьте себе их использование для примера с простыми числами. Попытка сформировать коллекцию из всех простых чисел привела бы к попаданию программы в бесконечный цикл вычисления все новых и новых простых чисел с добавлением их в коллекцию — цикл, который никогда не закончится, так что потребитель никогда не увидит коллекцию.

Рисунок 4.3 иллюстрирует разницу между потоком данных и коллекцией применительно к нашему примеру фильма на DVD по сравнению с потоковой трансляцией через Интернет.

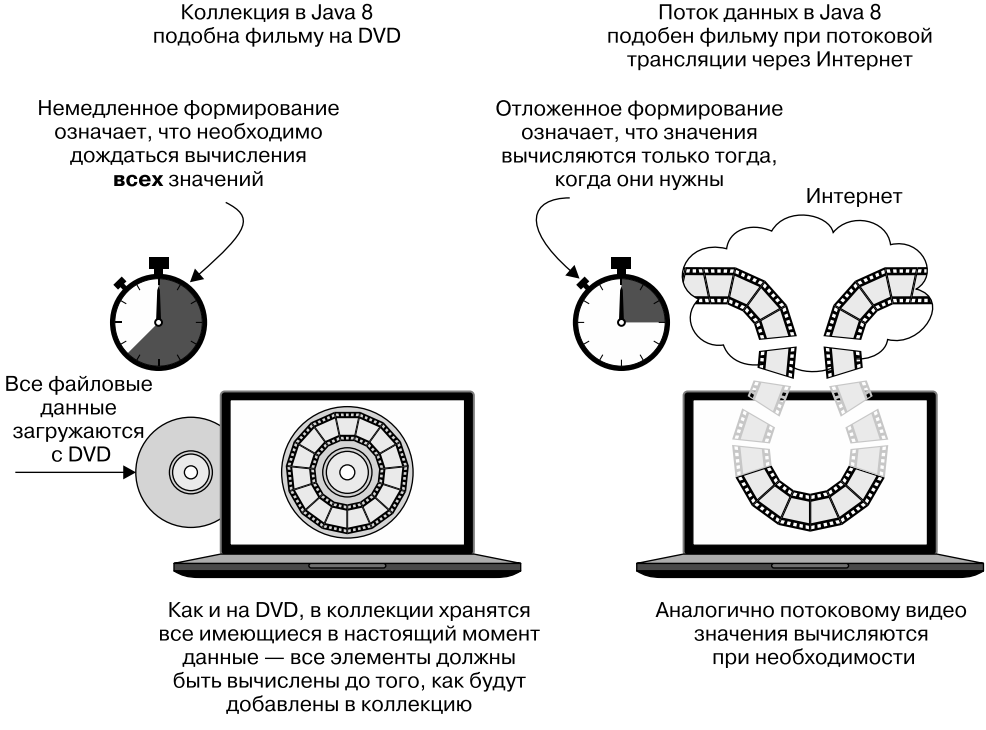


Рис. 4.3. Потоки данных по сравнению с коллекциями

Еще один пример — поиск в Интернете через браузер. Представьте себе, что вы ищете в Google или на сайте интернет-магазина фразу, для которой находится множество совпадений. Вам не нужно ждать скачивания полной коллекции результатов со всеми фотографиями — выдается поток из 10 или 20 наиболее релевантных результатов и кнопка, нажав которую можно получить следующие 10 или 20. Когда вы, потребитель, нажимаете кнопку для получения следующих 10, объект-поставщик вычисляет их (по запросу), а затем возвращает браузеру для отображения.

4.3.1. Строго однократный проход

Стоит отметить, что, подобно итераторам, поток данных проходится ровно один раз. После этого он считается прочитанным (потребленным). При необходимости можно создать новый поток на основе исходного источника данных и пройти его снова аналогично итератору (при условии, что источник данных — воспроизводимый, как коллекция; если же речь идет о канале ввода/вывода — вам не повезло). Например, следующий код сгенерирует исключение, указывающее, что поток данных уже был прочитан:

```
List<String> title = Arrays.asList("Современный", "язык", "Java");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println);
```

Выводит каждое слово из названия данной книги

Исключение java.lang.IllegalStateException: поток данных уже был обработан или закрыт

Потоки данных и коллекции с философской точки зрения

Философски настроенные читатели могут рассматривать поток данных как набор значений, разбросанных во времени. Напротив, коллекция представляет собой набор значений, разбросанных в пространстве (в данном случае в памяти компьютера), которые сосуществуют в один момент времени и к которым можно получить доступ с помощью итератора, внутри цикла `for-each`.

Еще одно ключевое различие между коллекциями и потоками данных — организация циклов по данным.

4.3.2. Внутренняя и внешняя итерация

При использовании интерфейса `Collection` организовать цикл должен пользователь (например, с помощью `for-each`); это называется *внешней итерацией* (external iteration). В библиотеке потоков данных, наоборот, применяется *внутренняя итерация* (internal iteration) — библиотека сама выполняет проход по данным и сохраняет где-то полученные потоковые значения; вам же достаточно указать функцию, которая определяет, что нужно сделать. Следующий код иллюстрирует описанное различие (листинг 4.1).

Листинг 4.1. Коллекции: внешняя итерация с помощью цикла for-each

```
List<String> names = new ArrayList<>();
for(Dish dish: menu) {
    names.add(dish.getName());
}
```

Последовательно проходим по списку menu явным образом

Извлекаем название и добавляем его в накопитель

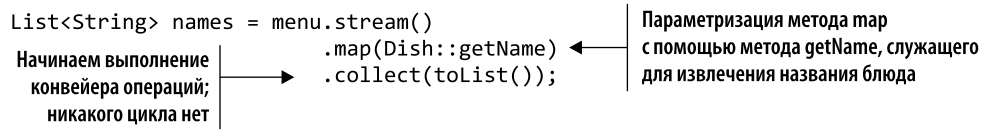
Отметим, что конструкция `for-each` скрывает часть сложности организации цикла. Конструкция `for-each` представляет собой «синтаксический сахар», за которым скрывается намного более уродливый код, использующий объект `Iterator` (листинги 4.2, 4.3).

Листинг 4.2. Коллекции: использование «под капотом» объекта Iterator для внешней итерации

```
List<String> names = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish dish = iterator.next();
    names.add(dish.getName());
}
```

Организация цикла явным образом

Листинг 4.3. Потоки данных: внутренняя итерация



Проведем аналогию, чтобы лучше разобраться в отличиях и преимуществах внутренней итерации. Представьте себе, что вы разговариваете со своей двухлетней дочерью Соней и хотите, чтобы она убрала свои игрушки:

Вы: «Соня, давай уберем игрушки. Есть на полу какая-нибудь игрушка?»

Соня: «Да, мячик».

Вы: «Хорошо, давай положим мячик в коробку. Что-нибудь еще?»

Соня: «Да, моя кукла».

Вы: «Хорошо, давай положим куклу в коробку. Что-нибудь еще?»

Соня: «Да, моя книжка».

Вы: «Хорошо, давай положим книжку в коробку. Что-нибудь еще?»

Соня: «Нет, больше ничего».

Вы: «Замечательно, готово».

Именно это вы каждый день делаете со своими коллекциями Java: выполняете итерации по коллекции *внешним образом*, явно извлекая и обрабатывая элементы один за другим. Гораздо удобнее было бы сказать Соне: «Положи в коробку все свои игрушки, лежащие на полу». Внутренняя итерация предпочтительнее еще по двум причинам: во-первых, Соня может одновременно взять куклу одной рукой, а мяч — другой; во-вторых, она может сначала взять объекты, находящиеся ближе к коробке, а затем — все остальные. Аналогично при внутренней итерации можно выполнять обработку элементов параллельно или в другом, более оптимальном порядке. При привычной вам в Java внешней итерации по коллекциям оптимизировать код подобным образом непросто. Хотя это может показаться какими-то мелочными придирками, но они во многом и явились главными причинами появления в Java 8 потоков данных. Внутренняя итерация в библиотеке потоков может автоматически выбирать представление данных и реализацию параллельного выполнения, наиболее подходящую для вашего аппаратного обеспечения. В то же время, если вы выбрали внешнюю итерацию и написали конструкцию `for-each`, вы вынуждены будете самостоятельно позаботиться о параллельном выполнении (на практике *«самостоятельно»* означает или «в один прекрасный день мы распараллелим это», или «начинаем долгую и трудную борьбу с задачами и ключевым словом `synchronized`»). Java 8 был необходим интерфейс, подобный `Collection`, но без итераторов, поэтому и появился интерфейс `Stream`!

Рисунок 4.4 иллюстрирует различие между потоком данных (внутренняя итерация) и коллекцией (внешняя итерация).

Итак, мы описали принципиальные различия между коллекциями и потоками данных. А именно: потоки данных используют внутреннюю итерацию, при которой всю работу за вас выполняет библиотека. Но это полезно только при наличии списка заранее определенных операций (например, `filter` или `map`), скрывающих детали организации цикла. Большая часть этих операций принимает лямбда-выражения

в качестве аргументов, благодаря чему можно параметризовать их поведение так, как показано в предыдущей главе. Создатели языка Java снабдили Stream API внушительным списком операций, с помощью которых можно выражать сложные запросы обработки данных. Мы сейчас вкратце рассмотрим этот список, а подробнее займемся им в следующей главе, на примерах.

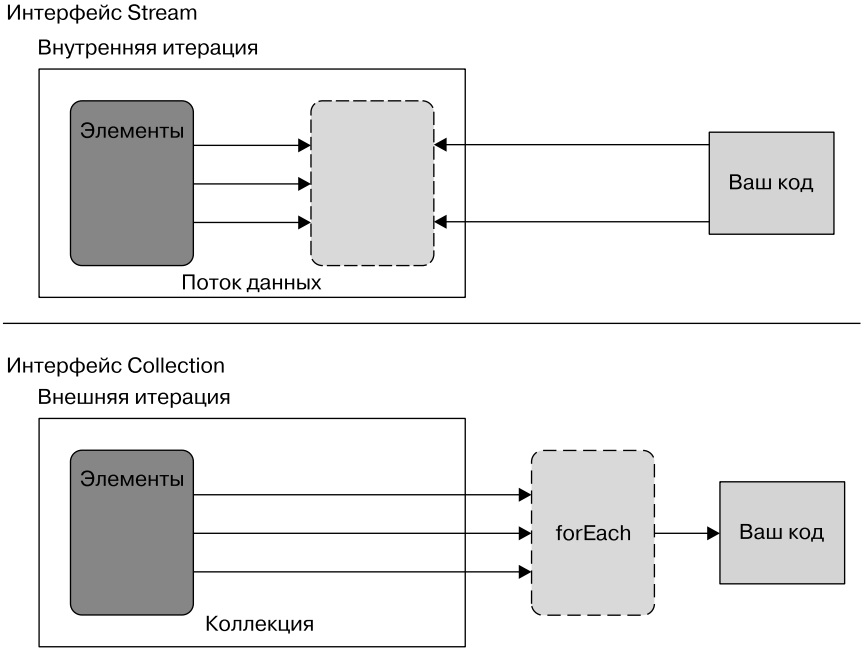


Рис. 4.4. Внешняя и внутренняя итерация

Чтобы проверить, насколько хорошо вы понимаете различия между внешней и внутренней итерацией, попробуйте выполнить следующее контрольное задание 4.1.

Контрольное задание 4.1. Внешняя и внутренняя итерация

Какой из потоковых операций вы бы воспользовались для рефакторинга следующего кода на основе полученных из листингов 4.1 и 4.2 знаний о внешней итерации?

```
List<String> highCaloricDishes = new ArrayList<>();
Iterator<Dish> iterator = menu.iterator();
while(iterator.hasNext()) {
    Dish dish = iterator.next();
    if(dish.getCalories() > 300) {
        highCaloricDishes.add(dish.getName());
    }
}
```

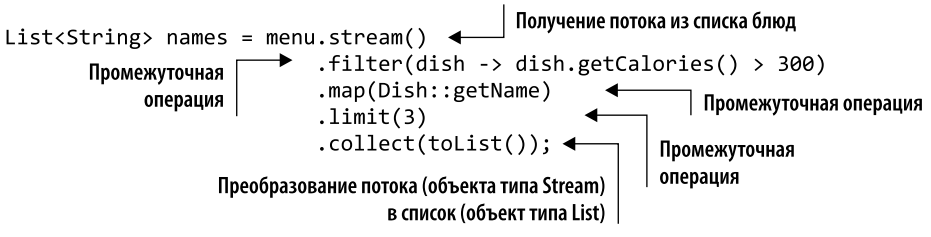
Ответ: здесь необходимо воспользоваться паттерном `filter`.

```
List<String> highCaloricDish =
    menu.stream()
        .filter(dish -> dish.getCalories() > 300)
        .map(dish -> dish.getName())
        .collect(toList());
```

Не волнуйтесь, если вы пока плохо понимаете все нюансы написания потоковых запросов, в следующей главе мы подробнее поговорим об этом.

4.4. Потоковые операции

В интерфейсе потоков данных в `java.util.stream.Stream` описано множество операций. Их можно разбить на две категории. Взглянем еще раз на наш предыдущий пример:



Здесь можно видеть две группы операций:

- `filter`, `map` и `limit`, которые соединяются цепочкой и образуют конвейер;
- вызов `collect`, который запускает выполнение конвейера, а затем завершает его работу.

Потоковые операции, допускающие соединение в цепочку, называются *промежуточными* (intermediate), а закрывающие поток данных операции — *завершающими* (terminal). На рис. 4.5 приведены эти две группы операций. Почему же это различие так важно?

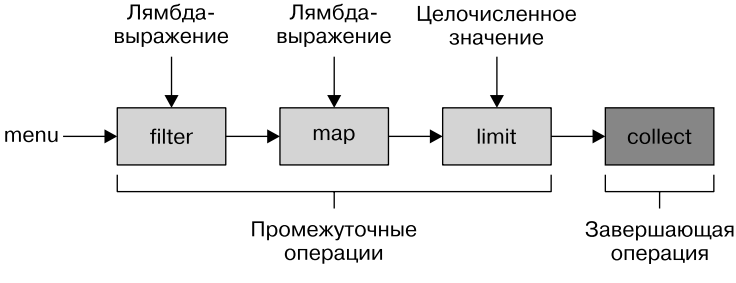


Рис. 4.5. Промежуточные и завершающие операции

4.4.1. Промежуточные операции

Промежуточные операции, например `filter` и `sorted`, возвращают другой объект-поток. Благодаря этому становится возможным формирование запроса путем

соединения этих операций цепочкой. Важно то, что промежуточные операции не выполняют никаких действий до тех пор, пока не будет вызвана завершающая операция, поскольку они носят отложенный характер. Причина в том, что обычно промежуточные операции сливаются воедино и обрабатываются завершающей операцией за один проход.

Чтобы разобраться в происходящем внутри потокового конвейера, изменим код так, чтобы все лямбда-выражения выводили текущие обрабатываемые ими блюда (подобно многим демонстрационным методикам и методикам отладки; для промышленного кода такой стиль программирования ужасен, но во время учебы он удобен для пояснения порядка выполнения вычислений).

```
List<String> names =
    menu.stream()
        .filter(dish -> {
                        System.out.println("filtering:" + dish.getName());
                        return dish.getCalories() > 300;
                    })
        .map(dish -> {
                      System.out.println("mapping:" + dish.getName());
                      return dish.getName();
                  })
        .limit(3)
        .collect(toList());
System.out.println(names);
```

Выводит блюда по мере фильтрации

Выводит блюда по мере извлечения их названий

При выполнении этого кода выводятся следующие результаты:

```
filtering:pork
mapping:pork
filtering:beef
mapping:beef
filtering:chicken
mapping:chicken
[pork, beef, chicken]
```

Можно заметить, что библиотека потоков данных производит несколько оптимизаций, в основе которых лежит отложенная природа выполнения потоков. Во-первых, несмотря на то, что во многих блюдах более 300 калорий, выбираются только первые три! Это происходит благодаря операции `limit`, а также методике *сокращенного вычисления* (short-circuiting), как мы расскажем в следующей главе. Во-вторых, несмотря на то, что `filter` и `map` — две отдельные операции, они объединяются для выполнения за один проход (специалисты по компиляторам называют это *слиянием циклов* (loop fusion)).

4.4.2. Завершающие операции

Генерация результата потокового конвейера происходит в результате вызова завершающей операции. Результат может представлять собой произвольное непотоковое значение, например, типа `List`, `Integer` или даже `void`. В частности, метод `forEach` в следующем конвейере — завершающая операция, возвращающая `void` и применяющая лямбда-выражение к каждому блюду из источника. Передача методу `forEach`

функции `System.out.println` приводит к выводу в консоль всех объектов `Dish` потока данных, созданного на основе объекта `menu`:

```
menu.stream().forEach(System.out::println);
```

Для проверки своего понимания различий промежуточных и завершающих операций попробуйте выполнить контрольное задание 4.2.

Контрольное задание 4.2. Промежуточные или завершающие операции?

Найдите в следующем потоковом конвейере промежуточные и завершающие операции:

```
long count = menu.stream()
                 .filter(dish -> dish.getCalories() > 300)
                 .distinct()
                 .limit(3)
                 .count();
```

Ответ: последняя операция в потоковом конвейере — `count` — возвращает значение типа `long`, не являющееся потоковым. Следовательно, она завершающая. Все предыдущие операции — `filter`, `distinct`, `limit` — связаны цепочкой, возвращают объект-поток и, следовательно, являются *промежуточными*.

4.4.3. Работа с потоками данных

Подытожим вышесказанное. Для работы с потоками данных в целом необходимы три элемента:

- *источник данных* (например, коллекция), к которому производится запрос;
- цепочка *промежуточных операций*, составляющая потоковый конвейер;
- *завершающая операция*, которая выполняет потоковый конвейер и выдает результат.

Концепция потокового конвейера напоминает паттерн «Строитель» (см. https://ru.wikipedia.org/wiki/Строитель_(шаблон_проектирования)). Паттерн проектирования «Строитель» описывает цепочку вызовов, предназначенную для задания настроек (в случае потоков — цепочку промежуточных операций), за которой следует вызов метода `build` (в случае потоков — завершающая операция).

Для удобства мы кратко подытожим в табл. 4.1 и 4.2 уже встречавшиеся вам в примерах кода промежуточные и завершающие потоковые операции. Заметим, что это далеко не полный список операций, доступных в Stream API; в следующей главе вы увидите еще несколько!

Таблица 4.1. Промежуточные операции

Операция	Тип	Возвращаемый тип	Аргумент операции	Функциональный дескриптор
filter	Промежуточная	Stream<T>	Predicate<T>	T -> boolean
map	Промежуточная	Stream<R>	Function<T, R>	T -> R

Операция	Тип	Возвращаемый тип	Аргумент операции	Функциональный дескриптор
limit	Промежуточная	Stream<T>	—	—
sorted	Промежуточная	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Промежуточная	Stream<T>	—	—

Таблица 4.2. Завершающие операции

Операция	Тип	Возвращаемый тип	Назначение
forEach	Завершающая	void	Потребляет все элементы из потока данных, применяя к каждому из них заданное лямбда-выражение
count	Завершающая	long	Возвращает количество элементов в потоке данных
collect	Завершающая	(обобщенный)	Производит свертку потока данных с целью создания коллекции, например, List или Map. См. подробности в главе 6

В следующей главе мы подробно изучим доступные потоковые операции, приведем конкретные сценарии использования и продемонстрируем вам, какие виды запросов можно выражать с их помощью. Мы рассмотрим множество типовых способов, таких как фильтрация, срезы, сопоставление с шаблоном, отображение и свертка, пригодные для выражения сложных запросов обработки данных.

Далее, в главе 6, мы подробно изучим сборщики данных — объекты-коллекторы. В текущей главе мы воспользовались для потоков данных лишь завершающей операцией `collect()` в виде вызова `collect(toList())` с целью создания объекта `List` с элементами того же типа, что и у исходного потока.

Резюме

- Поток данных — последовательность полученных из источника элементов, поддерживающая операции обработки данных.
- В потоках данных используется внутренняя итерация: организация цикла отделяется от выполнения за счет таких операций, как `filter`, `map` и `sorted`.
- Существует два вида потоковых операций: промежуточные и завершающие.
- Промежуточные операции, такие как `filter` и `map`, возвращают поток данных (объект `Stream`), их можно соединять цепочкой. Они применяются для организации конвейеров операций, но не генерируют сами по себе никаких результатов.
- Завершающие операции, такие как `forEach` и `count`, возвращают непотоковое значение и инициируют выполнение обработки потокового конвейера для возвращения результата.
- Вычисление элементов потока данных производится по запросу (отложенным образом).

5 Работа с потоками данных

В этой главе

- Фильтрация, срезы и отображение.
- Поиск, сопоставление с шаблоном и сверка.
- Числовые потоки данных (версия потоков для простых типов данных).
- Создание потоков данных из нескольких источников.
- Бесконечные потоки.

В предыдущей главе мы показали, как потоки данных позволяют перейти от *внешней итерации* к *внутренней*. Вместо того чтобы писать подобный следующему код с явным управлением проходом в цикле по коллекции данных (внешняя итерация):

```
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu) {
    if(d.isVegetarian()){
        vegetarianDishes.add(d);
    }
}
```

можно воспользоваться Stream API (внутренняя итерация), поддерживающим операции `filter` и `collect`, — он возьмет на себя организацию прохода в цикле по коллекции данных. Вам останется только передать нужное для фильтрации поведение в качестве аргумента методу `filter`:

```
import static java.util.stream.Collectors.toList;
List<Dish> vegetarianDishes =
    menu.stream()
        .filter(Dish::isVegetarian)
        .collect(toList());
```

Подобный способ работы с данными удобен тем, что управление обработкой данных берет на себя Stream API. В результате Stream API может неявно оптимизировать выполнение кода различными способами. Кроме того, при внутренней итерации Stream API может самостоятельно принять решение о параллельном выполнении кода. При внешней итерации это невозможно, ведь вы должны выполнять цикл однопоточно, пошагово и последовательно.

В этой главе мы подробно рассмотрим различные операции, поддерживаемые Stream API. Вы узнаете о том, какие операции существуют в Java 8, а также о добавленных в Java 9 новинках. С помощью этих операций можно выражать сложные запросы обработки данных, такие как фильтрация, срезы, отображение, сопоставление с шаблоном и свертка. Далее мы поговорим об особых разновидностях потоков данных: о числовых потоках, потоках на основе нескольких источников (таких как файлы и массивы) и, наконец, о бесконечных потоках.

5.1. Фильтрация

В этом разделе мы рассмотрим два способа выбора элементов потока: фильтрацию с помощью предикатов и фильтрацию только уникальных элементов.

5.1.1. Фильтрация с помощью предиката

Интерфейс `Stream` поддерживает метод `filter` (с которым вы уже хорошо знакомы). Он принимает в качестве аргумента *предикат*, или *условие* (функцию, возвращающую `boolean`), и возвращает поток данных, включающий все соответствующие этому предикату элементы. Например, можно сформировать вегетарианское меню, отфильтровав все вегетарианские блюда из обычного меню, как показано на рис. 5.1 и в следующем за ним коде.

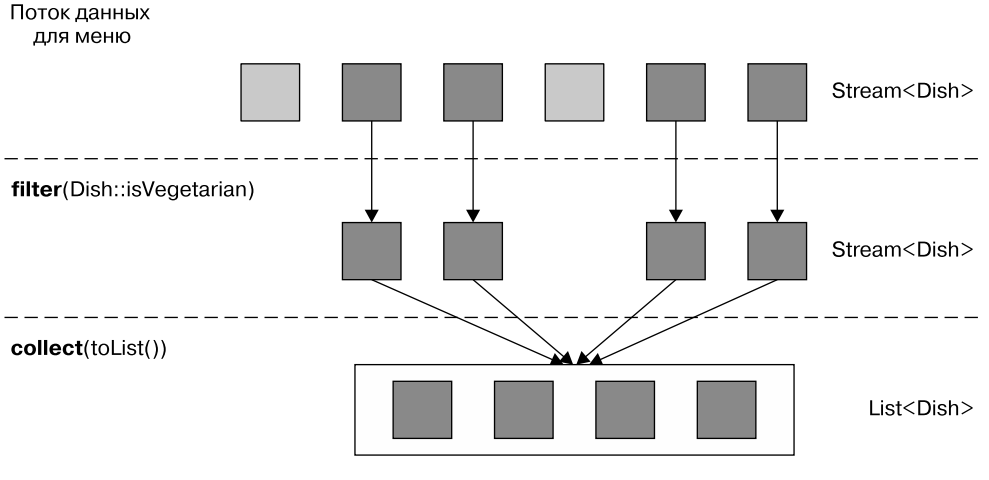


Рис. 5.1. Фильтрация потока данных с помощью предиката

```
List<Dish> vegetarianMenu = menu.stream()
                              .filter(Dish::isVegetarian)
                              .collect(toList());
```

С помощью ссылки на метод проверяем, вегетарианское ли блюдо

5.1.2. Фильтрация уникальных элементов

Потоки также поддерживают метод `distinct`, который возвращает поток данных, содержащий только уникальные элементы (в соответствии с реализацией методов `hashcode` и `equals`, генерируемых потоком объектов). Например, следующий код фильтрует все четные числа из списка, после чего удаляет дубликаты (для сравнения применяется метод `equals`). На рис. 5.2 этот процесс показан наглядно.

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
       .filter(i -> i % 2 == 0)
       .distinct()
       .forEach(System.out::println);
```

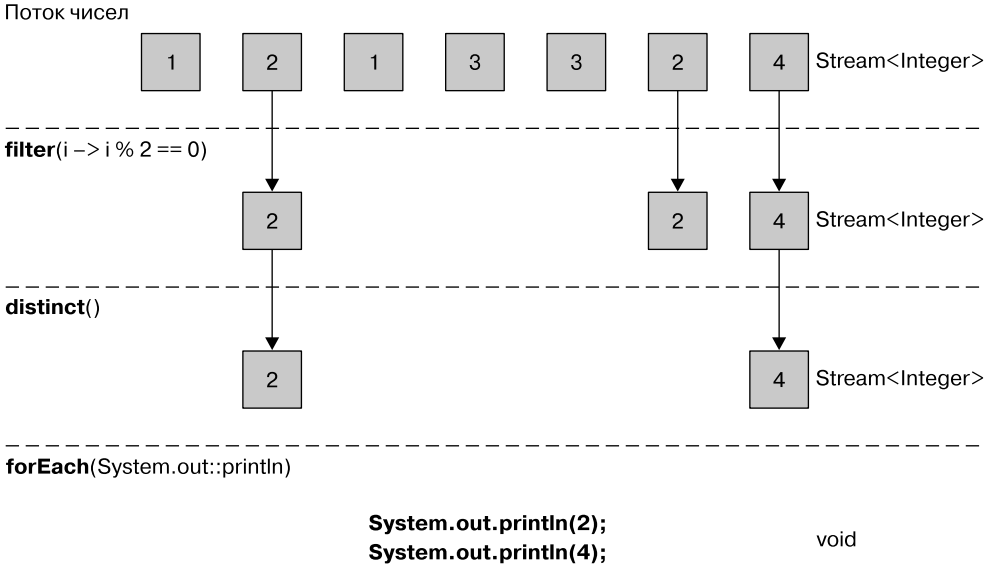


Рис. 5.2. Фильтрация уникальных элементов потока данных

5.2. Срез потока данных

В этом разделе мы обсудим, как различными способами выбирать и пропускать элементы потока данных. Существуют операции для эффективного выбора или отбрасывания элементов с помощью предиката, пропуска нескольких первых элементов потока или усечения потока до заданного размера.

5.2.1. Срез с помощью предиката

В Java 9 появилось два новых метода, полезных для выбора элементов потока с хорошей производительностью: `takeWhile` и `dropWhile`.

Метод takeWhile

Допустим, у нас есть следующий список блюд:

```
List<Dish> specialMenu = Arrays.asList(
    new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("chicken", false, 400, Dish.Type.MEAT),
    new Dish("french fries", true, 530, Dish.Type.OTHER));
```

Как выбрать блюда, содержащие менее 320 калорий? Вы уже знаете из предыдущего раздела, что можно воспользоваться операцией `filter`, вот так:

```
List<Dish> filteredMenu
    = specialMenu.stream()
                 .filter(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

Список, включающий сезонные фрукты и креветки

Но вы, наверное, обратили внимание, что исходный список уже отсортирован по числу калорий! Недостаток операции `filter` в том, что она требует прохода в цикле по всему потоку данных с применением предиката ко всем элементам. Вместо этого можно прекратить работу сразу же после обнаружения блюда, содержащего 320 калорий или более. В случае небольшого списка это может показаться не таким уж громадным преимуществом, но при работе с потенциально большим потоком элементов окажется весьма полезным. Но как же реализовать это? Вам на помощь придет операция `takeWhile`! Она позволяет выполнить срез любого потока данных (даже бесконечного, как вы узнаете далее) с помощью предиката. И, к счастью, она прекращает работу сразу же по обнаружении неподходящего элемента. Вот как ее следует использовать:

```
List<Dish> slicedMenu1
    = specialMenu.stream()
                 .takeWhile(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

Список, включающий сезонные фрукты и креветки

Метод dropWhile

А как получить остальные элементы? Как выбрать блюда, содержащие более 320 калорий? Для этого можно воспользоваться операцией `dropWhile`:

```
List<Dish> slicedMenu2
    = specialMenu.stream()
                 .dropWhile(dish -> dish.getCalories() < 320)
                 .collect(toList());
```

Список, включающий рис, цыплят, картофель фри

Операция `dropWhile` служит дополнением к операции `takeWhile`. Она отбрасывает первые элементы, для которых предикат ложен. Как только результат вычисления предиката становится истинным, она прекращает работу и возвращает все оставшиеся элементы, причем работает даже в том случае, если число оставшихся элементов бесконечно!

5.2.2. Усечение потока данных

Потоки данных поддерживают метод `limit(n)`. Он возвращает еще один поток, не превышающий заданную длину, переданную в `limit` в виде аргумента. Если поток данных упорядоченный, возвращаются первые элементы, но не более чем `n`. Например, с помощью следующего кода можно создать `List`, выбрав первые три блюда, содержащие более 300 калорий:

```
List<Dish> dishes = specialMenu
                    .stream()
                    .filter(dish -> dish.getCalories() > 300)
                    .limit(3)
                    .collect(toList());
```

Список, включающий рис, цыплят, картофель фри

Рисунок 5.3 иллюстрирует совместное использование методов `filter` и `limit`. Как видите, выбираются только первые три элемента, удовлетворяющие предикату, после чего немедленно возвращается полученный результат.

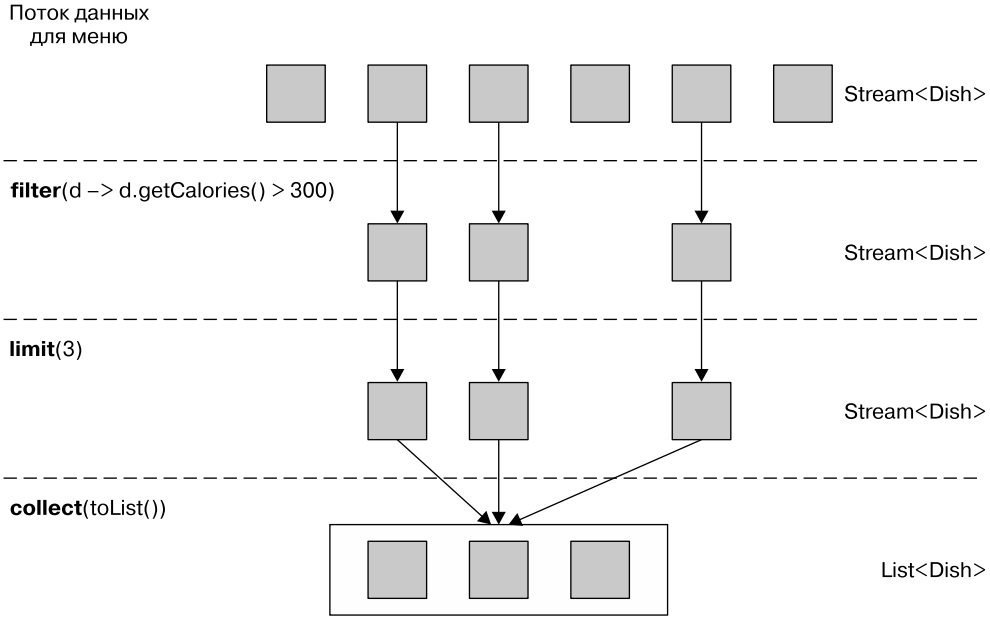


Рис. 5.3. Усечение потока данных

Отметим, что метод `limit` работает и для неупорядоченных потоков данных (например, когда источник представляет собой объект типа `Set`). В этом случае не следует исходить из допущения о какой-либо упорядоченности результата, возвращенного методом `limit`.

5.2.3. Пропуск элементов

Потоки данных поддерживают метод `skip(n)`, возвращающий поток данных с отброшенными первыми `n` элементами источника. Если в потоке-источнике было меньше `n` элементов, возвращается пустой поток. Заметим, что методы `limit(n)` и `skip(n)` дополняют друг друга!

Например, следующий код пропускает первые два блюда, содержащие более 300 калорий, и возвращает остальные. Приведенный запрос проиллюстрирован на рис. 5.4.

```
List<Dish> dishes = menu.stream()
                        .filter(d -> d.getCalories() > 300)
                        .skip(2)
                        .collect(toList());
```

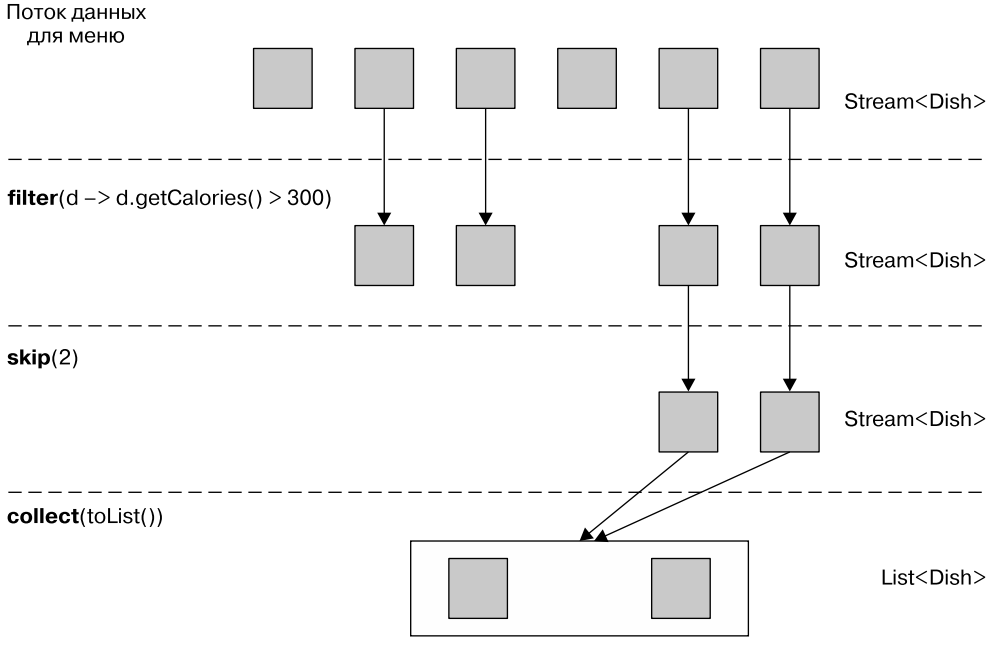


Рис. 5.4. Пропуск элементов потока данных

Прежде чем перейти к операциям отображения, попробуйте на практике все, что вы узнали из этого раздела, выполнив контрольное задание 5.1.

Контрольное задание 5.1. Фильтрация

Как бы вы отфильтровали два первых мясных блюда с помощью потоков данных?

Ответ: для решения этой задачи можно воспользоваться сочетанием методов `filter` и `limit`, а также вызовом `collect(toList())` для преобразования потока данных в список, вот так:

```
List<Dish> dishes =
    menu.stream()
        .filter(dish -> dish.getType() == Dish.Type.MEAT)
        .limit(2)
        .collect(toList());
```

5.3. Отображение

Распространенная идиома обработки данных — выбор информации из определенных объектов. Например, на языке SQL можно выбрать из таблицы конкретный столбец. Stream API предоставляет схожие возможности посредством методов `map` и `flatMap`.

5.3.1. Применение функции к каждому из элементов потока данных

Потоки данных поддерживают метод `map`, принимающий в качестве аргумента функцию. Эта функция затем применяется к каждому из элементов потока данных, отображая его в новый элемент (это называется термином *«отображение»* (mapping), потому что он синонимичен термину *«преобразование»* (transforming), но с дополнительным оттенком смысла «создание новой версии», а не просто «модификация»). Например, в следующем коде мы передаем ссылку на метод `Dish::getName` методу `map`, чтобы *извлечь* названия блюд из потока данных:

```
List<String> dishNames = menu.stream()
                              .map(Dish::getName)
                              .collect(toList());
```

Благодаря тому что метод `getName` возвращает строку, `map` возвращает поток данных типа `Stream<String>`.

Рассмотрим несколько иной пример, чтобы лучше разобраться в работе метода `map`. Для заданного списка слов необходимо вернуть список с количеством символов в каждом из слов. Как это сделать? Следует применить функцию к каждому из элементов списка. Похоже на работу как раз для метода `map`! Применяемая функция должна возвращать длину передаваемого ей слова. Решить эту задачу можно путем передачи в метод `map` ссылки на метод вида `String::length`:

```
List<String> words = Arrays.asList("Modern", "Java", "In", "Action");
List<Integer> wordLengths = words.stream().map(String::length)
                                  .collect(toList());
```

Вернемся к примеру с извлечением названий блюд. Что, если нам нужно узнать длину названия каждого блюда? Для этого достаточно добавить в цепочку вызовов еще один вызов метода `map`:

```
List<Integer> dishNameLengths = menu.stream()
                                    .map(Dish::getName)
                                    .map(String::length)
                                    .collect(toList());
```

5.3.2. Схлопывание потоков данных

В предыдущем разделе вы видели, как получить длину каждого из слов в списке с помощью метода `map`. Немного обобщим эту идею и попробуем вернуть список всех *уникальных символов*, содержащихся в списке слов. Например, при заданных словах `["Hello," "World"]` должен возвращаться список `["H," "e," "l," "o," "W," "r," "d"]`.

Вам может показаться, что это несложная задача и достаточно просто отобразить каждое из слов в список символов, после чего вызвать `distinct` для фильтрации дубликатов. На первый взгляд, что-то вроде следующего:

```
words.stream()
     .map(word -> word.split(""))
     .distinct()
     .collect(toList());
```

Проблема с этим подходом состоит в том, что передаваемое в метод `map` лямбда-выражение возвращает `String[]` (массив строк) для каждого из слов. Метод `map` возвращает поток данных типа `Stream<String[]>`. А нам нужен `Stream<String>`, отражающий поток символов.

Рисунок 5.5 иллюстрирует эту проблему.

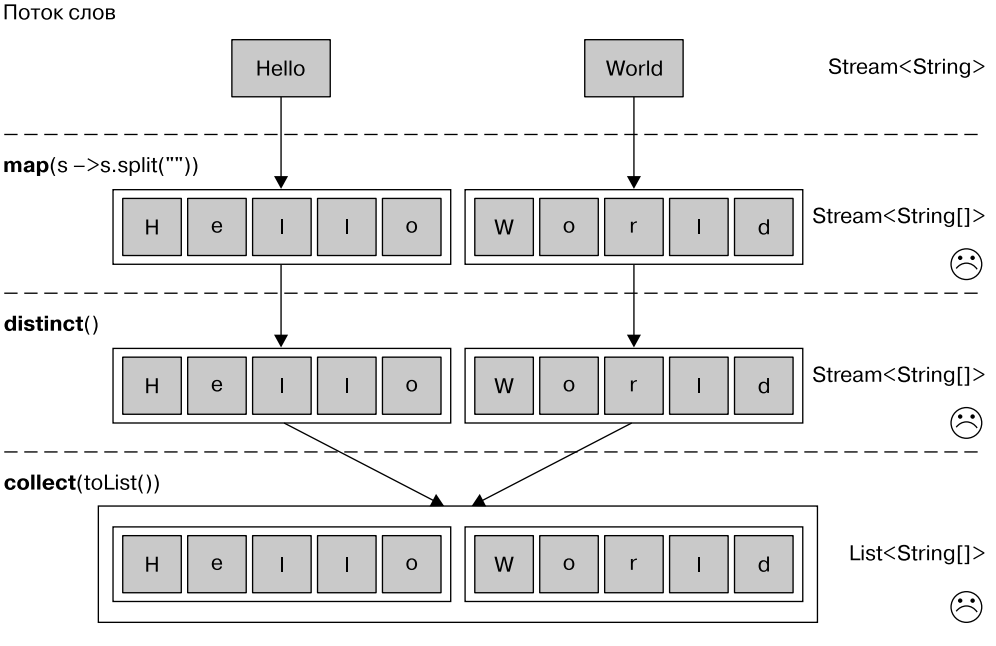


Рис. 5.5. Неправильное использование метода map для получения уникальных символов из списка слов

К счастью, эту проблему можно решить с помощью метода `flatMap`! Рассмотрим шаг за шагом, как это сделать.

Попробуем воспользоваться методами map и Arrays.stream

Во-первых, нам понадобится поток символов вместо потока массивов. Существует метод `Arrays.stream()`, принимающий на входе массив и возвращающий поток данных:

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfwords = Arrays.stream(arrayOfWords);
```

Включим его в предыдущий конвейер и посмотрим, что получится:

```
words.stream()
     .map(word -> word.split(""))
     .map(Arrays::stream)
     .distinct()
     .collect(toList());
```

Преобразует каждое из слов в массив его букв

Превращает каждый из массивов в отдельный поток

Полученное решение все еще не работает! Дело в том, что мы получили список потоков (точнее, `List<Stream<String>>`). И действительно, мы преобразовали каждое из слов в массив его букв, а затем сделали из каждого массива отдельный поток.

Метод flatMap

Решить указанную проблему можно с помощью метода `flatMap` следующим образом:

```
List<String> uniqueCharacters =
  words.stream()
       .map(word -> word.split(""))
       .flatMap(Arrays::stream)
       .distinct()
       .collect(toList());
```

Преобразуем каждое из слов в массив его букв

Схлопываем все сгенерированные потоки данных в единый поток

При использовании метода `flatMap` каждый из массивов отображается не в отдельный поток данных, а в *содержимое единого потока*. Все потоки, генерируемые при использовании `map(Arrays::stream)`, сливаются (схлопываются) в единый поток. Последствия применения метода `flatMap` показаны на рис. 5.6. Сравните его с работой метода `map` на рис. 5.5.

В двух словах, метод `flatMap` дает возможность заменить каждое значение потока другим потоком, после чего конкатенировать все полученные потоки в один.

Мы вернемся к методу `flatMap` в главе 11, когда будем обсуждать такие продвинутые паттерны Java 8, как использование нового библиотечного класса `Optional` для проверки на `null`. Для закрепления изложенного материала выполните контрольное задание 5.2.

Контрольное задание 5.2. Отображение

1. По заданному списку чисел вернуть список их квадратов. Например, при списке `[1, 2, 3, 4, 5]` необходимо вернуть `[1, 4, 9, 16, 25]`.

Ответ: решить эту задачу можно с помощью метода `map` и лямбда-выражения, принимающего на входе число и возвращающее квадрат этого числа:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares =
    numbers.stream()
           .map(n -> n * n)
           .collect(toList());
```

2. По двум заданным спискам чисел вернуть все их попарные сочетания. Например, при получении списков `[1, 2, 3]` и `[3, 4]` нужно вернуть `[(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)]`. Для простоты можно представить каждую пару в виде массива из двух элементов.

Ответ: можно воспользоваться двумя вызовами метода `map` для организации цикла по обоим спискам и генерации пар. Но при этом будет возвращаться `Stream<Stream<Integer[]>>`. Значит, необходимо схлопнуть все сгенерированные потоки так, чтобы получить `Stream<Integer[]>`. Для этого и предназначен метод `flatMap`:

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
            .flatMap(i -> numbers2.stream()
                                  .map(j -> new int[]{i, j})
                    )
            .collect(toList());
```

3. Как бы вы обобщили предыдущий пример так, чтобы возвращались только те пары, сумма которых делится на 3?

Ответ: как вы видели выше, операция `filter` с предикатом может использоваться для фильтрации элементов потока данных. А поскольку операция `flatMap` возвращает массив `int[]`, представляющий пару чисел, то нам достаточно предиката для проверки деления суммы на 3:

```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs =
    numbers1.stream()
            .flatMap(i ->
                       numbers2.stream()
                               .filter(j -> (i + j) % 3 == 0)
                               .map(j -> new int[]{i, j})
                    )
            .collect(toList());
```

Возвращаемый результат: `[(2, 4), (3, 3)]`.

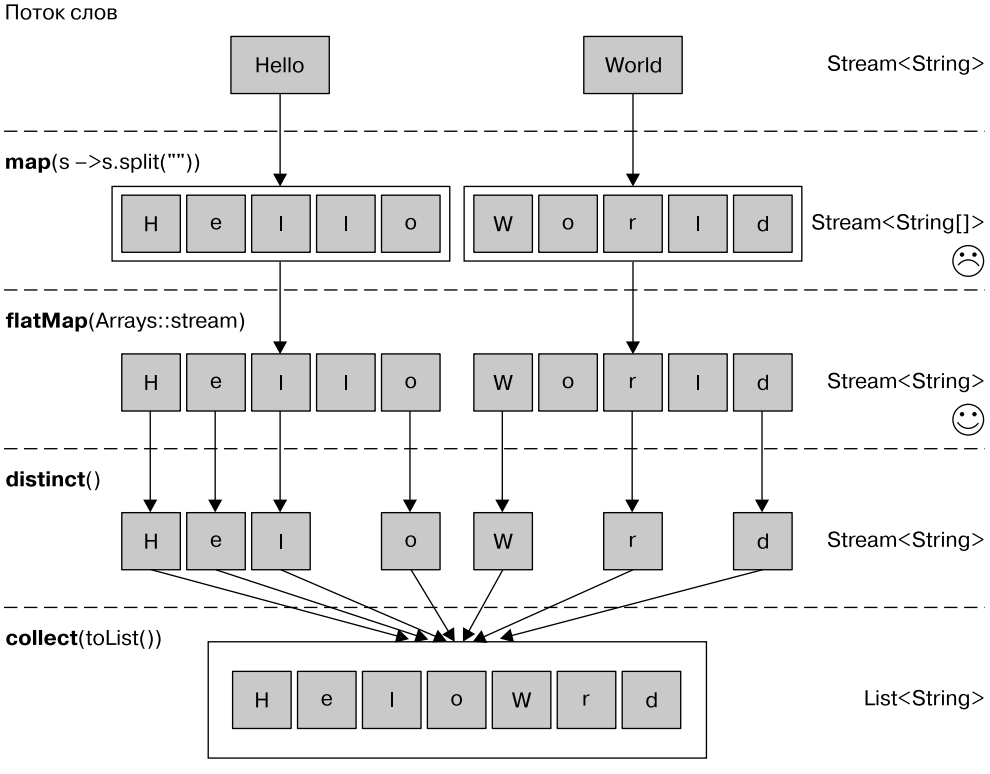


Рис. 5.6. Использование метода flatMap для получения уникальных символов из списка слов

5.4. Поиск и сопоставление с шаблоном

Еще одна распространенная идиома обработки данных: поиск элементов из набора данных, которые удовлетворяют заданному свойству. Stream API предоставляет такие возможности с помощью методов `allMatch`, `anyMatch`, `noneMatch`, `findFirst` и `findAny`.

5.4.1. Проверяем, удовлетворяет ли предикату хоть один элемент

Метод `anyMatch` служит для ответа на вопрос: «Удовлетворяет ли заданному предикату хотя бы один элемент из потока данных?» Например, им можно воспользоваться, чтобы выяснить, есть ли в меню хоть какие-то вегетарианские блюда:

```
if(menu.stream().anyMatch(Dish::isVegetarian)) {
    System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

Метод `anyMatch` возвращает `boolean`, а значит, является завершающей операцией.

5.4.2. Проверяем, удовлетворяют ли предикату все элементы

Метод `allMatch` работает аналогично `anyMatch`, но проверяет, удовлетворяют ли заданному предикату все элементы потока данных. Например, можно использовать его, чтобы выяснить, полезно ли меню для здоровья (все блюда в нем содержат менее 1000 калорий):

```
boolean isHealthy = menu.stream()
                        .allMatch(dish -> dish.getCalories() < 1000);
```

Противоположностью `allMatch` является метод `noneMatch`. Он проверяет, точно ли ни один элемент списка не соответствует заданному предикату. Например, можно переписать предыдущий пример следующим образом с помощью метода `noneMatch`:

```
boolean isHealthy = menu.stream()
                        .noneMatch(d -> d.getCalories() >= 1000);
```

Эти три операции — `anyMatch`, `allMatch` и `noneMatch` — используют то, что обычно называется *сокращенной схемой вычислений* (short-circuiting) потоковой версией схемы, которая в обычном языке Java представлена хорошо знакомыми вам операторами `&&` и `||`.

Сокращенная схема вычислений

Некоторым операциям для получения результата не требуется проходить полный цикл вычислений. Например, вам нужно вычислить большое булево выражение, части которого соединяются с помощью операторов `and`. Для того чтобы установить, что все выражение равно `false`, достаточно обнаружить одно-единственное подвыражение, равное `false`, вне зависимости от того, насколько велико выражение в целом. Вычислять все выражение необходимости нет. Это и называется *сокращенной схемой вычислений*.

Применительно к потокам данных при некоторых операциях (таких как `allMatch`, `noneMatch`, `findFirst` и `findAny`) не нужно обрабатывать весь поток для получения результата. Результат можно вернуть сразу по обнаружении искомого элемента. Сокращенная схема вычислений используется и для операции `limit`, создающей поток данных заданного размера, без обработки всех элементов входного потока. Подобные операции очень удобны (например, когда речь идет об обработке потока бесконечного размера, который такие операции могут преобразовать в поток конечного размера). Мы покажем примеры потоков данных бесконечного размера в разделе 5.7.

5.4.3. Поиск элемента в потоке

Метод `findAny` возвращает произвольный элемент текущего потока данных. Его можно использовать в сочетании с другими потоковыми операциями. Например, пусть нужно найти в меню любое вегетарианское блюдо. Для такого запроса можно скомбинировать метод `filter` с методом `findAny`:

```
Optional<Dish> dish =
  menu.stream()
```

```
    .filter(Dish::isVegetarian)
    .findAny();
```

Подобный конвейер, возможно, «за кулисами» будет оптимизирован так, чтобы выполнить поиск за один проход и завершить работу сразу же по обнаружении искомого благодаря использованию сокращенной схемы вычислений. Но постойте-ка, что это за `Optional` в предыдущем фрагменте кода?

Коротко об Optional. Класс `Optional<T>` (`java.util.Optional`) представляет собой класс-контейнер, отражающий наличие или отсутствие значения. В предыдущем коде существует вероятность того, что метод `findAny` не найдет ни одного элемента. Вместо того чтобы возвращать `null` — стратегия, часто приводящая к возникновению ошибок, — создатели библиотеки языка Java 8 ввели в нее класс `Optional<T>`. Мы не будем углубляться в технические подробности `Optional`, поскольку в главе 11 собираемся подробно обсудить возможные выгоды от использования `Optional` для предотвращения связанных с проверкой на `null` ошибок. Пока вам достаточно знать, что в классе `Optional` есть несколько методов, заставляющих вас явным образом проверять наличие значения или обрабатывать случай его отсутствия.

- Метод `isPresent()` возвращает `true`, если объект `Optional` содержит значение, и `false` в противном случае.
- Метод `ifPresent(Consumer<T> block)` выполняет заданный блок кода, если значение есть. Мы познакомили вас с функциональным интерфейсом `Consumer` в главе 3; с его помощью можно передавать лямбда-выражения, принимающие аргумент типа `T` и возвращающие `void`.
- `T get()` возвращает значение, если оно есть; в противном случае генерирует исключение `NoSuchElementException`.
- `T orElse(T other)` возвращает значение, если оно есть; в противном случае возвращает значение по умолчанию.

Например, в предыдущем коде необходимо явным образом проверить наличие блюда в объекте `Optional`, чтобы получить доступ к его названию.

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(dish -> System.out.println(dish.getName());
```

Возвращает Optional<Dish>

Если значение есть, оно выводится в консоль; в противном случае ничего не происходит

5.4.4. Поиск первого элемента

Для некоторых потоков данных определен так называемый *порядок обнаружения* (encounter order), то есть естественный порядок элементов в потоке (в качестве примера можно привести поток данных, сгенерированный из списка или отсортированной последовательности данных). Для подобных потоков может понадобиться найти первый элемент. Для этой цели предназначен метод `findFirst`, работающий аналогично методу `findAny` (например, следующий код находит в заданном списке чисел первый квадрат натурального числа, который делится на 3):

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
  someNumbers.stream()
             .map(n -> n * n)
             .filter(n -> n % 3 == 0)
             .findFirst(); // Результат равен 9
```

Когда имеет смысл применять методы findFirst и findAny

Наверное, вы недоумеваете, зачем нужны и метод `findFirst`, и метод `findAny`. Все дело в параллелизме. Поиск первого элемента налагает больше ограничений при параллельном выполнении. Если вам неважно, какой именно элемент будет возвращен, задействуйте `findAny`, поскольку он налагает меньше ограничений (а значит, обеспечивает более высокую производительность) при использовании параллельных потоков.

5.5. Свертка

Ранее упоминавшиеся завершающие операции возвращают `boolean` (`allMatch` и т. п.), `void` (`forEach`) или объект `Optional` (`findAny` и т. п.). Мы также использовали метод `collect` для объединения всех элементов потока в список.

В этом разделе вы увидите, как сгруппировать элементы потока для выражения более сложных запросов, таких как «Вычислить общую сумму калорий в меню» или «Какое блюдо в меню самое калорийное?» с помощью операции `reduce`. В подобных запросах все элементы потока группируются многократно для получения в итоге одного значения, например `Integer`. Подобные запросы относятся к *операциям свертки* (reduction operations, поток данных сворачивается в одно значение). На жаргоне из сферы функционального программирования о них говорят как о *складывании* (fold), поскольку такую операцию можно рассматривать как многократное складывание длинной полоски бумаги (потока данных), до тех пор пока она не превратится в маленький квадратик — результат операции складывания.

5.5.1. Суммирование элементов

Прежде чем приступить к обсуждению метода `reduce`, не помешает посмотреть, как можно просуммировать элементы списка с помощью цикла `for-each`:

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

Для получения результата все элементы списка `numbers` группируются с помощью оператора сложения. Иначе говоря, мы производим *свертку* списка в одно число посредством многократного сложения. В этом коде присутствует два параметра:

- исходное значение переменной `sum`, в данном случае `0`;
- операция группировки элементов списка, в данном случае `+`.

Как замечательно было бы иметь возможность умножать все числа без копирования и вставки этого кода. Именно здесь может быть полезна операция `reduce` (она представляет собой абстракцию, которая несколько раз воплощает паттерн применения какой-либо другой операции). Просуммировать все элементы потока данных можно следующим образом:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

Метод `reduce` принимает два аргумента:

- начальное значение, в данном случае `0`;
- оператор `BinaryOperator<T>`, который группирует два элемента и выдает новое значение; в данном случае мы используем лямбда-выражение `(a, b) -> a + b`.

Точно так же просто получить произведение всех элементов, для этого достаточно передать в операцию `reduce` лямбда-выражение `(a, b) -> a * b`:

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

Рисунок 5.7 иллюстрирует использование операции `reduce` для потока данных: лямбда-выражение группирует элементы до тех пор, пока поток, содержащий числа 4, 5, 3, 9, не окажется свернут в одно значение.

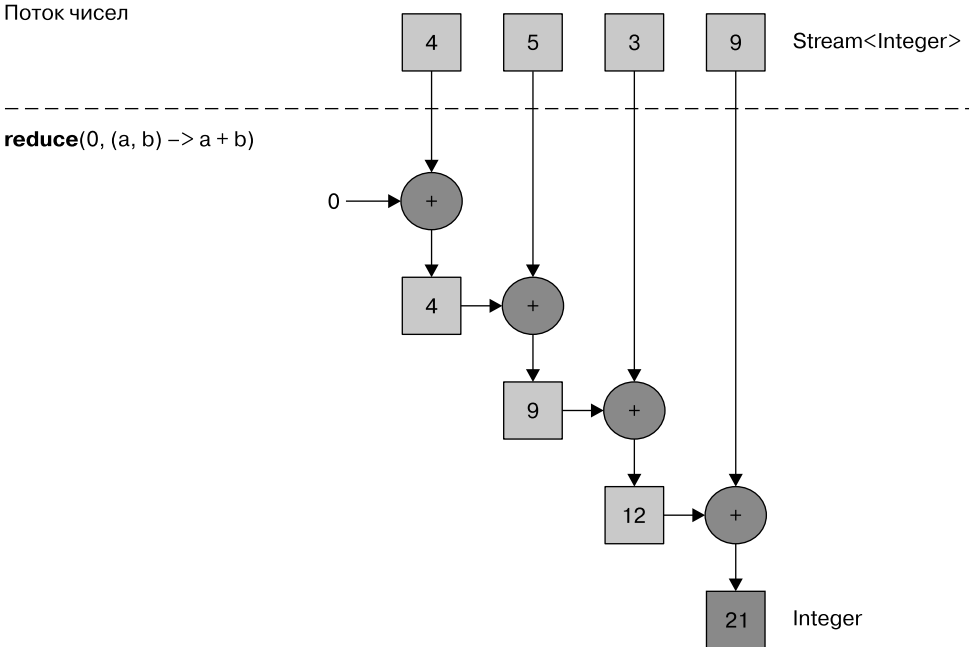


Рис. 5.7. Использование метода reduce для суммирования чисел из потока данных

Разберем подробнее, как именно операция `reduce` суммирует поток чисел. Во-первых, в качестве первого параметра лямбда-выражения `(a)` используется `0`, а из потока читается число `4`, которое служит в качестве второго параметра `(b)`. Сумма

0 + 4 составляет 4, и этот результат становится новым накопленным значением. Затем лямбда-выражение вызывается снова с этим новым накопленным значением и следующим элементом потока — 5, в результате чего получается новое накопленное значение — 9. Далее лямбда-выражение вызывается вновь с накопленным значением и следующим элементом — 3, в результате чего получается 12. Наконец, лямбда-выражение вызывается со значением 12 и последним элементом потока — 9, и мы получаем итоговое значение — 21.

Этот код можно сделать более лаконичным с помощью ссылки на метод. Начиная с Java 8, в классе Integer имеется статический метод sum, предназначенный для сложения двух чисел, который гораздо более удобен по сравнению с написанием одного и того же кода лямбда-выражения.

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

Вариант без начального значения. Существует также перегруженный вариант метода reduce, не принимающий начального значения в качестве параметра и возвращающий Optional:

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

Почему он возвращает интерфейс Optional? Рассмотрим случай, когда поток данных не содержит элементов. Операция reduce при этом не может вернуть сумму, потому что начального значения нет. Именно поэтому результат обернут в объект Optional в качестве указания на то, что сумма может отсутствовать. Посмотрим теперь, что еще можно делать с помощью reduce.

5.5.2. Максимум и минимум

Оказывается, что свертки вполне достаточно для вычисления максимума и минимума! Посмотрим, как применить наши только что полученные знания об операции reduce для вычисления максимального и минимального элементов потока. Как вы видели, метод reduce принимает на входе два параметра:

- начальное значение;
- лямбда-выражение, которое группирует два элемента потока данных и формирует новое значение.

Это лямбда-выражение применяется пошагово к каждому из элементов потока данных, как показано на рис. 5.7 для случая оператора сложения. Теперь же нам требуется лямбда-выражение, которое бы возвращало максимальный из двух заданных элементов. Операция reduce затем вычисляет максимальное значение между этим и следующим элементом потока, и так до тех пор, пока не будет прочитан весь поток! Для вычисления максимального элемента потока операцию reduce можно использовать, как проиллюстрировано на рис. 5.8.

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```

Для вычисления минимума достаточно передать операции reduce Integer.min вместо Integer.max:

```
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

Можно было воспользоваться и лямбда-выражением `(x, y) -> x < y ? x : y` вместо `Integer::min`, но последнее намного более удобочитаемо!

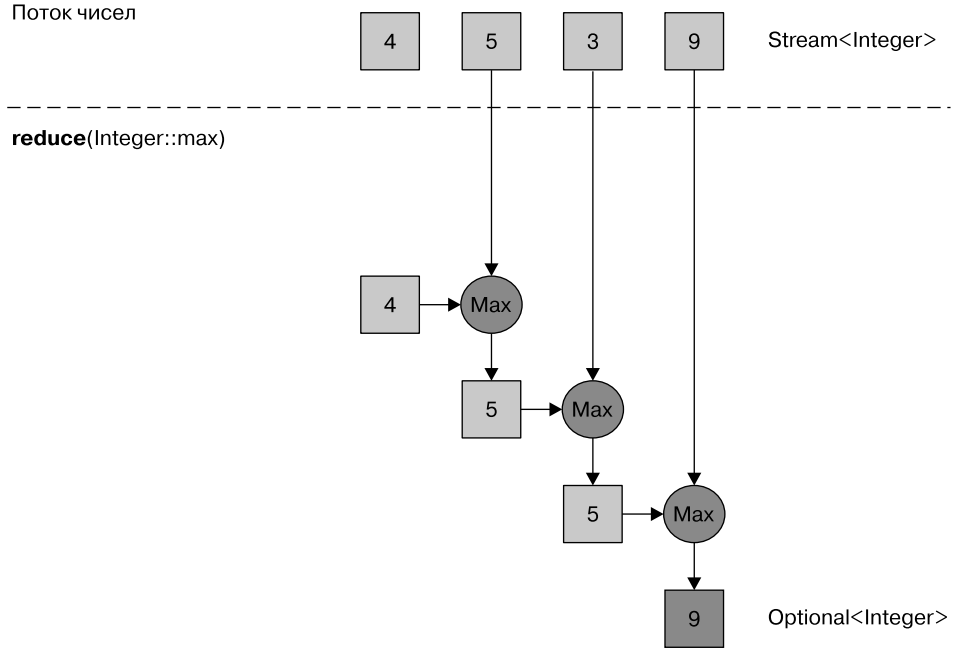


Рис. 5.8. Операция reduce — вычисление максимума

Чтобы проверить, насколько хорошо вы понимаете функционирование операции `reduce`, попробуйте выполнить контрольное задание 5.3.

Контрольное задание 5.3. Свертка

Как бы вы подсчитали число блюд в потоке данных с помощью методов `map` и `reduce`?

Ответ: решить эту задачу можно с помощью отображения всех элементов потока в число 1 с последующим суммированием их с помощью `reduce`! Это эквивалентно подсчету числа элементов в потоке:

```
int count = menu.stream()
                .map(d -> 1)
                .reduce(0, (a, b) -> a + b);
```

Цепочка операций `map` и `reduce` широко известна под названием паттерна «Отображение-свертка» (map-reduce), легко распараллеливаемого и потому примененного компанией Google в своей системе веб-поиска. Отметим, что в главе 4 мы встречали встроенный метод `count`, предназначенный для подсчета количества элементов в потоке данных:

```
long count = menu.stream().count();
```

Преимущества метода reduce и параллелизм

Основное преимущество использования метода `reduce` по сравнению с (приведенным выше) пошаговым суммированием в цикле — абстрагирование цикла с помощью внутренней итерации, благодаря которому у внутреннего механизма Java появляется возможность параллельного выполнения операции `reduce`. Пример с суммированием в цикле включает разделяемые обновления переменной `sum`, которые не слишком хорошо распараллеливаются. А если учесть еще и необходимость синхронизации, то, скорее всего, окажется, что конкуренция потоков выполнения нивелирует все ожидаемое от параллелизма повышение производительности! Распараллеливание таких вычислений требует иного подхода: секционирования входных данных, суммирования данных по секциям и группировки результатов. Но код при этом выглядит совсем иначе. Как именно — вы увидите в главе 7, когда мы будем использовать фреймворк ветвления-объединения (fork/join). А пока важно понимать, что распараллеливание в случае паттерна изменяемого накопителя — совершенно бесперспективная идея. Нам нужен новый паттерн, такой, как предлагает операция `reduce`. В главе 7 мы также увидим, что для параллельного суммирования всех элементов потока практически не требуется менять код: просто `stream()` превращается в `parallelStream()`:

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

Но за параллельное выполнение этого кода приходится расплачиваться, как мы объясним позднее, тем, что передаваемое в `reduce` лямбда-выражение не может менять состояние (например, переменные экземпляра), а операция должна быть ассоциативной и коммутативной, чтобы можно было выполнять ее в произвольном порядке.

Вы уже видели примеры свертки, в результате которых получался объект типа `Integer`: сумма значений элементов потока, максимум значений элементов потока или число элементов потока. В разделе 5.6 вы увидите, что существуют дополнительные встроенные методы, такие как `sum` и `max`, которые позволяют писать чуть более лаконичный код для распространенных паттернов свертки. В следующей главе мы рассмотрим и выполнение более сложных разновидностей свертки — с помощью метода `collect`. Например, вместо свертки потока данных в объект типа `Integer` можно свернуть его в `Map`, если требуется сгруппировать блюда по типам.

Потоковые операции: с сохранением состояния и без сохранения состояния

Вы уже видели множество потоковых операций. На первый взгляд они могут показаться универсальным средством на все случаи жизни. Все работает идеально, переход к параллельному выполнению требует лишь замены метода `stream` на `parallelStream` при получении потока данных из коллекции.

Конечно, для многих типов приложений так и есть, как вы видели в предыдущих примерах. Список блюд можно преобразовать в поток данных, воспользоваться `filter` для выбора различных блюд определенного типа, после чего произвести отображение

(`map`) для дальнейшего суммирования, затем выполнить `reduce` для получения общего количества калорий в меню. Можно даже производить подобные потоковые вычисления параллельно. Но характеристики этих операций различаются. Существуют нюансы относительно того, с каким внутренним состоянием они должны работать.

Такие операции, как `map` и `filter`, обрабатывают *по одному* элементу из входящего потока данных, выводя в выходной поток *не более одного* результирующего значения. Это операции, вообще говоря, *без сохранения состояния*: у них нет внутреннего состояния (при условии, что у передаваемого пользователем лямбда-выражения или ссылки на метод нет внутреннего изменяемого состояния).

Но таким операциям, как `reduce`, `sum` и `max`, внутреннее состояние необходимо для накопления результата. В подобных случаях внутреннее состояние невелико. В нашем примере оно представляет собой значение типа `int` или `double`. Вне зависимости от количества элементов в обрабатываемом потоке данных размер внутреннего состояния *ограничен*.

В отличие от них такие операции, как `sorted` и `distinct`, на первый взгляд, ведут себя подобно `map` и `filter` — принимают на входе поток и генерируют другой поток (промежуточная операция), — но между ними существует ключевое различие. Как для сортировки, так и для удаления дубликатов из потока нужно знать предыдущие состояния. Например, сортировка требует *буферизации всех элементов* до вывода первого элемента результатов; плюс важна *неограниченность* хранилища. Это может доставить проблемы в случае большого или бесконечного потока данных. (Например, как вывести поток всех простых чисел в обратном порядке? Для этого сначала нужно вывести наибольшее простое число, которое, как утверждают математики, не существует.) Такие операции называются *операциями с сохранением состояния* (stateful operations).

Вы уже знаете множество потоковых операций, пригодных для выражения сложных запросов обработки данных. В табл. 5.1 вкратце перечислены встречавшиеся до сих пор операции. Вам предстоит попробовать свои силы по работе с ними в упражнении, приведенном в следующем разделе.

Таблица 5.1. Промежуточные и завершающие операции

Операция	Тип	Возвращаемый тип	Используемый тип/функциональный интерфейс	Функциональный дескриптор
filter	Промежуточная	Stream<T>	Predicate<T>	T -> boolean
distinct	Промежуточная (с сохранением состояния/неограниченная)	Stream<T>		T -> boolean
takeWhile	Промежуточная	Stream<T>	Predicate<T>	T -> boolean
dropWhile	Промежуточная	Stream<T>	Predicate<T>	
skip	Промежуточная (с сохранением состояния/ограниченная)	Stream<T>	long	

Операция	Тип	Возвращаемый тип	Используемый тип/функциональный интерфейс	Функциональный дескриптор
limit	Промежуточная (с сохранением состояния/ограниченная)	Stream<T>	long	
map	Промежуточная	Stream<T>	Function<T, R>	T -> R
flatMap	Промежуточная	Stream<T>	Function<T, Stream<R>>	T -> Stream<R>
sorted	Промежуточная (с сохранением состояния/неограниченная)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	Завершающая	boolean	Predicate<T>	T -> boolean
noneMatch	Завершающая	boolean	Predicate<T>	T -> boolean
allMatch	Завершающая	boolean	Predicate<T>	T -> boolean
findAny	Завершающая	Optional<T>		
findFirst	Завершающая	Optional<T>		
forEach	Завершающая	void	Consumer<T>	T -> void
collect	Завершающая	R	Collector<T, A, R>	
reduce	Завершающая (с сохранением состояния/ограниченная)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	Завершающая	long		

5.6. Переходим к практике

В этом разделе вы сможете применить на практике все полученные до сих пор теоретические знания о потоках данных. Мы обратимся теперь к другой предметной области: транзакциям, проводимым трейдерами на бирже. Менеджер попросил вас выполнить восемь заданий. Под силу ли это вам? Решения будут приведены в подразделе 5.6.2, но мы рекомендуем вам сначала попробовать выполнить задание самостоятельно, для практики.

1. Найти все транзакции за 2011 год и отсортировать их по сумме (от меньшей к большей).
2. Вывести список неповторяющихся городов, в которых работают трейдеры.
3. Найти всех трейдеров из Кембриджа и отсортировать их по именам.
4. Вернуть строку со всеми именами трейдеров, отсортированными в алфавитном порядке.
5. Выяснить, существует ли хоть один трейдер из Милана.
6. Вывести суммы всех транзакций трейдеров из Кембриджа.
7. Какова максимальная сумма среди всех транзакций?
8. Найти транзакцию с минимальной суммой.

5.6.1. Предметная область: трейдеры и транзакции

Вот предметная область, с которой вам предстоит работать, в виде списка объектов классов `Trader` и `Transaction`:

```
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario","Milan");
Trader alan = new Trader("Alan","Cambridge");
Trader brian = new Trader("Brian","Cambridge");
List<Transaction> transactions = Arrays.asList(
    new Transaction(brian, 2011, 300),
    new Transaction(raoul, 2012, 1000),
    new Transaction(raoul, 2011, 400),
    new Transaction(mario, 2012, 710),
    new Transaction(mario, 2012, 700),
    new Transaction(alan, 2012, 950)
);
```

Классы `Trader` и `Transaction` описаны следующим образом:

```
public class Trader{
    private final String name;
    private final String city;
    public Trader(String n, String c){
        this.name = n;
        this.city = c;
    }
    public String getName(){
        return this.name;
    }
    public String getCity(){
        return this.city;
    }
    public String toString(){
        return "Trader:"+this.name + " in " + this.city;
    }
}
public class Transaction{
    private final Trader trader;
    private final int year;
    private final int value;
    public Transaction(Trader trader, int year, int value){
        this.trader = trader;
        this.year = year;
        this.value = value;
    }
    public Trader getTrader(){
        return this.trader;
    }
    public int getYear(){
        return this.year;
    }
    public int getValue(){
        return this.value;
    }
```

```
    public String toString(){
        return "{" + this.trader + ", " +
               "year: "+this.year+", " +
               "value:" + this.value +"}";
    }
}
```

5.6.2. Решения

Приведем решения в листингах 5.1–5.8, чтобы вы могли проверить, правильно ли понимаете пройденный материал. Удачи!

Листинг 5.1. Найти все транзакции за 2011 год и отсортировать их по сумме (от маленькой к большой)

```
List<Transaction> tr2011 =
    transactions.stream()
                .filter(transaction -> transaction.getYear() == 2011)
                .sorted(comparing(Transaction::getValue))
                .collect(toList());
```

Передаем в функцию filter предикат для выбора транзакций только за 2011 год

Собираем все элементы получившегося потока данных в объект List

Сортируем их по сумме транзакции

Листинг 5.2. Выводим список неповторяющихся городов, в которых работают трейдеры

```
List<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .distinct()
                .collect(toList());
```

Извлекаем из каждой транзакции информацию о городе, где живет соответствующий трейдер

Выбираем только неповторяющиеся города

Мы еще этого вам не рассказывали, но вместо `distinct()` можно также воспользоваться операцией `toSet()` для преобразования потока данных в множество. Вы узнаете об этом больше из главы 6.

```
Set<String> cities =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getCity())
                .collect(toSet());
```

Листинг 5.3. Находим всех трейдеров из Кембриджа и сортируем их по именам

```
List<Trader> traders =
    transactions.stream()
                .map(Transaction::getTrader)
                .filter(trader -> trader.getCity().equals("Cambridge"))
                .distinct()
                .sorted(comparing(Trader::getName))
                .collect(toList());
```

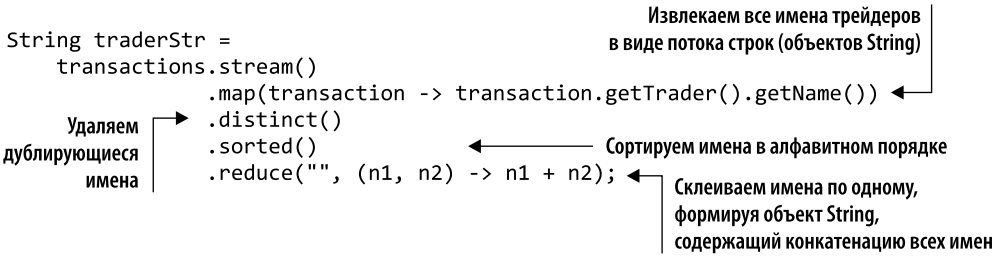
Извлекаем из транзакций информацию о трейдерах

Выбираем только трейдеров из Кембриджа

Удаляем дубликаты

Сортируем получившийся поток данных трейдеров по именам

Листинг 5.4. Возвращаем строку со всеми именами трейдеров, отсортированными в алфавитном порядке

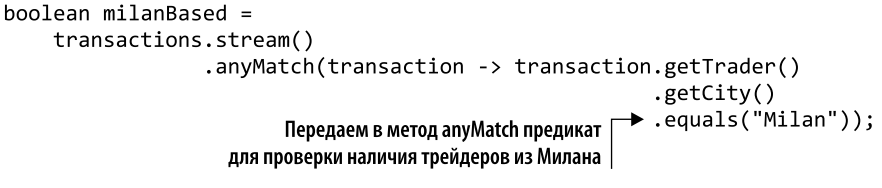



```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .reduce("", (n1, n2) -> n1 + n2);
```

Отметим, что это решение неоптимально в смысле расхода ресурсов (при подобной последовательной конкатенации строк каждый раз создается новый объект `String`). В следующей главе мы покажем вам более эффективное решение, основанное на функции `joining()`, используемой следующим образом (при этом неявным образом применяется `StringBuilder`):

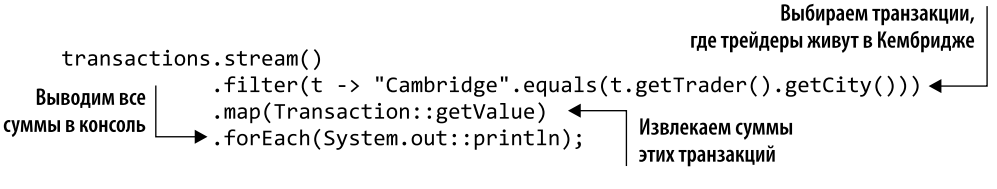
```
String traderStr =
    transactions.stream()
                .map(transaction -> transaction.getTrader().getName())
                .distinct()
                .sorted()
                .collect(joining());
```

Листинг 5.5. Существует ли хоть один трейдер из Милана?



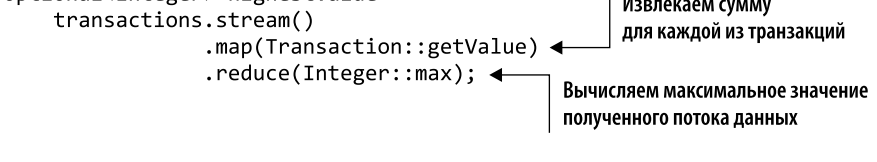

```
boolean milanBased =
    transactions.stream()
                .anyMatch(transaction -> transaction.getTrader()
                                                    .getCity()
                                                    .equals("Milan"));
```

Листинг 5.6. Выводим суммы всех транзакций трейдеров из Кембриджа




```
transactions.stream()
            .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
            .map(Transaction::getValue)
            .forEach(System.out::println);
```

Листинг 5.7. Какова максимальная сумма среди всех транзакций?




```
Optional<Integer> highestValue =
    transactions.stream()
                .map(Transaction::getValue)
                .reduce(Integer::max);
```

Листинг 5.8. Находим транзакцию с минимальной суммой

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
              .reduce((t1, t2) ->
                       t1.getValue() < t2.getValue() ? t1 : t2);
```

Находим транзакцию на самую маленькую сумму, последовательно сравнивая суммы всех транзакций

Существует более подходящее решение. Потоки данных поддерживают методы `min` и `max`, принимающие в качестве аргумента объект `Comparator`, определяющий, что сравнивается при вычислении минимума или максимума:

```
Optional<Transaction> smallestTransaction =
  transactions.stream()
              .min(comparing(Transaction::getValue));
```

5.7. Числовые потоки данных

Как вы видели ранее, для вычисления суммы элементов потока можно использовать метод `reduce`. Например, можно вычислить число калорий в меню следующим образом:

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

Проблема этого кода состоит в скрытых затратах на упаковку. Каждый объект `Integer` необходимо неявно распаковать до простого типа данных, прежде чем прибавить его к сумме. Кроме того, разве не приятнее было бы вызывать метод `sum()` напрямую, вот так:

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .sum();
```

Но это невозможно. Проблема в том, что метод `map` генерирует `Stream<T>`. И хотя тип элементов этого потока данных — `Integer`, в интерфейсе класса `Stream` не определен метод `sum`. Почему нет? Допустим, вы работаете с потоком `Stream<Dish>`, таким как объект `menu`, а сложение элементов такого типа не имеет никакого смысла. Но не беспокойтесь, в Stream API есть также специализированные *версии потоков для примитивных типов данных* (primitive stream specializations), поддерживающие специальные методы для работы с потоками чисел.

5.7.1. Версии потоков для простых типов данных

В Java 8 появились три специализированных интерфейса потоков для решения вышеприведенной задачи: `IntStream`, `DoubleStream` и `LongStream`. Они предназначены для работы соответственно с элементами типов `int`, `double` и `long`, причем без дополнительных скрытых затрат на упаковку. В каждом из этих интерфейсов есть новые методы для выполнения распространенных числовых сверток, например `sum` для вычисления суммы числового потока и `max` для получения максимального элемента. Кроме того, в них есть методы для обратного их преобразования в поток объектов

(при необходимости). Главное — не забывать, что повышенная сложность этих специализированных версий не является свойством самих потоков данных, а отражает сложность упаковки — разницу в производительности между `int` и `Integer` и т. д.

Отображение в числовой поток данных

Самые распространенные методы для преобразования потока данных в специализированные версии — это `mapToInt`, `mapToDouble` и `mapToLong`. Они полностью аналогичны методу `map`, который вы уже встречали ранее, только возвращают специализированную версию вместо `Stream<T>`. Например, можно воспользоваться методом `mapToInt`, чтобы вычислить сумму калорий в объекте `menu`, вот так:

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
```

Возвращает Stream<Dish>

Возвращает IntStream

В этом коде метод `mapToInt` извлекает информацию о количестве калорий из каждого блюда (в виде объекта типа `Integer`) и возвращает в качестве результата `IntStream` (а не `Stream<Integer>`). После этого можно получить общее количество калорий с помощью вызова метода `sum`, определенного в интерфейсе `IntStream`! Отметим, что, если поток данных пуст, `sum` по умолчанию вернет `0`. Интерфейс `IntStream` поддерживает и другие удобные методы, такие как `max`, `min` и `average`.

Преобразование обратно в поток объектов

Аналогично может понадобиться преобразовать числовой поток данных обратно в неспециализированный поток. Например, доступные для интерфейса `IntStream` операции могут возвращать только целочисленные простые типы данных: операция `map` интерфейса `IntStream` принимает на входе лямбда-выражение, принимающее, в свою очередь, в качестве параметра `int` и возвращающее также `int` (`IntUnaryOperator`). Но, возможно, вам нужно вернуть другой тип значения, например `Dish`. Для этого необходим доступ к операциям, описанным в более общем интерфейсе `Stream`. Для преобразования специализированного потока данных в общий (в котором каждый `int` упакован в `Integer`) можно воспользоваться методом `boxed`, вот так:

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> stream = intStream.boxed();
```

Преобразует объект Stream в числовой поток данных

Преобразует числовой поток данных в объект Stream

Из следующего раздела вы узнаете, что метод `boxed` особенно удобен в случае диапазонов чисел, которые нужно упаковать в объектный поток данных.

Значения по умолчанию: класс OptionalInt

Пример с суммированием удобен наличием значения по умолчанию: `0`. Но для вычисления максимального элемента потока `IntStream` это не подойдет, поскольку результат `0` некорректен. Как отличить отсутствие элементов в потоке и то, что максимальное значение действительно равно `0`? Ранее мы упоминали о классе

`Optional` — контейнере, указывающем на наличие или отсутствие значения. `Optional` можно параметризовать ссылочными типами, например `Integer`, `String` и т. д. Существуют и три версии типа `Optional` для потоков простых типов данных: `OptionalInt`, `OptionalDouble` и `OptionalLong`.

Например, чтобы найти максимальный элемент потока `IntStream`, можно вызвать метод `max`, который возвращает объект `OptionalInt`:

```
OptionalInt maxCalories = menu.stream()
                              .mapToInt(Dish::getCalories)
                              .max();
```

После этого можно явным образом обработать `OptionalInt` и задать значение по умолчанию, если максимум не был найден:

```
int max = maxCalories.orElse(1);
```

Задаем явным образом максимум по умолчанию, на случай, если значение отсутствует

5.7.2. Числовые диапазоны

При работе с числовыми данными часто приходится иметь дело с диапазонами числовых значений. Например, вам может понадобиться сгенерировать все целые числа от 1 до 100. В Java 8 появилось два статических метода в интерфейсах `IntStream` и `LongStream`, предназначенных для генерации подобных диапазонов значений: `range` и `rangeClosed`. Оба они принимают начальное значение диапазона в качестве первого параметра, а конечное значение — в качестве второго параметра. Различие лишь в том, что `rangeClosed` включает границы диапазона, а `range` — нет. Рассмотрим пример:

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100)
                                 .filter(n -> n % 2 == 0);
System.out.println(evenNumbers.count());
```

Представляет диапазон чисел от 1 до 100

Представляет поток четных чисел от 1 до 100

Выводит в консоль 50 — количество четных чисел от 1 до 100

Здесь с помощью метода `rangeClosed` мы генерируем диапазон всех целых чисел от 1 до 100. Метод возвращает поток данных, так что можно привязать к нему цепочкой метод `filter` и выбрать только четные числа. На этом этапе никаких фактических вычислений еще не выполняется. Затем мы вызываем для полученного потока данных метод `count`. Поскольку `count` — завершающая операция, она обрабатывает поток и возвращает результат `50` — число четных чисел от 1 до 100 включительно. Отметим, что если бы мы вызвали вместо этого `IntStream.range(1, 100)`, то получили бы результат `49`, поскольку метод `range` не включает границы диапазона.

5.7.3. Применяем числовые потоки данных на практике: пифагоровы тройки

Рассмотрим теперь более сложный пример, чтобы закрепить полученные вами знания о числовых потоках данных и потоковых операциях. Ваша задача, если вы на нее согласитесь, — создать поток пифагоровых троек.

Пифагорова тройка

Что такое пифагорова тройка? Вернемся на несколько лет назад. Как вы знаете из курса математики, знаменитый греческий математик Пифагор открыл[1], что некоторые тройки целых чисел `(a, b, c)` удовлетворяют уравнению `a * a + b * b = c * c`. Например, (3, 4, 5) — пифагорова тройка, поскольку $3 \cdot 3 + 4 \cdot 4 = 5 \cdot 5$, то есть $9 + 16 = 25$. Существует бесконечное множество подобных троек. Например, (5, 12, 13), (6, 8, 10) и (7, 24, 25) — пифагоровы тройки. Эти тройки полезны тем, что описывают длины трех сторон прямоугольного треугольника, как показано на рис. 5.9.

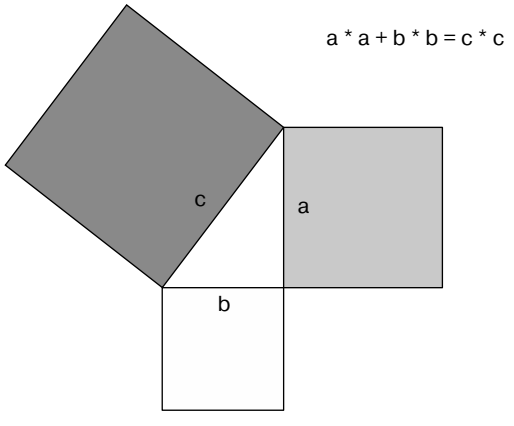


Рис. 5.9. Теорема Пифагора

Представление троек

С чего начать? Прежде всего — описать тройку. Вместо (более корректного варианта) описания нового класса, который бы представлял тройку чисел, можно воспользоваться массивом `int` из трех элементов. Например, с помощью команды `new int[]{3, 4, 5}` можно создать массив, который будет представлять тройку (3, 4, 5). Для обращения к отдельным компонентам троек можно при этом использовать обычную индексацию массивов.

Фильтрация подходящих сочетаний чисел

Допустим, вы получили от кого-то список первых двух чисел троек: `a` и `b`. Откуда вы знаете, что они подходят для пифагоровой тройки? Необходимо проверить, является ли квадратный корень из `a * a + b * b` целым числом. На языке Java это можно выразить так: `Math.sqrt(a*a + b*b) % 1 == 0`. (Дробную часть заданного числа с плавающей точкой `x` можно получить на языке Java с помощью выражения `x % 1.0`, а у целых

[1] Хотя такие тройки чисел и названы в честь Пифагора, открыты они были еще задолго до него. Старейший известный список таких троек приведен на вавилонской глиняной табличке Plimpton 322, датируемой примерно 1800 г. до н. э. — *Примеч. пер.*

чисел вроде 5,0 дробная часть равна 0.) В нашем коде эта идея используется в операции фильтрации (далее вы увидите, как сделать на основе этого полноценный код):

```
filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
```

Если в коде задано значение для a, а поток данных предоставляет возможные значения b, можно отфильтровать подобным образом значения b, которые формируют с заданным a пифагорову тройку.

Генерация троек

После фильтрации мы уже знаем, что a и b формируют подходящее сочетание чисел. Для создания тройки необходимо вычислить c. Для преобразования элементов в пифагорову тройку можно воспользоваться операцией map:

```
stream.filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
      .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Генерация значений для b

Почти готово! Осталось сгенерировать значения для b. Вы уже видели, что с помощью метода Stream.rangeClosed можно сгенерировать поток чисел в заданном диапазоне. Мы можем воспользоваться им для создания числовых значений для b от 1 до 100:

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .boxed()
         .map(b -> new int[]{a, b, (int) Math.sqrt(a * a + b * b)});
```

Обратите внимание на вызов boxed после filter для преобразования IntStream, возвращаемого методом rangeClosed, в поток Stream<Integer>. Дело в том, что метод map возвращает массив int для каждого из элементов потока данных. Метод map интерфейса IntStream ожидает, что для каждого из элементов потока будет возвращаться int, а нам нужно вовсе не это! Можно переписать этот код с помощью метода mapToObj интерфейса IntStream, который возвращает поток объектов-значений:

```
IntStream.rangeClosed(1, 100)
         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
         .mapToObj(b -> new int[]{a, b, (int) Math.sqrt(a*a + b*b)});
```

Генерация значений для a

Выше мы считали, что значения a у нас есть. И получили поток, который генерирует пифагоровы тройки для заданного значения a. Что же делать? То же, что мы сделали для b: сгенерировать числовые значения для a! Окончательное решение задачи выглядит так:

```
Stream<int[]> pythagoreanTriples =
    IntStream.rangeClosed(1, 100).boxed()
             .flatMap(a ->
                IntStream.rangeClosed(a, 100)
                         .filter(b -> Math.sqrt(a*a + b*b) % 1 == 0)
                         .mapToObj(b ->
                            new int[]{a, b, (int)Math.sqrt(a * a + b * b)})
                     );
```

Но что в этом коде делает `flatMap`? Сначала мы создаем числовой диапазон от 1 до 100 для генерации значений для `a`. Для каждого заданного значения `a` мы создаем поток троек. В результате отображения значения `a` в поток троек мы получили бы поток потоков! Метод `flatMap`, помимо собственно отображения, схлопывает все сгенерированные потоки троек в единый поток. В результате мы получаем поток троек. Обратите также внимание, что диапазон `b` — от `a` до 100. Начинать данный диапазон с 1 не имеет смысла, поскольку это приведет к дублирующимся тройкам (например, (3, 4, 5) и (4, 3, 5)).

Выполнение кода

Теперь можно запустить получившуюся программу и выбрать явным образом, с помощью уже известной вам операции `limit`, сколько троек нужно вернуть из полученного потока данных:

```
pythagoreanTriples.limit(5)
                  .forEach(t ->
                     System.out.println(t[0] + ", " + t[1] + ", " + t[2]));
```

В результате выполнения в консоль будет выведено:

```
3, 4, 5
5, 12, 13
6, 8, 10
7, 24, 25
8, 15, 17
```

Существует ли более оптимальное решение?

Наше текущее решение не оптимально, поскольку квадратный корень вычисляется дважды. Для большей лаконичности кода можно сгенерировать все тройки вида `(a*a, b*b, a*a+b*b)`, а затем отфильтровать те, которые соответствуют нужному критерию:

```
Stream<double[]> pythagoreanTriples2 =
  IntStream.rangeClosed(1, 100).boxed()
           .flatMap(a ->
              IntStream.rangeClosed(a, 100)
                       .mapToObj(
                           b -> new double[]{a, b, Math.sqrt(a*a + b*b)})
                       .filter(t -> t[2] % 1 == 0));
```

Генерация троек

Третий компонент тройки должен быть целым числом

5.8. Создание потоков данных

Надеемся, вы уже убедились, что потоки данных — полезное и весьма мощное средство выражения запросов обработки данных. Вы видели возможность создания потока данных из коллекции с помощью метода `Stream`. Кроме того, мы показали вам, как создавать числовые потоки на основе заданных диапазонов чисел. Но потоки данных можно создавать еще множеством других способов! В этом разделе демонстрируется, как создать поток из последовательности значений, массива, файла и даже как создавать бесконечные потоки данных на основе порождающей функции!

5.8.1. Создание потока данных из перечня значений

Потоки данных можно создавать на основе явно указываемых значений с помощью статического метода `Stream.of`, который способен принимать произвольное число

параметров. Например, в следующем коде мы создаем поток строк путем непосредственного вызова `Stream.of`. Затем, перед выводом строк в консоль по одной, мы преобразуем их в верхний регистр:

```
Stream<String> stream = Stream.of("Modern ", "Java ", "In ", "Action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

Для получения пустого потока данных можно воспользоваться методом `empty`:

```
Stream<String> emptyStream = Stream.empty();
```

5.8.2. Создание потока данных из объекта, допускающего неопределенное значение

В Java 9 появился новый метод для создания потока данных объекта, допускающего неопределенное значение (`null`). При работе с потоками данных вы можете столкнуться с ситуацией, когда извлеченный объект (возможно, с неопределенным значением) нужно преобразовать в поток данных (или пустой поток в случае объекта с неопределенным значением). Например, метод `System.getProperty` возвращает `null` при отсутствии свойства с заданным ключом. При использовании его с потоком данных необходимо явным образом проверить на `null`:

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream
    = homeValue == null ? Stream.empty() : Stream.of(value);
```

С помощью метода `Stream.ofNullable` можно упростить код:

```
Stream<String> homeValueStream
    = Stream.ofNullable(System.getProperty("home"));
```

Этот паттерн особенно удобен в сочетании с методом `flatMap` и потоком значений, среди которых могут быть объекты со значением `null`:

```
Stream<String> values =
   Stream.of("config", "home", "user")
         .flatMap(key -> Stream.ofNullable(System.getProperty(key)));
```

5.8.3. Создание потоков данных из массивов

Поток данных можно создать на основе массива, с помощью статического метода `Arrays.stream`, принимающего в качестве параметра массив. Например, можно преобразовать массив объектов простого типа `int` в объект `IntStream`, а затем просуммировать этот `IntStream`, получив результат `int`, вот так:

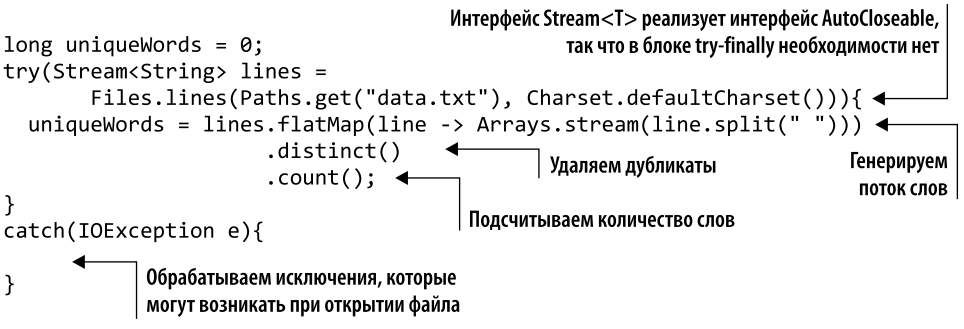
```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

Сумма равна 41

5.8.4. Создание потоков данных из файлов

API NIO (non-blocking I/O, неблокирующего ввода/вывода) языка Java, используемый для таких операций ввода/вывода, как обработка файлов, был расширен для того, чтобы использовать возможности Stream API. Многие статические методы из пакета `java.nio.file.Files` возвращают поток данных. Например, удобный метод

`Files.lines` возвращает поток строк файла в виде символьных значений. С помощью полученных ранее знаний мы можем применить этот метод для вычисления количества уникальных слов в файле:



Мы используем метод `Files.lines`, чтобы вернуть поток, каждый элемент которого представляет собой одну из строк заданного файла. Этот вызов заключен в блок `try/catch`, поскольку источник потока данных представляет собой ресурс ввода/вывода. Фактически вызов метода `Files.lines` открывает ресурс ввода/вывода, который необходимо потом закрыть во избежание утечек памяти. Ранее для этой цели требовался явный блок `finally`. Но, что очень удобно, интерфейс `Stream` реализует интерфейс `AutoCloseable`. Это значит, что управление ресурсом происходит автоматически внутри блока `try`. После получения потока строк файла можно разбить каждую из них на отдельные слова с помощью вызова метода `split` объектов `line`. Обратите внимание на использование операции `flatMap` для генерации одного схлопнутого потока слов вместо множества потоков слов для каждой из строк файла. И наконец, мы подсчитываем все уникальные слова в потоке, связывая цепочкой методы `distinct` и `count`.

5.8.5. Потоки на основе функций: создание бесконечных потоков

В Stream API есть два статических метода для генерации потока данных на основе функции: `Stream.iterate` и `Stream.generate`. С помощью этих двух операций можно создать то, что называется *бесконечным потоком данных* (infinite stream): поток данных без фиксированного размера, в отличие от потока, созданного из конкретной коллекции. Генерируемые с помощью методов `iterate` и `generate` потоки данных создают значения по мере необходимости с помощью заданной функции, а значит, могут делать это неограниченно долго! Обычно имеет смысл использовать для подобных потоков `limit(n)` во избежание вывода в консоль бесконечного количества значений.

Метод iterate

Рассмотрим сначала простой пример использования метода `iterate`:

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

Метод `iterate` принимает в качестве аргументов начальное значение (в данном случае `0`) и лямбда-выражение (типа `UnaryOperator<T>`), которое последовательно применяет к каждому из генерируемых значений. В данном случае мы возвращаем предыдущий элемент, к которому прибавляется `2` с помощью лямбда-выражения `n -> n + 2`. В результате метод `iterate` генерирует поток всех четных чисел: первый элемент — начальное значение `0`. Затем к начальному значению прибавляется `2` и получается новое значение `2`; снова прибавляется `2` и получается `4` и т. д. Операция `iterate` является принципиально последовательной, поскольку результат зависит от предыдущего значения. Отметим, что в результате использования этой операции мы получаем *бесконечный поток данных* — у потока нет последнего значения, поскольку значения вычисляются по требованию и этот процесс может продолжаться сколь угодно долго. В таком случае говорится, что поток данных — *неограниченный*. Как уже обсуждалось ранее, это ключевое различие между потоком данных и коллекцией. Для ограничения размера потока явным образом мы воспользовались методом `limit`. В данном случае мы выбираем первые десять четных чисел. А затем вызываем завершающую операцию `forEach` для потребления потока и вывода в консоль по отдельности каждого из элементов.

Вообще говоря, использовать метод `iterate` следует тогда, когда нужно сгенерировать ряд последовательных значений (например, ряд последовательных дат: 31 января, 1 февраля и т. д.). Более сложный пример использования метода `iterate` ждет вас в контрольном задании 5.4. Попробуйте его выполнить.

Контрольное задание 5.4. Ряд двоек чисел Фибоначчи

Последовательность чисел Фибоначчи — одно из классических упражнений при изучении программирования. Эта последовательность представляет собой числа из следующего ряда: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.... Первые два числа последовательности — 0 и 1, а каждое из последующих представляет собой сумму двух предыдущих.

Последовательность пар чисел Фибоначчи строится аналогично и представляет собой ряд, каждый элемент в котором состоит из двух соседних чисел в последовательности Фибоначчи: (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21)....

Ваша задача: сгенерировать первые 20 элементов последовательности пар чисел Фибоначчи с помощью метода `iterate`!

Для начала немного вам поможем. Первая проблема — метод `iterate` принимает в качестве аргумента объект `UnaryOperator<T>`, а нам нужен поток пар чисел вида (0, 1). Можно, хотя это и довольно неудобно, воспользоваться для представления подобной пары чисел массивом из двух элементов. Например, первый элемент последовательности пар чисел Фибоначчи будет представлен выражением `new int[] {0, 1}`. Оно и будет начальным значением для метода `iterate`:

```
Stream.iterate(new int[]{0, 1}, ???)
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

В этом контрольном задании ваша задача — написать код `???` из выделенного фрагмента. Помните, что метод `iterate` применяет переданное в него лямбда-выражение последовательно.

Ответ:

```
Stream.iterate(new int[]{0, 1},
               t -> new int[]{t[1], t[0]+t[1]})
      .limit(20)
      .forEach(t -> System.out.println("(" + t[0] + "," + t[1] +")"));
```

Как этот код работает? Методу `iterate` нужно лямбда-выражение, которое бы задавало правило вычисления следующего элемента последовательности. В случае пары чисел (3, 5) последующий элемент будет иметь вид (5, 3 + 5) = (5, 8). Следующий — (8, 5 + 8). Видите правило? Последующий элемент для заданной пары чисел имеет вид (t[1], t[0] + t[1]). Именно такое правило вычисления и должно задавать наше лямбда-выражение: `t -> new int[]{t[1],t[0] + t[1]}`. Запустив этот код, вы получите ряд (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21)... Заметим, что при необходимости вывода в консоль обычной последовательности чисел Фибоначчи можно воспользоваться операцией `map` для извлечения только первого элемента каждой из пар:

```
Stream.iterate(new int[]{0, 1},
               t -> new int[]{t[1],t[0] + t[1]})
      .limit(10)
      .map(t -> t[0])
      .forEach(System.out::println);
```

В результате выполнения этого кода будет выведен обычный ряд Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34....

В Java 9 в метод `iterate` была добавлена поддержка предикатов. Например, можно генерировать числа, начиная с 0, но остановить выполнение, как только встретится число, превышающее 100:

```
IntStream.iterate(0, n -> n < 100, n -> n + 4)
         .forEach(System.out::println);
```

Предикат, передаваемый в метод `iterate` в качестве второго аргумента, указывает, до какого момента должно продолжаться выполнение цикла. И хотя вам может показаться, что для достижения того же результата можно воспользоваться операцией `filter`:

```
IntStream.iterate(0, n -> n + 4)
         .filter(n -> n < 100)
         .forEach(System.out::println);
```

это не так. На самом деле выполнение кода не прекратится вовсе! Дело в том, что на этапе фильтрации неизвестно, что числа будут продолжать расти, так что их фильтрация продолжается бесконечно! Решить эту проблему можно с помощью операции `takeWhile` для использования сокращенной схемы вычисления потока данных:

```
IntStream.iterate(0, n -> n + 4)
         .takeWhile(n -> n < 100)
         .forEach(System.out::println);
```

Но нужно признать, что вариант метода `iterate` с предикатом более лаконичен!

Метод generate

Подобно методу `iterate`, метод `generate` дает возможность генерировать бесконечный поток значений, вычисляемых по требованию. Но метод `generate` не применяет функцию последовательно к каждому сгенерированному значению. Для генерации новых значений он получает в качестве параметра лямбда-выражение типа `Supplier<T>`. Рассмотрим пример его использования:

```
Stream.generate(Math::random)
      .limit(5)
      .forEach(System.out::println);
```

Этот код генерирует поток из пяти случайных чисел с двойной точностью в диапазоне от 0 до 1. Вот пример результатов его выполнения:

```
0.9410810294106129
0.6586270755634592
0.9592859117266873
0.13743396659487006
0.3942776037651241
```

В качестве генератора новых значений применяется статический метод `Math.random`. Опять-таки мы ограничиваем размер потока явным образом с помощью метода `limit`; в противном случае поток оказался бы неограниченным!

Наверное, вам интересно, можно ли сделать еще что-нибудь полезное с помощью метода `generate`. Мы использовали объект-поставщик (ссылку на метод для `Math.random`) без сохранения состояния: он нигде не фиксировал никаких значений, которые можно было бы задействовать при дальнейших вычислениях. Но объект-поставщик не обязан быть без сохранения состояния. Можно создать объект-поставщик, сохраняющий состояние, с возможностью его модификации и использования при генерации следующего значения потока данных.

В качестве примера покажем вам создание последовательности Фибоначчи из контрольного задания 5.4 с помощью метода `generate`, чтобы вы сравнили его с вариантом, в котором применялся метод `iterate`! Но важно отметить, что использовать объект-поставщик с сохранением состояния в параллельном коде небезопасно. Для полноты изложения в конце главы приведен класс `IntSupplier` с сохранением состояния для последовательности Фибоначчи, но в целом желательно избегать подобного подхода! Мы обсудим проблему операций с побочными эффектами и параллельные потоки данных в главе 7.

Воспользуемся интерфейсом `IntSupplier`, чтобы продемонстрировать, как можно избежать операций упаковки. Метод `generate` интерфейса `IntStream` принимает на входе объект `IntSupplier` вместо `Supplier<T>`. Например, сгенерировать бесконечный поток значений можно следующим образом:

```
IntStream ones = IntStream.generate(() -> 1);
```

Вы уже видели в главе 3, что с помощью лямбда-выражений можно создавать экземпляры функциональных интерфейсов путем передачи реализации метода непосредственно на месте. Можно также передать явный объект, как показано ниже,

реализовав описанный в интерфейсе `IntSupplier` метод `getAsInt` (хотя нам это кажется необоснованно громоздким):

```
IntStream twos = IntStream.generate(new IntSupplier(){
            public int getAsInt(){
                return 2;
            }
        });
```

Метод `generate` использует указанный объект-поставщик и многократно вызывает метод `getAsInt`, который всегда возвращает `2`. Но отличие между используемым здесь анонимным классом и лямбда-выражением состоит в том, что состояние в анонимном классе может задаваться посредством полей, которые способен модифицировать метод `getAsInt`. Это пример побочного эффекта. У всех встречавшихся вам до сих пор лямбда-выражений побочные эффекты отсутствовали; они не меняли состояния.

Вернемся к нашему примеру с числами Фибоначчи. Нам нужно создать объект `IntSupplier`, способный хранить в своем состоянии предыдущее значение ряда чисел, на основе которого метод `getAsInt` будет вычислять следующий элемент. Кроме того, он может обновлять состояние `IntSupplier` для использования при очередном запуске. В следующем коде показано, как создать объект `IntSupplier`, который бы при запуске возвращал очередное число Фибоначчи:

```
IntSupplier fib = new IntSupplier(){
    private int previous = 0;
    private int current = 1;
    public int getAsInt(){
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return oldPrevious;
    }
};
IntStream.generate(fib).limit(10).forEach(System.out::println);
```

Этот код создает экземпляр `IntSupplier` с *изменяемым* состоянием, который отслеживает предыдущее и текущее числа Фибоначчи в двух переменных экземпляра. Метод `getAsInt` при вызове меняет состояние этого объекта так, что при каждом вызове генерируются новые значения. А вот подход с использованием метода `iterate`, напротив, носил чисто *неизменяемый* характер: существующее состояние не менялось, а на каждой итерации создавались новые наборы чисел. Из главы 7 вы узнаете, что следует всегда предпочитать *неизменяемый подход*, чтобы получать правильные результаты при параллельной обработке потока данных.

Отметим, что, поскольку речь идет о потоке данных бесконечного размера, необходимо ограничивать его размер явным образом с помощью операции `limit`; в противном случае завершающая операция (здесь `forEach`) будет выполняться бесконечно. Аналогично нельзя сортировать бесконечный поток данных или выполнять его свертку, поскольку для этого нужно обработать все его элементы, что для потока с бесконечным количеством элементов займет бесконечное время!

Резюме

Это была длинная, но полезная глава! Теперь вы можете эффективнее обрабатывать коллекции. В самом деле с помощью потоков данных можно более лаконично выражать сложные запросы обработки данных. Кроме того, они допускают прозрачное распараллеливание операций.

- С помощью Stream API можно выражать сложные запросы обработки данных. Распространенные потоковые операции перечислены в табл. 5.1.
- С помощью методов `filter`, `distinct`, `takeWhile` (Java 9), `dropWhile` (Java 9), `skip` и `limit` можно фильтровать потоки данных и получать их срезы.
- Методы `takeWhile`, `dropWhile` работают эффективнее, чем `filter`, в случае отсортированного источника данных.
- С помощью методов `map` и `flatMap` можно извлекать элементы потоков данных и преобразовывать их.
- Используя методы `findFirst` и `findAny`, можно находить элементы потока данных. С помощью методов `allMatch`, `noneMatch` и `anyMatch` можно сопоставлять элементы потока с заданным предикатом.
- Эти методы реализуют сокращенную схему вычислений: вычисление прекращается сразу по обнаружении искомого; обрабатывать весь поток данных не требуется.
- С помощью метода `reduce` можно последовательно группировать все элементы потока для получения единого результата, например для вычисления суммы или максимума значений элементов потока данных.
- Часть операций, например `filter` и `map`, не сохраняют состояние. А некоторые, например `reduce`, для вычисления значения сохраняют состояние. Отдельные операции, такие как `sorted` и `distinct`, также сохраняют состояние, поскольку им необходимо буферизовать все элементы потока, чтобы вернуть новый поток. Подобные операции называются операциями *с сохранением состояния*.
- Существует три специализированные версии потоков для простых типов данных: интерфейсы `IntStream`, `DoubleStream` и `LongStream`. Их операции также специализированы соответствующим образом.
- Потоки данных можно создавать на основе не только коллекций, но и значений, массивов, файлов и с помощью конкретных методов, например `iterate` и `generate`.
- У бесконечного потока данных — бесконечное количество элементов (например, все возможные строки). Это возможно благодаря тому, что элементы в подобных потоках генерируются *по требованию*. С помощью таких методов, как `limit`, можно получить из бесконечного потока данных конечный.

6 Сбор данных с помощью потоков

В этой главе

- Создание и применение коллекторов с помощью класса `Collectors`.
- Свертка потоков данных в единое значение.
- Суммирование как частный случай свертки.
- Создаем свои собственные коллекторы.

Из предыдущей главы вы узнали, что потоки данных дают возможность обрабатывать коллекции посредством операций, схожих с операциями баз данных. Потоки данных Java 8 можно рассматривать как своеобразные отложенные итераторы наборов данных. Они поддерживают два типа операций: промежуточные (например, `filter` или `map`) и завершающие (например, `count`, `findFirst`, `forEach` и `reduce`). Промежуточные операции можно объединять в цепочку для преобразования одного потока данных в другой. Такие операции не потребляют данные из потока, их задача состоит в формировании конвейера потоков данных. Завершающие операции же, напротив, *как раз* потребляют данные из потока для того, чтобы вычислить конечный результат (например, вернуть максимальный элемент потока). Зачастую благодаря им можно сократить время вычислений за счет оптимизации конвейера потока данных.

Мы уже использовали завершающую операцию `collect` для потоков в главах 4 и 5, но применяли ее в основном для объединения всех элементов потока данных в объект `List`. Из этой главы вы узнаете, что `collect` представляет собой операцию свертки, как и `reduce`, и принимает в качестве аргумента различные стратегии группировки элементов потока в итоговый результат. Эти стратегии описываются с помощью нового интерфейса `Collector`, так что не путайте `Collection`, `Collector` и `collect`!

Вот несколько примеров запросов, которые можно реализовать с помощью операции `collect` и объектов-коллекторов.

- Группировка списка транзакций по валюте с целью получения суммы значений всех транзакций, произведенных в заданной валюте (возвращается `Map<Currency, Integer>`).
- Секционирование списка транзакций на две группы: транзакции на значительную сумму и нет (возвращается `Map<Boolean, List<Transaction>>`).
- Создание многоуровневых группировок, например группировки транзакций по городам с последующей разбивкой на категории в зависимости от суммы транзакции (возвращается `Map<String, Map<Boolean, List<Transaction>>>`).

Звучит захватывающе? Отлично. Начнем с примера, использующего возможности коллекторов. Представьте себе, что у вас есть список транзакций, который вы хотели бы сгруппировать по валюте. До Java 8 реализовать даже такой простой сценарий было непросто, как показано в листинге 6.1.

Листинг 6.1. Группировка транзакций по валюте, императивный подход

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
                                                    new HashMap<>();
for (Transaction transaction : transactions) {
    Currency currency = transaction.getCurrency();
    List<Transaction> transactionsForCurrency =
                                        transactionsByCurrencies.get(currency);
    if (transactionsForCurrency == null) {
        transactionsForCurrency = new ArrayList<>();
        transactionsByCurrencies
                                .put(currency, transactionsForCurrency);
    }
    transactionsForCurrency.add(transaction);
}
```

Цикл по списку транзакций

Создаем ассоциативный массив для накопления в нем сгруппированных транзакций

Извлекаем информацию о валюте транзакции

Создаем запись в ассоциативном массиве для текущей валюты, если ее еще нет

Добавляем только что обработанную транзакцию в список транзакций с той же валютой

Если вы опытный Java-разработчик, то, наверное, с легкостью пишете подобный код, хотя и понимаете, что для такой простой задачи он слишком длинный. Но хуже всего, что читать его еще сложнее, чем писать! Цель кода с первого взгляда неочевидна, хотя ее можно выразить простой фразой на русском языке: «Сгруппировать список транзакций по валюте». Как вы узнаете из этой главы, для решения такой задачи достаточно одного оператора благодаря использованию в методе `collect` обобщенного параметра `Collector` вместо частного случая `toList`, как в предыдущей главе:

```
Map<Currency, List<Transaction>> transactionsByCurrencies =
     transactions.stream().collect(groupingBy(Transaction::getCurrency));
```

Весьма обескураживающее сравнение, не правда ли?

6.1. Главное о коллекторах

Предыдущий пример ясно продемонстрировал одно из главных преимуществ программирования в функциональном стиле, в отличие от императивного подхода: достаточно сформулировать, *что* нужно получить, а не *какие* именно шаги необходимо выполнить для этого. В предыдущем примере в метод `collect` в качестве аргумента передавалась реализация интерфейса `Collector`, описывавшая, как получить сводный показатель элементов потока. В предыдущей главе `toList` означал «создать список всех элементов, взятых по очереди». В этом же примере `groupingBy` означает «создать `Map`, где ключами являются (валютные) корзины, а значениями — список элементов этих корзин».

Различие между императивной и функциональной версиями этого примера становится еще более явным при многоуровневых группировках: в этом случае императивный код очень быстро становится невозможно читать, сопровождать и модифицировать из-за множества глубоко вложенных циклов и условных операторов. При этом добавление дополнительных коллекторов в версию в функциональном стиле, как вы узнаете из раздела 6.3, не представляет трудностей.

6.1.1. Коллекторы как продвинутые средства свертки

Упомянутое выше наблюдение иллюстрирует еще одно типичное преимущество хорошо спроектированного функционального API: повышенные возможности сочетаемости компонентов и переиспользуемости. Коллекторы чрезвычайно удобны, поскольку с их помощью можно лаконично и гибко задавать критерии, на основе которых метод `collect` формирует итоговую коллекцию. Точнее говоря, вызов метода `collect` для потока данных запускает операцию свертки (параметризуемую объектом `Collector`) элементов указанного потока. Эта *операция свертки*, показанная на рис. 6.1, неявно выполняет вместо вас весь код из императивного листинга 6.1. Она обходит все элементы потока данных, а `Collector` их обрабатывает.

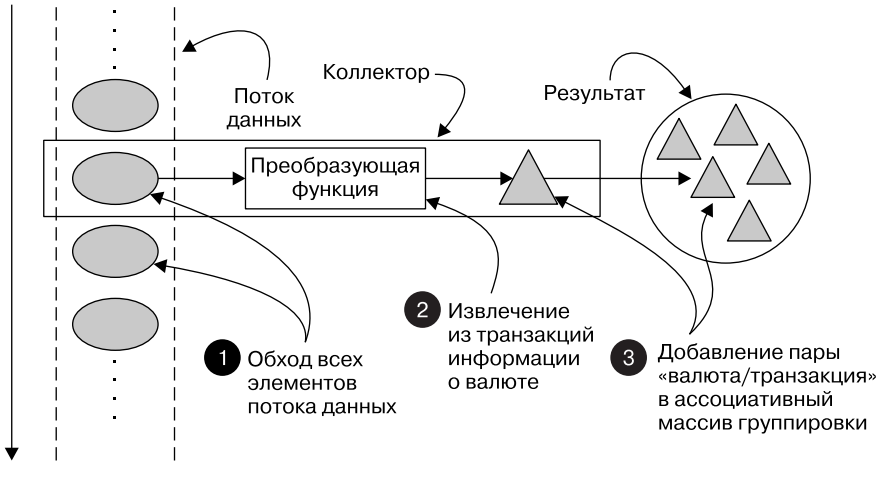


Рис. 6.1. Процесс свертки: группировка транзакций по валюте

Обычно `Collector` применяет к элементам преобразующую функцию. Довольно часто речь идет о тождественном преобразовании, не влияющем на данные (например, как в случае с преобразованием `toList`). Затем функция накапливает результаты в выводимой в конце структуре данных. Например, в показанном ранее примере с группировкой транзакций преобразующая функция извлекает из транзакций информацию о валюте, а далее сами транзакции накапливаются в итоговом ассоциативном массиве с валютой в качестве ключа.

Реализация методов интерфейса `Collector` определяет, как будет производиться над потоком операция свертки, например такая, как в примере с валютой. Мы обсудим создание специализированных коллекторов в разделах 6.5 и 6.6. Отметим, однако, что вспомогательный класс `Collectors` предоставляет множество статических фабричных методов для удобного создания готовых к применению экземпляров наиболее часто встречающихся коллекторов. Самый простой и чаще всего используемый коллектор — статический метод `toList`, собирающий все элементы потока данных в объект типа `List`:

```
List<Transaction> transactions =
    transactionStream.collect(Collectors.toList());
```

6.1.2. Встроенные коллекторы

Далее в главе мы в основном будем исследовать возможности встроенных коллекторов, создаваемых на основе предоставляемых классом `Collectors` фабричных методов (таких как `groupingBy`). Основные функциональные возможности, предоставляемые ими, включают следующее:

- свертку и сведение элементов потока данных в единое значение;
- группировку элементов;
- секционирование элементов.

Мы начнем с коллекторов, предназначенных для свертки и вычисления сводных показателей потоков данных. Они удобны во множестве сценариев использования, например при вычислении общей денежной суммы всего списка транзакций в предыдущем примере.

Далее мы покажем вам, как группировать элементы потока данных, обобщив предыдущий пример на случай нескольких уровней группировки и сочетания коллекторов с целью применения дальнейших операций свертки к каждой из получившихся подгрупп. Мы также расскажем про *секционирование* (partitioning) — частный случай группировки с предикатом (возвращающей булево значение функции с одним аргументом) в качестве группирующей функции.

В конце раздела 6.4 вы найдете сводную таблицу всех встроенных коллекторов, которые мы будем обсуждать в главе. Наконец, из раздела 6.5 вы узнаете больше об интерфейсе `Collector`, прежде чем мы расскажем вам (в разделе 6.6), как создать свои собственные пользовательские коллекторы для сценариев использования, не охваченных фабричными методами класса `Collectors`.

6.2. Свертка и вычисление сводных показателей

Для иллюстрации многообразия возможных экземпляров коллекторов, которые можно создать с помощью фабричного класса `Collectors`, мы воспользуемся предметной областью, обсуждавшейся в предыдущей главе: меню, состоящим из списка вкуснейших блюд!

Как вы уже знаете, коллекторы (параметры метода `collect` объекта `Stream`) обычно используются там, где нужно преобразовать элементы потока данных в коллекцию. Но, вообще говоря, их можно задействовать при любой группировке всех элементов потока данных в единый результат. Тип этого результата может быть любым: от отдельного целочисленного значения (скажем, соответствующего сумме калорий всего меню) до сложнейшего многоуровневого ассоциативного массива (соответствующего какой-то древовидной структуре). Мы рассмотрим оба этих типа результатов: отдельные целочисленные значения — в подразделе 6.2.2, а многоуровневые группировки — в подразделе 6.3.1.

В качестве первого простого примера подсчитаем количество блюд в меню с помощью коллектора, возвращаемого фабричным методом `counting`:

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

Конечно, это можно сделать намного более простым способом:

```
long howManyDishes = menu.stream().count();
```

но коллектор `counting` может оказаться полезен в сочетании с другими коллекторами, как мы продемонстрируем далее.

В оставшейся части данной главы предполагается, что все статические фабричные методы класса `Collectors` уже импортированы с помощью оператора:

```
import static java.util.stream.Collectors.*;
```

так что можно писать `counting()` вместо `Collectors.counting()` и т. д.

Продолжим обсуждение простых встроенных коллекторов на примере вычисления максимального и минимального значений потока.

6.2.1. Поиск максимума и минимума в потоке данных

Пусть вам нужно найти самое калорийное блюдо в меню. Для вычисления максимального или минимального значения в потоке данных можно использовать два коллектора — `Collectors.maxBy` и `Collectors.minBy`. Они принимают в качестве аргумента объект `Comparator` для сравнения элементов потока данных. В данном случае мы создадим `Comparator` для сравнения блюд на основе их калорийности и передадим его `Collectors.maxBy`:

```
Comparator<Dish> dishCaloriesComparator =
    Comparator.comparingInt(Dish::getCalories);
Optional<Dish> mostCalorieDish =
    menu.stream()
        .collect(maxBy(dishCaloriesComparator));
```

Наверное, вам интересно, что здесь делает `Optional<Dish>`. Для ответа на этот вопрос стоит задуматься: «А что, если `menu` пусто?» Возвращать нечего! В Java 8 для

этих целей появился класс `Optional` — контейнер, который может содержать или не содержать значение. В данном случае он прекрасно отражает идею отсутствия возвращаемого блюда. Мы вкратце упоминали его в главе 5, при описании метода `findAny`. Можете пока о нем не задумываться; мы посвятим всю главу 11 изучению класса `Optional<T>` и имеющихся в нем операций.

Еще одна часто встречающаяся операция свертки, возвращающая одно значение, — операция суммирования числового поля объектов потока данных. Или, скажем, усреднение его значений. Подобные операции называются *операциями вычисления сводных показателей* (summarization). Посмотрим, как их можно выразить с помощью коллекторов.

6.2.2. Вычисление сводных показателей

В классе `Collectors` имеется специальный фабричный метод для суммирования: `Collectors.summingInt`. Он принимает на входе функцию, отображающую объект в суммируемое значение типа `int`, и возвращает коллектор, который при передаче обычному методу `collect` вычисляет нужные сводные показатели. Например, можно найти общее число калорий в меню с помощью следующего кода:

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

В этом случае процесс вычисления производится так, как показано на рис. 6.2. При проходе по потоку данных каждому из блюд ставится в соответствие количество калорий в нем и эти значения прибавляются к накопителю, начиная с начального значения (в данном случае оно равно `0`).

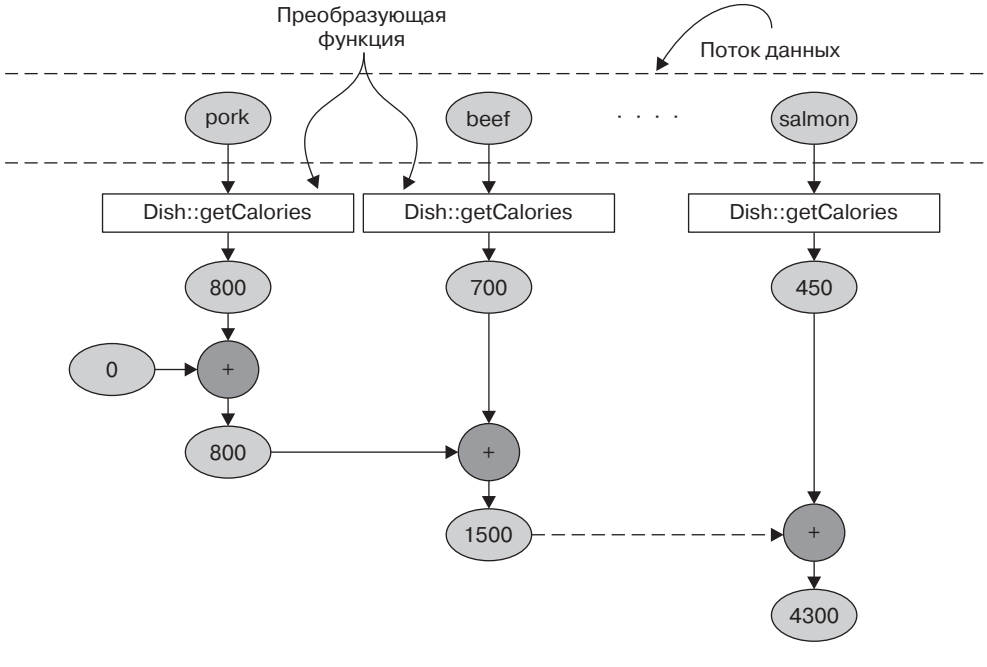


Рис. 6.2. Процесс агрегирования при использовании коллектора summingInt

Методы `Collectors.summingLong` и `Collectors.summingDouble` ведут себя точно так же и применяются в случае, когда тип суммируемого поля `long` или `double` соответственно.

Но вычисление сводных показателей — это отнюдь не только суммирование. Метод `Collectors.averagingInt` вместе с его аналогами `averagingLong` и `averagingDouble` позволяет вычислить среднее значение набора числовых значений:

```
double avgCalories =
    menu.stream().collect(averagingInt(Dish::getCalories));
```

До сих пор мы изучили, как с помощью коллекторов можно подсчитать количество элементов в потоке данных, найти максимальное и минимальное значения одного из числовых свойств этих элементов, а также вычислить их сумму и среднее значение. Довольно часто, однако, бывает нужно извлечь два таких результата или более, причем за одну операцию. В этом случае можно воспользоваться коллектором, возвращаемым фабричным методом `summarizingInt`. Например, можно подсчитать количество позиций меню и получить сумму, среднее значение, максимум и минимум числа содержащихся в блюдах калорий с помощью одной операции `summarizing`:

```
IntSummaryStatistics menuStatistics =
        menu.stream().collect(summarizingInt(Dish::getCalories));
```

Вся информация собирается этим коллектором в объект класса `IntSummaryStatistics`, из которого удобно получать результаты с помощью его методов-геттеров. В результате вывода в консоль объекта `menuStatistics` получаем следующее:

```
IntSummaryStatistics{count=9, sum=4300, min=120,
                     average=477.777778, max=800}
```

Как обычно, существуют соответствующие фабричные методы `summarizingLong` и `summarizingDouble` со связанными с ними типами `LongSummaryStatistics` и `DoubleSummaryStatistics`. Они применяются в том случае, когда тип поля, по которому производится вычисление сводных показателей, `long` или `double` соответственно.

6.2.3. Объединение строк

Возвращаемый фабричным методом `joining` коллектор выполняет конкатенацию всех строк, получаемых в результате вызовов метода `toString` для каждого из объектов потока данных. Это значит, что с помощью следующего кода можно склеить названия всех блюд из меню:

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

Обратите внимание, что внутри метода `joining` используется `StringBuilder` для слияния сгенерированных строк в одну. Отметим также, что если в классе `Dish` есть метод `toString`, возвращающий название блюда, то можно получить тот же результат без необходимости вызова `map(Dish::getName)`:

```
String shortMenu = menu.stream().collect(joining());
```

В результате обоих операторов получается довольно трудно читаемое строковое значение:

```
porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

К счастью, фабричный метод `joining` перегружен, и один из его перегруженных вариантов принимает в качестве параметра строку — разделитель соседних элементов, так что для получения разделенного запятыми списка названий блюд достаточно выполнить:

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

который сгенерирует следующее:

```
pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

До сих пор мы изучали различные коллекторы для свертки потока данных в одно значение. В следующем разделе мы покажем, что все подобные процессы свертки представляют собой частные случаи более общего коллектора свертки, возвращаемого фабричным методом `Collectors.reducing`.

6.2.4. Обобщение вычисления сводных показателей с помощью свертки

Все обсуждавшиеся до сих пор коллекторы, по существу, представляют собой лишь удобные специализированные варианты процесса свертки, задаваемого с помощью фабричного метода `reducing`. Фабричный метод `Collectors.reducing` — обобщение всех ранее упоминавшихся. Можно сказать, что обсуждавшиеся ранее частные случаи существуют лишь для удобства программистов (но не забывайте, что удобство для программистов и удобочитаемость кода чрезвычайно важны). Например, вычислить суммарное количество калорий в меню можно с помощью коллектора, созданного на основе метода `reducing`:

```
int totalCalories = menu.stream().collect(reducing(
                                    0, Dish::getCalories, (i, j) -> i + j));
```

Он принимает три аргумента.

- Первый аргумент — начальное значение для операции свертки, а равно и значение, возвращаемое при отсутствии элементов в потоке, так что в случае суммирования чисел явно лучше всего подходит `0`.
- Второй аргумент — та же функция, с помощью которой мы в подразделе 6.2.2 преобразовывали блюдо в значение типа `int`, соответствующее количеству содержащихся в нем калорий.
- Третий аргумент — `BinaryOperator`, который агрегирует два элемента в одно значение того же типа. В данном случае он суммирует два значения типа `int`.

Аналогичным образом можно найти самое калорийное блюдо с помощью одноаргументной версии метода `reducing`:

```
Optional<Dish> mostCalorieDish =
    menu.stream().collect(reducing(
        (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2));
```

Созданный с помощью одноаргументной версии фабричного метода `reducing` коллектор можно рассматривать как частный случай версии с тремя аргументами, с первым элементом потока данных в качестве начального значения и *тождественной функцией* (которая возвращает аргумент в неизменном виде) в качестве функции преобразования. Это также означает, что при передаче методу `collect` пустого потока у коллектора `reducing` с одним аргументом не окажется никакого начального значения и поэтому, как мы объясняли в подразделе 6.2.1, он вернет объект `Optional<Dish>`.

Методы collect и reduce

Мы много обсуждали вопрос свертки в этой и предыдущей главах. Наверное, вам интересно, в чем состоят различия методов `collect` и `reduce`, поскольку часто можно получить одни и те же результаты с помощью любого из них. Например, результаты работы коллектора `toList` можно получить и с помощью метода `reduce`:

```
Stream<Integer> stream = Arrays.asList(1, 2, 3, 4, 5, 6).stream();
List<Integer> numbers = stream.reduce(
                              new ArrayList<Integer>(),
                              (List<Integer> l, Integer e) -> {
                                        l.add(e);
                                        return l; },
                              (List<Integer> l1, List<Integer> l2) -> {
                                        l1.addAll(l2);
                                        return l1; });
```

У этого решения есть два недостатка: семантический и практический. Семантический заключается в том, что метод `reduce` предназначен для группировки двух значений в одно новое, то есть неизменяемой свертки. Напротив, метод `collect` накапливает требуемый результат путем изменения контейнера. Это значит, что в предыдущем фрагменте кода мы неправильно использовали метод `reduce`, поскольку он менял на месте используемый в качестве накопителя объект `List`. Как вы увидите в следующей главе, применение метода `reduce` с неправильной семантикой может привести и к чисто практической проблеме: такой процесс свертки не способен функционировать в параллельном режиме, поскольку параллельная модификация одной и той же структуры данных, используемой в нескольких потоках выполнения, может привести к порче самого объекта `List`. В этом случае, если вы хотите уверенности в типобезопасности, вам придется каждый раз выделять память под новый объект `List`, что отрицательно повлияет на производительность. Именно поэтому, главным образом, метод `collect` и полезен для свертки (на основе изменяемого контейнера, но, что очень важно, подходящим для параллельного выполнения образом), как вы узнаете далее в этой главе.

Гибкость фреймворка коллекций: выполнение одних операций разными способами

Еще более упростить предыдущий пример суммирования с коллектором `reducing` можно, воспользовавшись ссылкой на метод `sum` класса `Integer` вместо применяемого для задания той же операции лямбда-выражения. В результате получаем следующее:

```
int totalCalories = menu.stream().collect(reducing(0,
                                  Dish::getCalories,
                                  Integer::sum));
```

Начальное значение

Агрегирующая функция

Преобразующая функция

Схема работы этой операции свертки показана на рис. 6.3, где накопитель (инициализированный начальным значением) в цикле объединяется посредством агрегирующей функции с результатом применения преобразующей функции к каждому из элементов потока.

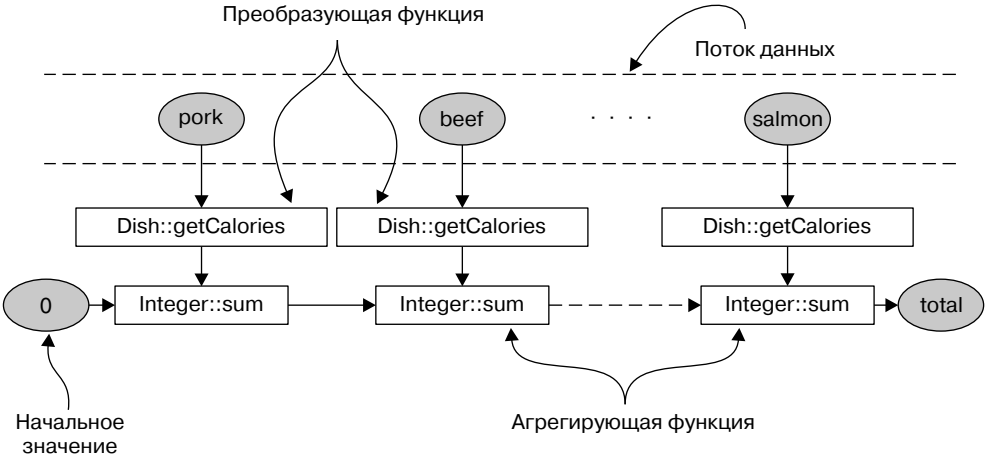


Рис. 6.3. Вычисление суммарного числа калорий в меню с использованием процесса свертки

Коллектор `counting`, упоминавшийся в начале раздела 6.2, в действительности реализуется аналогично с помощью фабричного метода `reducing` с тремя аргументами. Он преобразует каждый из элементов потока данных в объект типа `Long` со значением `1`, после чего суммирует их. Он реализован следующим образом:

```
public static <T> Collector<T, ?, Long> counting() {
    return reducing(0L, e -> 1L, Long::sum);
}
```

Джокерный символ обобщенных классов (?)

Вероятно, вы заметили в предыдущем фрагменте кода джокерный символ (`?`) в качестве второго обобщенного типа в сигнатуре возвращаемого фабричным методом `counting` коллектора. Наверное, вы уже знакомы с такой нотацией, особенно если часто используете фреймворк коллекций Java. Но в данном случае он означает лишь то, что тип накопителя коллектора неизвестен или (что эквивалентно) что сам накопитель может быть любого типа. Мы привели его здесь, чтобы в точности отразить сигнатуру метода в том виде, как она описана в классе `Collectors`, но в оставшейся части данной главы для простоты мы будем избегать подобной нотации.

Мы уже видели в главе 5, что ту же операцию можно выполнить и без коллектора — путем отображения потока блюд в число калорий каждого блюда с последующей сверткой получившегося потока данных, применяя ту же ссылку на метод:

```
int totalCalories =
    menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
```

Отметим, что вызов для потока данных операции `reduce(Integer::sum)`, как и любой другой одноаргументной операции, возвращает не `int`, а `Optional<Integer>` для безопасной свертки пустого потока данных. В этом случае мы извлекаем значение из объекта `Optional` с помощью его метода `get`. Обратите внимание, что мы можем без опаски использовать здесь метод `get` лишь потому, что уверены — поток не пуст. В целом, как вы узнаете из главы 10, безопаснее было бы распаковать содержащееся в `Optional` значение с помощью метода, который позволял бы задавать значение по умолчанию, например `orElse` и `orElseGet`. Наконец, можно получить тот же результат с помощью еще более лаконичного кода, отобразив поток данных в `IntStream` и вызвав для него метод `sum`:

```
int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();
```

Выбор оптимального решения для конкретной ситуации

И снова мы видим, как функциональное программирование вообще (и добавленные во фреймворк коллекций Java 8 новые API, основанные на функциональном стиле программирования, в частности) позволяют решать одну задачу различными способами. Этот пример также демонстрирует, что коллекторы, хотя они и сложнее в использовании, чем непосредственно методы интерфейса `Stream`, обеспечивают более высокий уровень абстракции и обобщения, их проще переиспользовать и адаптировать к конкретной задаче.

Мы рекомендуем вам разобрать все возможные решения текущей задачи, но выбирать наиболее специализированное и в то же время достаточно общее. Зачастую оно окажется лучшим в смысле как удобочитаемости, так и производительности. Например, для вычисления суммарного числа калорий в меню лучше выбрать последнее из представленных решений (с помощью `IntStream`) — наиболее лаконичное и, вероятно, наиболее удобочитаемое. В то же время оно и оптимальное с точки зрения производительности, ведь благодаря `IntStream` оказываются не нужны все операции *автораспаковки* (неявного преобразования из `Integer` в `int`).

А теперь проверьте, насколько хорошо вы понимаете применение `reducing` в качестве обобщения других коллекторов, и выполните упражнение из контрольного задания 6.1.

Контрольное задание 6.1. Объединение строк с помощью свертки

Какие из нижеприведенных операторов (в которых используется коллектор `reducing`) подойдут в качестве замены коллектора `joining` (из подраздела 6.2.3)?

```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

1.
```
String shortMenu = menu.stream().map(Dish::getName)
    .collect( reducing( (s1, s2) -> s1 + s2 ) ).get();
```

2. `String shortMenu = menu.stream().`
   `collect( reducing( (d1, d2) -> d1.getName() + d2.getName() )).get();`
3. `String shortMenu = menu.stream().`
   `collect( reducing( "", Dish::getName, (s1, s2) -> s1 + s2 ) );`

Ответ: операторы 1 и 3 написаны правильно, а оператор 2 вообще не скомпилируется.

1. Этот оператор преобразует каждое из блюд в его название точно так же, как и наш исходный оператор с коллектором `joining`, после чего производит свертку полученного потока строк с объектом `String` в качестве накопителя, к которому поочередно добавляются в конец строки названия блюд.
2. Этот код не скомпилируется, поскольку `reducing` принимает на входе один аргумент, а именно `BinaryOperator<T>` — частный случай `BiFunction<T,T,T>`. Это значит, что ему требуется функция, принимающая два аргумента и возвращающая значение того же типа, а приведенное в данном коде лямбда-выражение принимает в качестве аргументов два блюда и возвращает строку.
3. Этот код начинает процесс свертки с пустой строки в качестве накопителя, после чего обходит поток блюд, преобразуя каждое из блюд в название и добавляя это название в конец строки-накопителя. Отметим, что можно было бы использовать метод `reducing` без трех аргументов, который вернул бы `Optional`, но в случае пустого потока данных лучше вернуть более осмысленное значение — пустую строку (начальное значение накопителя).

Хотя операторы 1 и 3 можно применять в качестве замены коллектора `joining`, здесь они использовались лишь для демонстрации того, что `reducing` можно (по крайней мере теоретически) рассматривать как обобщение всех прочих обсуждавшихся в этой главе коллекторов. Тем не менее на практике мы рекомендуем всегда использовать коллектор `joining` для повышения удобочитаемости и производительности.

6.3. Группировка

В базах данных часто встречается операция группировки элементов в некий набор на основе одного или нескольких свойств. Как вы видели в предыдущем примере группировки транзакций по валюте, при реализации в императивном стиле код такой операции может оказаться громоздким, неуклюжим и подверженным ошибкам. Но его легко можно выразить в виде одного понятного оператора, если переписать в функциональном стиле, поощряемом Java 8. Приведем еще один пример этой возможности. Допустим, вам нужно классифицировать блюда меню по типу: в одной группе — блюда с мясом, в другой — с рыбой, в третьей — все остальные. Можно с легкостью решить эту задачу с помощью коллектора, возвращаемого фабричным методом `Collectors.groupingBy`:

```
Map<Dish.Type, List<Dish>> dishesByType =
                        menu.stream().collect(groupingBy(Dish::getType));
```

В результате получаем следующий ассоциативный массив:

```
{FISH=[prawns, salmon], OTHER=[french fries, rice, season fruit, pizza],
 MEAT=[pork, beef, chicken]}
```

Здесь мы передаем в метод `groupingBy` экземпляр `Function` (в виде ссылки на метод), который извлекает соответствующий `Dish.Type` каждого из объектов `Dish` потока. Мы будем называть этот объект `Function` функцией *классификации*, поскольку его задача состоит в классификации элементов потока данных по различным группам. В результате этой операции группировки, показанной на рис. 6.4, мы получаем ассоциативный массив (`Map`) с возвращаемым функцией классификации значением в качестве ключа и списком всех соответствующих элементов потока в качестве значения. В примере с классификацией меню ключ — тип блюда, а значение — список всех блюд соответствующего типа.

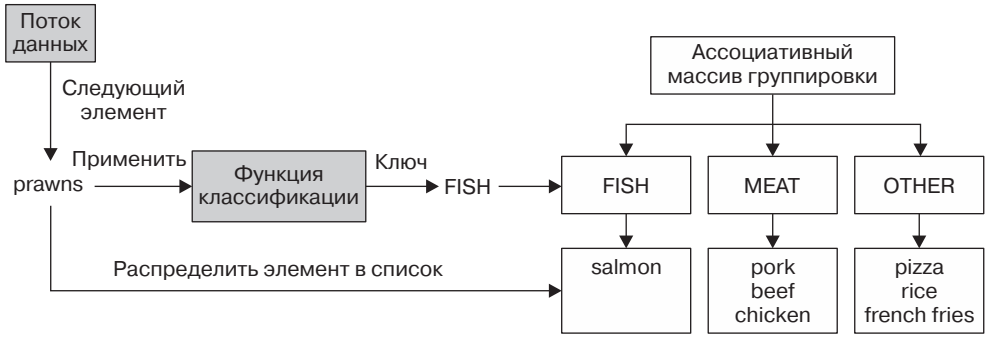


Рис. 6.4. Классификация элементов потока в процессе группировки

Но ссылка на метод не всегда подходит в качестве функции классификации, ведь иногда простого извлечения свойств объекта для классификации недостаточно. Например, вы захотели классифицировать как «диетические» (`DIET`) все блюда, содержащие не более 400 калорий, как «обычные» (`NORMAL`) — содержащие от 400 до 700 калорий и как «слишком жирные» (`FAT`) — содержащие более 700 калорий. Поскольку автор класса `Dish`, увы, не позаботился предоставить метод, реализующий подобную операцию, воспользоваться ссылкой на метод в данном случае не получится, но приведенную логику можно выразить с помощью лямбда-выражения:

```
public enum CaloricLevel { DIET, NORMAL, FAT }
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel = menu.stream().collect(
      groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT;
      } ));
```

Мы обсудили, как сгруппировать блюда в меню как по типу, так и по количеству калорий. Однако часто бывает нужно продолжить обработку результатов полученной группировки, и в следующем разделе мы покажем вам, как это сделать.

6.3.1. Дальнейшие операции со сгруппированными элементами

После группировки часто требуется произвести какие-либо действия над элементами каждой из получившихся групп. Допустим, вам нужно отфильтровать только высококалорийные блюда, скажем, содержащие более 500 калорий. Вероятно, вы

возразите, что в подобном случае можно воспользоваться предикатом фильтрации перед группировкой, вот так:

```
Map<Dish.Type, List<Dish>> caloricDishesByType =
                    menu.stream().filter(dish -> dish.getCalories() > 500)
                                 .collect(groupingBy(Dish::getType));
```

У этого решения, хотя и вполне работоспособного, есть иногда существенный недостаток. Если попытаться применить его к блюдам из нашего меню, мы получим следующий ассоциативный массив:

```
{OTHER=[french fries, pizza], MEAT=[pork, beef]}
```

Замечаете проблему? Поскольку ни одно из блюд категории `FISH` не удовлетворяет нашему предикату фильтрации, соответствующий ключ полностью исчезает из итогового ассоциативного массива. Для обхода этой проблемы в классе `Collectors` есть перегруженный фабричный метод `groupingBy`, принимающий второй аргумент типа `Collector` наряду с обычной функцией классификации. Таким образом, появляется возможность перенести предикат фильтрации внутрь этого второго коллектора, вот так:

```
Map<Dish.Type, List<Dish>> caloricDishesByType =
      menu.stream()
          .collect(groupingBy(Dish::getType,
                   filtering(dish -> dish.getCalories() > 500, toList())));
```

`filtering` — еще один статический фабричный метод класса `Collectors`, принимающий в качестве параметров предикат для фильтрации элементов в каждой из групп и еще один коллектор для перегруппировки профильтрованных элементов. Таким образом, в получившемся ассоциативном массиве останется запись для типа `FISH`, хотя и представляющая собой пустой список:

```
{OTHER=[french fries, pizza], MEAT=[pork, beef], FISH=[]}
```

Еще более распространенный удобный способ обработки сгруппированных элементов — преобразование с помощью функции отображения. Для этого аналогично коллектору `filtering` класс `Collectors` предоставляет еще один коллектор в виде метода `mapping`. Он принимает на входе функцию отображения и другой коллектор, предназначенный для группировки элементов, полученных в результате применения этой функции. С его помощью можно, например, преобразовать все `Dish` в группах в их названия:

```
Map<Dish.Type, List<String>> dishNamesByType =
      menu.stream()
          .collect(groupingBy(Dish::getType,
                   mapping(Dish::getName, toList())));
```

Отметим, что в этом случае каждая из групп получившегося ассоциативного массива представляет собой список строк, а не одно из блюд, как в предыдущих примерах. Можно также воспользоваться третьим коллектором в сочетании с методом `groupingBy` для схлопывания (`flatMap`) вместо простого отображения (`map`). Для наглядной демонстрации предположим, что с каждым блюдом списка связан ассоциативный массив пометок:

```
Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork", asList("greasy", "salty"));
```

```
dishTags.put("beef", asList("salty", "roasted"));
dishTags.put("chicken", asList("fried", "crisp"));
dishTags.put("french fries", asList("greasy", "fried"));
dishTags.put("rice", asList("light", "natural"));
dishTags.put("season fruit", asList("fresh", "natural"));
dishTags.put("pizza", asList("tasty", "salty"));
dishTags.put("prawns", asList("tasty", "roasted"));
dishTags.put("salmon", asList("delicious", "fresh"));
```

При необходимости можно легко извлечь эти пометки для каждой группы типа блюд с помощью коллектора `flatMapping`:

```
Map<Dish.Type, Set<String>> dishNamesByType =
   menu.stream()
      .collect(groupingBy(Dish::getType,
               flatMapping(dish -> dishTags.get( dish.getName() ).stream(),
                           toSet())));
```

Для каждого объекта `Dish` мы получаем `List` пометок. Поэтому аналогично изложенному в предыдущей главе нам нужно выполнить `flatMap` для схлопывания полученного двухуровневого списка в одноуровневый. Отметим также, что теперь мы получаем результаты схлопывания по группам в виде объектов `Set`, а не `List`, как раньше, чтобы избежать повтора тегов, относящихся к нескольким блюдам одного типа. Получившийся в результате ассоциативный массив выглядит следующим образом:

```
{MEAT=[salty, greasy, roasted, fried, crisp], FISH=[roasted, tasty, fresh,
    delicious], OTHER=[salty, greasy, natural, light, tasty, fresh, fried]}
```

До сих пор мы использовали только один критерий группировки блюд в меню, например, по их типу или числу калорий, но что, если нам нужно одновременно несколько критериев? Сила операции группировки — в эффективности агрегирования элементов. Посмотрим, как этого добиться.

6.3.2. Многоуровневая группировка

Двумя аргументами фабричного метода `Collectors.groupingBy`, с помощью которых в предыдущем разделе мы манипулировали элементами групп, можно воспользоваться и для группировки второго уровня. Для этого можно передать внешнему методу `groupingBy` второй внутренний `groupingBy`, задав в нем критерий второго уровня для классификации элементов потока, как показано в листинге 6.2.

Листинг 6.2. Многоуровневая группировка

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
menu.stream().collect(
      groupingBy(Dish::getType,                    ← Функция классификации первого уровня
          groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
          } )
      )
);
```

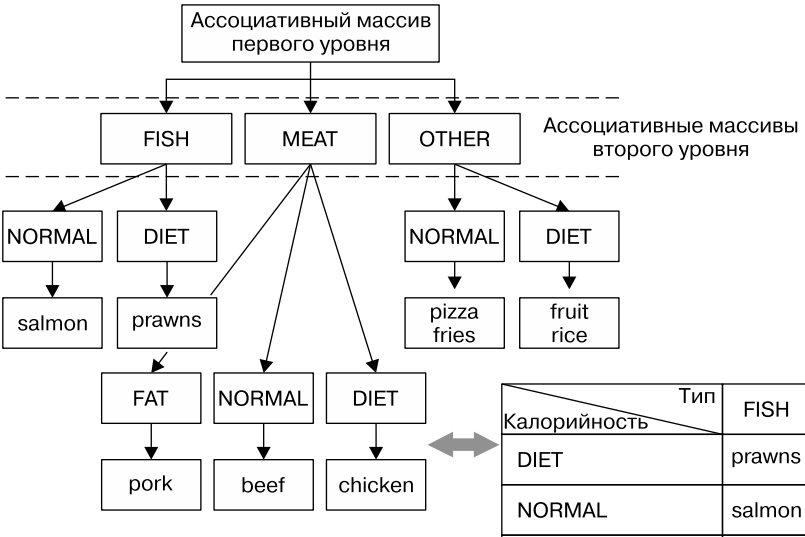
Функция классификации второго уровня

Результат этой двухуровневой группировки представляет собой следующий ассоциативный массив:

```
{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
 FISH={DIET=[prawns], NORMAL=[salmon]},
 OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}
```

Здесь ключами внешнего ассоциативного массива (объекта `Map`) являются значения, сгенерированные функцией классификации первого уровня: `MEAT`, `FISH` и `OTHER`. Значения этого ассоциативного массива также являются, в свою очередь, ассоциативными массивами, ключи которых — значения, сгенерированные функцией классификации второго уровня: `DIET`, `NORMAL` и `FAT`. Наконец, значения ассоциативных массивов второго уровня: списки (`List`) элементов потока (`salmon`, `pizza` и т. д.), для которых функции классификации первого и второго уровня возвращают соответствующие ключи первого и второго уровня. Подобную многоуровневую операцию группировки можно обобщить на произвольное число уровней, при этом результатом *n*-уровневой группировки будет *n*-уровневый ассоциативный массив, моделирующий *n*-уровневую древовидную структуру.

Как видно из рис. 6.5, который подчеркивает цель операции группировки — классификацию, эта структура эквивалентна *n*-мерной таблице.



Тип / Калорийность	FISH	MEAT	OTHER
DIET	prawns	chicken	fruit rice
NORMAL	salmon	beef	pizza fries
FAT		pork	

Рис. 6.5. Эквивалентность ассоциативного массива с n уровнями вложенности и n-мерной таблицы классификации

В целом удобно считать, что `groupingBy` работает с термином «корзина». Первый `groupingBy` создает корзины для всех ключей. А затем уже можно с помощью расположенного далее по конвейеру коллектора собрать элементы потока в каждой из корзин и т. д., вплоть до *n*-уровневой группировки!

6.3.3. Сбор данных в подгруппы

В предыдущем подразделе мы показали, что можно выполнить многоуровневую группировку за счет передачи во внешний коллектор `groupingBy` еще одного, дополнительного. Но в общем случае можно передавать в первый `groupingBy` коллектор произвольного типа, а не только еще один `groupingBy`. Например, можно подсчитать число блюд в меню для каждого из типов блюд, передав коллектор `counting` в качестве второго аргумента в коллектор `groupingBy`:

```
Map<Dish.Type, Long> typesCount = menu.stream().collect(
                    groupingBy(Dish::getType, counting()));
```

Результат представляет собой следующий ассоциативный массив:

```
{MEAT=3, FISH=2, OTHER=4}
```

Отметим также, что обычная операция `groupingBy(f)` с одним аргументом `f` — функцией классификации — фактически является сокращенной записью `groupingBy(f, toList())`.

В качестве еще одного примера мы можем переделать коллектор, уже применявшийся для поиска самого калорийного блюда в меню, и получить аналогичный результат, только с классификацией по *типу* блюда:

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =
    menu.stream()
        .collect(groupingBy(Dish::getType,
                            maxBy(comparingInt(Dish::getCalories))));
```

В результате этой группировки мы, конечно, получаем ассоциативный массив, ключами в котором служат имеющиеся типы блюд, а значениями — объекты `Optional<Dish>`, обертки для соответствующих наиболее калорийных (в пределах данного типа) блюд:

```
{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}
```

ПРИМЕЧАНИЕ

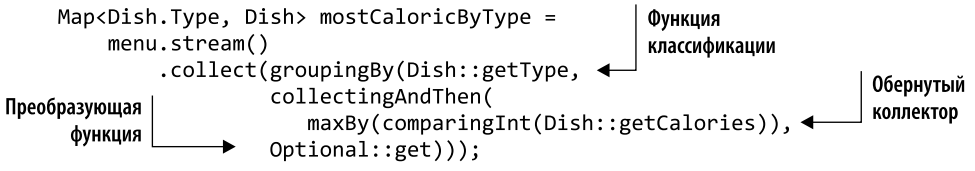
> Значениями в этом ассоциативном массиве служат объекты Optional, поскольку именно таков тип результатов коллектора, генерируемого фабричным методом maxBy. Однако фактически при отсутствии блюд данного типа в меню значение не будет равно Optional.empty(); в ассоциативном массиве вообще не окажется соответствующего ключа. Коллектор groupingBy добавляет новый ключ в итоговый ассоциативный массив (отложенным образом) только при первом обнаружении в потоке данных элемента, такого, что при применении к нему критерия группировки получается этот ключ. Иными словами, в данном случае адаптер Optional никакой пользы не несет, поскольку не отражает факт возможного отсутствия значения, а оказался здесь случайно, лишь потому, что коллектор свертки возвращает такой тип данных.

Преобразование результатов работы коллектора к другому типу

Поскольку обертывание в тип `Optional` всех значений ассоциативного массива, получившегося в результате последней операции группировки, в данном случае

никакой пользы не несет, имеет смысл избавиться от этих оберток. Для этого или в общем случае для преобразования возвращенного коллектором результата к другому типу можно воспользоваться коллектором, возвращаемым фабричным методом `Collectors.collectingAndThen`, как показано в листинге 6.3.

Листинг 6.3. Поиск самого калорийного блюда в каждой подгруппе



Этот фабричный метод принимает два аргумента — обертываемый коллектор и преобразующую функцию — и возвращает еще один коллектор. Данный дополнительный коллектор служит адаптером для предыдущего коллектора и отображает возвращаемое им значение с помощью преобразующей функции в качестве последнего шага операции `collect`. В данном случае обертывается созданный с помощью операции `maxBy` коллектор, а функция преобразования, `Optional::get`, извлекает содержащееся в возвращенном объекте `Optional` значение. Как мы уже упоминали, в данном случае это можно делать без опаски, поскольку коллектор `reducing` никогда не вернет `Optional.empty()`. В результате получаем следующий ассоциативный массив:

```
{FISH=salmon, OTHER=pizza, MEAT=pork}
```

Несколько вложенных коллекторов — достаточно распространенная практика, причем способ их взаимодействия далеко не всегда сразу очевиден. Рисунок 6.6 наглядно демонстрирует их взаимодействие.

Обратите внимание на следующее (в направлении с внешнего слоя внутрь).

- Коллекторы обозначены штриховыми линиями, `groupingBy` — внешний, он разбивает (группирует) поток блюд из меню на три подпотока, по различным типам блюд.
- Коллектор `groupingBy` служит адаптером для коллектора `collectingAndThen`, и каждый подпоток, получающийся в результате операции группировки, подвергается дальнейшей свертке этим вторым коллектором.
- Коллектор `collectingAndThen`, в свою очередь, служит оберткой для третьего коллектора, `maxBy`.
- Далее операция свертки подпотоков выполняется коллектором свертки, но включающий ее коллектор `collectingAndThen` применяет к ее результату преобразующую функцию `Optional::get`.
- Три преобразованных значения — наиболее калорийные блюда для каждого из типов блюд (полученные в результате выполнения вышеописанного процесса для каждого из трех подпотоков) — и будут значениями, соответствующими ключам классификации (типам блюд) в возвращаемом коллектором `groupingBy` ассоциативном массиве.

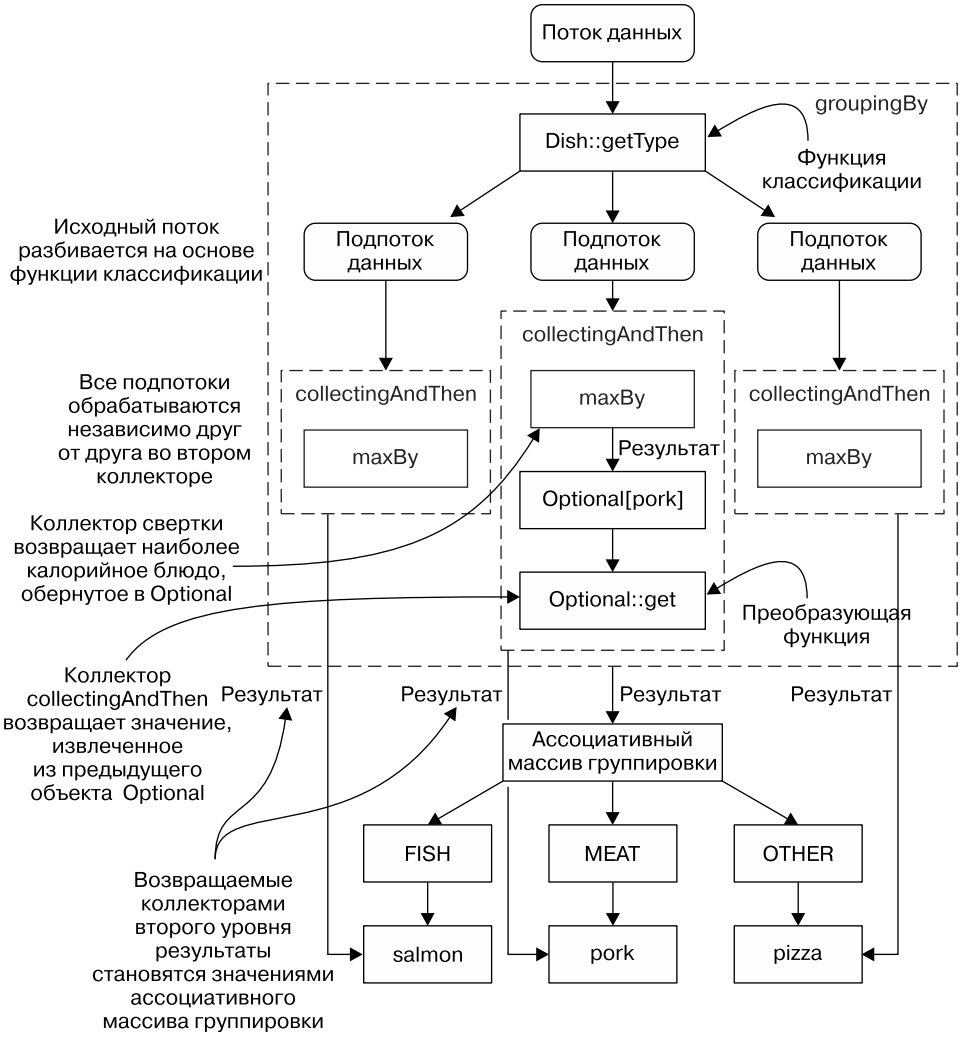


Рис. 6.6. Комбинируем воздействие нескольких коллекторов за счет вложения одного в другой

Другие примеры коллекторов, применяемых совместно с groupingBy

В общем случае коллектор, передаваемый в качестве второго аргумента в фабричный метод `groupingBy`, затем используется для последующей свертки всех элементов потока данных, отнесенных при классификации в одну группу. Например, можно переиспользовать коллектор, предназначенный для суммирования калорийности всех блюд меню, и получить аналогичные суммы, но уже для каждой из групп блюд:

```
Map<Dish.Type, Integer> totalCaloriesByType =
               menu.stream().collect(groupingBy(Dish::getType,
                        summingInt(Dish::getCalories)));
```

Кроме того, в сочетании с `groupingBy` часто применяется коллектор, генерируемый методом `mapping`. Этот метод принимает два аргумента: функцию преобразования элементов потока и еще один коллектор, служащий для накопления полученных в результате такого преобразования объектов. Он выступает в качестве адаптера между коллектором, принимающим объекты одного типа, и коллектором, работающим с объектами другого типа, за счет того, что перед попаданием в накопитель к каждому из входящих элементов применяется функция отображения. Рассмотрим практический пример использования этого коллектора. Допустим, мы хотим узнать, какие уровни калорийности присутствуют в меню для каждого из типов блюд. Для этого можно воспользоваться сочетанием коллекторов `groupingBy` и `mapping`, как показано ниже:

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
   groupingBy(Dish::getType, mapping(dish -> {
          if (dish.getCalories() <= 400) return CaloricLevel.DIET;
          else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
          else return CaloricLevel.FAT; },
    toSet() )));
```

В данном случае передаваемая в метод `mapping` преобразующая функция возвращает для каждого блюда его уровень калорийности, как вы уже видели ранее. Затем полученный поток объектов `CaloricLevel` передается коллектору `toSet()` (аналогичен `toList`, но накапливает элементы потока данных в объекте `Set` вместо `List`, чтобы исключить дубликаты). Как и в предыдущих примерах, затем с помощью этого коллектора `mapping` производится сбор элементов всех сгенерированных функцией группировки подпотоков, что позволяет получить конечный результат в виде следующего ассоциативного массива:

```
{OTHER=[DIET, NORMAL], MEAT=[DIET, NORMAL, FAT], FISH=[DIET, NORMAL]}
```

Теперь становится понятно, какие альтернативы у вас есть. Если вы на диете и не возражаете против рыбы, то легко можете найти подходящее для себя блюдо; если же вы голодны и хотите съесть что-то высококалорийное, то легко можете удовлетворить свой здоровый аппетит, выбрав что-то из мясного раздела меню. Отметим, что в предыдущем примере никакой уверенности в возвращаемом типе `Set` у нас не было. Для расширения контроля за этим можно воспользоваться методом `toCollection`. Например, можно потребовать, чтобы был возвращен объект типа `HashSet`, путем передачи ссылки на соответствующий конструктор:

```
Map<Dish.Type, Set<CaloricLevel>> caloricLevelsByType =
menu.stream().collect(
   groupingBy(Dish::getType, mapping(dish -> {
          if (dish.getCalories() <= 400) return CaloricLevel.DIET;
          else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
          else return CaloricLevel.FAT; },
    toCollection(HashSet::new) )));
```

6.4. Секционирование

Секционирование — частный случай группировки, при котором в качестве функции классификации используется предикат, называемый *секционирующей функцией* (partitioning function). Поскольку секционирующая функция возвращает булево значение, то тип ключа итогового ассоциативного массива группировки должен быть `Boolean`, а, следовательно, количество различных групп не может превышать двух — одна для `true` и одна для `false`. Например, если вы вегетарианец или позвали на обед друга-вегетарианца, то вам может понадобиться разделить блюда в меню на вегетарианские и невегетарианские:

```
Map<Boolean, List<Dish>> partitionedMenu =
             menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

Секционирующая функция

В результате получаем следующий ассоциативный массив:

```
{false=[pork, beef, chicken, prawns, salmon],
 true=[french fries, rice, season fruit, pizza]}
```

Таким образом, для выяснения всех вегетарианских блюд необходимо извлечь из этого ассоциативного массива значение, индексированное ключом `true`:

```
List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

Отметим, что получить тот же результат можно путем фильтрации списка, созданного из объекта `menu`, на основе того же предиката, что применялся для секционирования, с последующим сбором результата в еще один список:

```
List<Dish> vegetarianDishes =
              menu.stream().filter(Dish::isVegetarian).collect(toList());
```

6.4.1. Преимущества секционирования

Преимущество секционирования в том, что сохраняются оба списка элементов потока данных, применение к которым секционирующей функции дает `true` или `false`. В предыдущем примере получить список невегетарианских блюд можно путем извлечения значений для ключа `false` ассоциативного массива `partitionedMenu`, вместо того чтобы задействовать две отдельные операции фильтрации: одну с вышеупомянутым предикатом, а вторую — с его отрицанием. Также, как вы уже видели в случае группировки, у фабричного метода `partitioningBy` есть перегруженная версия, в которую можно передать второй коллектор, вот так:

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
menu.stream().collect(
        partitioningBy(Dish::isVegetarian,
                       groupingBy(Dish::getType)));
```

Секционирующая функция

Второй коллектор

В результате получаем двухуровневый ассоциативный массив:

```
{false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},
 true={OTHER=[french fries, rice, season fruit, pizza]}}
```

При этом группировка блюд по типу производится отдельно для каждого из двух подпотоков вегетарианских и невегетарианских блюд, полученных после секционирования. В результате этого мы получаем двухуровневый ассоциативный массив, аналогичный полученному при двухуровневой группировке в подразделе 6.3.1. В качестве еще одного примера можно повторно использовать вышеприведенный код и найти наиболее калорийные блюда среди вегетарианских и невегетарианских:

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
menu.stream().collect(
    partitioningBy(Dish::isVegetarian,
         collectingAndThen(maxBy(comparingInt(Dish::getCalories)),
                           Optional::get)));
```

В результате получаем:

```
{false=pork, true=pizza}
```

Мы начали этот раздел с утверждения, что секционирование можно рассматривать как частный случай группировки. Стоит также отметить, что возвращаемая `partitioningBy` реализация ассоциативного массива является более сжатой и эффективной, ведь она содержит только два ключа: `true` и `false`. На самом деле внутренняя реализация представляет собой специализированный ассоциативный массив с двумя полями. Аналогии между коллекторами `groupingBy` и `partitioningBy` на этом не заканчиваются: как вы увидите в следующем контрольном задании, можно производить многоуровневое секционирование аналогично многоуровневой группировке из подраздела 6.3.1.

Приведем еще один, последний пример использования коллектора `partitioningBy`: отложим в сторону модель данных меню и взглянем на чуть более сложную, но и более интересную задачу: секционирование чисел на простые и составные.

Контрольное задание 6.2. Использование коллектора partitioningBy

Как вы уже видели, коллектор `partitioningBy`, как и `groupingBy`, можно использовать совместно с другими коллекторами. В частности, для реализации многоуровневого секционирования его можно применять вместе со вторым коллектором `partitioningBy`. Каковы будут результаты следующих операций многоуровневого секционирования?

1.
```
menu.stream().collect(partitioningBy(Dish::isVegetarian,
                      partitioningBy(d -> d.getCalories() > 500)));
```
2.
```
menu.stream().collect(partitioningBy(Dish::isVegetarian,
                      partitioningBy(Dish::getType)));
```
3.
```
menu.stream().collect(partitioningBy(Dish::isVegetarian,
                      counting()));
```

Ответы

1. Это допустимый оператор многоуровневого секционирования, в результате которого возвращается следующий двухуровневый ассоциативный массив:

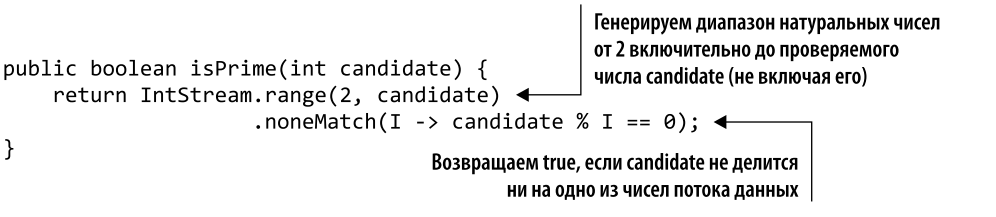
```
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},
    true={false=[rice, season fruit], true=[french fries, pizza]}}
```

2. Этот код не скомпилируется, поскольку коллектор `partitioningBy` требует в качестве параметра предикат, то есть функцию, возвращающую булево значение. А ссылку на метод `Dish::getType` нельзя использовать как предикат.
3. Этот оператор подсчитывает количество элементов в каждой из секций, в результате возвращается следующий ассоциативный массив:

   ```
   {false=5, true=4}
   ```

6.4.2. Секционирование чисел на простые и составные

Предположим, вы хотите написать метод, принимающий в качестве аргумента число *n* типа `int` и разбивающий первые *n* натуральных чисел на простые и составные. Но сначала нужно написать предикат для проверки, является ли заданное число простым:




```
public boolean isPrime(int candidate) {
    return IntStream.range(2, candidate)
                    .noneMatch(I -> candidate % I == 0);
}
```

Немного оптимизируем код: проверяем только множители, не превышающие квадратного корня из `candidate`:

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

Основная часть работы позади. Для секционирования первых *n* натуральных чисел на простые и составные достаточно создать поток данных, содержащий эти *n* чисел, и выполнить его свертку с помощью коллектора `partitioningBy` и только что написанного нами метода `isPrime` в качестве предиката:

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
                    .collect(
                      partitioningBy(candidate -> isPrime(candidate)));
}
```

Мы обсудили все коллекторы, которые только можно создать с помощью статических фабричных методов класса `Collectors`, и привели примеры их применения на практике. Перечислим их еще раз в табл. 6.1 вместе с типами, которые они возвращают при применении к `Stream<T>`, а также с примерами использования их на практике (для объекта `menuStream` типа `Stream<Dish>`).

Таблица 6.1. Основные статические фабричные методы класса Collectors

Фабричный метод	Возвращаемый тип	Назначение
toList	List<T>	Собирает все элементы потока данных в объект List
Пример использования: List<Dish> dishes = menuStream.collect(toList());		
toSet	Set<T>	Собирает все элементы потока данных в объект Set, исключая дубликаты
Пример использования: Set<Dish> dishes = menuStream.collect(toSet());		
toCollection	Collection<T>	Собирает все элементы потока данных в коллекцию, создаваемую переданным в качестве второго аргумента объектом-поставщиком
Пример использования: Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);		
counting	Long	Вычисляет число элементов в потоке данных
Пример использования: long howManyDishes = menuStream.collect(counting());		
summingInt	Integer	Суммирует значения указанного свойства типа Integer элементов потока данных
Пример использования: int totalCalories = menuStream.collect(summingInt(Dish::getCalories));		
averagingInt	Double	Вычисляет среднее значение указанного свойства типа Integer элементов потока данных
Пример использования: double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));		
summarizingInt	IntSummaryStatistics	Вычисляет сводные показатели, связанные со свойством типа Integer элементов потока данных: максимум, минимум, сумму, среднее значение и количество
Пример использования: IntSummaryStatistics menuStatistics = menuStream.collect(summarizingInt(Dish::getCalories));		
joining	String	Конкатенация строковых значений, получающихся в результате вызова метода toString для каждого из элементов потока данных
Пример использования: String shortMenu = menuStream.map(Dish::getName).collect(joining(“,”));		
maxBy	Optional<T>	Возвращает обертку Optional для максимального (в смысле указанного компаратора) элемента данного потока или Optional.empty(), если поток данных пуст
Пример использования: Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));		
minBy	Optional<T>	Возвращает обертку Optional для минимального (в смысле указанного компаратора) элемента данного потока или Optional.empty(), если поток данных пуст

Фабричный метод	Возвращаемый тип	Назначение
Пример использования: Optional<Dish> lightest = menuStream.collect(minBy(comparingInt(Dish::getCalories)));		
reducing	Тип, получающийся в результате операции свертки	Производит свертку потока данных до одного значения, последовательно группируя значение-накопитель (изначально равное заданному начальному значению) с элементами потока данных, с помощью заданного BinaryOperator
Пример использования: int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));		
collectingAndThen	Тип, возвращаемый преобразующей функцией	Обертывает другой коллектор, применяя к полученному результату заданную преобразующую функцию
Пример использования: int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));		
groupingBy	Map<K, List<T>>	Группирует элементы потока данных согласно значениям одного из их свойств, используя эти значения в качестве ключей в итоговом ассоциативном массиве
Пример использования: Map<Dish.Type,List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));		
partitioningBy	Map<Boolean, List<T>>	Секционирует элементы потока данных согласно результатам применения к ним предиката
Пример использования: Map<Boolean,List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));		

Как мы уже упоминали в начале главы, все эти коллекторы реализуют интерфейс `Collector`, так что в оставшейся части главы мы займемся более подробным его изучением. Мы рассмотрим имеющиеся в этом интерфейсе методы, а затем узнаем, как реализовывать пользовательские коллекторы.

6.5. Интерфейс Collector

Интерфейс `Collector` состоит из набора методов, служащих шаблонами для реализации конкретных операций свертки (коллекторов). Мы уже сталкивались со многими коллекторами, реализующими интерфейс `Collector`, например `toList` и `groupingBy`. Это значит также, что можно создавать пользовательские операции свертки путем реализации интерфейса `Collector`. В разделе 6.6 мы покажем, как реализовать интерфейс `Collector` для создания коллектора, с помощью которого можно было бы эффективнее секционировать поток чисел на простые и составные, чем было показано выше.

Сначала в рамках обсуждения интерфейса `Collector` мы сосредоточим внимание на одном из первых встреченных вами в этой главе коллекторов: на фабричном методе `toList`, собирающем все элементы потока данных в объект `List`. Мы уже упоминали, что вам часто придется использовать этот коллектор в своей повседневной работе, кроме того, его разработка наиболее проста, по крайней мере теоретически.

Подробное изучение реализации этого коллектора — отличный способ разобраться с тем, как описан интерфейс `Collector`, и с внутренним механизмом использования методом `collect` функций, возвращаемых методами данного интерфейса.

Начнем с того, что изучим описание интерфейса `Collector`, приведенное в листинге 6.4. Оно включает сигнатуру интерфейса, а также пять объявленных в нем методов.

Листинг 6.4. Интерфейс Collector

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

В этом листинге используются следующие обозначения:

- `T` — обобщенный тип собираемых элементов потока данных;
- `A` — тип накопителя, то есть объекта, в котором будет накапливаться частичный результат во время процесса сбора данных;
- `R` — тип объекта — результата операции `collect` (обычно, хотя и не всегда, коллекция).

Например, можно реализовать класс `ToListCollector<T>`, собирающий все элементы потока `Stream<T>` в объект `List<T>`, со следующей сигнатурой:

```
public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

где, как мы вскоре увидим, используемый в процессе накопления объект станет и итоговым результатом процесса сбора данных.

6.5.1. Разбираемся с объявленными в интерфейсе Collector методами

Проанализируем по очереди объявленные в интерфейсе `Collector` методы. Обратите внимание, что каждый из первых четырех методов возвращает функцию, предназначенную для вызова методом `collect`, а пятый, `characteristics`, возвращает набор характеристик, служащих в качестве подсказок методу `collect` относительно допустимых при операции свертки оптимизаций (например, распараллеливания).

Создаем новый контейнер для результата: метод supplier

Метод `supplier` должен возвращать объект типа `Supplier` — функцию без параметров, создающую при вызове экземпляр пустого накопителя для использования в процессе сбора данных. Разумеется, для коллектора, возвращающего в качестве результата сам накопитель, например `ToListCollector`, этот пустой накопитель является результатом процесса сбора данных в случае пустого потока. В нашем `ToListCollector` метод `supplier` будет возвращать пустой объект `List`:

```
public Supplier<List<T>> supplier() {
    return () -> new ArrayList<T>();
}
```

Отметим, что можно воспользоваться и ссылкой на конструктор:

```
public Supplier<List<T>> supplier() {
    return ArrayList::new;
}
```

Добавление элемента в контейнер для результата: метод accumulator

Метод `accumulator` возвращает функцию, осуществляющую операцию свертки. При обходе n-го элемента потока данных эта функция применяется с двумя аргументами — накопителем (результатом свертки предыдущих $n - 1$ элементов потока) и самим n-м элементом потока. Функция возвращает `void`, поскольку накопитель меняется «на месте», то есть она меняет его внутреннее состояние, отражая тем самым результат обработки очередного элемента. В случае `ToListCollector` данной функции нужно просто добавить текущий элемент в список, содержащий уже обработанные элементы:

```
public BiConsumer<List<T>, T> accumulator() {
    return (list, item) -> list.add(item);
```

Для лаконичности можно воспользоваться ссылкой на метод:

```
public BiConsumer<List<T>, T> accumulator() {
    return List::add;
}
```

Применяем к контейнеру итоговое преобразование для получения результата: метод finisher

Метод `finisher` должен возвращать функцию, вызываемую в конце процесса накопления, по завершении прохода по потоку данных для преобразования объекта-накопителя в конечный результат всей операции сбора данных. Зачастую, как и в случае `ToListCollector`, объект-накопитель совпадает с ожидаемым конечным результатом. Вследствие этого необходимости выполнять какое-либо преобразование нет, так что метод `finisher` должен возвращать функцию `identity`:

```
public Function<List<T>, List<T>> finisher() {
    return Function.identity();
}
```

Описанных трех методов достаточно для выполнения последовательной свертки потока данных, осуществляемой, по крайней мере с точки зрения логики работы, так, как показано на рис. 6.7. На практике реализация несколько более сложна по причине отложенной природы потоков данных, из-за которой может потребоваться конвейер других, выполняемых до `collect`, промежуточных операций, а также из-за наличия возможности (хотя бы теоретической) выполнения свертки в параллельном режиме.

Объединение двух контейнеров для результатов: метод combiner

Метод `combiner`, последний из четырех методов, возвращающий используемую операцией свертки функцию, определяет, каким образом объединяются накопители,

полученные в результате параллельной обработки частей потока данных. В случае коллектора `ToList` реализация этого метода проста: список, содержащий собранные из второй части потока элементы, добавляется в конец списка, полученного при обходе первой части:

```
public BinaryOperator<List<T>> combiner() {
    return (list1, list2) -> {
        list1.addAll(list2);
        return list1; }
}
```

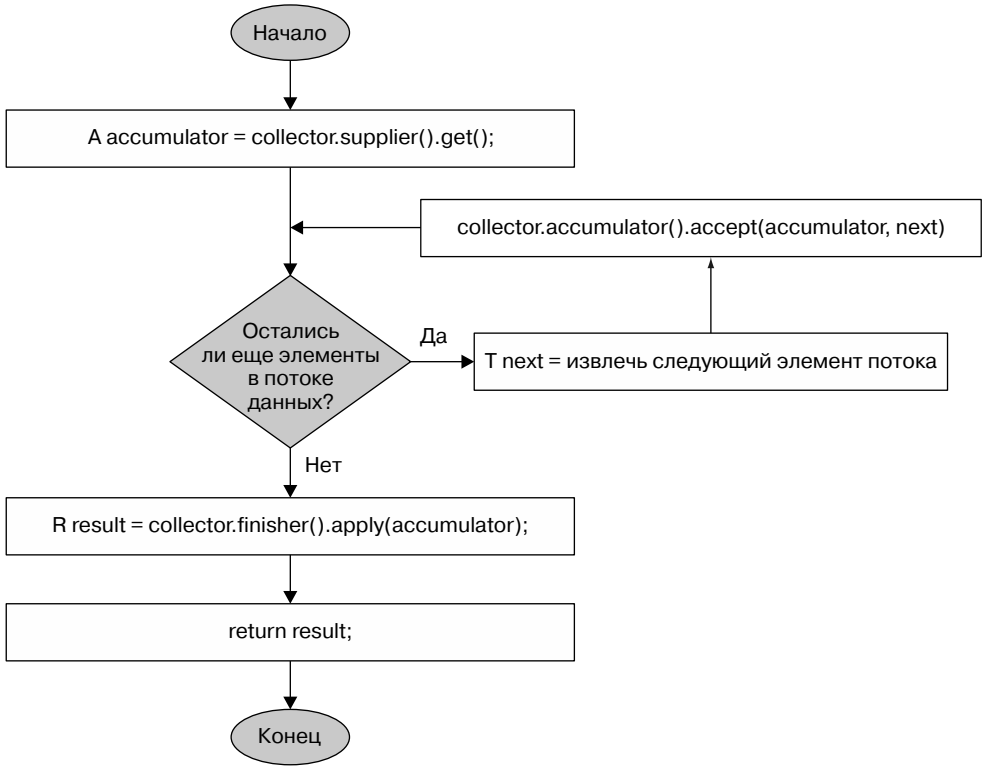


Рис. 6.7. Логические шаги процесса последовательной свертки

Благодаря добавлению четвертого метода появляется возможность параллельной свертки потоков данных. При этом используются фреймворк ветвления-объединения, появившийся в Java 7, а также абстракция `Spliterator`, о которой мы расскажем вам в следующей главе. Процесс выполнения при этом напоминает показанный на рис. 6.8 (и подробно описанный ниже).

- Исходный *поток данных* рекурсивно разбивается на подпотоки, пока условие разбиения потока не становится ложным (параллельная обработка часто вы-

полняется медленнее, чем последовательная, при слишком маленьких единицах работы; кроме того, нет смысла генерировать число параллельных задач, которое бы превышало количество ядер процессора).

- На этом этапе все *подпотоки* обрабатываются параллельно, каждый — с помощью последовательного алгоритма свертки, показанного на рис. 6.7.
- Наконец, все эти частичные результаты объединяются с помощью функции, возвращаемой методом `combiner` коллектора. Это делается путем объединения результатов, соответствующих каждому подпотоку, возникшему в результате разбиения исходного потока.

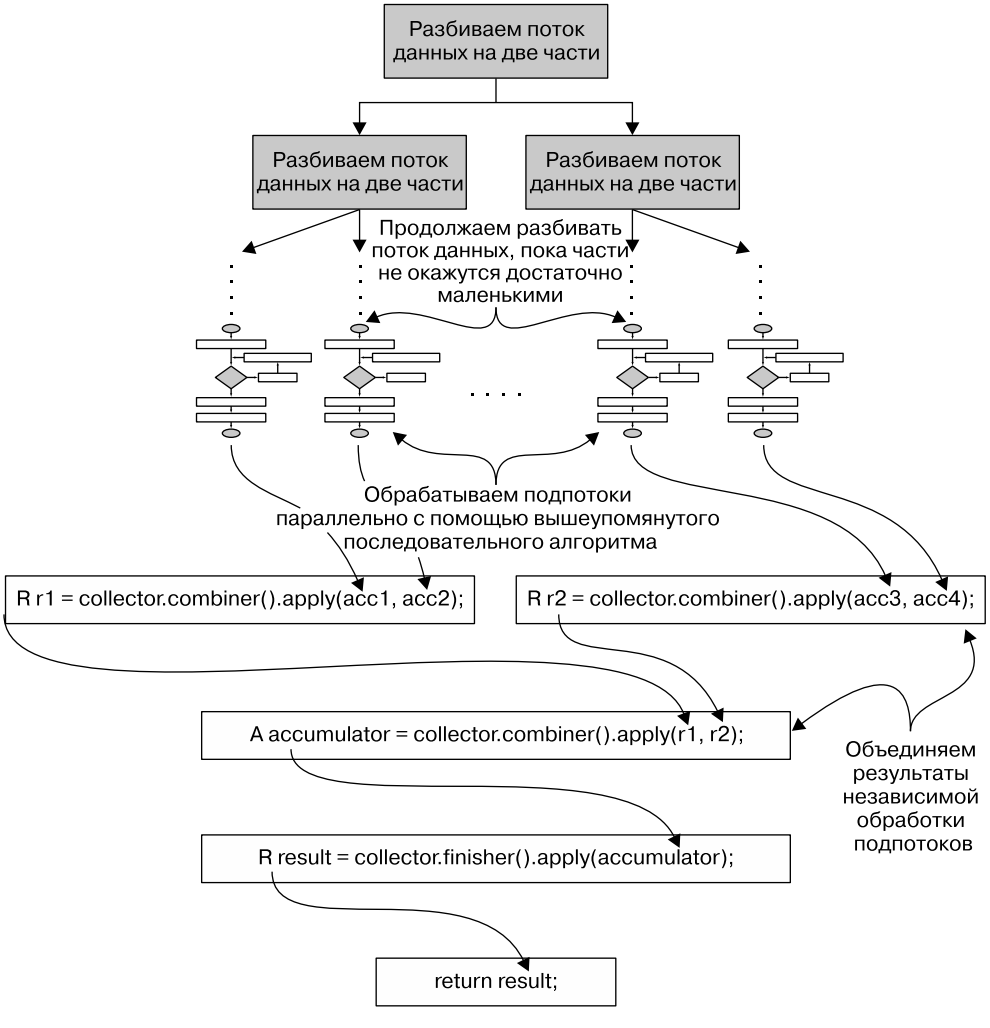


Рис. 6.8. Распараллеливание процесса свертки с помощью метода combiner

Метод characteristics

Последний из методов, `characteristics`, возвращает неизменяемый набор типа `Characteristics`, который определяет поведение коллектора — в частности, предоставляет «подсказки» относительно того, можно ли выполнять свертку параллельно и какие оптимизации допустимы при этом. `Characteristics` — перечисляемый тип, содержащий три элемента.

- `UNORDERED` — порядок обхода и накопления элементов потока данных не влияет на результат свертки.
- `CONCURRENT` — функцию `accumulator` можно вызывать параллельно из нескольких потоков выполнения, и, следовательно, этот коллектор может производить параллельную свертку потока данных. Если коллектор при этом не помечен как `UNORDERED`, то он может выполнять параллельную свертку только при работе с неупорядоченным источником данных.
- `IDENTITY_FINISH` — указывает, что возвращаемая методом `finisher` функция — тождественная и ее можно не вызывать. В этом случае объект-накопитель непосредственно используется в качестве окончательного результата процесса свертки. Означает также, что можно без проверки выполнять приведение типа накопителя `A` к типу результата `R`.

Разработанный нами выше `ToListCollector` обладает свойством `IDENTITY_FINISH`, поскольку используемый для накопления результатов объект `List` представляет собой конечный результат, не требующий дальнейших преобразований, но не `UNORDERED`, так как при обработке упорядоченного потока данных необходимо, чтобы порядок сохранялся в получаемом списке. Наконец, он `CONCURRENT`, но в соответствии со сказанным выше обрабатывать поток данных параллельно может только в случае, если источник данных потока не упорядочен.

6.5.2. Собираем все вместе

Обсуждавшихся в предыдущем разделе пяти методов вполне достаточно для разработки своего собственного `ToListCollector`, так что мы реализуем его, собрав их все вместе, как показано в листинге 6.5.

Листинг 6.5. Класс ToListCollector

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collector;
import static java.util.stream.Collector.Characteristics.*;
public class ToListCollector<T> implements Collector<T, List<T>, List<T>> {
    @Override
    public Supplier<List<T>> supplier() {
        return ArrayList::new;
    }
    @Override
    public BiConsumer<List<T>, T> accumulator() {
        return List::add;
    }
```

Создаем отправной пункт для операции сбора данных

Добавляем обработанный элемент в накопитель (на месте)

```
    @Override
    public Function<List<T>, List<T>> finisher() {
        return Function.identity();
    }
    @Override
    public BinaryOperator<List<T>> combiner() {
        return (list1, list2) -> {
            list1.addAll(list2);
            return list1;
        };
    }
    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(
            IDENTITY_FINISH, CONCURRENT));
    }
}
```

Тождественная функция

Модифицируем первый накопитель, объединяя его с содержимым второго

Возвращаем модифицированный первый накопитель

Помечаем коллектор как IDENTITY_FINISH и CONCURRENT

Отметим, что эта реализация не идентична возвращаемой методом `Collectors.toList`, но различие лишь в нескольких мелких оптимизациях, связанных в основном с тем, что предоставляемый API Java коллектор использует (возвращающий объект-одиночку) метод `Collections.emptyList()` в том случае, когда нужно вернуть пустой список. Это значит, что его можно спокойно использовать вместо нашего первоначального Java-кода в качестве примера получения списка всех блюд из потока данных меню:

```
List<Dish> dishes = menuStream.collect(new ToListCollector<Dish>());
```

Еще одно, последнее различие между этим и стандартным способом:

```
List<Dish> dishes = menuStream.collect(toList());
```

состоит в том, что `toList()` — фабричный метод, а для создания экземпляра вашего `ToListCollector` необходимо использовать `new`.

Нестандартный сбор данных без реализации интерфейса Collector. В случае операции сбора данных с характеристикой `IDENTITY_FINISH` существует возможность получить тот же результат без разработки полностью новой реализации интерфейса `Collector`. У потоков данных есть перегруженный метод `collect`, принимающий в качестве аргументов три другие функции — `supplier`, `accumulator` и `combiner` — точно с такой же семантикой, что возвращаемые соответствующими методами интерфейса `Collector`.

Например, собрать в список все элементы потока блюд можно следующим образом:

```
List<Dish> dishes = menuStream.collect(
                        ArrayList::new,
                        List::add,
                        List::addAll);
```

Метод Supplier

Метод Accumulator

Метод Combiner

Мне кажется, что вторая форма записи, хотя и более лаконичная, чем предыдущая, менее удобочитаема. Кроме того, создание реализации своего пользовательского коллектора в виде соответствующего класса способствует его переиспользованию и помогает избегать дублирования кода. Стоит также отметить, что в этот второй метод `collect` нельзя передавать никакие характеристики, так что он всегда ведет себя как коллектор типа `IDENTITY_FINISH` и `CONCURRENT`, но не `UNORDERED`.

В следующем разделе ваши познания в сфере создания коллекторов поднимутся на новый уровень. Вам предстоит разработать собственный пользовательский коллектор для более сложного, но, надеемся, более необычного и захватывающего сценария применения.

6.6. Разрабатываем собственный, более производительный коллектор

В разделе 6.4 при обсуждении секционирования мы создали коллектор с помощью одного из множества удобных фабричных методов класса `Collectors`, который разбивал первые *n* натуральных чисел на простые и составные, как показано в листинге 6.6.

Листинг 6.6. Секционирование первых n натуральных чисел на простые и составные

```
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
return IntStream.rangeClosed(2, n).boxed()
                .collect(partitioningBy(candidate -> isPrime(candidate));
}
```

Мы уже усовершенствовали исходный метод `isPrime` тем, что проверяются лишь те возможные делители потенциального простого числа, которые не превышают квадратного корня из него:

```
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

Можно ли добиться еще более высокой производительности? Ответ — да, но для этого необходимо разработать пользовательский коллектор.

6.6.1. Деление только на простые числа

Одна из возможных оптимизаций — проверять делимость числа — кандидата в простые числа только на простые числа. Не имеет смысла проверять, делится ли оно на составное число! Можно ограничить проверку только уже обнаруженными до текущего числа-кандидата простыми числами. Проблема с используемыми до сих пор встроенными коллекторами (и причина, по которой нам придется разработать пользовательский коллектор): отсутствие доступа к промежуточному результату во время процесса сбора данных. Это значит, что при проверке простоты текущего числа-кандидата у нас нет доступа к списку уже обнаруженных ранее простых чисел.

Допустим, у нас есть этот список. Тогда можно передать его в метод `isPrime` и переписать последний следующим образом:

```
public static boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}
```

Имеет также смысл реализовать вышеупомянутую оптимизацию и проверять только простые числа, которые меньше квадратного корня из числа-кандидата. Нам требуется найти способ прекратить проверку на делимость числа-кандидата в момент, когда следующее простое число больше, чем корень из числа-кандидата. Это можно легко сделать с помощью метода `takeWhile` интерфейса `Stream`:

```
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return primes.stream()
                 .takeWhile(i -> i <= candidateRoot)
                 .noneMatch(i -> candidate % i == 0);
}
```

Контрольное задание 6.3. Имитация метода takeWhile в Java 8

Метод `takeWhile` появился только в Java 9, так что, к сожалению, такое решение недоступно тем, кто использует Java 8. Как бы вы обошли это ограничение и создали подобное решение в Java 8?

Ответ: вы можете реализовать собственный аналог метода `takeWhile`, который по заданному отсортированному списку и предикату возвращает начальную часть этого списка, элементы которой удовлетворяют предикату:

```
public static <A> List<A> takeWhile(List<A> list, Predicate<A> p) {
    int i = 0;
    for (A item : list) {
            if (!p.test(item)) {
                    return list.subList(0, i);
            }
            i++;
    }
    return list;
}
```

Проверяем, удовлетворяет ли текущий элемент списка заданному условию

Если нет, возвращаем часть списка, с начала, вплоть до элемента, предшествующего проверяемому

Все элементы списка удовлетворяют условию, так что возвращаем его целиком

С помощью такого метода можно переписать метод `isPrime` и тоже проверять делимость потенциального простого числа только на те простые числа, которые не превышают квадратного корня из него.

```
public static boolean isPrime(List<Integer> primes, int candidate){
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return takeWhile(primes, i -> i <= candidateRoot)
                .stream()
                .noneMatch(p -> candidate % p == 0);
}
```

Отметим, что, в отличие от метода из Stream API, данная реализация метода `takeWhile` производит вычисления немедленно. По возможности всегда используйте вычисляемую отложенным образом версию `takeWhile` из Stream API Java 9, чтобы ее можно было сочетать с операцией `noneMatch`.

Благодаря этому новому методу `isPrime` мы готовы реализовать наш собственный пользовательский коллектор. Во-первых, нам понадобится создать новый класс, реализующий интерфейс `Collector`. Далее необходимо будет разработать пять требуемых интерфейсом `Collector` методов.

Шаг 1: описание сигнатуры класса Collector

Начнем с сигнатуры класса. Напомним, что интерфейс `Collector` описан следующим образом:

```
public interface Collector<T, A, R>
```

где `T`, `A` и `R` — тип элементов потока данных, тип объекта, используемого для накопления промежуточных результатов, и тип конечного результата операции `collect` соответственно. В данном случае нам нужно собирать потоки объектов `Integer`, притом что тип как накопителя, так и результата — `Map<Boolean, List<Integer>>` (такой же, как и у результата вышеописанной операции секционирования в листинге 6.6). В качестве ключей используются `true` и `false`, а в качестве значений — соответственно списки простых и составных чисел:

```
public class PrimeNumbersCollector
             implements Collector<Integer,
                                  Map<Boolean, List<Integer>>,
                                  Map<Boolean, List<Integer>>>
```

Тип элементов потока

Тип результата операции сбора данных

Тип накопителя

Шаг 2: реализация процесса свертки

Далее нам нужно реализовать пять объявленных в интерфейсе `Collector` методов. Метод `supplier` должен возвращать функцию, при вызове которой создается накопитель:

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {{
        put(true, new ArrayList<Integer>());
        put(false, new ArrayList<Integer>());
    }};
}
```

В этом коде ассоциативный массив — будущий накопитель — не только создается, но и инициализируется двумя пустыми списками для ключей `true` и `false`. Именно в них мы будем добавлять простые и составные числа соответственно во время процесса сбора данных. Важнейший метод нашего коллектора — `accumulator`,

содержащий логику сбора элементов потока данных. В данном случае этот метод также является ключом к выполнению описанной выше оптимизации. На любой итерации цикла у нас теперь есть доступ к промежуточным результатам процесса сбора данных — к накопителю, содержащему найденные на этот момент простые числа:

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime(acc.get(true), candidate) )
               .add(candidate);
    };
}
```

Получаем список простых или составных чисел, в зависимости от результата выполнения isPrime

Добавляем число-кандидат в соответствующий список

В этом методе мы вызываем метод `isPrime`, передавая в него (вместе с проверяемым на простоту числом) список уже найденных простых чисел (то есть значения, индексированные ключом `true` в ассоциативном массиве-накопителе). Результат этого вызова далее используется в качестве ключа для получения списка простых или составных чисел, так что новое число добавляется в нужный список.

Шаг 3: обеспечиваем параллельное выполнение коллектора (если это возможно)

Задача следующего метода состоит в объединении двух частичных накопителей при параллельном процессе сбора данных, то есть в данном случае в объединении двух ассоциативных массивов: добавлении всех чисел из списков простых/составных чисел второго ассоциативного массива в соответствующие списки первого:

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1,
            Map<Boolean, List<Integer>> map2) -> {
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
                return map1;
    };
}
```

Отметим, что на самом деле не получится использовать этот коллектор в параллельном режиме, поскольку данный алгоритм по своей сути является последовательным. Это значит, что метод `combiner` никогда не будет вызываться и его реализацию можно оставить пустой (или, что предпочтительнее, генерировать в ней исключение `UnsupportedOperationException`). Мы приняли решение реализовать его просто для полноты.

Шаг 4: методы finisher и characteristics коллектора

Реализация последних двух методов очень проста. Как мы уже говорили, наш накопитель совпадает с результатом работы коллектора, так что дальнейших его преобразований не требуется и метод `finisher` возвращает тождественную функцию:

```
public Function<Map<Boolean, List<Integer>>,
                Map<Boolean, List<Integer>>> finisher() {
```

```
        return Function.identity();
}
```

Что же касается метода `characteristics`, то мы уже упоминали, что наш коллектор `IDENTITY_FINISH`, но не `CONCURRENT` и не `UNORDERED`:

```
public Set<Characteristics> characteristics() {
    return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
}
```

Листинг 6.7 демонстрирует законченную реализацию `PrimeNumbersCollector`.

Листинг 6.7. Класс PrimeNumbersCollector

```
public class PrimeNumbersCollector
    implements Collector<Integer,
           Map<Boolean, List<Integer>>,
           Map<Boolean, List<Integer>>> {
    @Override
    public Supplier<Map<Boolean, List<Integer>>> supplier() {
        return () -> new HashMap<Boolean, List<Integer>>() {{
             put(true, new ArrayList<Integer>());
             put(false, new ArrayList<Integer>());
          }};
    }
    @Override
    public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
        return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
            acc.get( isPrime( acc.get(true),
                candidate) )
                .add(candidate);
        };
    }
    @Override
    public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
            return (Map<Boolean, List<Integer>> map1,
                    Map<Boolean, List<Integer>> map2) -> {
                            map1.get(true).addAll(map2.get(true));
                            map1.get(false).addAll(map2.get(false));
                            return map1;
                    };
    }
    @Override
    public Function<Map<Boolean, List<Integer>>,
                    Map<Boolean, List<Integer>>> finisher() {
                            return Function.identity();
    }
    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }
}
```

Начинаем процесс сбора с ассоциативного массива, содержащего два пустых списка

Передаем в метод isPrime список уже найденных простых чисел

Получаем из ассоциативного массива список простых или составных чисел, в зависимости от возвращенного методом isPrime результата, и добавляем в него текущее число-кандидат

Объединяем второй ассоциативный массив с первым

В конце сбора данных не требуются никакие дополнительные преобразования, так что завершаем его тождественной функцией

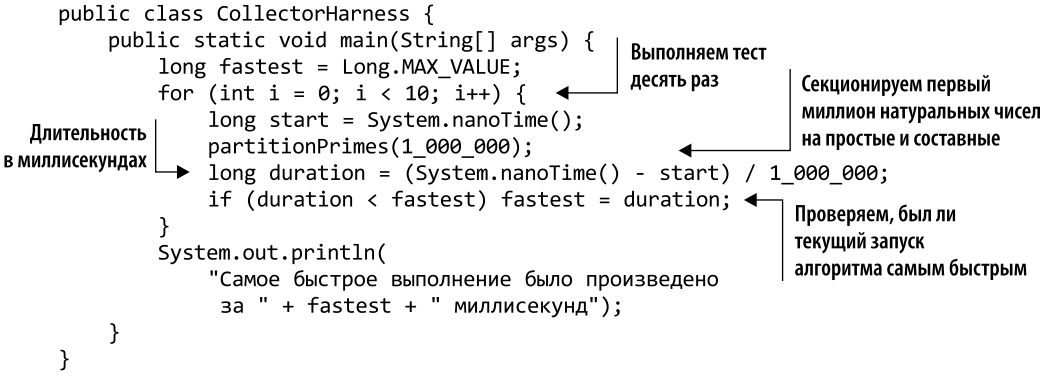
Коллектор обладает свойством IDENTITY_FINISH, но не CONCURRENT и не UNORDERED, поскольку в его основе лежит последовательный поиск простых чисел

Теперь можно воспользоваться этим новым коллектором вместо того, что создавался с помощью фабричного метода `partitioningBy` в разделе 6.4, и получить в точности тот же результат:

```
public Map<Boolean, List<Integer>>
                    partitionPrimesWithCustomCollector(int n) {
        return IntStream.rangeClosed(2, n).boxed()
                        .collect(new PrimeNumbersCollector());
}
```

6.6.2. Сравнение производительности коллекторов

Функционально коллектор, созданный с помощью фабричного метода `partitioningBy`, и только что созданный нами пользовательский коллектор идентичны, но удалось ли нам путем создания пользовательского коллектора повысить производительность? Напишем небольшую тестовую оснастку для проверки:



Отметим, что для более научного подхода к оценке производительности следовало бы воспользоваться фреймворком вроде Java Microbenchmark Harness (JMH), но мы не хотим усложнять изложение за счет его использования и для данного сценария вышеприведенный маленький класс возвращает достаточно точные результаты. Этот класс секционирует первый миллион натуральных чисел на простые и составные, вызывая созданный с помощью фабричного метода `partitioningBy` коллектор десять раз и фиксируя длительность самого быстрого выполнения. При запуске его на процессоре Intel i5 2.4 Ггц были выведены следующие результаты:

```
Самое быстрое выполнение было произведено за 4716 миллисекунд
```

Теперь меняем в нашей тестовой оснастке `partitionPrimes` на `partitionPrimesWithCustomCollector` и тестируем производительность разработанного нами пользовательского коллектора. Теперь программа выводит в консоль следующее:

```
Самое быстрое выполнение было произведено за 3201 миллисекунду
```

Неплохо! Это значит, что вы не напрасно потратили время на разработку пользовательского коллектора по двум причинам. Во-первых, вы научились создавать

свои коллекторы. И во-вторых, добились повышения производительности примерно на 32 %.

Наконец, важно отметить, что аналогично тому, как мы делали с `ToListCollector` в листинге 6.5, можно получить тот же результат, передав три реализующие основную логику `PrimeNumbersCollector` функции в качестве аргументов в перегруженную версию метода `collect`:

```
public Map<Boolean, List<Integer>> partitionPrimesWithCustomCollector
             (int n) {
    IntStream.rangeClosed(2, n).boxed()                          Метод supplier
        .collect(
                     () -> new HashMap<Boolean, List<Integer>>() {{
                        put(true, new ArrayList<Integer>());
                        put(false, new ArrayList<Integer>());
                  }},                                  Метод accumulator
             (acc, candidate) -> {
                acc.get( isPrime(acc.get(true), candidate) )
                   .add(candidate);
            },                                  Метод combiner
              (map1, map2) -> {
                map1.get(true).addAll(map2.get(true));
                map1.get(false).addAll(map2.get(false));
            });
}
```

Как вы можете видеть, нам не требуется создавать совершенно новый класс, реализующий интерфейс `Collector`; получившийся при этом код более лаконичен, хотя, возможно, менее удобочитаем и явно хуже подходит для переиспользования.

Резюме

- `collect` — завершающая операция, принимающая в качестве аргументов различные стратегии (так называемые коллекторы) агрегирования элементов потока в сводном результате.
- Встроенные коллекторы включают свертку элементов потока данных в единое значение и вычисление сводных показателей, например минимума, максимума и среднего значения. Список этих коллекторов приведен в табл. 6.1.
- С помощью встроенных коллекторов можно группировать элементы потока (коллектор `groupingBy`) и секционировать их (коллектор `partitioningBy`).
- Коллекторы можно эффективно комбинировать для многоуровневой группировки, секционирования и свертки.
- Можно разрабатывать свои собственные коллекторы путем реализации описанных в интерфейсе `Collector` методов.

7 Параллельная обработка данных и производительность

В этой главе

- Параллельная обработка данных с помощью параллельных потоков.
- Анализ производительности параллельных потоков данных.
- Фреймворк ветвления-объединения.
- Разбиение потока данных с помощью интерфейса `Spliterator`.

В предыдущих трех главах рассказывалось, как с помощью нового интерфейса `Stream` можно декларативным образом задавать различные действия над коллекциями данных. Мы также объясняли, что переход от внешней к внутренней итерации повышает степень контроля нативной библиотеки Java над обработкой элементов потока данных. Подобный подход избавляет Java-разработчиков от необходимости явным образом воплощать оптимизации, необходимые для ускорения обработки коллекций данных. Наиболее важное преимущество — возможность автоматического использования нескольких ядер процессора при выполнении конвейера операций над коллекциями.

Например, до Java 7 для параллельной обработки коллекций данных требовался исключительно громоздкий код. Во-первых, необходимо было явным образом разбить обрабатываемую структуру данных на части. Во-вторых, нужно было распределить эти части по различным потокам выполнения. В-третьих, требовалось так синхронизовать их, чтобы избежать нежелательных состояний гонки, дождаться завершения всех потоков выполнения и, наконец, объединить частичные результаты. В Java 7 появился фреймворк *ветвления-объединения* (fork/join), благодаря которому выполнение подобных операций стало более единообразным и менее подверженным ошибкам. Мы поговорим об этом фреймворке в разделе 7.2.

В этой главе вы увидите, насколько простым стало параллельное выполнение операций над коллекциями данных благодаря интерфейсу `Stream`. С его помощью можно декларативно преобразовать последовательный поток данных в параллельный. Более того, вам предстоит увидеть, каким образом Java делает это возможным или, точнее говоря, что происходит «под капотом» параллельных потоков при использовании появившегося в Java 7 фреймворка ветвления-объединения. Вы также обнаружите, насколько важно представлять себе, как работают параллельные потоки данных, ведь без этих знаний легко можно применить их неправильно и получить неожиданные (и, вероятно, неверные) результаты.

В частности, мы покажем, что в некоторых случаях источником неправильных и необъяснимых результатов может быть разбиение потока данных на порции перед их параллельной обработкой. Поэтому мы расскажем вам, как реализовать свой собственный сплитератор для управления процессом разбиения.

7.1. Параллельные потоки данных

В главе 4 мы вкратце упоминали, что интерфейс `Stream` предоставляет удобную возможность параллельной обработки элементов; с помощью вызова метода `parallelStream` коллекции-источника можно преобразовать коллекцию в параллельный поток данных. *Параллельный* поток данных разбивает элементы на несколько порций, каждая из которых обрабатывается в отдельном потоке выполнения. Таким образом, весь объем работы заданной операции автоматически равномерно распределяется по всем ядрам многоядерного процессора. Поэкспериментируем с этой идеей на простом примере.

Пусть необходимо написать метод, принимающий в качестве аргумента число *n* и возвращающий сумму всех чисел от 1 до *n*. Простейший (и, возможно, наивный) способ решения такой задачи — сгенерировать бесконечный поток чисел, ограничив его переданным в метод числом, и выполнить свертку полученного потока данных с помощью `BinaryOperator`, суммирующего два числа:

```
public long sequentialSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
                 .limit(n)
                 .reduce(0L, Long::sum);
}
```

Генерируем бесконечный поток натуральных чисел

Ограничиваем его первыми n числами

Производим свертку полученного потока, суммируя все числа

В более традиционном для Java виде эквивалентный итеративный аналог этого кода:

```
public long iterativeSum(long n) {
    long result = 0;
    for (long i = 1L; i <= n; i++) {
        result += i;
    }
    return result;
}
```

Эта операция — очевидный кандидат на распараллеливание, особенно при больших значениях *n*. Но с чего же начать? Нужно ли синхронизировать переменную, в которой хранится результат? Сколько потоков выполнения использовать? Кто будет отвечать за генерацию чисел? Кто будет отвечать за их суммирование?

Не забивайте себе этим голову. Задача сильно упрощается, если воспользоваться параллельными потоками данных!

7.1.1. Превращаем последовательный поток данных в параллельный

Организовать параллельное выполнение вышеприведенного функционального процесса свертки (суммирования) можно путем преобразования потока данных в параллельный. Вызовем метод `parallel` последовательного потока:

```
public long parallelSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
                 .limit(n)
                 .parallel()
                 .reduce(0L, Long::sum);
}
```

Превращаем поток данных в параллельный

В предыдущем коде с помощью процесса свертки мы просуммировали все числа в потоке данных аналогично тому, как это происходило в подразделе 5.4.1. Различие в том, что «под капотом» поток данных делится на несколько порций. В результате операция свертки выполняется параллельно над различными порциями данных, как показано на рис. 7.1. Наконец, та же самая операция свертки объединяет все значения, полученные в результате частных операций свертки каждого из подпотоков, и получается итоговый результат процесса свертки всего исходного потока данных.

Отметим, что на практике вызов метода `parallel` для последовательного потока не означает какого-либо конкретного преобразования самого потока. «Под капотом» происходит установка булева флага, который указывает, что все следующие за `parallel` операции необходимо выполнять параллельно. Аналогично можно преобразовать параллельный поток данных в последовательный, вызвав для него метод `sequential`. Вам может показаться, что путем комбинирования этих двух методов можно добиться очень точного контроля над тем, какие операции при обходе потока данных производить параллельно, а какие — последовательно. Например, сделать что-то вроде:

```
stream.parallel()
      .filter(...)
      .sequential()
      .map(...)
      .parallel()
      .reduce();
```

Но приоритет всегда будет у последнего вызова `parallel` или `sequential`, который влияет на весь конвейер. В этом примере конвейер будет выполняться параллельно, поскольку последним в нем был вызов метода `parallel`.

Настройка пула потоков выполнения, используемого параллельными потоками данных

Возможно, при взгляде на метод `parallel` потока у вас возникнет вопрос, откуда берутся потоки выполнения, используемые параллельным потоком данных, и как можно настроить этот процесс под свои потребности.

Внутри механизма параллельных потоков данных используется по умолчанию `ForkJoinPool` (больше про фреймворк ветвления-объединения мы расскажем в разделе 7.2), количество потоков выполнения которого по умолчанию равно количеству процессоров, возвращаемому методом `Runtime.getRuntime().availableProcessors()`.

Впрочем, с помощью системного свойства `java.util.concurrent.ForkJoinPool.common.parallelism` можно изменить размер этого пула так, как показано в следующем примере:

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "12");
```

Это глобальный параметр, так что он влияет сразу на все параллельные потоки данных в коде. В настоящий момент возможность изменения этого значения для отдельного параллельного потока данных отсутствует. В целом размер `ForkJoinPool`, равный числу процессоров машины, — вполне разумная настройка, и мы настойчиво рекомендуем вам ее не менять без уважительных причин.

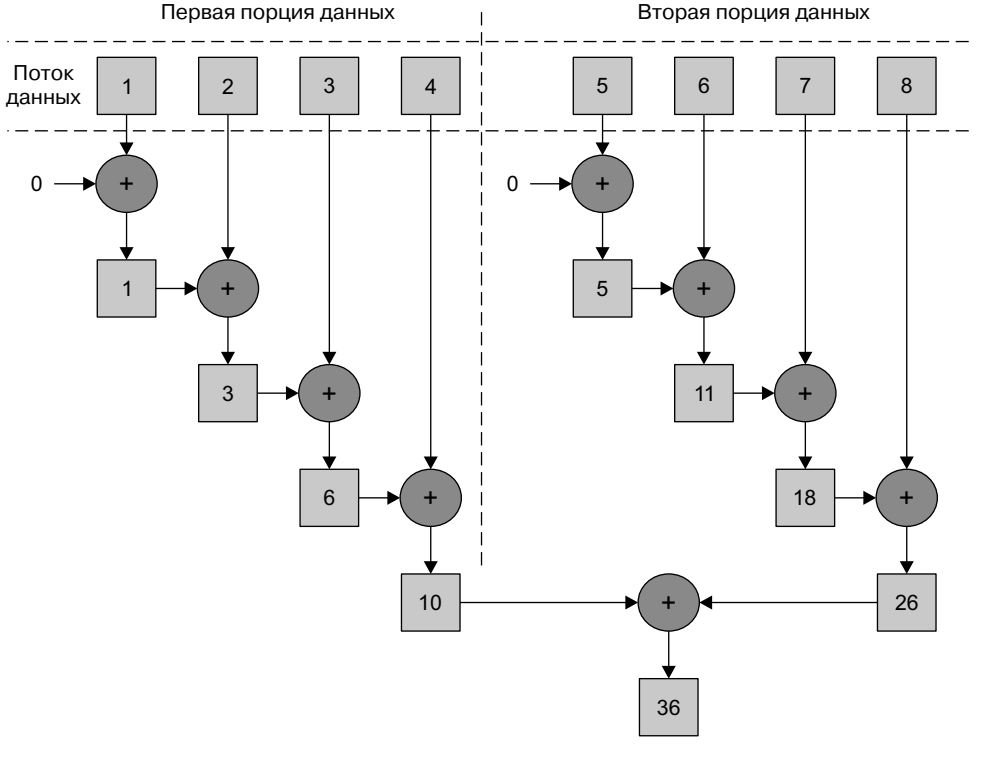


Рис. 7.1. Параллельная операция свертки

Вернемся к примеру с суммированием чисел. Мы говорили, что от параллельной версии можно ожидать существенного роста производительности при работе на

многоядерном процессоре. Теперь проверим, какой из трех имеющихся у нас методов, решающих одну задачу тремя различными способами (итеративный стиль программирования, последовательная свертка и параллельная свертка), работает быстрее всего!

7.1.2. Измерение быстродействия потока данных

Мы заявили, что параллельный метод суммирования должен работать быстрее последовательного и итеративного методов. Однако при разработке программного обеспечения гипотезы всегда требуют проверки! При оптимизации производительности следует всегда соблюдать три золотых правила: измерять, измерять и еще раз измерять. Для этой цели мы реализуем тест производительности с помощью библиотеки под названием Java Microbenchmark Harness (JMH). Этот набор инструментов предназначен для удобного создания с помощью аннотаций надежных микротестов производительности для Java и любых других языков программирования, использующих виртуальную машину Java (JVM). На самом деле разработка правильных и осмысленных тестов производительности для работающих на JVM программ — задача непростая, ведь при этом приходится учитывать множество факторов, например время, которое требуется HotSpot для оптимизации байт-кода, и накладные расходы на сборку мусора. В случае системы сборки Maven для использования JMH в проекте необходимо добавить несколько зависимостей в файл `pom.xml` (который описывает процесс сборки Maven).

```
<dependency>
  <groupId>org.openjdk.jmh</groupId>
  <artifactId>jmh-core</artifactId>
  <version>1.17.4</version>
</dependency>
<dependency>
  <groupId>org.openjdk.jmh</groupId>
  <artifactId>jmh-generator-annprocess</artifactId>
  <version>1.17.4</version>
</dependency>
```

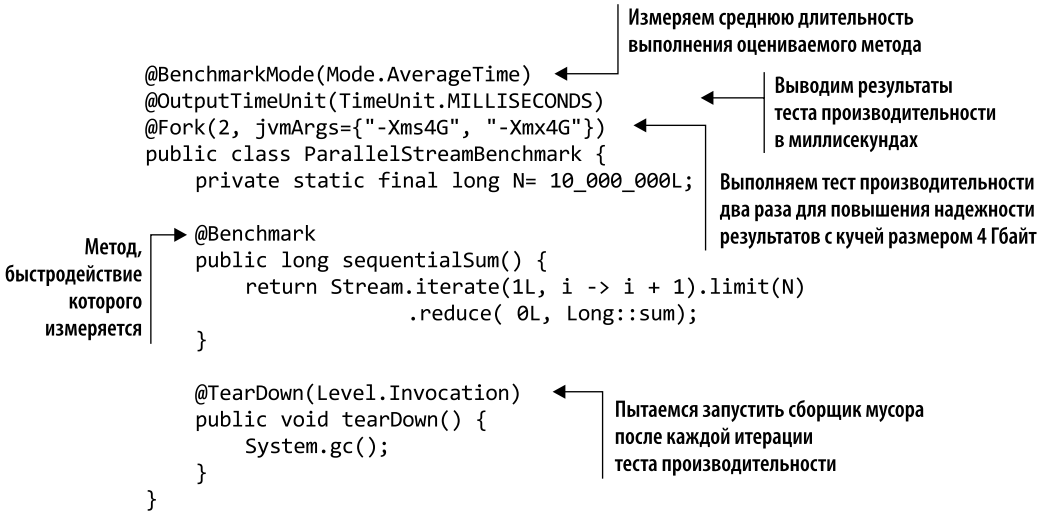
Первая из описанных библиотек — основная реализация JMH, а вторая содержит обработчик аннотаций для генерации файла Java-архива (JAR), с помощью которого можно легко запустить тест производительности после добавления в конфигурацию Maven следующего плагина:

```
<build>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <executions>
        <execution>
          <phase>package</phase>
          <goals><goal>shade</goal></goals>
          <configuration>
            <finalName>benchmarks</finalName>
            <transformers>
```

```
            <transformer implementation="org.apache.maven.plugins.shade.
                                    resource.ManifestResourceTransformer">
              <mainClass>org.openjdk.jmh.Main</mainClass>
            </transformer>
          </transformers>
        </configuration>
      </execution>
    </executions>
  </plugin>
 </plugins>
</build>
```

После этого можно легко измерить производительность приведенного в начале раздела метода `sequentialSum`, как показано в листинге 7.1.

Листинг 7.1. Измерение производительности функции, суммирующей первые n натуральных чисел



После компиляции этого класса настроенный выше плагин Maven генерирует второй JAR-файл с названием `benchmarks.jar`, который можно выполнить следующим образом:

```
java -jar ./target/benchmarks.jar ParallelStreamBenchmark
```

Мы задали в настройках теста производительности повышенный размер кучи, чтобы по возможности избежать влияния сборщика мусора, и по той же причине пытаемся запустить сборку мусора после каждой итерации нашего теста производительности. Несмотря на все эти предосторожности, следует отметить, что к результатам нужно относиться с долей скептицизма. На время выполнения может повлиять множество факторов, например количество ядер процессора на машине. Можете запустить этот тест на своей машине, воспользовавшись кодом из сопровождающего книгу репозитория.

Запуск вышеприведенного кода приводит к тому, что JMH выполняет тестируемый метод сначала 20 раз, для того чтобы HotSpot оптимизировал код, а потом еще 20 раз — для вычисления итогового результата. Подобное выполнение 20 + 20 итераций — поведение JMH по умолчанию, но оба значения можно поменять или с помощью специальных аннотаций JMH, или, что еще удобнее, путем добавления в командную строку флагов `-w` и `-i`. При выполнении этого кода на компьютере с четырехъядерным процессором Intel i7-4600U 2.1 ГГц были выведены следующие результаты:

```
Benchmark                                  Mode  Cnt    Score    Error  Units
ParallelStreamBenchmark.sequentialSum      avgt   40  121.843 ±  3.062  ms/op
```

Следует ожидать, что итеративная версия с обычным циклом `for` будет работать намного быстрее благодаря своей низкоуровневости и, что главное, отсутствию необходимости упаковки/распаковки значений простых типов данных. Проверим эту догадку, добавив в класс теста производительности из листинга 7.1 еще один метод, также снабженный аннотацией `@Benchmark`:

```
@Benchmark
public long iterativeSum() {
    long result = 0;
    for (long i = 1L; i <= N; i++) {
        result += i;
    }
    return result;
}
```

Запускаем этот второй тест (закомментировав первый, чтобы не запустить его еще раз) на тестовой машине и получаем такие результаты:

```
Benchmark                                  Mode  Cnt    Score    Error  Units
ParallelStreamBenchmark.iterativeSum       avgt   40    3.278 ±  0.192  ms/op
```

Как и следовало ожидать, итеративная версия оказалась почти в 40 раз быстрее использующей последовательный поток данных по вышеупомянутым причинам. Теперь проделаем то же самое с версией, в которой применяется параллельный поток данных, добавив еще один метод в класс теста производительности. Получаем следующее:

```
Benchmark                                  Mode  Cnt    Score    Error  Units
ParallelStreamBenchmark.parallelSum        avgt   40  604.059 ± 55.288  ms/op
```

Весьма неутешительный результат: параллельная версия метода суммирования не извлекает никакой выгоды из четырехъядерного CPU и работает почти в пять раз медленнее последовательной. Чем объясняется такой неожиданный результат? Двумя факторами:

- метод `iterate` генерирует упакованные объекты, которые приходится распаковывать в обычные числа, прежде чем складывать;
- операция `iterate` плохо поддается разбиению на отдельные порции для параллельного выполнения.

Особый интерес представляет второй фактор, ведь нужно не забывать: некоторые потоковые операции лучше поддаются распараллеливанию, чем другие. А именно,

наша операция `iterate` плохо поддается разбиению на отдельные порции, которые можно было бы выполнять независимо друг от друга, поскольку результаты очередного применения этой функции всегда зависят от результата выполнения предыдущей, как показано на рис. 7.2.

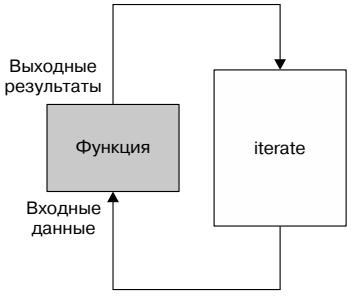


Рис. 7.2. Операция iterate по своей природе является последовательной

Это значит, что в данном частном случае процесс свертки не происходит так, как показано на рис. 7.1: полный список чисел в начале процесса свертки недоступен, из-за чего невозможно разумным образом разбить поток данных на порции для параллельной обработки. А указывая, что поток данных параллельный, мы привносим, по сравнению с последовательной обработкой, дополнительные накладные расходы в виде распределения операций суммирования по различным потокам выполнения.

Этот пример демонстрирует, насколько запутанным и порой неочевидным может быть параллельное программирование. При неправильном применении (скажем, использовании принципиально не распараллеливаемой операции вроде `iterate`) оно может даже ухудшить общую производительность программы, так что при вызове «волшебного» метода `parallel` необходимо четко понимать, что происходит за его кулисами.

Более специализированные методы. Как же эффективно использовать возможности многоядерных процессоров и потоков данных для параллельного суммирования? В главе 5 мы обсуждали метод `LongStream.rangeClosed`. У него есть два преимущества по сравнению с `iterate`.

- Метод `LongStream.rangeClosed` работает непосредственно с простым типом данных `long`, так что не требует дополнительных расходов ресурсов на упаковку и распаковку.
- Метод `LongStream.rangeClosed` генерирует диапазоны целых чисел, легко разбиваемые на отдельные порции. Например, диапазон 1–20 можно разбить на порции 1–5, 6–10, 11–15 и 16–20.

Взглянем сначала на его быстродействие в случае последовательного потока данных, добавив новый метод в наш класс тестирования производительности для проверки того, насколько существенны накладные расходы распаковки:

```
@Benchmark
public long rangedSum() {
    return LongStream.rangeClosed(1, N)
                     .reduce(0L, Long::sum);
}
```

Результаты:

```
Benchmark                                  Mode  Cnt    Score    Error  Units
ParallelStreamBenchmark.rangedSum          avgt   40    5.315 ±  0.285  ms/op
```

Этот специализированный числовой поток отработал намного быстрее, чем предыдущая последовательная версия, сгенерированная с помощью фабричного метода `iterate`, благодаря отсутствию накладных расходов на ненужные автоупаковку и автораспаковку. Это свидетельствует о том, что выбор правильных структур данных зачастую более важен, чем распараллеливание алгоритма, в котором они используются. Но что будет, если запустить следующую параллельную версию потока?

```
@Benchmark
public long parallelRangedSum() {
    return LongStream.rangeClosed(1, N)
                     .parallel()
                     .reduce(0L, Long::sum);
}
```

При добавлении этого метода в класс теста производительности мы получили следующие результаты:

```
Benchmark                                    Mode  Cnt  Score    Error  Units
ParallelStreamBenchmark.parallelRangedSum    avgt   40  2.677 ±  0.214  ms/op
```

Наконец-то мы создали алгоритм параллельной свертки, более быстрый, чем его последовательный аналог, поскольку теперь операция свертки выполняется так, как показано на рис. 7.1. Это также демонстрирует, что использование правильной структуры данных в параллельном режиме гарантирует максимальное быстродействие. Отметим также, что эта параллельная версия примерно на 20 % быстрее исходной итеративной, так что при должном применении функциональный стиль программирования упрощает использование параллелизма современных многоядерных процессоров, по сравнению с императивным аналогом.

Тем не менее помните, что распараллеливание имеет свою цену. Процесс распараллеливания требует рекурсивного разбиения потока данных, распределения операций свертки по различным потокам выполнения и последующего объединения результатов этих операций в единое значение. Но перемещение данных между различными ядрами процессора требует больше ресурсов, чем вы можете предположить, так что выполняемые параллельно на другом ядре вычисления должны занимать больше времени, чем перемещение данных с одного ядра на другое. Вообще говоря, во многих сценариях невозможно или не имеет смысла задействовать параллелизм. И, прежде чем воспользоваться параллельным потоком данных для ускорения работы кода, нужно убедиться в правильности логики; нет смысла в быстром получении результатов, если они неправильны. Рассмотрим один из распространенных подводных камней такого подхода.

7.1.3. Правильное применение параллельных потоков данных

Основная причина ошибок, возникающих вследствие неправильного применения параллельных потоков данных, — применение алгоритмов, меняющих какое-либо совместно используемое состояние. Ниже приведен вариант вычисления суммы первых *n* натуральных чисел с помощью изменения совместно используемого накопителя:

```
public long sideEffectSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).forEach(accumulator::add);
    return accumulator.total;
}
public class Accumulator {
    public long total = 0;
    public void add(long value) { total += value; }
}
```

Подобный код пишут достаточно часто, особенно разработчики, знакомые с парадигмами императивного программирования. Он очень похож на императивную обработку в цикле списка чисел: инициализацию накопителя с последующим обходом элементов списка по очереди, с прибавлением каждого из них к накопителю.

Что же не так с этим кодом? К сожалению, он совершенно безнадежен, поскольку по своей сути является последовательным. При каждом обращении к `total` будет возникать состояние гонки по данным. А если попытаться исправить эту ситуацию с помощью синхронизации, то потеряется весь параллелизм. Чтобы лучше разобраться с этим, попробуем преобразовать поток в параллельный:

```
public long sideEffectParallelSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);
    return accumulator.total;
}
```

Запустим этот последний метод с помощью тестовой оснастки из листинга 7.1, выводя результаты каждого из выполнений:

```
System.out.println("SideEffect parallel sum done in: " +
    measurePerf(ParallelStreams::sideEffectParallelSum, 10_000_000L) + " msecs" );
```

При этом вы получите что-то вроде:

```
Result: 5959989000692
Result: 7425264100768
Result: 6827235020033
Result: 7192970417739
Result: 6714157975331
Result: 7497810541907
Result: 6435348440385
Result: 6999349840672
Result: 7435914379978
Result: 7715125932481
SideEffect parallel sum done in: 49 msecs
```

На этот раз производительность метода не имеет значения. Важно лишь то, что при каждом выполнении возвращается другой результат, причем все эти результаты весьма далеки от правильного значения `50000005000000`. Причина этого состоит в конкурентном обращении нескольких потоков выполнения к одному накопителю и, в частности, в выполнении оператора `total += value`, который, вопреки первому впечатлению, отнюдь не атомарный. Источник проблемы — побочный эффект метода, вызываемого внутри блока `forEach`, в виде изменения состояния объекта, совместно используемого в нескольких потоках выполнения. Избегайте подобных ситуаций, если не хотите сталкиваться с неприятными неожиданностями при использовании параллельных потоков данных!

Итак, вы уже знаете, что разделяемое изменяемое состояние плохо сочетается с параллельными потоками данных, да и вообще с параллельными вычислениями. Мы вернемся к вопросу о том, как избежать изменения состояния, в главах 18 и 19, когда будем подробнее обсуждать функциональное программирование. Пока просто запомните: избегайте совместного использования изменяемого состояния, и ваш параллельный поток данных вернет правильный результат. Далее мы рассмотрим несколько практических советов относительно того, в каких случаях параллельные потоки данных позволяют повысить производительность.

7.1.4. Эффективное использование параллельных потоков данных

Вообще говоря, невозможно (да и не имеет смысла) привести количественные критерии того, когда следует использовать параллельные потоки данных, ведь любой конкретный критерий вида «только для потоков данных более чем из тысячи элементов» подойдет лишь для конкретной операции на конкретной машине, но окажется совершенно неправильным при работе в принципиально другом контексте. Тем не менее можно по крайней мере дать некоторые качественные советы, полезные для принятия решения о целесообразности использования параллельного потока данных в конкретной ситуации.

- Если сомневаетесь — измеряйте. Превратить последовательный поток данных в параллельный очень просто, но далеко не всегда имеет смысл это делать. Как мы уже продемонстрировали в этом разделе, параллельный поток данных не всегда работает быстрее своего последовательного аналога. Более того, параллельные потоки данных иногда работают неожиданным образом, так что важнейшая рекомендация при выборе между последовательным и параллельным потоками данных — всегда проверять их быстродействие с помощью соответствующего теста производительности.
- Остерегайтесь упаковки. Автоматические операции упаковки и распаковки могут значительно снизить производительность. Чтобы можно было избежать подобных операций, в Java 8 предусмотрены версии потоков для простых типов данных (`IntStream`, `LongStream` и `DoubleStream`), которые следует использовать везде, где только возможно.
- Некоторые операции по своей природе работают на параллельных потоках данных хуже, чем на последовательных. В частности, значительных ресурсов требует выполнение для параллельных потоков данных таких операций, как `limit` и `findFirst`, зависящих от порядка элементов. Например, операция `findAny` будет

работать быстрее, чем `findFirst`, поскольку не обязана анализировать элементы в порядке обнаружения. Упорядоченный поток данных всегда можно преобразовать в неупорядоченный, вызвав для него метод `unordered`. Например, с целью получения N элементов потока, причем не обязательно *первых* N элементов, эффективнее будет вызвать метод `limit` для неупорядоченного параллельного потока, а не потока с заданным порядком обнаружения (такого как, например, в случае, когда источник представляет собой список).

- Оцените общие вычислительные затраты конвейера операций над потоком. Если обозначить N число обрабатываемых элементов, а Q — приблизительные затраты на обработку одного элемента в рамках конвейера потока, то суммарные затраты можно приближенно оценить как произведение $N \cdot Q$. Чем выше значение Q, тем больше шансов на хорошую производительность при использовании параллельного потока данных.
- При небольших объемах данных выбирать параллельный поток данных почти никогда не имеет смысла. Преимущества параллельной обработки недостаточны, чтобы перевесить дополнительные затраты на процесс распараллеливания.
- Необходимо учитывать, насколько хорошо допускает разбиение на порции структура данных, лежащая в основе потока. Например, список `ArrayList` разбивается намного эффективнее, чем `LinkedList`, поскольку первый можно разбить на равные порции без его обхода, в отличие от второго. Кроме того, легко разбиваются на порции специализированные версии потоков для простых типов данных, созданные с помощью фабричного метода `range`. Наконец, как вы узнаете из раздела 7.3, управлять всеми нюансами этого процесса разбиения можно с помощью создания своего собственного сплитератора.
- Характеристики потока данных, а также модификация этих характеристик промежуточными операциями на протяжении конвейера могут менять производительность процесса разбиения. Например, поток с характеристикой `SIZED` можно разбить на две равные части, которые более эффективно — обработать параллельно, но при фильтрации может оказаться отброшено непредсказуемое число элементов, вследствие чего размер потока окажется неизвестным.
- Следует учесть, насколько велики затраты на шаг объединения в завершающей операции (например, метод `combiner` коллектора). Если этот шаг дорогостоящий, то затраты на объединение промежуточных результатов отдельных потоков могут перевесить выгоды параллельного потока данных в смысле производительности.

В табл. 7.1 приведена краткая сводка степеней дружественности к параллелизму различных источников потоков данных в смысле возможностей их разбиения.

Наконец, необходимо подчеркнуть, что роль инфраструктуры, применяемой во внутренних механизмах параллельных потоков данных для параллельного выполнения операций, играет появившийся в Java 7 фреймворк ветвления-объединения. Пример параллельного суммирования подтверждает, что для правильного использования параллельных потоков данных нужно хорошо представлять себе их внутреннее устройство, так что в следующем разделе мы подробно изучим фреймворк ветвления-объединения.

Таблица 7.1. Пригодность различных источников потоков данных к разбиению

Источник	Пригодность к разбиению
ArrayList	Превосходная
LinkedList	Плохая
IntStream.range	Превосходная
Stream.iterate	Плохая
HashSet	Хорошая
TreeSet	Хорошая

7.2. Фреймворк ветвления-объединения

Назначение фреймворка ветвления-объединения состоит в рекурсивном разбиении параллелизуемой задачи на меньшие задачи с последующим объединением результатов работы всех этих подзадач в общий результат. Он представляет собой реализацию интерфейса `ExecutorService` и распределяет эти подзадачи по потокам-исполнителям из пула потоков выполнения — `ForkJoinPool`. Начнем с описания задачи и подзадач.

7.2.1. Работа с классом RecursiveTask

Для отправки задачи в этот пул необходимо создать подкласс класса `RecursiveTask<R>`, где `R` — тип возвращаемого распараллеленной задачей (и всеми ее подзадачами) результата, или `RecursiveAction`, если задача ничего не возвращает (обновляет данные в других, не локальных, структурах). Для описания класса `RecursiveTask` достаточно реализовать его единственный абстрактный метод `compute`:

```
protected abstract R compute();
```

Этот метод задает как логику разбиения задачи на подзадачи, так и алгоритм получения результата отдельной задачи в случае, когда дальнейшее разбиение невозможно или не имеет смысла. Поэтому реализация данного метода часто напоминает следующий псевдокод:

```
if (задача достаточно мала, или ее нельзя далее делить) {
    произвести необходимые для задачи вычисления последовательным образом
} else {
    разбить задачу на две подзадачи
    вызвать текущий метод рекурсивно, возможно далее разбивая подзадачи
    ожидать завершения всех подзадач
    объединить результаты выполнения всех подзадач
}
```

Вообще говоря, не существует никакого точного критерия того, следует ли делить конкретную задачу; есть лишь различные эвристические алгоритмы, облегчающие принятие подобного решения. Мы поговорим о них подробнее в подразделе 7.2.2. Рекурсивный процесс разбиения задач визуализирован на рис. 7.3.

Как вы, наверное, обратили внимание, это лишь известный принцип «разделяй и властвуй» применительно к параллельному программированию. Для демонстрации практического примера использования фреймворка ветвления-

объединения на основе наших предыдущих примеров попробуем с его помощью вычислить сумму диапазона чисел (в данном случае представленного в виде массива целых чисел `long[]`). Как уже объяснялось, сначала необходимо создать реализацию для класса `RecursiveTask` — класс `ForkJoinSumCalculator`, приведенный в листинге 7.2.

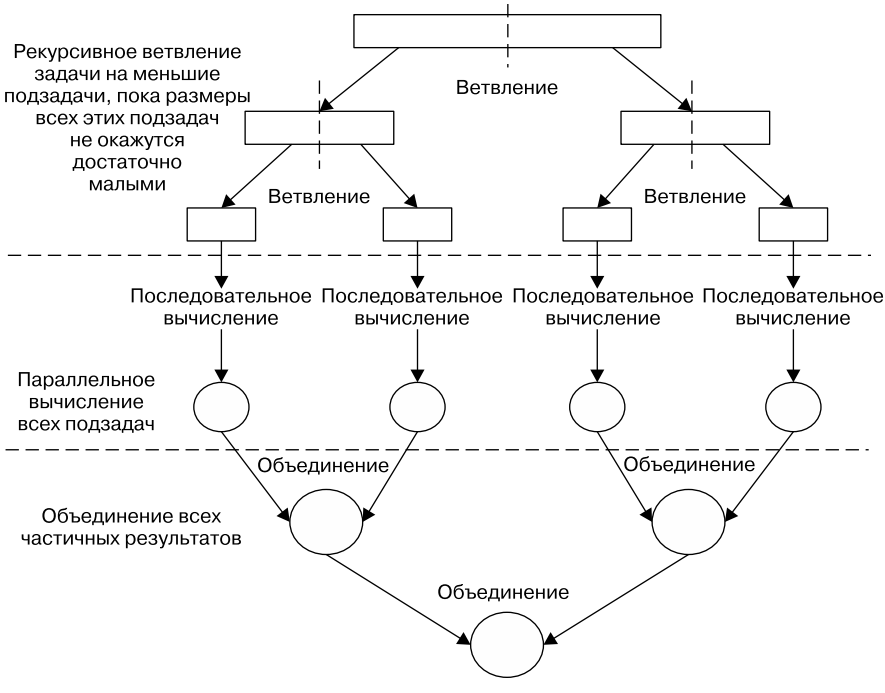


Рис. 7.3. Процесс ветвления-объединения

Листинг 7.2. Параллельное суммирование с помощью фреймворка ветвления-объединения

```
public class ForkJoinSumCalculator
                extends java.util.concurrent.RecursiveTask<Long> {
    private final long[] numbers;
    private final int start;
    private final int end;
    public static final long THRESHOLD = 10_000;
    public ForkJoinSumCalculator(long[] numbers) {
        this(numbers, 0, numbers.length);
    }
    private ForkJoinSumCalculator(long[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
```

Начальный и конечный индекс обрабатываемого этой подзадачей подмассива

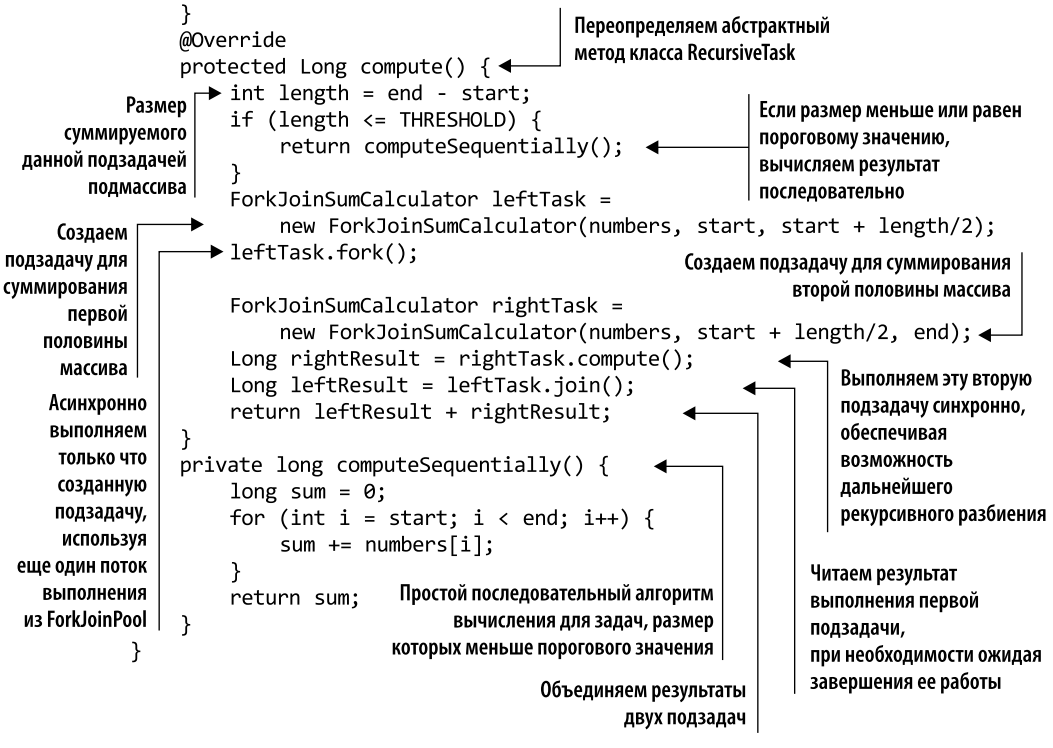
Расширяем класс RecursiveTask, создавая задачу, пригодную для работы с фреймворком ветвления-объединения

Массив суммируемых чисел

Пороговое значение размера для разбиения на подзадачи

Общедоступный конструктор для создания основной задачи

Приватный конструктор для создания подзадач основной задачи



Теперь написание метода для параллельного суммирования первых *n* натуральных чисел становится тривиальной задачей. Необходимо передать нужный массив конструктору класса `ForkJoinSumCalculator`:

```
public static long forkJoinSum(long n) {
    long[] numbers = LongStream.rangeClosed(1, n).toArray();
    ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);
    return new ForkJoinPool().invoke(task);
}
```

В этом фрагменте кода с помощью `LongStream` мы генерируем массив, содержащий первые *n* натуральных чисел. Далее мы создаем объект `ForkJoinTask` (суперкласса `RecursiveTask`), передавая вышеупомянутый массив общедоступному конструктору класса `ForkJoinSumCalculator`, приведенного в листинге 7.2. Наконец мы создаем новый `ForkJoinPool` и передаем эту задачу в его метод `invoke`. Возвращаемое последним методом значение представляет собой результат выполнения (внутри `ForkJoinPool`) задачи, задаваемой `ForkJoinSumCalculator`.

Отметим, что в настоящих приложениях не имеет смысла использовать более одного объекта `ForkJoinPool`. Поэтому обычно создают только один экземпляр данного класса и хранят ссылку на него в статическом поле, превращая его, таким образом, в объект-одиночку, благодаря чему его можно переиспользовать в любой части вашего программного обеспечения. Для его создания мы здесь используем конструктор по

умолчанию без аргументов, а значит, разрешаем пулу задействовать все доступные JVM процессоры. Точнее говоря, для определения количества используемых пулом потоков выполнения этот конструктор берет значение, возвращаемое методом `Runtime.availableProcessors`. Отметим, что метод `Runtime.availableProcessors` вопреки названию возвращает число доступных *ядер* процессоров, включая виртуальные (обеспечиваемые технологией Hyper-Threading).

Выполнение задачи ForkJoinSumCalculator. Передаваемая `ForkJoinPool` задача `ForkJoinSumCalculator` выполняется одним из его потоков выполнения, который, в свою очередь, вызывает метод `compute` этой задачи. В этом методе проверяется, достаточно ли задача мала для последовательного выполнения; если нет, массив суммируемых чисел разбивается пополам, а половины распределяются по двум новым объектам `ForkJoinSumCalculator` для дальнейшего выполнения с помощью `ForkJoinPool`. В результате процесс повторяется рекурсивно, разбивая исходную задачу на все меньшие подзадачи до тех пор, пока не перестанет выполняться условие дальнейшего разбиения (в данном случае количество суммируемых чисел меньше или равно 10 000). После этого результаты всех подзадач вычисляются последовательно и производится обход созданного в результате ветвления (неявного) бинарного дерева задач — от листьев к корню. Далее частичные результаты подзадач объединяются в единый результат. Этот процесс показан на рис. 7.4.

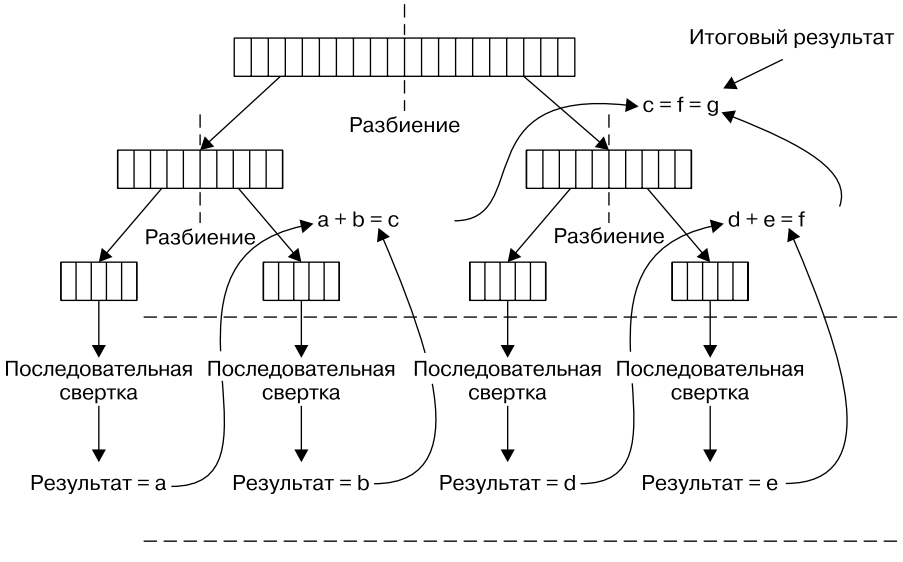


Рис. 7.4. Алгоритм ветвления-объединения

Как и ранее, проверить производительность метода суммирования, явно использующего фреймворк ветвления-объединения, можно с помощью разработанной в начале этой главы оснастки:

```
System.out.println("ForkJoin sum done in: " + measureSumPerf(
        ForkJoinSumCalculator::forkJoinSum, 10_000_000) + " msecs" );
```

В данном случае получаем следующий результат:

```
ForkJoin sum done in: 41 msecs
```

Производительность оказывается хуже, чем у версии с параллельным потоком данных, но лишь потому, что мы вынуждены поместить весь поток чисел в массив `long[]` перед его использованием в задаче `ForkJoinSumCalculator`.

7.2.2. Рекомендуемые практики применения фреймворка ветвления-объединения

Хотя использовать фреймворк ветвления-объединения довольно легко, его столь же легко использовать неправильно. Вот несколько рекомендуемых практик его эффективного применения.

- Вызов для задачи метода `join` блокирует вызывающую программу вплоть до момента готовности результата. Поэтому вызывать его необходимо после начала вычисления обеих подзадач. В противном случае ваш алгоритм превратится в более медленную и запутанную версию нашего первоначального последовательного алгоритма, ведь каждая из задач, прежде чем начать работать, должна будет дождаться завершения предыдущей.
- Метод `invoke` класса `ForkJoinPool` не следует использовать внутри `RecursiveTask`. Вместо него всегда нужно вызывать непосредственно методы `compute` и `fork`; `invoke` должен использоваться лишь в последовательном коде для запуска параллельных вычислений.
- Чтобы запланировать выполнение подзадачи в `ForkJoinPool`, следует вызывать для нее метод `fork`. Может показаться логичным вызвать его как для правой, так и для левой подзадачи, но эффективнее будет вызвать для одной из них метод `compute`. Это позволит переиспользовать один и тот же поток выполнения для одной из двух подзадач и избежать затрат на выделение пулом дополнительных ресурсов.
- Отладка кода для параллельных вычислений, использующих фреймворк ветвления-объединения, может оказаться непростой задачей. В частности, для разработчиков вполне обычным делом является просмотр стека трассы вызовов в своем любимом IDE, чтобы обнаружить источник проблемы. Но при вычислениях типа «ветвление-объединение» этот способ не сработает, ведь вызов метода `compute` происходит в потоке выполнения, отличном от потока вызвавшего `fork` кода.
- Как мы уже видели в случае с параллельными потоками данных, не стоит считать самой собой разумеющейся более высокую скорость работы вычислений с помощью фреймворка ветвления-объединения на многоядерном процессоре, по сравнению с его последовательным аналогом. Мы уже говорили, что для возможности распараллеливания задачи (и получения соответствующих преимуществ в смысле производительности) она должна быть подходящей для разбиения на несколько независимых подзадач. Выполнение каждой из этих подзадач должно занимать больше времени, чем ветвление новой задачи; один из вариантов состоит в том, чтобы поместить операции ввода/вывода в одну подзадачу, а вычисления — в другую, таким образом совмещая вычисления с операциями ввода/вывода. Более

того, важно учитывать и другие факторы при сравнении производительности последовательной и параллельной версий одного алгоритма. Как и любой другой код на языке Java, фреймворк ветвления-объединения необходимо немного «разогреть» — выполнить несколько раз, — чтобы компилятор JIT его оптимизировал. Именно поэтому перед измерением быстродействия программы требуется запустить ее несколько раз, как и было сделано в нашей тестовой оснастке. Учтите также, что из-за встроенных оптимизаций компилятора последовательная версия может получить незаслуженное преимущество (например, за счет анализа исполняемости кода — удаления никогда не используемых фрагментов).

Еще одно, последнее замечание относительно стратегии разбиения по типу «ветвление-объединение»: вам необходимо выбрать критерий, определяющий, нужно ли далее разбивать подзадачу, или ее размеры уже достаточно малы для последовательного выполнения. Мы дадим несколько советов по его выбору в следующем подразделе.

7.2.3. Перехват работы

В примере с `ForkJoinSumCalculator` мы решили прекратить создание дальнейших подзадач, когда массивы суммируемых чисел будут содержать не более 10 000 элементов. Это значение было взято с потолка, но в большинстве случаев единственный способ найти хороший эвристический алгоритм — провести несколько экспериментов с различными входными данными. В нашем тестовом сценарии мы начали с массива из 10 миллионов элементов, так что `ForkJoinSumCalculator` придется создать как минимум 1000 подзадач. Это может показаться бессмысленным разбазариванием ресурсов, поскольку будет выполняться на машине всего с четырьмя ядрами процессора. В данном конкретном случае это справедливо, поскольку предполагается, что все задачи займут примерно одинаковое время и все они ограничиваются возможностями процессора.

Однако в общем случае ветвление задачи на большое число мелких подзадач — выигрышная стратегия, поскольку в идеальном варианте вычислительная нагрузка параллелизуемой задачи должна разбиваться так, чтобы все подзадачи занимали одинаковое время, а значит, все ядра процессора были одинаково загружены. К сожалению, занимаемое подзадачами время может сильно варьироваться вследствие как неэффективной стратегии разбиения, так и непредсказуемых причин, таких как медленный доступ к диску или необходимость согласования выполнения с внешними сервисами. Особенно сильно это проявляется в сценариях, более близких к реальности, чем приведенный нами простой пример.

Фреймворк ветвления-объединения обходит указанную проблему с помощью так называемого *перехвата работы* (work stealing). На практике он означает более или менее равномерное распределение задач между потоками выполнения из `ForkJoinPool`. У каждого из потоков выполнения есть двусвязная очередь назначенных ему задач, и по завершении очередной задачи поток извлекает из головы очереди следующую и начинает ее выполнять. По вышеупомянутым причинам один поток выполнения может завершить все назначенные ему задачи намного быстрее остальных потоков, в результате чего его очередь опустеет, в то время как в остальных очередях еще останется немало задач. В подобном случае, вместо того чтобы простаивать, этот поток выполнения случайным образом выбирает очередь одного из других

потоков выполнения и «перехватывает» у него задачу, находящуюся в конце очереди. Этот процесс продолжается вплоть до завершения всех задач и опустения всех очередей. Именно поэтому большое количество маленьких задач (вместо нескольких больших) помогает лучше распределить нагрузку между потоками-исполнителями.

В общем случае такой алгоритм перехвата работы применяется для перераспределения и балансировки задач между потоками-исполнителями в пуле. На рис. 7.5 показан этот процесс. При разбиении задачи из очереди на две подзадачи одна из них перехватывается другим простаивающим исполнителем. Как уже описывалось ранее, такой процесс может продолжаться рекурсивно до тех пор, пока не станет истинным условие последовательного выполнения подзадачи.

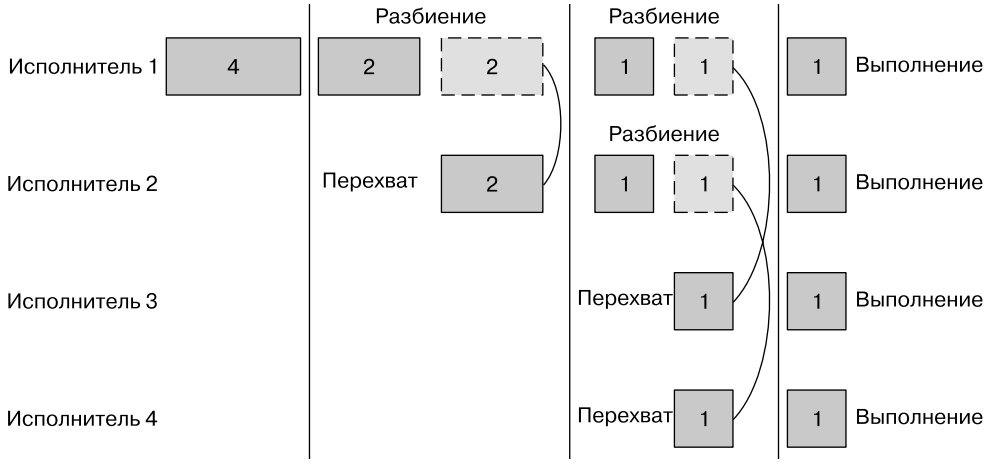


Рис. 7.5. Алгоритм перехвата работы, используемый фреймворком ветвления-объединения

Вам должно быть уже ясно, каким образом поток данных должен использовать фреймворк ветвления-объединения для параллельной обработки своих элементов, но одного ингредиента в нашем блюде еще недостает. В этом разделе мы разбирали пример, в котором явным образом создавали логику разбиения массива чисел на несколько задач. Тем не менее при использовании параллельных потоков данных в начале данной главы нам не приходилось делать ничего подобного, а значит, существует автоматический механизм разбиения потока данных. Этот новый автоматический механизм — интерфейс `Spliterator`, и мы изучим его в следующем разделе.

7.3. Интерфейс Spliterator

`Spliterator` — еще один добавленный в Java 8 интерфейс; его название означает «разбиваемый итератор» (splittable iterator). Подобно обычным итераторам, сплитераторы предназначены для обхода элементов источника данных, причем в параллельном режиме. Хотя, возможно, вам и не придется разрабатывать свой собственный сплитератор, понимание того, как это делается, позволит лучше разобраться, как вообще работают параллельные потоки данных. В Java 8 есть реализации интерфейса `Spliterator`

по умолчанию для всех структур данных из фреймворка коллекций. В интерфейсе `Collection` появился метод с реализацией по умолчанию `spliterator()` (больше о методах с реализацией по умолчанию мы расскажем в главе 13), возвращающий объект типа `Spliterator`. В интерфейсе `Spliterator` определено несколько методов, как показано в листинге 7.3.

Листинг 7.3. Интерфейс Spliterator

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

Как обычно, `T` — тип обрабатываемых сплитератором элементов. Метод `tryAdvance` ведет себя аналогично обычному итератору в том смысле, что используется для последовательного потребления элементов сплитератора по очереди; он возвращает `true`, если обработаны еще не все элементы. Но для интерфейса `Spliterator` более специфичен метод `trySplit`, используемый для выделения части элементов во второй сплитератор (возвращаемый этим методом) и обработки обоих параллельно. Сплитератор может также оценивать количество оставшихся элементов посредством `estimateSize`, поскольку даже неточное, но не требующее длительных вычислений значение может помочь более равномерно разбить структуру.

Важно также понимать внутренний механизм процесса разбиения, чтобы реализовать его самим при необходимости. Поэтому в следующем подразделе мы проанализируем его подробнее.

7.3.1. Процесс разбиения

Алгоритм разбиения потока данных на несколько частей представляет собой рекурсивный процесс и происходит так, как показано на рис. 7.6. На первом шаге для первого сплитератора вызывается метод `trySplit`, который создает второй сплитератор. Далее, на втором шаге, он вызывается снова для этих двух сплитераторов, в результате чего всего их получается четыре. Фреймворк продолжает вызывать для сплитераторов метод `trySplit` до тех пор, пока он не вернет `null` как признак того, что обрабатываемая структура данных более не поддается разбиению (это показано на рис. 7.6 в шаге 3). Наконец, этот рекурсивный процесс разбиения завершается на шаге 4, когда вызов метода `trySplit` для всех сплитераторов возвращает `null`.

На процесс разбиения также могут влиять характеристики самого сплитератора, объявляемые с помощью метода `characteristics`.

Характеристики сплитератора. Последний из объявленных в интерфейсе `Spliterator` абстрактных методов — `characteristics`, возвращающий значение типа `int`, кодирующее набор характеристик самого сплитератора. Благодаря этим характеристикам клиенты сплитератора могут лучше контролировать и оптимизировать его использование. Их список приведен в табл. 7.2 (к сожалению, хотя они, по существу, и пересекаются с характеристиками коллекторов, но обозначаются иначе). Характеристики представляют собой константы типа `int`, описанные в интерфейсе `Spliterator`.

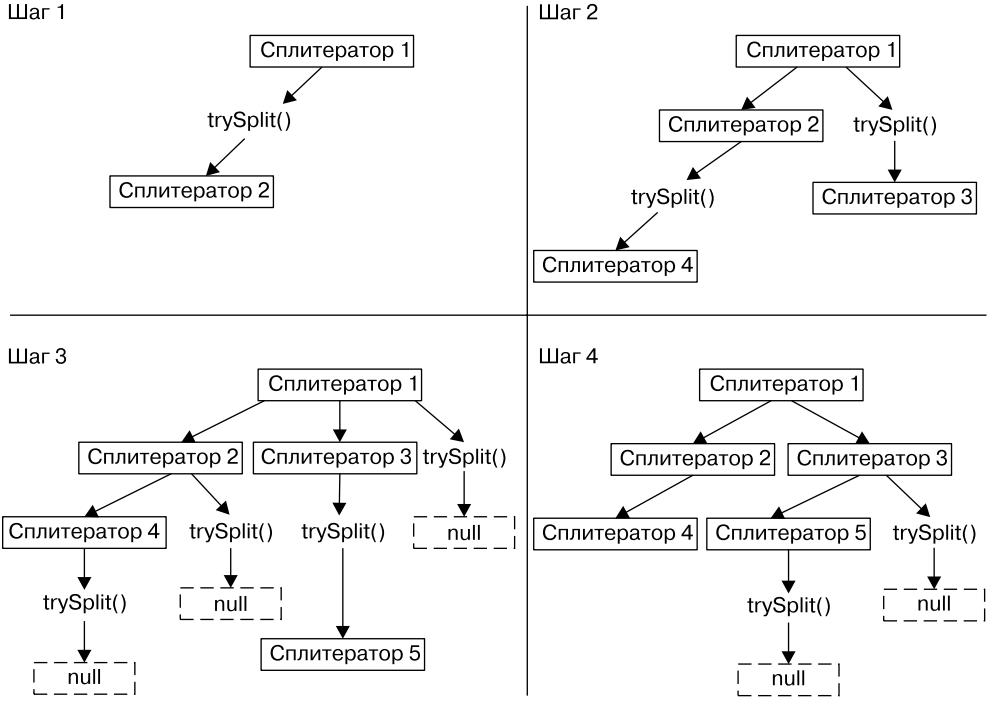


Рис. 7.6. Рекурсивный процесс разбиения

Таблица 7.2. Характеристики сплитераторов

Характеристика	Значение
ORDERED	Для элементов задан определенный порядок (например, источником является список), так что сплитератор должен учитывать его при их обходе и разбиении
DISTINCT	Для любой пары обрабатываемых элементов x и y вызов x.equals(y) возвращает false
SORTED	Обрабатываемые элементы следуют определенному порядку сортировки
SIZED	Сплитератор был создан на основе источника с известным размером (например, объекта Set), так что метод estimatedSize() возвращает точное значение
NON-NULL	Гарантируется, что обрабатываемые элементы не пусты (не равны null)
IMMUTABLE	Источник данного сплитератора не допускает изменения. Это значит, что во время его обхода нельзя добавлять, удалять или модифицировать элементы
CONCURRENT	Другие потоки выполнения могут безопасно конкурентно модифицировать источник данного сплитератора без какой-либо синхронизации
SUBSIZED	Как этот сплитератор, так и все будущие сплитераторы, полученные в результате его разбиения, обладают характеристикой SIZED

Теперь, когда мы уже знаем, что представляет собой интерфейс `Spliterator` и какие методы в нем описаны, можно попробовать создать собственную реализацию сплитератора.

7.3.2. Реализация собственного сплитератора

Рассмотрим пример реальной ситуации, в которой может понадобиться реализовать свой собственный сплитератор. Мы создадим простой метод для подсчета количества слов в объекте `String`. Итеративная версия подобного метода может выглядеть так, как показано в листинге 7.4.

Листинг 7.4. Итеративный метод для подсчета количества слов

```
public int countWordsIteratively(String s) {
    int counter = 0;
    boolean lastSpace = true;
    for (char c : s.toCharArray()) {
        if (Character.isWhitespace(c)) {
            lastSpace = true;
        } else {
            if (lastSpace) counter++;
            lastSpace = false;
        }
    }
    return counter;
}
```

Обход всех символов в строке одного за другим

Увеличение счетчика слов в случае, когда предыдущий символ представляет собой пробел, а текущий — нет

Проверим работу этого метода на первом предложении части «Ад» из «Божественной комедии» Данте (см. http://en.wikipedia.org/wiki/Inferno_(Dante) и https://ru.wikipedia.org/wiki/Ад_(Божественная_комедия)):

```
final String SENTENCE =
            " Nel   mezzo del cammin  di nostra  vita " +
            "mi  ritrovai in una  selva oscura" +
            " ché la  dritta via era   smarrita ";
System.out.println("Найдено " + countWordsIteratively(SENTENCE) + " слов");
```

Обратите внимание, что мы добавили в предложение несколько дополнительных пробелов в произвольных местах, чтобы продемонстрировать, что итеративная реализация работает правильно даже в случае лишних пробелов между соседними словами. Как и ожидалось, данный код выводит следующий результат:

```
Найдено 19 слов
```

Нам хотелось бы добиться того же результата при более функциональном стиле кода, поскольку это позволило бы нам, как было показано выше, распараллелить процесс вычислений с помощью параллельного потока данных, причем без необходимости явным образом заниматься потоками выполнения и их синхронизацией.

Переписываем счетчик слов в функциональном стиле

Во-первых, нужно преобразовать объект `String` в поток. К сожалению, специализированные потоки данных существуют только для типов `int`, `long` и `double`, так что нам придется воспользоваться типом `Stream<Character>`:

```
Stream<Character> stream = IntStream.range(0, SENTENCE.length())
                                    .mapToObj(SENTENCE::charAt);
```

Для подсчета количества слов можно воспользоваться операцией свертки этого потока данных. В процессе свертки нам понадобится состояние, включающее две переменные: переменную типа `int` для хранения числа найденных на этот момент слов и переменную типа `boolean` для запоминания того, был ли последний из обнаруженных объектов `Character` пробелом. Поскольку в языке Java нет кортежей (конструкций, отражающих упорядоченный список разнородных элементов без объекта-адаптера), придется создать новый класс, `WordCounter`, для инкапсуляции этого состояния, как показано в листинге 7.5.

Листинг 7.5. Класс для подсчета количества слов во время обхода потока символов

```
class WordCounter {
    private final int counter;
    private final boolean lastSpace;
    public WordCounter(int counter, boolean lastSpace) {
        this.counter = counter;
        this.lastSpace = lastSpace;
    }
    public WordCounter accumulate(Character c) {
        if (Character.isWhitespace(c)) {
            return lastSpace ?
                   this :
                   new WordCounter(counter, true);
        } else {
            return lastSpace ?
                   new WordCounter(counter+1, false) :
                   this;
        }
    }
    public WordCounter combine(WordCounter wordCounter) {
        return new WordCounter(counter + wordCounter.counter,
                               wordCounter.lastSpace);
    }
    public int getCounter() {
        return counter;
    }
}
```

Метод accumulate обходит объекты Character по одному, аналогично итеративному алгоритму

Увеличиваем значение счетчика слов на единицу, если предыдущий из символов был пробелом, а текущий символ — нет

Объединяет два объекта WordCounter путем суммирования их счетчиков

Используется только сумма счетчиков, так что значение lastSpace нас не волнует

В этом листинге метод `accumulate` определяет способ изменения состояния `WordCounter`, точнее, устанавливает, на основе какого состояния создать новый `WordCounter`, поскольку данный класс — неизменяемый. Это важный для понимания нюанс. Мы накапливаем состояние, используя неизменяемый класс, чтобы можно было распараллелить процесс на следующем шаге. Метод `accumulate` вызывается при обработке каждого нового символа из потока. В частности, аналогично методу `countWordsIteratively` из листинга 7.4 счетчик наращивается, когда обнаруживается новый непробельный символ, причем предыдущий символ был пробелом. Рисунок 7.7 иллюстрирует переходы `WordCounter` из одного состояния в другое при обработке нового символа методом `accumulate`.

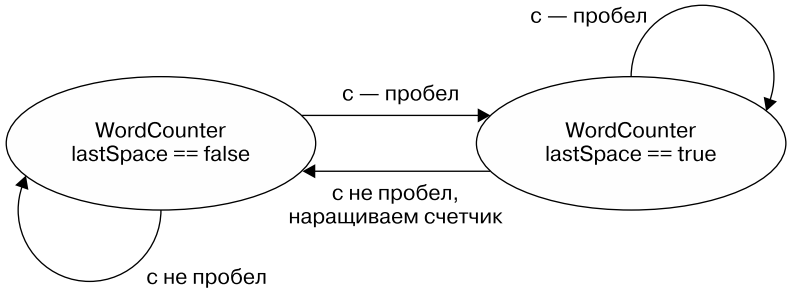


Рис. 7.7. Переходы WordCounter из одного состояния в другое при обработке нового символа

Второй метод, `combine`, вызывается для агрегирования частичных результатов двух `WordCounter`, обрабатывающих две различные части потока символов. Он объединяет два `WordCounter` путем суммирования их внутренних счетчиков.

Теперь, после завершения написания логики накопления счетчиков `WordCounter` и их объединения в самом `WordCounter`, написание метода для свертки потока символов не составляет труда:

```
private int countWords(Stream<Character> stream) {
    WordCounter wordCounter = stream.reduce(new WordCounter(0, true),
                                            WordCounter::accumulate,
                                            WordCounter::combine);
    return wordCounter.getCounter();
}
```

Проверим этот метод на потоке данных, созданном на основе строки, содержащей первое предложение из «Ада» Данте:

```
Stream<Character> stream = IntStream.range(0, SENTENCE.length())
                                    .mapToObj(SENTENCE::charAt);
System.out.println("Найдено " + countWords(stream) + " слов");
```

Можете убедиться, что результаты соответствуют результатам итеративной версии алгоритма:

```
Найдено 19 слов
```

Пока все хорошо, но, как мы говорили, одна из основных причин реализации `WordCounter` в функциональном стиле — возможность удобного распараллеливания данной операции, так что посмотрим, как это происходит.

Распараллеливаем выполнение WordCounter

Вы можете попытаться ускорить подсчет слов с помощью параллельного потока данных, вот так:

```
System.out.println("Найдено " + countWords(stream.parallel()) + " слов");
```

К сожалению, на этот раз результат выглядит так:

```
Найдено 25 слов
```

Явно что-то не так, но что именно? Найти причину этой проблемы несложно. Поскольку исходная строка разбивается в произвольных точках, иногда получается так, что одно слово разбивается на две части и впоследствии учитывается дважды. Вообще говоря, это демонстрирует тот факт, что переход от последовательного потока к параллельному может привести к ошибочному результату, если результат потенциально зависит от точек разбиения потока.

Как исправить эту проблему? Для этого необходимо гарантировать разбиение строки только в конце слов, вместо произвольных точек. Потребуется реализовать сплитератор символов, который разбивает строку только между двумя словами (как показано в листинге 7.6) и затем создает из нее параллельный поток данных.

Этот сплитератор, созданный на основе строки, в которой нужно подсчитать количество слов, проходит в цикле по ее символам, отслеживая индекс текущего обрабатываемого символа. Вкратце напомним методы класса `WordCounterSpliterator`, реализующего интерфейс `Spliterator`.

- Метод `tryAdvance` поставляет потребителю (объекту `Consumer`) очередной символ из строки, находящийся на соответствующем текущему значению индекса месте, и наращивает значение индекса на единицу. Передаваемый в качестве его аргумента объект типа `Consumer` — внутренний класс Java, направляющий прочитанный символ набору функций, которые должны быть применены к нему во время обхода потока. В данном случае они представляют собой лишь функцию свертки, а именно метод `accumulate` класса `WordCounter`. Метод `tryAdvance` возвращает `true`, если новая позиция курсора меньше общей длины строки и еще остались символы для обработки.
- Метод `trySplit` — важнейший метод итератора, поскольку именно в нем описывается логика разбиения обрабатываемой структуры данных. Подобно тому как мы делали в методе `compute` класса `RecursiveTask`, описанном в листинге 7.1, прежде всего необходимо задать пороговое значение, ниже которого не следует производить дальнейшие разбиения. В данном случае мы зададим небольшое пороговое значение, равное десяти символам, просто чтобы программа наверняка выполнила хотя бы несколько разбиений, с учетом относительно короткой анализируемой строки. Но в настоящих приложениях пороговое значение должно быть больше, как в нашем примере с ветвлением-объединением, во избежание слишком большого количества задач. Если число оставшихся символов меньше этого порогового значения, мы возвращаем `null` как признак того, что разбивать строку дальше не нужно. И напротив, если разбиение требуется, мы устанавливаем предварительную точку разбиения посередине оставшейся порции строки. Однако эта точка разбиения не используется непосредственно, во избежание разбиения посередине слова мы проходим по строке вперед, пока не найдем пробельный символ. После обнаружения подходящей точки разбиения мы создаем новый сплитератор для обхода части подстроки от текущей позиции до точки разбиения; текущая позиция `this` устанавливается равной точке разбиения, поскольку за предшествующую ей часть будет отвечать новый сплитератор, после чего мы возвращаем его.
- Метод `estimatedSize` (количество еще нуждающихся в обработке элементов) возвращает разницу между общей длиной обрабатываемой сплитератором строки и текущей позицией.

- Наконец, метод `characteristics` сообщает фреймворку, что данный сплитератор имеет следующие характеристики: `ORDERED` (порядок соответствует последовательности символов в строке), `SIZED` (метод `estimatedSize` возвращает точное значение), `SUBSIZED` (размер других созданных методом `trySplit` сплитераторов также точен), `NON-NULL` (в строке не может быть пустых символов) и `IMMUTABLE` (во время синтаксического разбора строки не может добавляться никаких новых символов, поскольку сам `String` — неизменяемый класс).

Листинг 7.6. Класс WordCounterSpliterator

```
class WordCounterSpliterator implements Spliterator<Character> {
    private final String string;
    private int currentChar = 0;
    public WordCounterSpliterator(String string) {
        this.string = string;
    }
    @Override
    public boolean tryAdvance(Consumer<? super Character> action) {
        action.accept(string.charAt(currentChar++));
        return currentChar < string.length();
    }
    @Override
    public Spliterator<Character> trySplit() {
        int currentSize = string.length() - currentChar;
        if (currentSize < 10) {
              return null;
        }
        for (int splitPos = currentSize / 2 + currentChar;
             splitPos < string.length(); splitPos++) {
            if (Character.isWhitespace(string.charAt(splitPos))) {
                Spliterator<Character> spliterator =
                   new WordCounterSpliterator(string.substring(currentChar,
                                                               splitPos));
                currentChar = splitPos;
                return spliterator;
            }
        }
        return null;
    }
    @Override
    public long estimateSize() {
        return string.length() - currentChar;
    }
    @Override
    public int characteristics() {
        return ORDERED + SIZED + SUBSIZED + NON-NULL + IMMUTABLE;
    }
```

Читает текущий символ

Возвращает true, если еще остались символы для чтения

Возвращает null, как указание, что строка уже достаточно мала для последовательной обработки

Сдвигает точку разбиения вперед, вплоть до следующего пробела

Задает предварительную точку разбиения строки посередине строки

Создает новый объект WordCounterSpliterator для синтаксического разбора строки, от ее начала и до точки разбиения

Устанавливает начальную точку текущего WordCounterSpliterator равной точке разбиения

Мы нашли пробел и создали новый сплитератор, так что покидаем цикл

Применяем класс WordCounterSpliterator на практике

Теперь мы можем воспользоваться параллельным потоком данных с помощью нового `WordCounterSpliterator` следующим образом:

```
Spliterator<Character> spliterator = new WordCounterSpliterator(SENTENCE);
Stream<Character> stream = StreamSupport.stream(spliterator, true);
```

Передаваемый в фабричный метод `StreamSupport.stream` второй булев аргумент означает, что мы хотим создать параллельный поток данных. Передаем этот параллельный поток данных в метод `countWords`:

```
System.out.println("Найдено " + countWords(stream) + " слов");
```

и получаем правильный результат, как и ожидалось:

```
Найдено 19 слов
```

Мы показали вам, как можно с помощью сплитераторов управлять стратегией разбиения структуры данных. Еще одна, последняя примечательная возможность сплитераторов — привязка источника обрабатываемых элементов к точке первого чтения элемента, первого разбиения или первого запроса предположительного размера данных, а не ко времени создания сплитератора. Такие сплитераторы называются сплитераторами с *динамическим связыванием* (late-binding). Мы посвятили приложение В демонстрации создания вспомогательного класса, предназначенного для параллельного выполнения нескольких различных операций над одним потоком данных.

Резюме

- С помощью внутренней итерации можно обрабатывать поток данных в параллельном режиме без необходимости явно использовать и синхронизировать в своем коде различные потоки выполнения.
- Несмотря на простоту параллельной обработки потоков данных, нет никаких гарантий, что благодаря этому ваши программы станут работать быстрее во всех случаях. Поведение и быстродействие параллельного программного обеспечения не всегда поддаются логике, поэтому следует всегда измерять производительность, чтобы убедиться, что параллельная обработка не замедляет работу программы.
- Параллельное выполнение операций над набором данных с помощью параллельного потока может серьезно повысить производительность, особенно когда количество обрабатываемых элементов очень велико или обработка каждого из них занимает очень много времени.
- С точки зрения производительности использование правильной структуры данных, например применение везде, где только возможно, специализированных версий потоков для простых типов данных вместо неспециализированных, практически всегда важнее распараллеливания операций.
- С помощью фреймворка ветвления-объединения можно рекурсивно разбить параллелизуемую задачу на меньшие подзадачи, выполнить их в различных потоках выполнения, а затем объединить результаты всех подзадач в единый общий результат.
- Сплитераторы задают способ разбиения параллельным потоком обрабатываемых им данных.

Часть III

Эффективное программирование с помощью потоков и лямбда-выражений

Третья часть этой книги изучает различные функции Java 8 и Java 9, которые помогут вам эффективнее использовать язык и обогатить вашу базу кода современными идиомами. А поскольку речь в ней идет о продвинутых концепциях программирования, то для понимания изложенного далее в книге не потребуется знаний этих методик.

Глава 8 написана специально для второго издания, в ней мы изучаем расширения Collection API Java 8 и Java 9. Она охватывает вопросы использования фабрик коллекций и новые идиоматические паттерны для работы со списками и множествами (коллекциями типа `List` и `Set`), помимо идиоматических паттернов для ассоциативных массивов (коллекций `Map`).

Глава 9 посвящена вопросам усовершенствования существующего кода с помощью новых возможностей Java 8 и включает несколько полезных рецептов. Кроме того, в ней мы исследуем такие жизненно важные методики разработки программного обеспечения, как паттерны проектирования, рефакторинг, тестирование и отладку.

Глава 10 также написана специально для второго издания. В ней изучается идея API на основе предметно-ориентированного языка (DSL). Это многообещающий метод проектирования API, популярность которого неуклонно растет. Его уже можно встретить в таких интерфейсах Java, как `Comparator`, `Stream` и `Collector`.

8 Расширения Collection API

В этой главе

- Использование фабрик коллекций.
- Новые идиоматические паттерны для работы со списками и множествами.
- Идиоматические паттерны для работы с ассоциативными массивами.

Ваша жизнь как Java-разработчика была бы очень скучной без Collection API. Коллекции применяются практически во всех Java-приложениях. В предыдущих главах было продемонстрировано, насколько полезно сочетание коллекций со Stream API для выражения запросов обработки данных. Тем не менее у Collection API есть множество недостатков, вследствие которых основанный на нем код порой излишне «многословен» и подвержен ошибкам.

В этой главе мы расскажем про добавления, внесенные в Collection API в Java 8 и Java 9, которые ощутимо упрощают жизнь программистов. Прежде всего вам предстоит узнать о фабриках коллекций в Java 9 — новинке, облегчающей процесс создания маленьких списков, множеств и ассоциативных массивов. Далее вы научитесь применять идиоматические паттерны удаления и подстановки в списках и множествах благодаря расширениям Java 8. Наконец, вы узнаете о новых удобных операциях для работы с ассоциативными массивами.

В главе 9 изучается более широкий круг методов рефакторинга Java-кода, написанного в старом стиле.

8.1. Фабрики коллекций

В Java 9 появилось несколько новых удобных способов создания небольших объектов коллекций. Во-первых, обсудим, почему программистов не устраивает существующее положение вещей; а затем мы покажем вам, как применять новые фабричные методы.

Как бы вы создали маленький список элементов на Java? Например, пусть вам нужно сгруппировать имена ваших друзей, отправляющихся в путешествие. Вот один из способов:

```
List<String> friends = new ArrayList<>();
friends.add("Raphael");
friends.add("Olivia");
friends.add("Thibaut");
```

Прямо скажем, изрядное количество кода ради сохранения всего лишь трех строковых значений! Удобнее будет воспользоваться для написания этого кода фабричным методом `Arrays.asList()`:

```
List<String> friends
   = Arrays.asList("Raphael", "Olivia", "Thibaut");
```

При этом создается список фиксированного размера, значения в котором можно менять, нельзя лишь добавлять/удалять элементы. Попытка добавления элементов, например, приводит к генерации исключения `UnsupportedModificationException`, обновление же списка с помощью метода `set` вполне допустимо:

```
List<String> friends = Arrays.asList("Raphael", "Olivia");
friends.set(0, "Richard");
friends.add("Thibaut");
```

Генерирует исключение UnsupportedModificationException

Подобное поведение выглядит довольно странным, поскольку в основе этого лежит изменяемый список фиксированного размера.

А как насчет множеств? К сожалению, фабричного метода `Arrays.asSet()` не существует, так что придется воспользоваться другой уловкой, а именно конструктором `HashSet`, принимающим в качестве аргумента список:

```
Set<String> friends
   = new HashSet<>(Arrays.asList("Raphael", "Olivia", "Thibaut"));
```

Или же можно воспользоваться Stream API:

```
Set<String> friends
   = Stream.of("Raphael", "Olivia", "Thibaut")
           .collect(Collectors.toSet());
```

Оба этих решения, впрочем, трудно назвать изящными, они требуют выделения памяти под лишние объекты. Отметим также, что в результате мы получаем изменяемый объект `Set`.

А как насчет ассоциативных массивов? Изящного способа создания маленьких ассоциативных массивов не существует, но не переживайте: в Java 9 были добавлены

фабричные методы, упрощающие жизнь программиста при создании маленьких списков, множеств и ассоциативных массивов.

Мы начнем наш обзор новых способов создания коллекций в языке Java с демонстрации новых возможностей, касающихся списков.

Коллекции-литералы

Некоторые языки программирования, в том числе Python и Groovy, поддерживают коллекции-литералы, то есть создание коллекций с помощью специального синтаксиса (например, [42, 1, 5] для создания списка из трех чисел). Java не предоставляет синтаксической поддержки подобного, поскольку изменения языка означают серьезные затраты на сопровождение и ограничивают использование возможного синтаксиса в будущем. Вместо этого в Java 9 была добавлена поддержка коллекций-литералов посредством расширения Collection API.

8.1.1. Фабрика списков

Для создания списка достаточно вызвать фабричный метод `List.of`:

```
List<String> friends = List.of("Raphael", "Olivia", "Thibaut");
System.out.println(friends);
```

[Raphael, Olivia, Thibaut]

Однако вы можете заметить одну странность. Попробуйте добавить в свой список друзей новый элемент:

```
List<String> friends = List.of("Raphael", "Olivia", "Thibaut");
friends.add("Chih-Chun");
```

Запуск этого кода вызовет генерацию исключения `java.lang.UnsupportedOperationException`. На самом деле полученный список — неизменяемый. Попытка замены элемента с помощью метода `set()` приводит к генерации аналогичного исключения. Не получится у вас и модифицировать его с помощью метода `set()`. Это ограничение, однако, весьма удобно, поскольку защищает от нежелательных изменений коллекций. Ничто ведь не мешает использовать изменяемые элементы. Если вам требуется изменяемый список, всегда можно создать экземпляр вручную. Наконец, обратите внимание, что пустые (равные `null`) элементы не разрешены во избежание неожиданных ошибок и ради большей компактности внутреннего представления.

Перегрузка и аргумент переменной длины

Если вы посмотрите на интерфейс `List` внимательнее, то обнаружите несколько перегруженных вариантов метода `List.of`:

```
static <E> List<E> of(E e1, E e2, E e3, E e4)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5)
```

Наверное, вы недоумеваете, почему в Java вместо них не используется один метод, в котором бы применялась сигнатура аргумента переменной длины для получения произвольного количества элементов, вот так:

```
static <E> List<E> of(E... elements)
```

Дело в том, что версия с аргументом переменной длины неявно выделяет память под дополнительный массив, обернутый в список. В итоге пришлось бы тратить ресурсы на выделение памяти для массива, его инициализацию, а позднее и на сборку мусора. А при указании фиксированного числа элементов (вплоть до десяти) эти накладные расходы отсутствуют. Отметим, что можно создать и список из более чем десяти элементов, но в этом случае будет использоваться вышеупомянутая сигнатура с аргументом переменной длины. Аналогичную схему можно наблюдать и для методов `Set.of` и `Map.of`.

Вероятно, вы задумались: не использовать ли Stream API вместо новых фабричных методов коллекций для создания подобных списков? В конце концов, в предыдущих главах вы видели, как можно применять коллектор `Collectors.toList()` для преобразования потока данных в список. Мы рекомендуем вам все же использовать фабричные методы, за исключением случаев, когда нужно задать какую-либо последовательность обработки и преобразования данных. Они проще в применении, да и реализация фабричных методов лучше подходит для этого.

Мы рассказали вам про новый фабричный метод для интерфейса `List`. В следующем подразделе мы поговорим об интерфейсе `Set`.

8.1.2. Фабрика множеств

Аналогично методу `List.of` существует и метод для создания из списка элементов неизменяемого объекта `Set`:

```
Set<String> friends = Set.of("Raphael", "Olivia", "Thibaut");
System.out.println(friends);
```

[Raphael, Olivia, Thibaut]

Если попытаться создать множество с повторяющимся элементом, будет сгенерировано исключение `IllegalArgumentException`. Это исключение отражает контракт на обеспечение множествами уникальности их элементов:

```
Set<String> friends = Set.of("Raphael", "Olivia", "Olivia");
```

java.lang.IllegalArgumentException:
duplicate element: Olivia

Еще одна популярная структура данных в Java — ассоциативные массивы (тип `Map`). В следующем разделе мы расскажем о новых способах создания ассоциативных массивов.

8.1.3. Фабрики ассоциативных массивов

Создание ассоциативного массива — несколько более сложная задача, чем создание списков и множеств, поскольку необходимо указать как ключи, так и значения. Существует два способа инициализации неизменяемых ассоциативных массивов

в Java 9. Первый способ — воспользоваться фабричным методом `Map.of`, который умеет отличать ключи от значений:

```
Map<String, Integer> ageOfFriends
   = Map.of("Raphael", 30, "Olivia", 25, "Thibaut", 26);
System.out.println(ageOfFriends);
```

{Olivia=25, Raphael=30, Thibaut=26}

Метод удобен для создания маленьких ассоциативных массивов из не более чем десяти ключей и значений. Если этого недостаточно, можно воспользоваться другим фабричным методом, `Map.ofEntries`, который принимает объекты `Map.Entry<K, V>`, но реализован на основе аргумента переменной длины. Данный метод требует выделения дополнительной памяти для объектов, служащих адаптерами для ключа и значения:

```
import static java.util.Map.entry;
Map<String, Integer> ageOfFriends
       = Map.ofEntries(entry("Raphael", 30),
                       entry("Olivia", 25),
                       entry("Thibaut", 26));
System.out.println(ageOfFriends);
```

{Olivia=25, Raphael=30, Thibaut=26}

`Map.entry` — новый фабричный метод, предназначенный для создания объектов `Map.Entry`.

Контрольное задание 8.1

Как вы думаете, какой результат вернет следующий фрагмент кода?

```
List<String> actors = List.of("Keanu", "Jessica");
actors.set(0, "Brad");
System.out.println(actors);
```

Ответ: будет сгенерировано исключение `UnsupportedOperationException`. Метод `List.of` создает неизменяемые коллекции.

Пока мы узнали, что новые фабричные методы Java 9 упрощают создание коллекций. Но на практике эти коллекции необходимо еще и обрабатывать. В следующем разделе мы расскажем про несколько новых расширений интерфейсов `List` и `Set`, реализующих уже готовые паттерны обработки, которые часто встречаются на практике.

8.2. Работа со списками и множествами

В Java 8 в интерфейсах `List` и `Set` появилось несколько новых методов.

- Метод `removeIf` удаляет соответствующий предикату элемент. Он присутствует во всех классах, реализующих интерфейсы `List` или `Set` (унаследован от интерфейса `Collection`).
- Метод `replaceAll` присутствует в интерфейсе `List` и служит для замены элементов результатом применения к ним функции вида `(UnaryOperator)`.

- Метод `sort` также присутствует в интерфейсе `List` и служит для сортировки самого списка.

Все эти методы модифицируют коллекции, для которых вызываются. Другими словами, они меняют саму коллекцию, в отличие от потоковых операций, возвращающих новый (скопированный) объект. Зачем были добавлены подобные методы? Дело в том, что модификация коллекций чревата ошибками и код для ее осуществления не слишком компактен. Для решения этих проблем в Java 8 и были добавлены методы `removeIf` и `replaceAll`.

8.2.1. Метод removeIf

Рассмотрим следующий код, предназначенный для удаления транзакций, идентификатор которых начинается с цифры:

```
for (Transaction transaction : transactions) {
   if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
       transactions.remove(transaction);
   }
}
```

Видите, в чем проблема? К сожалению, этот код может привести к генерации исключения `ConcurrentModificationException`. Почему? А потому, что «под капотом» цикла `for-each` используется объект-итератор, так что этот код выполняется следующим образом:

```
for (Iterator<Transaction> iterator = transactions.iterator();
iterator.hasNext(); ) {
   Transaction transaction = iterator.next();
   if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
       transactions.remove(transaction);
   }
}
```

Проблема в том, что мы организуем цикл и модифицируем коллекцию через два отдельных объекта

Обратите внимание на два отдельных объекта, с помощью которых осуществляется работа с коллекцией.

- Объект-итератор, с помощью которого производится опрос источника (применяются методы `next()` и `hasNext()`).

Сам объект-коллекция, служащий для удаления элемента посредством вызова метода `remove()`. В результате состояние итератора перестает быть согласованным с состоянием коллекции и наоборот. Для решения этой проблемы необходимо использовать объект-итератор непосредственно и вызывать его метод `remove()`:

```
for (Iterator<Transaction> iterator = transactions.iterator();
iterator.hasNext(); ) {
   Transaction transaction = iterator.next();
   if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
```

```
        iterator.remove();
    }
}
```

Этот код становится весьма длинным. К счастью, для выражения подобного паттерна кода можно теперь воспользоваться методом `removeIf` Java 8, что не только проще, но и защищает разработчика от вышеупомянутых программных ошибок. Он принимает в качестве параметра предикат, указывающий, какие элементы необходимо удалить:

```
transactions.removeIf(transaction ->
      Character.isDigit(transaction.getReferenceCode().charAt(0)));
```

Иногда, впрочем, нужно не удалить элемент, а заменить его другим. Для этой цели в Java 8 был добавлен метод `replaceAll`.

8.2.2. Метод replaceAll

С помощью метода `replaceAll` интерфейса `List` можно заменить все элементы списка другими. Благодаря Stream API можно решить эту задачу следующим образом:

```
referenceCodes.stream()                    <-- [a12, C14, b13]
              .map(code -> Character.toUpperCase(code.charAt(0)) +
code.substring(1))
              .collect(Collectors.toList())
              .forEach(System.out::println);   <-- Выводит в консоль A12, C14, B13
```

Однако в результате работы этого кода получается новая коллекция строк. Нам же требуется способ модификации уже существующей коллекции. Для этого можно воспользоваться объектом типа `ListIterator` (который поддерживает предназначенный для замены элементов метод `set()`):

```
for (ListIterator<String> iterator = referenceCodes.listIterator();
iterator.hasNext(); ) {
    String code = iterator.next();
    iterator.set(Character.toUpperCase(code.charAt(0)) + code.substring(1));
}
```

Как вы можете видеть, этот код весьма «многословен». Кроме того, как мы уже объясняли ранее, совместное использование итераторов с коллекциями чревато возникновением ошибок из-за смешивания итерации и модификации коллекции. В Java 8 можно просто написать:

```
referenceCodes.replaceAll(code -> Character.toUpperCase(code.charAt(0)) +
    code.substring(1));
```

Вы узнали о новинках в интерфейсах `List` и `Set`, но не забывайте, что есть еще и `Map`. Новые расширения интерфейса `Map` описаны в следующем разделе.

8.3. Работа с ассоциативными массивами

В Java 8 интерфейс `Map` стал поддерживать несколько новых методов с реализацией по умолчанию. (Методы с реализацией по умолчанию подробнее рассматриваются в главе 13, пока вы можете считать их просто заранее реализованными методами этого интерфейса.) Задача этих новых методов — упростить создание более лаконичного кода за счет использования готовых идиоматических паттернов вместо реализации их самостоятельно. Мы рассмотрим эти операции в следующем разделе, начиная с блистательной новой версии `forEach`.

8.3.1. Метод forEach

Обход в цикле ключей и значений ассоциативного массива всегда требовал написания довольно неуклюжего кода. Фактически для этого нужно организовать цикл с помощью итератора `Map.Entry<K, V>`, проходящего по результату, возвращаемому методом `entrySet()` ассоциативного массива:

```
for(Map.Entry<String, Integer> entry: ageOfFriends.entrySet()) {
   String friend = entry.getKey();
   Integer age = entry.getValue();
   System.out.println(friend + " is " + age + " years old");
}
```

Начиная с Java 8, интерфейс `Map` поддерживает метод `forEach`, принимающий в качестве параметра операцию типа `BiConsumer` с ключом и значением в качестве аргументов. Использование метода `forEach` существенно повышает лаконичность кода:

```
ageOfFriends.forEach((friend, age) -> System.out.println(friend + " is " +
     age + " years old"));
```

Одна из задач, связанных с организацией циклов по данным, — их сортировка. В Java 8 появилось несколько удобных способов сравнения записей ассоциативного массива.

8.3.2. Сортировка

Отсортировать записи ассоциативного массива по значениям или ключам можно с помощью следующих двух новых утилит:

- `Entry.comparingByValue`;
- `Entry.comparingByKey`.

При выполнении этого кода:

```
Map<String, String> favouriteMovies
       = Map.ofEntries(entry("Raphael", "Star Wars"),
       entry("Cristina", "Matrix"),
       entry("Olivia", "James Bond"));

favouriteMovies
  .entrySet()
```

```
  .stream()
  .sorted(Entry.comparingByKey())
  .forEachOrdered(System.out::println);
```

Обрабатывает элементы потока в алфавитном порядке имен

выводятся следующие результаты:

```
Cristina=Matrix
Olivia=James Bond
Raphael=Star Wars
```

Класс HashMap и быстродействие

Внутренняя структура класса `HashMap` была усовершенствована в Java 8 ради повышения производительности. Записи ассоциативного массива обычно хранятся в корзинах, доступ к которым производится по хеш-коду ключа. Но в случае одинакового хеш-кода у большого количества ключей производительность существенно падает, поскольку корзины реализованы в виде связных списков со сложностью извлечения порядка $O(n)$. Теперь же, если корзина слишком разрастается, она динамически заменяется отсортированным деревом со сложностью извлечения порядка $O(\log(n))$ и лучшими показателями поиска конфликтующих элементов. Отметим, что использовать подобным образом отсортированные деревья можно лишь тогда, когда ключи относятся к типам, реализующим интерфейс `Comparable` (таким как классы `String` и `Number`)[1].

Еще один распространенный паттерн описывает, что делать при отсутствии в ассоциативном массиве искомого ключа. Здесь нам поможет новый метод `getOrDefault`.

8.3.3. Метод getOrDefault

При отсутствии в ассоциативном массиве искомого ключа возвращается пустая ссылка, и такой случай нужно отслеживать, чтобы не получить `NullPointerException`. Чаще всего в этом случае задают возвращаемое значение по умолчанию. Теперь можно проще реализовать эту идею с помощью метода `getOrDefault`. Данный метод принимает в качестве первого аргумента ключ, а в качестве второго аргумента — значение по умолчанию (используемое при отсутствии ключа в ассоциативном массиве):

```
Map<String, String> favouriteMovies
       = Map.ofEntries(entry("Raphael", "Star Wars"),
       entry("Olivia", "James Bond"));
System.out.println(favouriteMovies.getOrDefault("Olivia", "Matrix"));
System.out.println(favouriteMovies.getOrDefault("Thibaut", "Matrix"));
```

Выводит James Bond

Выводит Matrix

[1] Сам класс Number не реализует интерфейс Comparable, хотя многие из его подклассов (Double, Integer, Long, Short и т. п.) — реализуют. — *Примеч. пер.*

Обратите внимание, что если ключ в ассоциативном массиве присутствует, но случайно был связан с пустым значением, то `getOrDefault` может вернуть `null`. Отметим также, что передаваемое в качестве резервного варианта значение вычисляется в любом случае, независимо от того, существует ключ или нет.

Java 8 включает также несколько более продвинутых паттернов обработки случаев наличия/отсутствия значений для заданного ключа. Мы расскажем о них в следующем подразделе.

8.3.4. Паттерны вычисления

Иногда требуется выполнить (или не выполнить) операцию и сохранить ее результат в зависимости от того, присутствует ключ в ассоциативном массиве или нет. Например, речь может идти о кэшировании результата дорогостоящей операции по ключу. Если ключ присутствует в массиве, то вычислять результат заново не нужно. При этом могут быть полезны следующие три операции.

- `computeIfAbsent` — если для ключа значение не задано (ключ отсутствует или значение для него равно `null`), вычисляет новое значение для этого ключа и добавляет его в ассоциативный массив.
- `computeIfPresent` — если ключ присутствует в ассоциативном массиве, вычисляет новое значение для него и добавляет полученное в ассоциативный массив.
- `compute` — эта операция вычисляет новое значение для заданного ключа и сохраняет его в ассоциативном массиве.

Метод `computeIfAbsent`, в частности, применяется для кэширования информации. Допустим, вы выполняете синтаксический разбор каждой из строк набора файлов и вычисляете их SHA-256-хеши. Если данные уже были обработаны, необходимости вычислять их заново нет.

Теперь предположим, что ваш кэш реализован на основе ассоциативного массива, а для вычисления SHA-256-хешей используется экземпляр класса `MessageDigest`:

```
Map<String, byte[]> dataToHash = new HashMap<>();
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
```

Далее вы проходите в цикле по данным и кэшируете результаты:

```
lines.forEach(line ->
   dataToHash.computeIfAbsent(line,
                              this::calculateDigest));

private byte[] calculateDigest(String key) {
   return messageDigest.digest(key.getBytes(StandardCharsets.UTF_8));
}
```

line представляет собой искомый ключ

Эта операция выполняется, если ключа не найдено

Эта вспомогательная функция вычисляет хеш для заданного ключа

Этот способ также удобен для работы с ассоциативными массивами, хранящими несколько значений для одного ключа. При необходимости добавить элемент в `Map<K, List<V>>` нужно обеспечить инициализацию записи. Потребуется написать

немалый объем кода. Допустим, вы хотите сформировать список фильмов для вашего друга Рафаэля:

```
String friend = "Raphael";
List<String> movies = friendsToMovies.get(friend);
if(movies == null) {
   movies = new ArrayList<>();
   friendsToMovies.put(friend, movies);
}
movies.add("Star Wars");

System.out.println(friendsToMovies);
```

Проверяем, что список был инициализирован

Добавляем фильм

{Raphael: [Star Wars]}

Можно вместо этого воспользоваться методом `computeIfAbsent`. Если ключ не найден, этот метод добавляет вычисленное значение в ассоциативный массив и возвращает его; в противном случае возвращается уже имеющееся значение. Использовать его можно следующим образом:

```
friendsToMovies.computeIfAbsent("Raphael", name -> new ArrayList<>())
               .add("Star Wars");
```

{Raphael: [Star Wars]}

Метод `computeIfAbsent` вычисляет новое значение, если соответствующее ключу текущее значение присутствует в ассоциативном массиве и не равно `null`. Обратите внимание на небольшой нюанс: если вычисляющая значение функция вернет `null`, то текущее соответствие ключа/значения удаляется из ассоциативного массива. Впрочем, для удаления соответствия лучше подходит перегруженная версия метода `remove`. Мы расскажем о ней в следующем подразделе.

8.3.5. Паттерны удаления

Вы уже знаете про метод `remove`, служащий для удаления из ассоциативного массива записи для конкретного ключа. Начиная с Java 8, существует перегруженная версия, которая удаляет запись только в том случае, когда ключу соответствует конкретное значение. Ранее подобное поведение можно было реализовать с помощью такого кода (мы ничего не имеем против Тома Круза, но отзывы о фильме «Джек Ричер — 2» были довольно плохими):

```
String key = "Raphael";
String value = "Jack Reacher 2";
if (favouriteMovies.containsKey(key) &&
       Objects.equals(favouriteMovies.get(key), value)) {
   favouriteMovies.remove(key);
   return true;
}
else {
   return false;
}
```

Теперь же для этого достаточно написать такой код. Согласитесь, что он выглядит намного лучше:

```
favouriteMovies.remove(key, value);
```

В следующем разделе мы расскажем вам о способах замены элементов в ассоциативном массиве и удаления элементов из него.

8.3.6. Способы замены элементов

В интерфейсе `Map` появилось два новых метода для замены записей внутри ассоциативного массива.

- `replaceAll` — заменяет значения во всех записях результатом применения к ним `BiFunction`. Этот метод работает аналогично методу `replaceAll` интерфейса `List`, который вы уже видели ранее.
- `Replace` — позволяет заменить значение в ассоциативном массиве при наличии в нем определенного ключа. Дополнительный перегруженный метод заменяет значения только в том случае, если ключ связан с определенным значением.

Например, можно отформатировать все значения в ассоциативном массиве следующим образом:

```
Map<String, String> favouriteMovies = new HashMap<>();
favouriteMovies.put("Raphael", "Star Wars");
favouriteMovies.put("Olivia", "james bond");
favouriteMovies.replaceAll((friend, movie) -> movie.toUpperCase());
System.out.println(favouriteMovies);
```

Нам пришлось воспользоваться изменяемым ассоциативным массивом, поскольку мы далее будем применять к нему метод replaceAll

{Olivia=JAMES BOND, Raphael=STAR WARS}

Описанные способы замены предназначены для работы с одним ассоциативным массивом. Но что делать, если потребуется сочетать и заменять значения из двух ассоциативных массивов? Для этой цели можно взять новый метод `merge`.

8.3.7. Метод merge

Пусть вам нужно выполнить слияние двух промежуточных ассоциативных массивов, например двух отдельных ассоциативных массивов для двух групп контактов. Для этого можно использовать метод `putAll` следующим образом:

```
Map<String, String> family = Map.ofEntries(
   entry("Teo", "Star Wars"), entry("Cristina", "James Bond"));
Map<String, String> friends = Map.ofEntries(
   entry("Raphael", "Star Wars"));
Map<String, String> everyone = new HashMap<>(family);
everyone.putAll(friends);
System.out.println(everyone);
```

Копирует все записи из ассоциативного массива friends в ассоциативный массив everyone

{Cristina=James Bond, Raphael=Star Wars, Teo=Star Wars}

При отсутствии дублирующихся ключей этот код работает так, как и ожидалось. Если же вам нужна повышенная гибкость при объединении значений, можете воспользоваться новым методом `merge`. Он принимает в качестве аргумента `BiFunction` для слияния значений с дублирующимся ключом. Пусть Кристина числится в двух

ассоциативных массивах — `family` и `friends`, но записи для нее содержат там различные фильмы:

```
Map<String, String> family = Map.ofEntries(
    entry("Teo", "Star Wars"), entry("Cristina", "James Bond"));
Map<String, String> friends = Map.ofEntries(
    entry("Raphael", "Star Wars"), entry("Cristina", "Matrix"));
```

В подобном случае можно воспользоваться методом `merge` вместе с `forEach` для разрешения конфликта. Следующий код производит конкатенацию строковых названий двух указанных фильмов:

```
Map<String, String> everyone = new HashMap<>(family);
friends.forEach((k, v) ->
   everyone.merge(k, v, (movie1, movie2) -> movie1 + " & " + movie2));
System.out.println(everyone);
```

По заданному дублирующемуся ключу производит конкатенацию двух значений

Выводит {Raphael=Star Wars, Cristina=James Bond & Matrix, Teo=Star Wars}

Отметим, что метод `merge` обрабатывает значения `null` довольно запутанным способом, описанным в Javadoc так:

> *«Если заданному ключу не соответствует никакое значение или соответствует пустое значение (null), метод merge связывает его с заданным непустым значением. В противном случае метод merge заменяет соответствующее ему значение результатом выполнения заданной функции повторного отображения или удаляет его, если результат выполнения — null».*

Можно также задействовать метод `merge` для проверки задания начального значения. Пусть у нас есть ассоциативный массив с информацией о количестве просмотров фильмов. Прежде чем увеличивать значение счетчика для фильма, необходимо проверить наличие в ассоциативном массиве соответствующего этому фильму ключа:

```
Map<String, Long> moviesToCount = new HashMap<>();
String movieName = "JamesBond";
long count = moviesToCount.get(movieName);
if(count == null) {
   moviesToCount.put(movieName, 1);
}
else {
   moviesToCount.put(moviename, count + 1);
}
```

Этот код можно переписать в следующем виде:

```
moviesToCount.merge(movieName, 1L, (key, count) -> count + 1L);
```

Второй аргумент метода `merge` в данном случае — `1L`. Javadoc гласит, что этот аргумент «представляет собой непустое значение, которое сливается с соответствующим ключу уже существующим значением или связывается с ключом, если тому не соответствует никакого значения или соответствует пустое». А поскольку возвращаемое для этого ключа значение равно `null`, то на первый раз указывается `1`.

В следующий раз, поскольку уже было задано равное 1 начальное значение ключа, для увеличения счетчика применяется указанная BiFunction.

Мы рассказали вам о дополнениях к интерфейсу Map. Новые расширения были добавлены и в родственный ему класс ConcurrentHashMap, о котором мы расскажем далее.

Контрольное задание 8.2

Попробуйте разобраться, что делает следующий код и какими идиоматическими операциями можно воспользоваться для упрощения кода, решающего эту же задачу:

```
Map<String, Integer> movies = new HashMap<>();
movies.put("JamesBond", 20);
movies.put("Matrix", 15);
movies.put("Harry Potter", 5);
Iterator<Map.Entry<String, Integer>> iterator =
             movies.entrySet().iterator();
while(iterator.hasNext()) {
   Map.Entry<String, Integer> entry = iterator.next();
   if(entry.getValue() < 10) {
       iterator.remove();
   }
}
System.out.println(movies);
```

{Matrix=15, JamesBond=20}

Ответ: для этой цели можно применить к результату, возвращаемому методом entrySet() ассоциативного массива, метод removeIf, который принимает на входе предикат и удаляет соответствующие элементы:

```
movies.entrySet().removeIf(entry -> entry.getValue() < 10);
```

8.4. Усовершенствованный класс ConcurrentHashMap

Класс ConcurrentHashMap был включен в язык Java в качестве альтернативы HashMap — более современной и поддерживающей конкурентную обработку. Класс ConcurrentHashMap содержит конкурентные операции добавления (add) и обновления (update), блокирующие лишь определенные части внутренней структуры данных. В результате этого производительность операций чтения и записи повысилась, по сравнению с синхронизированным Hashtable (обратите внимание, что обычный HashMap не синхронизирован).

8.4.1. Свертка и поиск

Класс ConcurrentHashMap поддерживает три новых типа операций, напоминающие операции с потоками данных:

- forEach — выполняет заданное действие для каждой из пар «ключ, значение»;
- reduce — объединяет все пары «ключ, значение» в единый результат с помощью заданной функции свертки;

- `search` — применяет функцию к каждой из пар «ключ, значение», пока не будет получен непустой результат.

Каждый из этих видов операций существует в четырех формах, принимающих функции с ключами, значениями, объектами `Map.Entry` и парами «ключ, значение» в качестве аргументов:

- работающие с ключами и значениями (`forEach`, `reduce`, `search`);
- работающие с ключами (`forEachKey`, `reduceKeys`, `searchKeys`);
- работающие со значениями (`forEachValue`, `reduceValues`, `searchValues`);
- работающие с объектами `Map.Entry` (`forEachEntry`, `reduceEntries`, `searchEntries`).

Обратите внимание, что эти операции не блокируют состояние объекта `ConcurrentHashMap`; они работают с отдельными элементами по мере обхода. Передаваемые в эти операции функции не зависят от какого-либо упорядочения, каких-либо других объектов или значений, которые могут поменяться за время вычислений.

Кроме того, для всех этих операций необходимо указывать пороговое значение распараллеливания. Если текущий размер ассоциативного массива меньше заданного порогового значения, эти операции будут выполняться последовательно. Пороговое значение `1` означает максимальный параллелизм на основе общего пула потоков выполнения. Пороговое значение, равное `Long.MAX_VALUE`, означает выполнение операции в одном потоке выполнения. Обычно имеет смысл придерживаться этих значений, разве что архитектура вашего приложения включает расширенную оптимизацию ресурсов.

В следующем примере мы воспользуемся методом `reduceValues` для поиска максимального значения ассоциативного массива:

```
                                    Объект ConcurrentHashMap, который где-то в промежутке
                                    обновляется и содержит несколько ключей и значений
ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();
long parallelismThreshold = 1;
Optional<Integer> maxValue =
   Optional.ofNullable(map.reduceValues(parallelismThreshold, Long::max));
```

Обратите внимание на существование специализированных версий простых типов данных (`int`, `long` и `double`) для всех операций свертки (`reduceValuesToInt`, `reduceKeysToLong` и т. д.) — более эффективных благодаря отсутствию упаковки.

8.4.2. Счетчики

Класс `ConcurrentHashMap` содержит новый метод — `mappingCount`, который возвращает количество соответствий в ассоциативном массиве в виде значения типа `long`. Мы рекомендуем вам использовать его при написании нового кода вместо метода `size`, возвращающего `int`. Это гарантирует работоспособность кода в будущем, когда число соответствий превысит размеры `int`.

8.4.3. Представление в виде множества

Класс `ConcurrentHashMap` включает еще один новый метод — `keySet`, возвращающий представление `ConcurrentHashMap` в виде объекта `Set` (изменения в ассоциативном массиве отражаются в множестве, и наоборот). Можно также создать множество на основе объекта `ConcurrentHashMap` с помощью нового статического метода `newKeySet`.

Резюме

- Java 9 поддерживает фабрики коллекций, с помощью которых можно создавать маленькие неизменяемые списки, множества и ассоциативные массивы, используя методы `List.of`, `Set.of`, `Map.of` и `Map.ofEntries`.
- Возвращаемые этими фабриками коллекций объекты являются неизменяемыми, то есть невозможно изменить их состояние после создания.
- Интерфейс `List` поддерживает методы с реализацией по умолчанию `removeIf`, `replaceAll` и `sort`.
- Интерфейс `Set` поддерживает метод с реализацией по умолчанию `removeIf`.
- Интерфейс `Map` включает несколько новых методов с реализацией по умолчанию для распространенных паттернов, уменьшая тем самым количество потенциальных ошибок.
- Класс `ConcurrentHashMap` поддерживает унаследованные от интерфейса `Map` новые методы с реализацией по умолчанию, но уже с подходящими для многопоточного выполнения реализациями.

9 Рефакторинг, тестирование и отладка

В этой главе

- Рефакторинг кода под лямбда-выражения.
- Оцениваем влияние лямбда-выражений на объектно-ориентированные паттерны проектирования.
- Тестирование лямбда-выражений.
- Отладка кода, использующего лямбда-выражения и Stream API.

В первых восьми главах мы продемонстрировали впечатляющие возможности лямбда-выражений и Stream API. В основном вы создавали на базе этих возможностей новый код. В новых проектах Java можно сразу же использовать лямбда-выражения и потоки данных.

К сожалению, начинать новый проект с нуля получается далеко не всегда. Чаще всего приходится иметь дело с уже существующим кодом, написанным на более старых версиях Java.

В этой главе приведено несколько рецептов по рефакторингу существующего кода для применения лямбда-выражений и повышения удобочитаемости и гибкости кода. Кроме того, мы обсудим, как за счет лямбда-выражений повысить лаконичность нескольких объектно-ориентированных паттернов проектирования (в том числе таких, как «Стратегия», «Шаблонный метод», «Наблюдатель», «Цепочка обязанностей» и «Фабрика»). Наконец, мы выясним, как тестировать и отлаживать код, использующий лямбда-выражения и Stream API.

В главе 10 мы поговорим о более масштабном способе рефакторинга кода для повышения понятности логики приложения: создании предметно-ориентированных языков.

9.1. Рефакторинг с целью повышения удобочитаемости и гибкости кода

Начиная с первой страницы книги, мы приводили все новые аргументы в пользу того, что с помощью лямбда-выражений можно писать более лаконичный и гибкий код. Более лаконичный потому, что лямбда-выражения позволяют выражать варианты поведения в более сжатой, по сравнению с анонимными классами, форме. В главе 3 мы также показали вам, что можно писать еще более компактный код с помощью ссылок на методы, если нужно лишь передать существующий метод в качестве аргумента другому методу.

А более гибким код становится потому, что лямбда-выражения поощряют стиль программирования с параметризацией поведения, с которым мы познакомили вас в главе 2. Код может адаптироваться к изменениям требований за счет использования и выполнения множества различных вариантов поведения, передаваемых в виде аргументов.

В этом разделе мы соберем все это воедино и покажем простой путь рефакторинга кода с целью повышения его удобочитаемости и гибкости на основе возможностей, о которых вы узнали в предыдущих главах: лямбда-выражений, ссылок на методы и потоков данных.

9.1.1. Повышаем удобочитаемость кода

Что такое удобочитаемость кода? Хорошая удобочитаемость — довольно-таки субъективное понятие. Обычно считают, что этот термин равнозначен понятности кода для человека. Повышение удобочитаемости кода гарантирует, что в коде сможет разобраться (и сопровождать его) кто-то еще, кроме вас. Чтобы обеспечить понятность кода другим людям, можно предпринять несколько простых шагов, например создать подробную документацию и следовать при написании кода общепринятым стандартам.

Повысить удобочитаемость кода, по сравнению с предыдущими версиями, могут также появившиеся в Java 8 новые функциональные возможности. С их помощью можно повысить лаконичность кода и сделать его более понятным. Кроме того, используя ссылки на методы и Stream API, можно сделать назначение кода более прозрачным для читающего.

В этой главе мы опишем три простых варианта рефакторинга с использованием лямбда-выражений, ссылок на методы и потоков данных, с помощью которых вы сможете повысить удобочитаемость своего кода:

- преобразование анонимных классов в лямбда-выражения;
- преобразование лямбда-выражений в ссылки на методы;
- преобразование императивной обработки данных в потоковую.

9.1.2. Из анонимных классов — в лямбда-выражения

Первый простой рефакторинг, который имеет смысл выполнить: преобразовать анонимные классы, где реализован один-единственный абстрактный метод, в лямб-

да-выражения. Почему? Надеемся, в предыдущих главах мы убедительно продемонстрировали, что анонимные классы подвержены ошибкам и не отличаются компактностью. Благодаря переходу на лямбда-выражения можно создавать более лаконичный и читабельный код. Как показано в главе 3, анонимный класс для создания объекта типа `Runnable` и его аналог с применением лямбда-выражения выглядят следующим образом:

```
Runnable r1 = new Runnable() {                    ← До, с помощью анонимного класса
    public void run(){
        System.out.println("Hello");
    }
};
Runnable r2 = () -> System.out.println("Hello");  ← После, с помощью лямбда-выражения
```

Но в определенных случаях преобразование анонимных классов в лямбда-выражения может оказаться непростой задачей[1]. Во-первых, смысл ключевых слов `this` и `super` различен для анонимных классов и лямбда-выражений. Внутри анонимного класса `this` ссылается на сам анонимный класс, а внутри лямбда-выражения — на охватывающий его класс. Во-вторых, анонимные классы могут затенять переменные из охватывающего их класса. Лямбда-выражения — нет (в подобном случае компилятор выдаст ошибку), как показано в следующем коде:

```
int a = 10;
Runnable r1 = () -> {
    int a = 2;                    ← Ошибка компиляции
    System.out.println(a);
};
Runnable r2 = new Runnable(){
    public void run(){
        int a = 2;                ← Все в порядке!
        System.out.println(a);
    }
};
```

Наконец, преобразование анонимного класса в лямбда-выражения может привести к неоднозначности кода в случае перегрузки. И действительно, тип анонимного класса задается явным образом при создании экземпляра, а тип лямбда-выражения зависит от контекста. Ниже приведен пример случая, когда это может привести к проблемам. Пусть вы объявили функциональный интерфейс `Task` с такой же сигнатурой, как и у `Runnable` (например, если вам захотелось более осмысленных названий интерфейсов в модели предметной области):

```
interface Task {
    public void execute();
}
public static void doSomething(Runnable r){ r.run(); }
public static void doSomething(Task a){ r.execute(); }
```

[1] Более подробно этот процесс описан в превосходной статье по адресу http://dig.cs.illinois.edu/papers/lambdaRefactoring.pdf.

Передать анонимный класс, реализующий интерфейс `Task`, можно без всяких проблем:

```
doSomething(new Task() {
    public void execute() {
        System.out.println("Danger danger!!");
    }
});
```

А преобразование этого анонимного класса в лямбда-выражение приведет к неоднозначности вызова метода, поскольку подходящим целевым типом будет и `Runnable`, и `Task`:

```
doSomething(() -> System.out.println("Danger danger!!"));
```

Проблема: подходит как doSomething(Runnable), так и doSomething(Task)

Разрешить эту неоднозначность можно путем явного преобразования типа `(Task)`:

```
doSomething((Task)() -> System.out.println("Danger danger!!"));
```

Впрочем, не пугайтесь этих проблем, есть и хорошие новости! Большинство интегрированных сред разработки — например, NetBeans, Eclipse и IntelliJ — поддерживают подобный рефакторинг и автоматически исключают возникновение таких «глюков».

9.1.3. Из лямбда-выражений — в ссылки на методы

Лямбда-выражения замечательно подходят для короткого кода, который нужно куда-либо передать. Но для повышения удобочитаемости кода лучше использовать ссылки на методы везде, где только возможно. В главе 6, например, был приведен следующий код для группировки блюд по степени калорийности:

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
  menu.stream()
      .collect(
          groupingBy(dish -> {
            if (dish.getCalories() <= 400) return CaloricLevel.DIET;
            else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
            else return CaloricLevel.FAT;
          }));
```

Приведенное лямбда-выражение можно выделить в отдельный метод и передать его как аргумент операции `groupingBy`. Код при этом становится лаконичнее, а его назначение — более явным:

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream().collect(groupingBy(Dish::getCaloricLevel));
```

Лямбда-выражение вынесено в отдельный метод

Чтобы этот код заработал, необходимо добавить метод `getCaloricLevel` в класс `Dish`:

```
public class Dish{
    ...
    public CaloricLevel getCaloricLevel() {
        if (this.getCalories() <= 400) return CaloricLevel.DIET;
        else if (this.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
    }
}
```

Кроме того, имеет смысл везде, где только возможно, использовать вспомогательные статические методы, такие как `comparing` и `maxBy`. Эти методы специально предназначены для использования со ссылками на методы! И действительно, назначение следующего кода намного понятнее, чем его аналога с лямбда-выражением, как мы продемонстрировали вам в главе 3:

```
inventory.sort(
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
inventory.sort(comparing(Apple::getWeight));
```

Приходится задумываться о том, как реализована операция сравнения

Воспринимается просто как описание формулировки задачи

Более того, для многих распространенных операций свертки, таких как *суммирование* и *поиск максимума*, существуют встроенные вспомогательные методы, которые можно сочетать со ссылками на методы. Мы уже показали вам, например, что с помощью API коллекторов находить максимум или сумму удобнее, чем при сочетании лямбда-выражения и низкоуровневой операции `reduce`. Вместо того чтобы писать:

```
int totalCalories =
    menu.stream().map(Dish::getCalories)
                 .reduce(0, (c1, c2) -> c1 + c2);
```

попробуйте воспользоваться встроенными коллекторами, позволяющими сформулировать задачу намного яснее. В данном случае применяется коллектор `summingInt` (названия очень важны при написании документации к коду):

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

9.1.4. От императивной обработки данных до потоков

В коде обработки коллекции лучше всего использовать Stream API вместо стандартных паттернов с итераторами. Почему? Stream API позволяет более ясно выразить назначение конвейера обработки данных. Кроме того, возможна неявная оптимизация потоков данных, включая сокращенные схемы вычислений, отложенную обработку и использование многоядерной архитектуры, как мы уже объясняли в главе 7.

Следующий императивный код выражает два способа (фильтрацию и извлечение), которые часто декорируются совместно, из-за чего программисту приходится

сначала разбираться в реализации, прежде чем выяснять, что делает код. Кроме того, при этом намного усложняется реализация выполняемого параллельно кода. Чтобы представить себе, какого труда это стоит, см. главу 7 (особенно раздел 7.2):

```
List<String> dishNames = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() > 300){
        dishNames.add(dish.getName());
    }
}
```

Альтернативный вариант с использованием Stream API читается как описание формулировки задачи и легко распараллеливается:

```
menu.parallelStream()
    .filter(d -> d.getCalories() > 300)
    .map(Dish::getName)
    .collect(toList());
```

К сожалению, переход от императивного кода к Stream API может оказаться непростой задачей, ведь приходится искать подходящие потоковые операции с учетом операторов управления потоком выполнения, таких как `break`, `continue` и `return`. Хорошая новость заключается в том, что существуют утилиты, упрощающие в том числе и решение этой задачи (например, LambdaFicator, https://ieeexplore.ieee.org/document/6606699).

9.1.5. Повышаем гибкость кода

В главах 2 и 3 мы доказывали, что лямбда-выражения поощряют использование параметризации поведения. Можно создать несколько лямбда-выражений для различных вариантов поведения и затем передавать их при необходимости. Такой стиль программирования помогает адаптироваться к изменениям требований (например, путем создания различных вариантов фильтрации с применением предиката или сравнения с помощью компаратора). Далее мы рассмотрим несколько способов, как получить немедленную отдачу от использования лямбда-выражений для вашей базы кода.

Внедрение функциональных интерфейсов

Во-первых, применять лямбда-выражения без функциональных интерфейсов невозможно; а значит, имеет смысл начать внедрять их в свою базу кода. Но когда именно это следует делать? В этой главе мы обсудим два распространенных паттерна кода, в которых можно воспользоваться лямбда-выражениями: условное отложенное и охватывающее выполнение. Кроме того, в следующем разделе мы покажем, как с помощью лямбда-выражений более лаконично переписать различные объектно-ориентированные паттерны проектирования, например «Стратегия» и «Шаблонный метод».

Условное отложенное выполнение

Операторы управления потоком выполнения часто декорируются внутри кода бизнес-логики. В числе типичных сценариев использования — проверки на безопасность

и журналирование. Рассмотрим следующий код, в котором используется встроенный класс `Logger` языка Java:

```
if (logger.isLoggable(Log.FINER)) {
    logger.finer("Problem: " + generateDiagnostic());
}
```

Что с этим кодом не так? А вот что:

- клиентскому коду приходится узнавать состояние журналирующего объекта (поддерживаемый им уровень журналирования) через метод `isLoggable`;
- какой смысл в запросе состояния журналирующего объекта перед журналированием каждого сообщения? Это только загромождает код.

Лучше воспользоваться методом `log`, внутри которого перед журналированием сообщения проверяется, правильный ли уровень задан для журналирующего объекта:

```
logger.log(Level.FINER, "Problem: " + generateDiagnostic());
```

Этот подход лучше, поскольку не загромождает код операторами `if`, а состояние механизма журналирования не делается доступным. К сожалению, в данном коде все равно есть проблема: сообщение, предназначенное для журналирования, всегда вычисляется, даже если механизм журналирования не приспособлен для передаваемого в качестве аргумента уровня сообщений.

Здесь нам помогут лямбда-выражения. Необходимо просто отложить формирование сообщения так, чтобы оно генерировалось только при заданном условии (в данном случае при уровне журналирования `FINER`). Оказывается, что создатели API Java 8 знали об этой проблеме и создали перегруженный вариант `log`, принимающий в качестве аргумента объект типа `Supplier`. Сигнатура этого варианта `log` выглядит таким образом:

```
public void log(Level level, Supplier<String> msgSupplier)
```

Так что теперь можно написать следующий вызов:

```
logger.log(Level.FINER, () -> "Problem: " + generateDiagnostic());
```

Передаваемое в качестве аргумента лямбда-выражение выполняется внутри метода `log` только при механизме журналирования соответствующего уровня. Внутренняя реализация метода `log` имеет примерно следующий вид:

```
public void log(Level level, Supplier<String> msgSupplier) {
    if(logger.isLoggable(level)){
        log(level, msgSupplier.get()); <-- Выполнение лямбда-выражения
    }
}
```

Какой вывод можно из этого сделать? Если в клиентском коде многократно выполняются запросы состояния объекта (например, состояния журналирующего объекта) лишь для того, чтобы вызвать для этого объекта какой-либо метод с аргументами (например, для журналирования сообщения), имеет смысл создать новый

метод, который бы вызывал вышеупомянутый, передаваемый в виде лямбда-выражения или ссылки на метод, только после внутренней проверки состояния объекта. При этом повысится удобочитаемость кода (он станет менее загроможденным) и его инкапсулированность, а состояние объекта не будет доступно в клиентском коде.

Охватывающее выполнение

В главе 3 мы обсуждали еще один способ: охватывающее выполнение (execute around). Если в вашей программе различный код оказывается окружен одними и теми же фазами подготовки и очистки, то часто имеет смысл вынести этот код в лямбда-выражение. Благодаря этому можно переиспользовать логику подготовки и очистки, уменьшая дублирование кода.

Ниже приведен код, который вы уже видели в главе 3. В нем переиспользуется одна и та же логика для открытия и закрытия файла, при этом параметризуемая различными лямбда-выражениями для обработки файлов:

```
String oneLine =
    processFile((BufferedReader b) -> b.readLine());
String twoLines =
    processFile((BufferedReader b) -> b.readLine() + b.readLine());
public static String processFile(BufferedReaderProcessor p) throws
IOException {
    try(BufferedReader br = new BufferedReader(new
FileReader("ModernJavaInAction/chap9/data.txt"))) {
        return p.process(br);
    }
}
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

Передаем лямбда-выражение

Передаем другое лямбда-выражение

Выполняем передаваемый в качестве аргумента BufferedReaderProcessor

Функциональный интерфейс для лямбда-выражения, который может генерировать IOException

Такой код стал возможен благодаря созданию функционального интерфейса `BufferedReaderProcessor`, с помощью которого мы передаем различные лямбда-выражения для обработки объекта `BufferedReader`.

В этом разделе мы привели различные рецепты повышения удобочитаемости и гибкости кода. Далее мы покажем, как с помощью лямбда-выражений устранять стереотипный код, связанный с популярными объектно-ориентированными паттернами проектирования.

9.2. Рефакторинг объектно-ориентированных паттернов проектирования с помощью лямбда-выражений

Новые возможности языков программирования часто снижают популярность существующих паттернов кода или идиом. Появление цикла `for-each` в Java 5, например, привело к замене множества явных итераторов в силу меньшей подверженности ошибкам и большей компактности. Появление ромбовидного оператора `<>` в Java 7

снизило объемы явного использования обобщенных типов при создании экземпляров (и понемногу вдохновило Java-программистов на применение вывода типов).

Особый класс паттернов составляют паттерны проектирования[1]. *Паттерны проектирования* (design patterns) — это, если хотите, переиспользуемые шаблоны для решения распространенных задач проектирования программного обеспечения. Они напоминают наборы стандартных решений, применяемых инженерами-строителями при конструировании мостов в конкретных сценариях (висячий мост, арочный мост и т. д.). *Паттерн проектирования «Посетитель»* (Visitor design pattern), например, представляет собой часто используемое решение для отделения алгоритма от структуры данных, с которой он работает. *Паттерн «Одиночка»* (Singleton pattern) — распространенное решение, позволяющее ограничить количество создаваемых экземпляров класса одним объектом.

Лямбда-выражения — еще один новый инструмент в арсенале программиста. Зачастую они позволяют решать те же задачи, что и паттерны проектирования, но проще и с меньшим объемом работы. Благодаря лямбда-выражениям многие из существующих паттернов проектирования становятся ненужными либо могут быть переписаны более лаконично.

В этом разделе мы рассмотрим пять паттернов проектирования:

- «Стратегия»;
- «Шаблонный метод»;
- «Наблюдатель»;
- «Цепочка обязанностей»;
- «Фабрика».

Мы покажем, как лямбда-выражения позволяют решить иначе все задачи, решаемые этими паттернами проектирования.

9.2.1. Стратегия

Паттерн *«Стратегия»* (Strategy) предлагает распространенный способ представления семейства алгоритмов и обеспечения возможности выбора между ними во время выполнения. Мы уже мельком встречались с этим паттерном проектирования в главе 2, когда демонстрировали фильтрацию склада с помощью различных предикатов (например, выбор тяжелых яблок или зеленых яблок). Этот паттерн можно использовать во множестве сценариев, в частности при проверке входных данных на соответствие различным критериям, использовании различных способов синтаксического разбора или форматировании входных данных.

Паттерн «Стратегия» состоит из трех частей, как показано на рис. 9.1.

- Интерфейс, отражающий какой-либо алгоритм (интерфейс `Strategy`).
- Одна или несколько конкретных реализаций этого интерфейса, отражающих несколько алгоритмов (конкретные классы `ConcreteStrategyA` и `ConcreteStrategyB`).
- Один или несколько клиентов, использующих объекты стратегий.

[1] См.: *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2010.

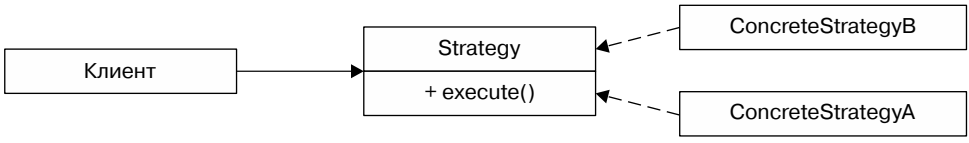


Рис. 9.1. Паттерн проектирования «Стратегия»

Пусть вам нужно проверить, корректно ли отформатирован входной текст по нескольким различным критериям (например, состоит только из символов нижнего регистра или цифр). Начнем с описания интерфейса для проверки текста (представленного в виде объекта `String`):

```
public interface ValidationStrategy {
    boolean execute(String s);
}
```

Далее описываем одну или несколько реализаций этого интерфейса:

```
public class IsAllLowerCase implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("[a-z]+");
    }
}
public class IsNumeric implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("\\d+");
    }
}
```

После этого можно использовать указанные стратегии проверки в своей программе:

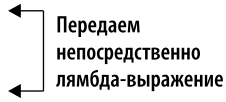
```
public class Validator {
    private final ValidationStrategy strategy;
    public Validator(ValidationStrategy v) {
        this.strategy = v;
    }
    public boolean validate(String s) {
        return strategy.execute(s);
    }
}
Validator numericValidator = new Validator(new IsNumeric());
boolean b1 = numericValidator.validate("aaaa");
Validator lowerCaseValidator = new Validator(new IsAllLowerCase ());
boolean b2 = lowerCaseValidator.validate("bbbb");
```

Возвращает false

Возвращает true

Использование лямбда-выражений. Вы уже, наверное, узнали в `ValidationStrategy` функциональный интерфейс. Кроме того, его функциональный дескриптор соответствует `Predicate<String>`. В результате вместо объявления новых классов для реализации различных стратегий можно передавать непосредственно более лаконичные лямбда-выражения:

```
Validator numericValidator =
    new Validator((String s) -> s.matches("[a-z]+"));
boolean b1 = numericValidator.validate("aaaa");
Validator lowerCaseValidator =
    new Validator((String s) -> s.matches("\\d+"));
boolean b2 = lowerCaseValidator.validate("bbbb");
```



Как вы можете видеть, лямбда-выражения позволяют устранить присущий этому паттерну проектирования стереотипный код. Если задуматься, становится понятно, что лямбда-выражения инкапсулируют элемент кода (стратегию) — то, для чего и создан паттерн, так что мы рекомендуем использовать вместо него (для решения соответствующих задач) лямбда-выражения.

9.2.2. Шаблонный метод

Паттерн проектирования *«Шаблонный метод»* (Template method) — решение, часто применяемое для представления основных контуров алгоритма с сохранением возможностей гибко менять определенные его части. Да, это звучит довольно абстрактно. Говоря другими словами, шаблонный метод удобен тогда, когда вы говорите себе: «Я бы с удовольствием воспользовался этим алгоритмом, но мне нужно поменять в нем несколько строк, чтобы он делал то, что мне требуется».

Вот пример работы этого паттерна. Пусть вам нужно написать простое приложение для онлайн-банкинга. Пользователи обычно вводят идентификатор клиента; приложение извлекает информацию о клиенте из базы данных банка и выполняет какие-то действия, чтобы осчастливить этого пользователя. Различные приложения онлайн-банкинга для разных подразделений банка могут осчастливить пользователя по-разному (например, добавлять бонусы на его счет или уменьшать поток документооборота с ним). Приложение для онлайн-банкинга можно представить с помощью следующего абстрактного класса:

```
abstract class OnlineBanking {
    public void processCustomer(int id){
        Customer c = Database.getCustomerWithId(id);
        makeCustomerHappy(c);
    }
    abstract void makeCustomerHappy(Customer c);
}
```

Метод `processCustomer` включает «скелет» алгоритма онлайн-банкинга: извлечь данные о клиенте по заданному идентификатору и осчастливить этого клиента. Теперь различные подразделения могут использовать различные реализации метода `makeCustomerHappy` благодаря наследованию класса `OnlineBanking`.

Использование лямбда-выражений. Решить ту же задачу (создание общей структуры алгоритма с возможностью подстановки отдельных частей) можно с помощью наших излюбленных лямбда-выражений. Подключаемые компоненты алгоритма в этом случае можно выразить в виде лямбда-выражений или ссылок на методы.

В следующем коде мы добавляем в метод `processCustomer` второй аргумент типа `Consumer<Customer>`, соответствующий сигнатуре описанного ранее метода `makeCustomerHappy`:

```
public void processCustomer(int id, Consumer<Customer> makeCustomerHappy) {
    Customer c = Database.getCustomerWithId(id);
    makeCustomerHappy.accept(c);
}
```

Теперь можно напрямую подключать различные варианты поведения, передавая лямбда-выражения, без наследования класса `OnlineBanking`:

```
new OnlineBankingLambda().processCustomer(1337, (Customer c) ->
     System.out.println("Hello " + c.getName());
```

Этот пример демонстрирует, как лямбда-выражения помогают устранять присущий паттернам проектирования стереотипный код.

9.2.3. Наблюдатель

Паттерн проектирования *«Наблюдатель»* (Observer) — распространенное решение для ситуации, когда объект (называемый *субъектом*) должен автоматически оповещать перечень других объектов (называемых *наблюдателями*) при возникновении какого-либо события (например, при изменении состояния). Обычно этот паттерн встречается при работе с GUI-приложениями. Для компонента GUI, такого как кнопка, регистрируется набор наблюдателей. При нажатии этой кнопки наблюдатели оповещаются и могут выполнять определенное действие. Но сфера использования паттерна «Наблюдатель» не ограничивается GUI. Он подойдет также в случае, когда несколько трейдеров (наблюдателей) хотели бы реагировать на изменение цены акции (субъекта). На рис. 9.2 приведена UML-диаграмма рассматриваемого паттерна.

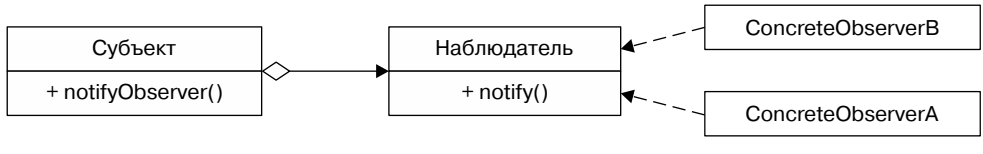


Рис. 9.2. Паттерн проектирования «Наблюдатель»

Теперь напишем код, чтобы узнать, насколько наблюдатель полезен на практике. Мы разработаем и реализуем свою систему оповещения для приложений вроде Twitter. Принцип ее работы прост: несколько новостных агентств (*The New York Times*, *The Guardian* и *Le Monde*) подписаны на ленту новостных твитов и хотели бы получать оповещение, если твит содержит заданное ключевое слово.

Во-первых, нам нужен интерфейс `Observer` для группировки наблюдателей. Он содержит один метод, `notify`, который субъект (`Feed`) будет вызывать при поступлении нового твита:

```
interface Observer {
    void notify(String tweet);
}
```

Теперь можно объявить различные наблюдатели (в данном случае три газеты) с различными действиями для каждого ключевого слова в твите:

```
class NYTimes implements Observer {
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("money")){
            System.out.println("Breaking news in NY! " + tweet);
        }
    }
}
class Guardian implements Observer {
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("queen")){
            System.out.println("Yet more news from London... " + tweet);
        }
    }
}
class LeMonde implements Observer {
    public void notify(String tweet) {
        if(tweet != null && tweet.contains("wine")){
            System.out.println("Today cheese, wine and news! " + tweet);
        }
    }
}
```

Здесь все еще не хватает важнейшей части — субъекта. Опишем интерфейс для субъекта следующим образом:

```
interface Subject {
    void registerObserver(Observer o);
    void notifyObservers(String tweet);
}
```

Субъект регистрирует новые наблюдатели с помощью метода `registerObserver` и оповещает их о появлении нового твита с помощью метода `notifyObservers`. Реализуем теперь класс `Feed`:

```
class Feed implements Subject {
    private final List<Observer> observers = new ArrayList<>();
    public void registerObserver(Observer o) {
        this.observers.add(o);
    }
    public void notifyObservers(String tweet) {
        observers.forEach(o -> o.notify(tweet));
    }
}
```

Эта реализация проста: внутри объекта `Feed` содержится список наблюдателей для оповещения при появлении нового твита. Создадим демонстрационное приложение, в котором субъект взаимодействовал бы с наблюдателями:

```
Feed f = new Feed();
f.registerObserver(new NYTimes());
```

```
f.registerObserver(new Guardian());
f.registerObserver(new LeMonde());
f.notifyObservers("The queen said her favourite book is Modern Java in
Action!");
```

Ничего удивительного, что этот твит заинтересовал *The Guardian*.

Использование лямбда-выражений. Наверное, вам интересно, как пользоваться лямбда-выражениями в сочетании с паттерном проектирования «Наблюдатель». Обратите внимание, что все классы, реализующие интерфейс `Observer`, включают реализацию одного-единственного метода — `notify`. Фактически это адаптеры для элемента поведения, выполняемого при поступлении твита. Лямбда-выражения как раз и предназначены для устранения подобного стереотипного кода. Вместо явного создания экземпляров трех объектов-наблюдателей можно передать непосредственно лямбда-выражения, соответствующие выполняемому поведению:

```
f.registerObserver((String tweet) -> {
        if(tweet != null && tweet.contains("money")){
            System.out.println("Breaking news in NY! " + tweet);
        }
});
f.registerObserver((String tweet) -> {
        if(tweet != null && tweet.contains("queen")){
            System.out.println("Yet more news from London... " + tweet);
        }
});
```

Всегда ли следует использовать лямбда-выражения? Нет. В вышеприведенном примере лямбда-выражения прекрасно подходят, поскольку выполняемое поведение — очень простое, так что с их помощью можно устранить стереотипный код. Но наблюдатели бывают и гораздо более сложными; они могут сохранять состояние, включать описание нескольких методов и т. п. В подобных случаях лучше использовать классы.

9.2.4. Цепочка обязанностей

Паттерн *«Цепочка обязанностей»* (Chain of responsibility) — решение, часто применяемое для создания последовательности обработки объектов (например, последовательности операций). Один объект-обработчик выполняет какие-либо действия и передает результат следующему объекту, который также выполняет какие-либо (другие) действия и передает еще одному объекту-обработчику и т. д.

Как правило, этот паттерн реализуется путем описания абстрактного класса для объекта-обработчика, в котором описывается поле для хранения следующего за ним объекта. По завершении работы объект-обработчик передает результаты следующему объекту.

Соответствующий код выглядит так:

```
public abstract class ProcessingObject<T> {
    protected ProcessingObject<T> successor;
```

```
    public void setSuccessor(ProcessingObject<T> successor){
        this.successor = successor;
    }
    public T handle(T input) {
        T r = handleWork(input);
        if(successor != null){
            return successor.handle(r);
        }
        return r;
    }
    abstract protected T handleWork(T input);
}
```

На рис. 9.3 приведен паттерн «Цепочка обязанностей» на языке UML.

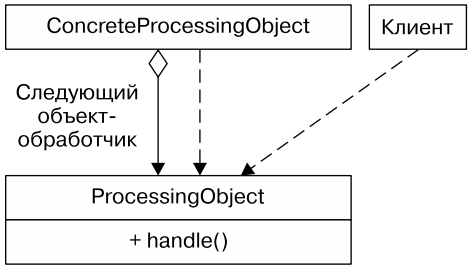


Рис. 9.3. Паттерн проектирования «Цепочка обязанностей»

Вы могли узнать здесь паттерн проектирования «Шаблонный метод», обсуждавшийся в подразделе 9.2.2. Метод `handle` содержит общую схему требуемых действий. Путем наследования класса `ProcessingObject` с реализацией метода `handleWork` можно создавать различные типы объектов-обработчиков.

Приведем пример использования этого паттерна. Создадим два объекта-обработчика для выполнения какой-то обработки текста:

```
public class HeaderTextProcessing extends ProcessingObject<String> {
    public String handleWork(String text) {
        return "From Raoul, Mario and Alan: " + text;
    }
}
public class SpellCheckerProcessing extends ProcessingObject<String> {
    public String handleWork(String text) {
        return text.replaceAll("labda", "lambda");
    }
}
```

Ой, мы забыли букву 'm' в "lambda"!

Теперь можно соединить оба объекта-обработчика в цепочку операций:

```
ProcessingObject<String> p1 = new HeaderTextProcessing();
ProcessingObject<String> p2 = new SpellCheckerProcessing();
p1.setSuccessor(p2);
```

Соединяем два объекта-обработчика в цепочку

```
String result = p1.handle("Aren't labdas really sexy?!!");
System.out.println(result);
```

Выводит "From Raoul, Mario and Alan: Aren't lambdas really sexy?!!"

Использование лямбда-выражений. Погодите-ка, этот паттерн очень напоминает связывание цепочкой (то есть композицию) функций. Мы уже обсуждали композицию лямбда-выражений в главе 3. Объекты-обработчики можно представить в виде экземпляров `Function<String, String>` или, точнее, `UnaryOperator<String>`. Для связывания их цепочкой необходимо объединить их с помощью метода `andThen`:

```
UnaryOperator<String> headerProcessing =
    (String text) -> "From Raoul, Mario and Alan: " + text;
UnaryOperator<String> spellCheckerProcessing =
    (String text) -> text.replaceAll("labda", "lambda");
Function<String, String> pipeline =
    headerProcessing.andThen(spellCheckerProcessing);
String result = pipeline.apply("Aren't labdas really sexy?!!");
```

Первый объект-обработчик

Второй объект-обработчик

В результате композиции этих двух функций получаем цепочку операций

9.2.5. Фабрика

С помощью паттерна проектирования «Фабрика» можно создавать объекты, не делая логику создания экземпляра видимой для клиента. Допустим, вы работаете на банк, которому нужно создавать различные финансовые продукты: займы (loans), облигации (bonds), акции (stocks) и т. д.

Обычно для этого создается класс `Factory`, содержащий метод, который отвечает за создание различных объектов, вот так:

```
public class ProductFactory {
    public static Product createProduct(String name) {
        switch(name){
            case "loan": return new Loan();
            case "stock": return new Stock();
            case "bond": return new Bond();
            default: throw new RuntimeException("No such product " + name);
        }
    }
}
```

Здесь `Loan`, `Stock` и `Bond` — подтипы типа `Product`. В методе `createProduct` может содержаться дополнительная логика задания настроек создаваемых продуктов. Преимущество состоит в том, что теперь можно создавать эти объекты, не делая конструктор и настройки доступными клиенту, что упрощает для последнего создание продуктов:

```
Product p = ProductFactory.createProduct("loan");
```

Использование лямбда-выражений. Вы уже видели в главе 3, что к конструкторам можно обращаться так же, как и к методам: с помощью ссылок на методы. Например, вот так можно ссылаться на конструктор класса `Loan`:

```
Supplier<Product> loanSupplier = Loan::new;
Loan loan = loanSupplier.get();
```

Воспользовавшись этой методикой, можно переписать предыдущий код, создав ассоциативный массив с названиями продуктов и конструкторов соответствующих классов:

```
final static Map<String, Supplier<Product>> map = new HashMap<>();
static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

С помощью этого ассоциативного массива можно создавать экземпляры различных продуктов аналогично тому, как мы делали при использовании фабрики:

```
public static Product createProduct(String name){
    Supplier<Product> p = map.get(name);
    if(p != null) return p.get();
    throw new IllegalArgumentException("No such product " + name);
}
```

Таким образом, благодаря этой возможности Java 8 можно легко достичь тех же целей, что и при использовании паттерна «Фабрика». Но возможности масштабирования при этом не столь хороши, например, если фабричному методу `createProduct` понадобится несколько аргументов для передачи конструкторам продуктов. Для этой цели лучше подойдет функциональный интерфейс, а не простой `Supplier`.

Пусть вам нужно ссылаться на конструкторы для продуктов, принимающие три аргумента (два объекта `Integer` и один `String`); для поддержки таких конструкторов необходимо создать специальный функциональный интерфейс `TriFunction`. В результате сигнатура нашего ассоциативного массива усложняется:

```
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}

Map<String, TriFunction<Integer, Integer, String, Product>> map
    = new HashMap<>();
```

Мы выяснили, как выполнять рефакторинг кода и писать новый код с лямбда-выражениями. В следующем разделе мы расскажем вам, как проверить правильность работы кода.

9.3. Тестирование лямбда-выражений

Мы приправили наш код лямбда-выражениями, и теперь он выглядит аккуратно и лаконично. Но большинству программистов платят деньги за написание не аккуратного, а прежде всего правильно работающего кода.

Чаще всего рекомендуемые практики разработки программного обеспечения включают модульное тестирование с целью проверки правильного поведения программы. С помощью тестовых сценариев контролируется правильность результатов, возвращаемых отдельными маленькими частями исходного кода. Рассмотрим следующий простой класс `Point` из приложения для работы с графикой:

```
public class Point {
    private final int x;
    private final int y;
    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public Point moveRightBy(int x) {
        return new Point(this.x + x, this.y);
    }
}
```

Следующий модульный тест проверяет, правильно ли работает метод `moveRightBy`:

```
@Test
public void testMoveRightBy() throws Exception {
    Point p1 = new Point(5, 5);
    Point p2 = p1.moveRightBy(10);
    assertEquals(15, p2.getX());
    assertEquals(5, p2.getY());
}
```

9.3.1. Тестирование поведения видимого лямбда-выражения

Этот код работает отлично, поскольку метод `moveRightBy` — общедоступный, а значит, его можно тестировать внутри тестового сценария. Но у лямбда-выражений нет названий (в конце концов, это же анонимные функции), так что тестировать их — непростая задача, поскольку нельзя обратиться к ним по названию.

Иногда лямбда-выражение доступно через какое-либо поле, так что его можно переиспользовать и хотелось бы протестировать заключенную в этом лямбда-выражении логику. Как это сделать? Можно протестировать лямбда-выражение аналогично тестированию вызова методов. Допустим, мы добавили в класс `Point` статическое поле `compareByXAndThenY`, предоставляющее доступ к объекту `Comparator`, сгенерированному на основе ссылок на методы:

```
public class Point {
    public final static Comparator<Point> compareByXAndThenY =
        comparing(Point::getX).thenComparing(Point::getY);
    ...
}
```

Напомним, что лямбда-выражения генерируют экземпляр функционального интерфейса. Таким образом, можно протестировать поведение этого экземпляра. Далее мы вызываем метод `compare` объекта-компаратора `compareByXAndThenY` с различными аргументами, чтобы протестировать правильность его поведения:

```
@Test
public void testComparingTwoPoints() throws Exception {
    Point p1 = new Point(10, 15);
    Point p2 = new Point(10, 20);
    int result = Point.compareByXAndThenY.compare(p1 , p2);
    assertTrue(result < 0);
}
```

9.3.2. Делаем упор на поведение метода, использующего лямбда-выражение

Задача лямбда-выражения все же состоит в инкапсуляции отдельного элемента поведения, используемого другим методом. В этом случае не стоит делать лямбда-выражения общедоступными; это всего лишь нюансы реализации. Вместо этого, мы убеждены, следует протестировать поведение использующего лямбда-выражение метода. Рассмотрим метод `moveAllPointsRightBy`:

```
public static List<Point> moveAllPointsRightBy(List<Point> points, int x) {
    return points.stream()
                 .map(p -> new Point(p.getX() + x, p.getY()))
                 .collect(toList());
}
```

Нет смысла в тестировании лямбда-выражения `p -> new Point(p.getX() + x, p.getY())`; это всего лишь нюанс реализации метода `moveAllPointsRightBy`. Вместо этого лучше сосредоточить внимание на тестировании метода `moveAllPointsRightBy`:

```
@Test
public void testMoveAllPointsRightBy() throws Exception {
    List<Point> points =
        Arrays.asList(new Point(5, 5), new Point(10, 5));
    List<Point> expectedPoints =
        Arrays.asList(new Point(15, 5), new Point(20, 5));
    List<Point> newPoints = Point.moveAllPointsRightBy(points, 10);
    assertEquals(expectedPoints, newPoints);
}
```

Отметим, что при модульном тестировании важно, чтобы в классе `Point` был правильно реализован метод `equals`, иначе будет использоваться реализация по умолчанию из класса `Object`.

9.3.3. Разносим сложные лямбда-выражения по отдельным методам

Допустим, вы столкнулись с действительно сложным лямбда-выражением, содержащим большой объем логики (что-то вроде алгоритма рыночного ценообразования с патологическими случаями). Сослаться на лямбда-выражение внутри теста невозможно, так что же делать? Одна из стратегий — преобразовать лямбда-выражение в ссылку на метод (для этого нужно объявить новый обычный метод), как мы объясняли в подразделе 9.1.3. Далее можно протестировать поведение этого нового метода аналогично любому другому методу.

9.3.4. Тестирование функций высшего порядка

Тестировать методы, принимающие функцию в качестве аргумента или возвращающие другую функцию (так называемые функции высшего порядка, о которых мы расскажем в главе 19), немного сложнее. Если метод принимает в качестве аргумента лямбда-выражение, можно протестировать его поведение при различных лямбда-выражениях. Например, можно протестировать созданный в главе 2 метод `filter` при различных предикатах:

```
@Test
public void testFilter() throws Exception {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
    List<Integer> even = filter(numbers, i -> i % 2 == 0);
    List<Integer> smallerThanThree = filter(numbers, i -> i < 3);
    assertEquals(Arrays.asList(2, 4), even);
    assertEquals(Arrays.asList(1, 2), smallerThanThree);
}
```

А что, если тестируемый метод возвращает другую функцию? Можно протестировать ее поведение, рассматривая ее как экземпляр функционального интерфейса, как мы показали вам ранее для случая компаратора.

К сожалению, код не всегда работает с первого раза, так что тесты могут показать наличие ошибок, связанных с использованием лямбда-выражений. Поэтому в следующем разделе мы займемся вопросом отладки.

9.4. Отладка

В арсенале разработчика есть два проверенных инструмента для отладки проблемного кода:

- изучение трассы вызовов в стеке;
- журналирование.

Лямбда-выражения и потоки данных могут внести новые проблемы в вашу обычную практику отладки. В этом разделе мы поговорим и о тех и о других.

9.4.1. Изучаем трассу вызовов в стеке

Если выполнение программы неожиданно прервалось (например, из-за генерации исключения), прежде всего нужно понять, в каком именно месте и как программа попала в это место. Для данной цели удобны фреймы стека. При каждом вызове метода программой генерируется информация о вызове, включая место вызова в программе, аргументы вызова и локальные переменные вызываемого метода. Эта информация хранится в одном из фреймов стека.

При сбое программы вам выдается *трасса вызовов в стеке* (stack trace) — краткая сводка предыстории этого сбоя, фрейм за фреймом. Другими словами, вы получаете ценный список вызовов методов вплоть до момента сбоя. Этот список дает возможность выяснить, как возникла проблема.

Использование лямбда-выражений. К сожалению, поскольку у лямбда-выражений отсутствуют названия, трасса вызовов может оказаться довольно запутанной. Рассмотрим следующий простой код, специально написанный так, чтобы привести к сбою:

```
import java.util.*;
public class Debugging{
    public static void main(String[] args) {
        List<Point> points = Arrays.asList(new Point(12, 2), null);
        points.stream().map(p -> p.getX()).forEach(System.out::println);
    }
}
```

Выполнение этого кода генерирует примерно следующую трассу вызовов (трасса может отличаться в зависимости от версии javac):

```
Exception in thread "main" java.lang.NullPointerException
    at Debugging.lambda$main$0(Debugging.java:6)
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
.java:193)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators
.java:948)
...
```

Что означает $0 в этой строке?

Тьфу! Что происходит? Конечно, программа завершилась сбоем, поскольку второй элемент списка точек равен `null` и мы пытаемся обработать пустую ссылку. А поскольку ошибка произошла в контейнере потока данных, мы видим всю последовательность вызовов методов, выполняемых при работе контейнера потока. Но, как вы видите, трасса вызовов включает следующие загадочные строки:

```
at Debugging.lambda$main$0(Debugging.java:6)
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
```

Эти строки означают, что ошибка произошла внутри лямбда-выражения. К сожалению, поскольку у лямбда-выражений отсутствуют названия, компилятору приходится генерировать какие-то названия, чтобы на них ссылаться. В данном случае название имеет вид `lambda$main$0` — не слишком интуитивное и малопонятное при наличии большого класса с несколькими лямбда-выражениями.

Даже если использовать ссылки на методы, трасса все равно может не содержать названия выполняемого метода. Замена вышеприведенного лямбда-выражения `p -> p.getX()` ссылкой на метод `Point::getX` также приводит к малопонятной трассе вызовов:

```
points.stream().map(Point::getX).forEach(System.out::println);
Exception in thread "main" java.lang.NullPointerException
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
.java:193)
...
```

Что означает эта строка?

Отметим, что если речь идет о ссылке на метод, объявленный в том же классе, где используется, то в трассе вызовов он будет отображен. В следующем примере:

```
import java.util.*;
public class Debugging{
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3);
        numbers.stream().map(Debugging::divideByZero).forEach(System.out::println);
    }
    public static int divideByZero(int n){
        return n / 0;
    }
}
```

метод `divideByZero` правильно отображается в трассе вызовов:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Debugging.divideByZero(Debugging.java:10)
    at Debugging$$Lambda$1/999966131.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)
...
```

Метод divideByZero указывается в трассе вызовов

Вообще говоря, имейте в виду, что разбираться в трассах вызовов в стеке, содержащих лямбда-выражения, сложнее. Это открывает широкое поле для усовершенствований в будущих версиях компилятора Java.

9.4.2. Журналирование информации

Допустим, вам нужно отладить конвейер операций потока данных. Как это сделать? Можно воспользоваться `forEach` для вывода в консоль или журналирования результата работы потока, вот так:

```
List<Integer> numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
       .map(x -> x + 17)
       .filter(x -> x % 2 == 0)
       .limit(3)
       .forEach(System.out::println);
```

Этот код выводит в консоль следующее:

```
20
22
```

К сожалению, после вызова `forEach` чтение потока завершается. А было бы удобно знать результат выполнения каждой из операций (`map`, `filter`, `limit`) в конвейере.

Здесь может оказаться полезной потоковая операция `peek`. Ее назначение — выполнять какое-либо действие над элементами потока данных по мере их потребления. Она не потребляет весь поток целиком, как `forEach`, а пересылает обработанный элемент далее, следующей операции в конвейере. Рисунок 9.4 иллюстрирует работу операции `peek`.

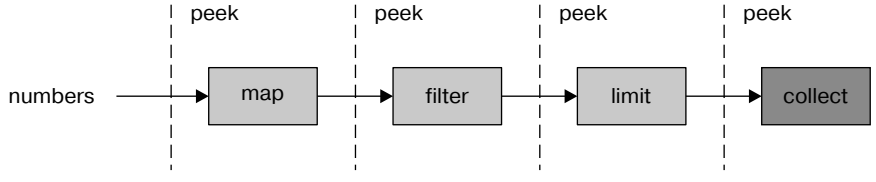


Рис. 9.4. Просмотр проходящих через поток данных значений с помощью операции peek

В следующем коде мы воспользуемся операцией `peek` для вывода промежуточных значений до и после каждой из операций потокового конвейера:

```
List<Integer> result =
  numbers.stream()
         .peek(x -> System.out.println("from stream: " + x))
         .map(x -> x + 17)
         .peek(x -> System.out.println("after map: " + x))
         .filter(x -> x % 2 == 0)
         .peek(x -> System.out.println("after filter: " + x))
         .limit(3)
         .peek(x -> System.out.println("after limit: " + x))
         .collect(toList());
```

Выводим в консоль текущий элемент, прочитанный из источника

Выводим в консоль результат выполнения операции map

Выводим в консоль выбранные числа после операции filter

Выводим в консоль выбранные числа после операции limit

Этот код выводит полезные сведения на каждом шаге конвейера:

```
from stream: 2
after map: 19
from stream: 3
after map: 20
after filter: 20
after limit: 20
from stream: 4
after map: 21
from stream: 5
after map: 22
after filter: 22
after limit: 22
```

Резюме

- Благодаря лямбда-выражениям можно сделать код более читабельным и гибким.
- Имеет смысл преобразовать анонимные классы в лямбда-выражения, не забывая при этом о небольших семантических различиях, например о смысле ключевого слова `this` и затенении переменных.
- Для повышения удобочитаемости кода вместо лямбда-выражений можно воспользоваться ссылками на методы.
- Имеет смысл рассмотреть вариант использования Stream API вместо итеративной обработки коллекций.
- Лямбда-выражения способны устранить стереотипный код, связанный с несколькими объектно-ориентированными паттернами проектирования, такими как «Стратегия», «Шаблонный метод», «Наблюдатель», «Цепочка обязанностей» и «Фабрика».
- Модульное тестирование лямбда-выражений возможно, но в целом лучше сосредоточить свое внимание на тестировании поведения методов, в которых эти лямбда-выражения встречаются.
- Сложные лямбда-выражения имеет смысл вынести в обычные методы.
- Лямбда-выражения могут снижать удобочитаемость трассы вызовов в стеке.
- Для журналирования промежуточных значений в определенных точках потокового конвейера удобен метод `peek` потока данных.

10 Предметно-ориентированные языки и лямбда-выражения

В этой главе

- Предметно-ориентированные языки (DSL) и их формы.
- За и против добавления предметно-ориентированного языка в API.
- Альтернативы простому DSL на основе Java, доступные на JVM.
- Учимся на примере предметно-ориентированных языков, уже существующих в интерфейсах и классах современного языка Java.
- Паттерны и методики реализации эффективных предметно-ориентированных языков на основе Java.
- Нюансы применения этих паттернов в распространенных библиотеках и утилитах языка Java.

Разработчики часто забывают, что язык программирования — прежде всего язык. Основная задача любого языка — доносить до адресата информацию наиболее понятным способом. И, вероятно, важнейшая черта хорошо написанного программного обеспечения — ясное выражение его назначения, или, как выразился знаменитый специалист в области компьютерных наук Гарольд Абельсон (Harold Abelson), «программы должны писаться главным образом для чтения людьми и лишь попутно — для выполнения машинами».

Удобочитаемость и понятность еще более важны в тех частях программного обеспечения, которые предназначены для решения основной задачи приложения. Производительность значительно повышается при написании кода, понятного как команде разработчиков, так и специалистам по предметной области. Последние могут участвовать в процессе разработки, проверяя правильность работы программного

обеспечения с коммерческой точки зрения. В результате можно на ранних стадиях обнаружить ошибки и неправильные представления.

Для этого бизнес-логику приложения часто выражают с помощью предметно-ориентированного языка (domain-specific language, DSL). *DSL* — это небольшой, обычно не универсальный язык программирования, специально спроектированный для работы с конкретной предметной областью. В DSL используется терминология, характерная для конкретной предметной области. Например, возможно, вы знакомы с Maven и Ant. Их можно рассматривать как предметно-ориентированные языки, предназначенные для выражения процесса сборки. Вы также знакомы с HTML — языком, специально предназначенным для описания структуры веб-страниц. Исторически вследствие отсутствия гибкости и лаконичности язык Java никогда не был популярен в качестве языка создания компактных DSL, подходящих для чтения неспециалистами. Однако теперь, когда Java поддерживает лямбда-выражения, в арсенале разработчика появляются новые инструменты! И действительно, из главы 3 вы узнали, что с помощью лямбда-выражений можно повысить лаконичность кода и соотношение «сигнал/шум» программ.

Рассмотрим реализованную на языке Java базу данных. Вероятно, глубоко в недрах этой базы можно найти огромный массив изощренного кода для определения места на диске для сохранения конкретной записи, для построения индексов таблиц и обработки конкурентных транзакций. Вероятно, такую базу данных должны разрабатывать достаточно опытные программисты. Допустим теперь, что нам нужно запрограммировать запрос, аналогичный приведенным в главах 4 и 5: «Найти все блюда из заданного меню, содержащие менее 400 калорий».

Ранее опытный программист мог быстро написать примерно следующий низкоуровневый код и думать, что задача оказалась легкой:

```
while (block != null) {
    read(block, buffer)
    for (every record in buffer) {
        if (record.calorie < 400) {
            System.out.println (record.name);
        }
    }
    block = buffer.next();
  }
```

В этом решении есть две проблемы: менее опытному программисту написать подобное будет непросто (могут понадобиться знания нюансов блокировки, операций ввода/вывода или выделения места на диске), и, что важнее, оно оперирует понятиями уровня системы, а не уровня приложения.

Программист-новичок, разрабатывающий клиентскую часть, может ответить на это: «Почему бы вам не сделать для меня SQL-интерфейс, чтобы я мог просто написать `SELECT name FROM menu WHERE calorie < 400`, где `menu` — SQL-таблица с меню ресторана? В этом случае я смог бы программировать гораздо эффективнее, чем при использовании всей этой ерунды системного уровня!» С таким утверждением не поспоришь! По существу, этому программисту нужен DSL для взаимодействия с базой данных вместо написания чистого Java-кода. Формально подобный тип DSL

называется *внешним*, поскольку от базы данных требуется API для синтаксического разбора и выполнения SQL-выражений, написанных в виде текста. Мы расскажем вам о различиях между внешним и внутренним DSL далее в этой главе.

Но если вы вспомните главы 4 и 5, то поймете, что этот код можно также написать на языке Java более лаконично с помощью Stream API, вот так:

```
menu.stream()
     .filter(d -> d.getCalories() < 400)
     .map(Dish::getName)
     .forEach(System.out::println)
```

Подобное применение связанных цепочкой методов, столь характерное для Stream API, часто называется *текучим стилем* (fluent style) в смысле легкости и понятности, в отличие от сложных потоков управления в циклах Java.

Такой стиль хорошо подходит для DSL. Это не внешний, а внутренний DSL. Во внутреннем DSL примитивы уровня приложения можно использовать в виде Java-методов для одного или нескольких типов классов, представляющих базу данных, в отличие от не-Java-синтаксиса примитивов внешнего DSL, таких как `SELECT FROM` из упомянутого выше SQL-запроса.

По существу, проектирование DSL состоит из выбора операций, необходимых для программиста, работающего на уровне приложения (причем важно избегать ненужного загрязнения понятиями уровня системы), и предоставления их ему.

В случае внутреннего DSL этот процесс включает предоставление соответствующих классов и методов для написания кода в текучем стиле. Внешний DSL требует бóльших усилий: требуется не только спроектировать синтаксис DSL, но и реализовать средство синтаксического анализа и выполнения кода на этом DSL. При правильной архитектуре, однако, даже менее опытные программисты могут писать код быстро и эффективно (таким образом зарабатывая деньги для компании) без необходимости писать прекрасный (но малопонятный для неспециалистов) код системного уровня!

В этой главе мы продемонстрируем, что такое DSL, на нескольких примерах и сценариях использования; вы узнаете, когда имеет смысл реализовать его и каковы выгоды от его использования. Далее мы рассмотрим некоторые из малых DSL, появившихся в API Java 8. Вы также узнаете, как применить вышеупомянутые паттерны для создания своих собственных DSL. Наконец, мы поговорим о широком использовании этих методик в популярных библиотеках и фреймворках языка Java для упрощения доступа к их API путем обращения к их функциональности через набор DSL.

10.1. Специальный язык для конкретной предметной области

DSL — это создаваемый пользователем язык, предназначенный для решения задач конкретной коммерческой предметной области. Представьте себе, что вы разрабатываете программное обеспечение для бухгалтерии. Ваша коммерческая предметная область включает такие понятия, как выписка о состоянии банковского счета, и такие операции, как согласование счетов. Для отражения задач в этой предметной области

можно создать пользовательский DSL. Для представления данной предметной области на языке Java необходимо создать набор классов и методов. В некотором смысле можно рассматривать DSL как API, предназначенный для взаимодействия с конкретной коммерческой предметной областью.

DSL — это не универсальный язык программирования; доступные операции и терминология ограничиваются конкретной предметной областью, а значит, у вас будет меньше забот и вы сможете уделить больше внимания решению бизнес-задач. Благодаря DSL пользователи должны иметь возможность думать только о сложностях предметной области. Все остальные нюансы низкоуровневой реализации должны быть скрыты — аналогично тому, как методы класса, отвечающие за нюансы низкоуровневой реализации, делаются приватными. В результате DSL будет удобным для использования.

Чем DSL не является? DSL — это не естественный язык. Это также не язык для реализации низкоуровневой бизнес-логики специалистами по предметной области. Назовем две причины для начала разработки DSL.

- *Главное — это передача информации.* Код должен четко выражать свои цели и быть понятным даже непрограммисту. Только в этом случае подобный непрограммист сможет внести свой вклад в проверки соответствия кода бизнес-требованиям.
- *Код пишут один раз, а читают многократно.* Удобочитаемость жизненно важна для сопровождения кода. Другими словами, писать код нужно всегда так, чтобы ваши сослуживцы благодарили вас за него, а не ругали!

У хорошо спроектированного DSL есть множество преимуществ. Тем не менее существуют аргументы как за, так и против разработки специализированных DSL. В подразделе 10.1.1 мы подробнее обсудим эти «за» и «против», чтобы вы смогли решить, подходит DSL для вашего конкретного сценария или нет.

10.1.1. За и против предметно-ориентированных языков

DSL, как и другие технологии и решения в сфере разработки программного обеспечения, вовсе не серебряная пуля[1]. Использование DSL для работы с вашей предметной областью может как помочь, так и помешать. Помочь за счет повышения уровня абстракции, благодаря чему можно яснее выразить бизнес-задачи кода и сделать его более удобочитаемым. А помешать потому, что реализация DSL — это тоже код, который нужно тестировать и сопровождать. Таким образом, имеет смысл заранее оценить стоимость внедрения DSL и выгоды от его использования, чтобы понять, окупятся ли затраты.

Выгоды от DSL включают следующее.

- *Краткость* — удобно инкапсулирующий бизнес-логику API позволяет избежать дублирования кода, благодаря чему код становится более лаконичным.

[1] См. https://ru.wikipedia.org/wiki/ Серебряной_пули_нет. — *Примеч. пер.*

- *Удобочитаемость* — благодаря использованию терминологии, присущей данной предметной области, код становится понятным даже для не специалистов в этой предметной области. А значит, код и знания предметной области сможет использовать более широкий круг работников организации.
- *Сопровождаемость* — написанный с помощью хорошо проектированного DSL код легче сопровождать и редактировать. Сопровождаемость особенно важна для кода бизнес-логики, поскольку эта часть приложения меняется чаще всего.
- *Более высокий уровень абстракции* — доступные в DSL операции работают на том же уровне абстракции, что и предметная область, скрывая таким образом детали реализации, не связанные напрямую с задачами предметной области.
- *Фокусировка* — наличие языка, созданного с единственной целью — выражать правила предметной области, помогает программисту сосредоточить внимание на данной конкретной части кода. В результате производительность работы программиста повышается.
- *Разделение обязанностей* — выражение бизнес-логики на специализированном языке упрощает изоляцию кода бизнес-логики от инфраструктурной части приложения. В результате получается более простой в сопровождении код.

И наоборот, у ввода DSL в базу кода есть несколько недостатков.

- *Сложность проектирования DSL* — выразить знания предметной области в сжатом, ограниченном языке весьма непросто.
- *Затраты на разработку* — добавление DSL в базу кода представляет собой долгосрочные инвестиции с высокими предварительными расходами, что может затянуть начальные стадии проекта. Кроме того, сопровождение DSL и его дальнейшая доработка требуют дополнительных затрат.
- *Дополнительный слой переадресации* — желательно, чтобы дополнительный слой, в который DSL обертывает модель предметной области, был как можно тоньше, во избежание проблем с производительностью.
- *Еще один язык, который нужно учить,* — сегодня программисты привыкли к работе с несколькими языками программирования. Добавление в проект DSL, однако, неявно означает, что вам и вашей команде нужно будет выучить на один язык больше. Хуже того, если вы захотите несколько DSL для различных частей предметной области, будет непросто идеально состыковать их, поскольку предметно-ориентированные языки склонны развиваться независимо друг от друга.
- *Ограничения, накладываемые базовым языком,* — некоторые универсальные языки программирования (в том числе Java) известны своей «многословностью» и негибким синтаксисом. Создать удобный в использовании DSL на их основе непросто. На самом деле разрабатываемые на основе подобного языка программирования DSL ограничиваются возможностями его громоздкого синтаксиса и вряд ли окажутся удобочитаемыми. Появившиеся в Java 8 лямбда-выражения — прекрасный новый инструмент для сглаживания данной проблемы.

При таких длинных списках аргументов за и против непросто решить, стоит ли разрабатывать DSL для конкретного проекта. Более того, существуют альтернативы языку Java для разработки своего собственного DSL. Прежде чем исследовать, с помощью каких паттернов и стратегий можно разработать удобочитаемый и легкий в использовании DSL на Java 8 и последующих версиях, мы вкратце перечислим эти альтернативы и расскажем, при каких обстоятельствах они могут пригодиться.

10.1.2. Доступные на JVM решения, подходящие для создания DSL

В этом разделе мы расскажем о различных типах DSL. Вы также узнаете о существовании множества других вариантов языков для создания DSL, помимо Java. В последующих же разделах мы сосредоточим наше внимание на реализации DSL с помощью возможностей Java.

Чаще всего для классификации DSL используется предложенное Мартином Фаулером разбиение их на внутренние и внешние DSL. Внутренние DSL (также называемые встраиваемыми) реализовываются поверх уже существующего базового языка программирования (например, обычного кода Java), в то время как внешние DSL называются автономными, поскольку разрабатываются с нуля и их синтаксис не зависит от базового языка.

Более того, JVM предоставляет еще и третий вариант, промежуточный: другой универсальный язык программирования, работающий на JVM, но более гибкий и выразительный, чем Java, например Scala или Groovy. Мы будем называть этот третий вариант многоязычным DSL.

Далее мы рассмотрим эти три типа DSL по очереди.

Внутренний DSL

Поскольку книга посвящена Java, то, говоря о внутреннем DSL, мы, конечно, подразумеваем написанный на Java DSL. Исторически Java никогда не считался подходящим для DSL языком программирования, поскольку его громоздкий и негибкий синтаксис делает почти невозможным создание удобочитаемого, лаконичного и выразительного DSL. Эту проблему в значительной степени свело на нет появление лямбда-выражений. Как вы уже видели в главе 3, лямбда-выражения пригодны для весьма лаконичной параметризации поведения. На самом деле использование лямбда-выражений приводит к DSL с намного более приемлемым соотношением «полезный сигнал/шум» за счет повышения лаконичности по сравнению с анонимными внутренними классами. Для демонстрации соотношения «полезный сигнал/шум» попробуйте вывести список строк с помощью синтаксиса Java 7, но используя появившийся в Java 8 новый метод `forEach`:

```
List<String> numbers = Arrays.asList("one", "two", "three");
numbers.forEach( new Consumer<String>() {
    @Override
    public void accept( String s ) {
        System.out.println(s);
    }
} );
```

В этом фрагменте кода выделенная жирным шрифтом часть заключает в себе полезный сигнал кода. Весь оставшийся код представляет собой синтаксический шум, не дающий никакой выгоды и (еще лучше) в Java 8 более совершенно ненужный. Этот анонимный внутренний класс можно заменить лямбда-выражением:

```
numbers.forEach(s -> System.out.println(s));
```

или даже еще более лаконично — ссылкой на метод:

```
numbers.forEach(System.out::println);
```

Построить DSL с помощью Java — замечательный вариант, если пользователи ожидаются технически грамотные. Если синтаксис Java не представляет сложностей, то у идеи разработки DSL на чистом Java есть множество достоинств.

- Затраты усилий на изучение паттернов и методик, необходимых для реализации хорошего DSL на Java, невелики по сравнению с усилиями, требуемыми для изучения нового языка программирования и утилит, обычно применяемых для разработки внешнего DSL.
- DSL, написанный на чистом Java, можно скомпилировать вместе с остальным вашим кодом. При сборке не нужно тратить ресурсы на интеграцию компилятора второго языка или утилиты, используемой для генерации внешнего DSL.
- Команде разработчиков не придется изучать другой язык программирования или разбираться в потенциально незнакомой и сложной внешней утилите.
- Для пользователей вашего DSL будут доступны все обычные возможности вашего любимого Java IDE, такие как автодополнение и средства рефакторинга. Современные IDE постепенно расширяют поддержку других популярных языков JVM, но эта поддержка все еще несравнима с имеющейся у Java-разработчиков.
- Если возникнет необходимость реализовать несколько DSL для различных частей предметной области или нескольких предметных областей, то при написании на чистом Java их можно будет без проблем сочетать.

Еще одна возможность — сочетание DSL, использующих одинаковый байт-код Java путем кобинирования основанных на JVM языков программирования. Такие DSL, называемые многоязычными, мы опишем далее.

Многоязычный DSL

В настоящее время на JVM работает более 100 языков программирования. Некоторые из них, например Scala и Groovy, весьма популярны, и найти опытных разработчиков на них несложно. Другие языки, в том числе JRuby и Jython, представляют собой адаптацию различных известных языков программирования к JVM. Наконец, еще часть активно развивающихся языков, например Kotlin и Ceylon, завоевывают популярность во многом благодаря тому, что заявляют о наличии возможностей, сравнимых с возможностями Scala, но при меньшей сложности и более пологой кривой обучения. Все эти языки программирования появились позже, чем Java, и отличаются менее жестким и более лаконичным синтаксисом. Вследствие такой

особенности этих языков программирования можно реализовывать на них более лаконичный DSL.

В частности, несколько полезных при разработке DSL возможностей есть у языка Scala, например каррирование и неявное преобразование типов. Обзор языка Scala и его сравнение с Java вы найдете в главе 20. А пока мы хотели бы лишь продемонстрировать вам на маленьком примере, что можно сделать благодаря этим возможностям.

Допустим, вам нужно написать вспомогательную функцию, которая бы повторяла выполнение другой функции, `f`, заданное количество раз. С первого раза у вас может получиться следующая рекурсивная реализация на Scala (не задумывайтесь насчет синтаксиса, нас интересует только общая идея):

```
def times(i: Int, f: => Unit): Unit = {
  f                                    ← Выполняем функцию f
  if (i > 1) times(i - 1, f)           ← Если счетчик i больше нуля, уменьшаем его
}                                        на единицу и рекурсивно вызываем функцию times
```

Обратите внимание, что на Scala вызов этой функции с большими значениями `i` не приведет к переполнению стека, как на Java, поскольку в языке Scala есть оптимизация хвостовой рекурсии, то есть рекурсивные вызовы функции `times` не будут добавляться в стек. Мы поговорим подробнее об этом в главах 18 и 19. С помощью этой функции можно рекурсивно вызывать другую функцию (три раза выводящую в консоль `"Hello World"`):

```
times(3, println("Hello World"))
```

При каррировании функции `times`, то есть разбиении ее аргументов на две группы (мы подробно обсудим каррирование в главе 19):

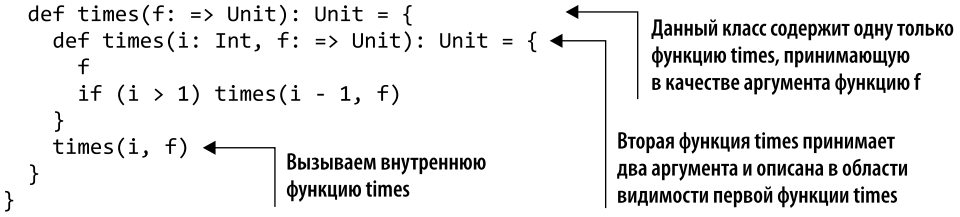
```
def times(i: Int)(f: => Unit): Unit = {
  f
  if (i > 1 times(i - 1)(f)
}
```

можно получить те же результаты, передав функцию для многократного выполнения в фигурных скобках:

```
times(3) {
  println("Hello World")
}
```

Наконец, в языке Scala можно задать неявное преобразование из типа `Int` в анонимный класс, объявив в нем только одну функцию. Ее аргументом будет служить функция, выполнение которой нужно повторять. Опять же не задумывайтесь о синтаксисе и нюансах. Цель этого примера — дать вам представление о возможностях, существующих за пределами языка Java.

```
implicit def intToTimes(i: Int) = new {   ← Описание неявного преобразования
                                            из типа Int в анонимный класс
```



Таким образом, пользователь вашего маленького DSL на основе Scala может выполнить функцию, которая три раза выводит в консоль `"Hello World"`:

```
3 times {
  println("Hello World")
}
```

Как видите, в получившемся коде отсутствует синтаксический шум — он понятен даже непрограммисту. Число `3` автоматически преобразуется компилятором в экземпляр класса, где оно хранится в поле `i`. После этого с помощью бесточечной нотации вызывается функция `times`, принимающая в качестве аргумента функцию, выполнение которой нужно повторять.

Добиться этого в Java невозможно, так что преимущества применения языка, более подходящего для создания DSL, очевидны. Впрочем, у такого варианта есть и явные недостатки.

- Необходимость изучать новый язык программирования или держать в команде человека, имеющего большой опыт работы с ним. Поверхностного знания будет недостаточно, поскольку разработка хорошего DSL на этих языках обычно требует применения относительно продвинутых возможностей.
- Процесс сборки несколько усложняется, необходима интеграция нескольких компиляторов для сборки исходного кода, написанного на двух языках программирования или более.
- Наконец, хотя большинство работающих на JVM языков преподносятся как на 100 % совместимые с Java, для взаимодействия их с Java часто приходится идти на компромиссы и выполнять различные неуклюжие трюки. Кроме того, подобное взаимодействие иногда приводит к снижению производительности. Коллекции Scala и Java не вполне совместимы, например, так что при передаче коллекции Scala в Java-функцию или наоборот приходится преобразовывать исходную коллекцию в коллекцию нативного API целевого языка.

Внешний DSL

Третий вариант добавления DSL в проект: реализация внешнего DSL. В этом случае необходимо спроектировать с нуля новый язык, с его собственным синтаксисом и семантикой. Необходимо также создать отдельную инфраструктуру для синтаксического разбора кода на этом новом языке, анализа результатов работы анализатора и генерации кода для выполнения команд этого внешнего DSL. Немалый объем

работы! Требуемые для выполнения этих задач навыки — далеко не самые распространенные, и обзавестись ими не так уж легко. Если же вы все-таки решите этим заниматься, то вам может помочь часто применяемый для данных целей генератор синтаксических анализаторов ANTLR, который отлично сочетается с Java.

Кроме того, даже просто спроектировать логически последовательный язык программирования с нуля — вовсе не тривиальная задача. Еще одна проблема: внешний DSL легко может выйти из-под контроля и распространиться на сферы и задачи, для которых не был предназначен.

Главная выгода от разработки внешнего DSL — практически неограниченная гибкость. Вы можете спроектировать такой язык, который будет идеально соответствовать потребностям и особенностям вашей предметной области. Если постараться, в результате можно получить исключительно удобочитаемый язык, идеально подогнанный под описание и решение задач вашего бизнеса. Еще один положительный результат — ясное разделение инфраструктурного кода на Java и бизнес-кода, написанного на внешнем DSL. Впрочем, такое разделение приводит к созданию искусственного слоя между DSL и базовым языком, так что это палка о двух концах.

Из оставшейся части главы вы узнаете о паттернах и методиках разработки эффективных внутренних DSL на основе современного языка Java. Мы начнем с обсуждения того, как эти идеи воплощались в архитектуре нативного API Java, особенно в дополнениях к API, появившихся в Java 8 и последующих версиях.

10.2. Малые DSL в современных API Java

Первые API, которые воспользовались новыми функциональными возможностями Java, — сами нативные API Java. До Java 8 в нативные API Java уже было включено несколько интерфейсов с одним абстрактным методом, но, как вы видели в разделе 10.1, их использование требовало реализации анонимного внутреннего класса с громоздким синтаксисом. Появление лямбда-выражений и (что, вероятно, еще важнее с точки зрения DSL) ссылок на методы полностью изменило правила игры и сделало функциональные интерфейсы краеугольным камнем архитектуры API Java.

В интерфейс `Comparator` в Java 8 было добавлено несколько новых методов. В главе 13 вы увидите, что интерфейс может включать как статические методы, так и методы с реализацией по умолчанию. А пока интерфейс `Comparator` может послужить для вас отличным примером того, как лямбда-выражения повышают переиспользуемость и сочетаемость методов нативного API Java.

Пусть у вас есть список объектов, отражающих различных людей (тип `Person`), и вам нужно отсортировать эти объекты по возрасту людей. До появления лямбда-выражений пришлось бы реализовать интерфейс `Comparator` с помощью внутреннего класса:

```
Collections.sort(persons, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
    }
});
```

Как вы уже видели во множестве примеров из этой книги, внутренний класс теперь можно заменить более лаконичным лямбда-выражением:

```
Collections.sort(people, (p1, p2) -> p1.getAge() - p2.getAge());
```

Даже еще лучше, можно заменить лямбда-выражение ссылкой на метод:

```
Collections.sort(persons, comparing(Person::getAge));
```

От такого подхода можно добиться еще больших выгод. Если нужно отсортировать людей по возрасту в порядке убывания, можно воспользоваться методом экземпляра `reverse` (также появившимся в Java 8):

```
Collections.sort(persons, comparing(Person::getAge).reverse());
```

Более того, чтобы отсортировать людей одинакового возраста в алфавитном порядке, можно скомпоновать этот компаратор с другим, для сравнения по именам:

```
Collections.sort(persons, comparing(Person::getAge)
                          .thenComparing(Person::getName));
```

Наконец, можно воспользоваться новым методом `sort`, добавленным в интерфейс `List`, чтобы сделать код еще более аккуратным:

```
persons.sort(comparing(Person::getAge)
             .thenComparing(Person::getName));
```

Этот малый DSL — минимальный DSL для предметной области сортировки коллекций. Несмотря на ограниченную область действия, он демонстрирует повышение производительности, переиспользуемости и сочетаемости кода за счет продуманного применения лямбда-выражений и ссылок на методы.

В следующем разделе мы изучим более ценный и шире используемый класс Java 8, в котором повышение удобочитаемости еще более очевидно: интерфейс `Stream`.

10.2.1. Stream API как DSL для работы с коллекциями

Интерфейс `Stream` — отличный пример малого внутреннего DSL в нативном API Java. На самом деле интерфейс `Stream` можно рассматривать как компактный, но обладающий широкими возможностями DSL для фильтрации, сортировки, преобразования, группировки и прочих операций над элементами коллекций. Допустим, вам нужно прочитать файл журнала и собрать в `List<String>` первые 40 его строк, начинающиеся со слова `"ERROR"`. Это можно сделать в императивном стиле, как показано в листинге 10.1.

Листинг 10.1. Чтение строк с сообщениями об ошибках из файла журнала в императивном стиле

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
BufferedReader bufferedReader
    = new BufferedReader(new FileReader(fileName));
String line = bufferedReader.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
```

```
            errors.add(line);
               errorCount++;
        }
        line = bufferedReader.readLine();
}
```

Для краткости мы опустили здесь код обработки ошибок. Несмотря на это, код получился, мягко говоря, не слишком компактным, да и его назначение неочевидно с первого взгляда. Кроме того, на удобство чтения и сопровождения этого кода отрицательно влияет отсутствие четкого разделения обязанностей. Фактически код с одинаковыми обязанностями разбросан по множеству операторов. Например, код для построчного чтения файла расположен в трех местах:

- там, где создается объект `FileReader`;
- во втором условии цикла `while`, которое проверяет, не закончился ли файл;
- в конце цикла `while`, при чтении следующей строки файла.

Аналогично код ограничения списка собираемых строк первыми 40 разбросан по трем операторам, таким как:

- оператор инициализации переменной `errorCount`;
- первое условие цикла `while`;
- оператор увеличения счетчика на единицу при обнаружении строки, начинающейся со слова `"ERROR"`.

Получить тот же результат в функциональном стиле с помощью интерфейса `Stream` можно гораздо проще, да и код окажется намного лаконичнее, как показано в листинге 10.2.

Листинг 10.2. Чтение строк с сообщениями об ошибках из файла журнала в функциональном стиле

```
List<String> errors = Files.lines(Paths.get(fileName))
                           .filter(line -> line.startsWith("ERROR"))
                           .limit(40)
                           .collect(toList());
```

Открываем файл и создаем поток строковых значений, каждое из которых соответствует строке файла

Отфильтровываем строки, начинающиеся с "ERROR"

Собираем полученные строковые значения в список

Ограничиваем результат первыми 40 строками

`Files.lines` — вспомогательный статический метод, возвращающий `Stream<String>`, каждый объект `String` в котором соответствует строке из анализируемого файла. Это единственная часть кода, которая выполняет построчное чтение файла. Аналогично оператора `limit(40)` оказывается вполне достаточно для ограничения списка собираемых строк с сообщениями об ошибках первыми 40. Можно ли представить себе нечто более понятное и удобочитаемое?

Текучий стиль Stream API — еще одна черта, характерная для хорошо спроектированных DSL. Выполнение всех промежуточных операций производится отложенным образом, они возвращают другой объект `Stream`, благодаря чему можно организовывать конвейер операций. Завершающая операция выполняется немедленно и запускает вычисление результата всего конвейера.

Пришло время познакомиться с API другого малого DSL, предназначенного для применения в связке с методом `collect` интерфейса `Stream`: `Collector`.

10.2.2. Коллекторы как предметно-ориентированный язык агрегирования данных

Вы уже видели, что интерфейс `Stream` можно рассматривать как DSL для работы с коллекциями данных. Аналогично интерфейс `Collector` можно рассматривать как DSL для агрегирования данных. В главе 6 мы исследовали интерфейс `Collector` и объясняли, как с его помощью собирать, группировать и секционировать элементы потока данных. Мы также обсудили статические фабричные методы, предоставляемые классом `Collectors` для удобного создания различных вариантов объектов-коллекторов и их комбинирования. Пришло время снова взглянуть на эти методы с точки зрения DSL. В частности, аналогично тому, как методы из интерфейса `Comparator` можно комбинировать для сортировки по нескольким полям, коллекторы можно комбинировать для многоуровневой группировки. Например, можно сгруппировать список машин сначала по их марке, а затем по цвету:

```
Map<String, Map<Color, List<Car>>> carsByBrandAndColor =
        cars.stream().collect(groupingBy(Car::getBrand,
                                         groupingBy(Car::getColor)));
```

Что здесь можно заметить по сравнению с тем, как мы выполняли конкатенацию двух компараторов? Мы описали компаратор для сравнения по нескольким полям посредством текучей композиции двух компараторов:

```
Comparator<Person> comparator =
        comparing(Person::getAge).thenComparing(Person::getName);
```

в то время как Collector API позволяет создать многоуровневый коллектор путем вложения коллекторов:

```
Collector<? super Car, ?, Map<Brand, Map<Color, List<Car>>>>
   carGroupingCollector =
       groupingBy(Car::getBrand, groupingBy(Car::getColor));
```

Обычно считается, что программы в текучем стиле читаются легче, чем программы во вложенном стиле, особенно если речь идет о композиции трех компонентов или более. Это различие в стиле отражает обдуманный выбор архитектуры, вызванный тем фактом, что сначала должен вычисляться сам внутренний коллектор, но с точки зрения логики данная группировка должна быть последней. В таком случае вложение операций создания коллекторов с помощью нескольких статических методов вместо текучей конкатенации их позволяет сначала выполнять саму внутреннюю группировку, хотя в коде она находится на последнем месте.

Проще будет (за исключением использования в описаниях обобщенных типов) реализовать класс `GroupingBuilder`, который бы передавал выполнение фабричному методу `groupingBy`, но предоставлял возможность текучей композиции нескольких операций группировки. В листинге 10.3 показано, как именно.

Листинг 10.3. Построитель коллекторов группировки

```
import static java.util.stream.Collectors.groupingBy;
public class GroupingBuilder<T, D, K> {
    private final Collector<? super T, ?, Map<K, D>> collector;
    private GroupingBuilder(Collector<? super T, ?, Map<K, D>> collector) {
        this.collector = collector;
    }
    public Collector<? super T, ?, Map<K, D>> get() {
        return collector;
    }
    public <J> GroupingBuilder<T, Map<K, D>, J>
                      after(Function<? super T, ? extends J> classifier) {
        return new GroupingBuilder<>(groupingBy(classifier, collector));
    }
    public static <T, D, K> GroupingBuilder<T, List<T>, K>
                     groupOn(Function<? super T, ? extends K> classifier) {
        return new GroupingBuilder<>(groupingBy(classifier));
    }
}
```

В чем проблема этого построителя (билдера, builder)? Она становится очевидной при попытке его использовать:

```
Collector<? super Car, ?, Map<Brand, Map<Color, List<Car>>>>
   carGroupingCollector =
       groupOn(Car::getColor).after(Car::getBrand).get()
```

Как можно видеть, использование этого вспомогательного класса не интуитивно, поскольку функции группировки приходится писать в обратном порядке относительно соответствующих вложенных уровней группировки. Если же попытаться переделать этот построитель так, чтобы решить проблему с упорядоченностью, то окажется, что система типов Java, увы, такой возможности не предоставляет.

Если вы взглянете поближе на нативный API Java и причины, лежащие в основе принятых при его проектировании решений, то сможете изучить несколько способов и полезных уловок для реализации удобочитаемых DSL. В следующем разделе мы продолжим изучать методики разработки эффективных DSL.

10.3. Паттерны и методики создания DSL на языке Java

DSL — это удобный и читабельный API для работы с моделью конкретной предметной области. Поэтому мы начнем данный раздел с описания простой модели предметной области, а затем обсудим способы, которые могут пригодиться при создании на ее основе DSL.

Наш пример модели предметной области состоит из трех элементов. Первый из них представляет собой простые компоненты Java, моделирующие акции, которые котируются на заданном рынке:

```
public class Stock {
    private String symbol;
    private String market;
    public String getSymbol() {
        return symbol;
    }
    public void setSymbol(String symbol) {
        this.symbol = symbol;
    }
    public String getMarket() {
        return market;
    }
    public void setMarket(String market) {
        this.market = market;
    }
}
```

Второй элемент — сделка по покупке или продаже заданного количества акций по указанной цене:

```
public class Trade {
    public enum Type { BUY, SELL }
    private Type type;
    private Stock stock;
    private int quantity;
    private double price;
    public Type getType() {
        return type;
    }
    public void setType(Type type) {
        this.type = type;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public Stock getStock() {
        return stock;
    }
    public void setStock(Stock stock) {
```

```
        this.stock = stock;
    }
    public double getValue() {
        return quantity * price;
    }
}
```

И наконец, заявка клиента на заключение одной или нескольких сделок:

```
public class Order {
    private String customer;
    private List<Trade> trades = new ArrayList<>();
    public void addTrade(Trade trade) {
        trades.add(trade);
    }
    public String getCustomer() {
        return customer;
    }
    public void setCustomer(String customer) {
        this.customer = customer;
    }
    public double getValue() {
        return trades.stream().mapToDouble(Trade::getValue).sum();
    }
}
```

Сама по себе модель предметной области очень проста. А вот создавать объекты, отражающие заявки, довольно неудобно. Попробуем описать простую заявку, включающую две сделки для клиента BigBank, как показано в листинге 10.4.

Листинг 10.4. Создание биржевой заявки непосредственно с помощью API объекта предметной области

```
Order order = new Order();
order.setCustomer("BigBank");

Trade trade1 = new Trade();
trade1.setType(Trade.Type.BUY);

Stock stock1 = new Stock();
stock1.setSymbol("IBM");
stock1.setMarket("NYSE");

trade1.setStock(stock1);
trade1.setPrice(125.00);
trade1.setQuantity(80);
order.addTrade(trade1);

Trade trade2 = new Trade();
trade2.setType(Trade.Type.BUY);

Stock stock2 = new Stock();
stock2.setSymbol("GOOGLE");
stock2.setMarket("NASDAQ");
```

```
trade2.setStock(stock2);
trade2.setPrice(375.00);
trade2.setQuantity(50);
order.addTrade(trade2);
```

Объемность этого кода вас вряд ли устроит; трудно ожидать, что специалист по предметной области, непрограммист, сможет понять и проверить его с первого взгляда. Нам нужен DSL, который бы отражал модель предметной области и позволял работать с ней более прямым, интуитивно понятным способом. Для достижения этого результата можно применить различные подходы. В оставшейся части данного раздела мы поговорим об аргументах за и против этих подходов.

10.3.1. Связывание методов цепочкой

Первый из обсуждаемых нами стилей DSL используется чаще всего. Он позволяет описать биржевую заявку с помощью единой цепочки вызовов методов. Пример такого типа DSL приведен в листинге 10.5.

Листинг 10.5. Создание заявки на покупку/продажу акций с помощью организации методов цепочкой

```
Order order = forCustomer( "BigBank" )
                  .buy( 80 )
                  .stock( "IBM" )
                      .on( "NYSE" )
                  .at( 125.00 )
                  .sell( 50 )
                  .stock( "GOOGLE" )
                      .on( "NASDAQ" )
                  .at( 375.00 )
              .end();
```

Этот код выглядит шагом вперед по сравнению с предыдущим, правда? Похоже, теперь ваш специалист по предметной области сможет без труда разобраться в нем. Но как реализовать DSL, чтобы добиться этого? Потребуется несколько построителей для создания объектов данной предметной области с использованием текучего API. Построитель верхнего уровня создает и обертывает заявку, благодаря чему появляется возможность добавления в нее сделок, как показано в листинге 10.6.

Листинг 10.6. Построитель заявок, предоставляющий DSL для организации методов цепочкой

```
public class MethodChainingOrderBuilder {

    public final Order order = new Order();

    private MethodChainingOrderBuilder(String customer) {
        order.setCustomer(customer);
    }

    public static MethodChainingOrderBuilder forCustomer(String customer) {
        return new MethodChainingOrderBuilder(customer);
```

Обернутая этим построителем заявка

Статический фабричный метод с целью создания построителя для размещенной заданным пользователем заявки

```
    }

    public TradeBuilder buy(int quantity) {
        return new TradeBuilder(this, Trade.Type.BUY, quantity);
    }

    public TradeBuilder sell(int quantity) {
        return new TradeBuilder(this, Trade.Type.SELL, quantity);
    }

    public MethodChainingOrderBuilder addTrade(Trade trade) {
        order.addTrade(trade);
        return this;
    }

    public Order end() {
        return order;
    }
}
```

Создает объект TradeBuilder для построения сделки покупки акций

Создает объект TradeBuilder для построения сделки продажи акций

Добавляет сделку в заявку

Возвращает сам построитель заявки, что позволяет создать и добавить последующие сделки

Завершает построение заявки и возвращает ее

Методы `buy()` и `sell()` построителя заявки создают и возвращают еще один построитель для формирования сделки и добавления ее в заявку:

```
public class TradeBuilder {
    private final MethodChainingOrderBuilder builder;
    public final Trade trade = new Trade();
    private TradeBuilder(MethodChainingOrderBuilder builder,
                         Trade.Type type, int quantity) {
        this.builder = builder;
        trade.setType( type );
        trade.setQuantity( quantity );
    }
    public StockBuilder stock(String symbol) {
        return new StockBuilder(builder, trade, symbol);
    }
}
```

Единственный общедоступный метод класса `TradeBuilder` служит для создания еще одного построителя, который затем формирует экземпляр класса `Stock`:

```
public class StockBuilder {
    private final MethodChainingOrderBuilder builder;
    private final Trade trade;
    private final Stock stock = new Stock();
    private StockBuilder(MethodChainingOrderBuilder builder,
                         Trade trade, String symbol) {
        this.builder = builder;
        this.trade = trade;
        stock.setSymbol(symbol);
    }
    public TradeBuilderWithStock on(String market) {
        stock.setMarket(market);
```

```
        trade.setStock(stock);
        return new TradeBuilderWithStock(builder, trade);
    }
}
```

Класс `StockBuilder` включает только один метод, `on()`, который задает рынок для акций, добавляет акции в сделку и возвращает еще один, последний построитель:

```
public class TradeBuilderWithStock {
    private final MethodChainingOrderBuilder builder;
    private final Trade trade;
    public TradeBuilderWithStock(MethodChainingOrderBuilder builder,
                                 Trade trade) {
        this.builder = builder;
        this.trade = trade;
    }
    public MethodChainingOrderBuilder at(double price) {
        trade.setPrice(price);
        return builder.addTrade(trade);
    }
}
```

Этот единственный общедоступный метод класса `TradeBuilderWithStock` задает цену за единицу для торгуемых акций и возвращает исходный построитель заявки. Как вы видели, с помощью этого метода можно добавлять другие сделки в заявки, пока не будет вызван метод `end` класса `MethodChainingOrderBuilder`. Мы решили, что лучше задействовать несколько классов построителей — и в частности два различных построителя сделок, — чтобы пользователям этого DSL пришлось вызывать методы его текучего API в заранее определенной последовательности, гарантируя таким образом правильную конфигурацию одной сделки перед началом создания следующей. Еще одно преимущество такого подхода заключается в том, что область видимости используемых для формирования заявки параметров ограничивается соответствующим построителем. При таком подходе применение статических методов сводится к минимуму, а названия методов могут выступать в роли именованных аргументов, что еще более повышает удобочитаемость DSL подобного типа. Наконец, уровень синтаксического шума у получаемого в результате этой методики текучего DSL минимален.

К сожалению, основная проблема метода связывания цепочкой — количество кода, необходимого для реализации построителей. Для комбинирования высокоуровневых построителей с низкоуровневыми потребуется много связующего кода. Еще один очевидный недостаток — невозможность обеспечить соблюдение привычных вам отступов для иерархии вложенности объектов предметной области.

В следующем подразделе мы поговорим о другом паттерне DSL, совсем с другими отличительными свойствами.

10.3.2. Вложенные функции

Своим названием паттерн DSL *«Вложенная функция»* обязан тому факту, что модель предметной области заполняется с помощью функций, вложенных в другие функции. Листинг 10.7 демонстрирует возникающий при таком подходе стиль DSL.

Листинг 10.7. Создание заявки на покупку/продажу акций с помощью вложенных функций

```
Order order = order("BigBank",
                    buy(80,
                        stock("IBM", on("NYSE")),
                        at(125.00)),
                    sell(50,
                         stock("GOOGLE", on("NASDAQ")),
                         at(375.00))
                   );
```

Для реализации подобного типа DSL требуется гораздо меньше кода, чем в варианте из подраздела 10.3.1.

Класс `NestedFunctionOrderBuilder` из листинга 10.8 демонстрирует возможность создания API с DSL подобного стиля (в этом листинге мы неявно предполагаем, что все статические методы импортированы).

Листинг 10.8. Построитель заявок с DSL на основе вложенных функций

```
public class NestedFunctionOrderBuilder {
    public static Order order(String customer, Trade... trades) {
        Order order = new Order();
        order.setCustomer(customer);
        Stream.of(trades).forEach(order::addTrade);
        return order;
    }

    public static Trade buy(int quantity, Stock stock, double price) {
        return buildTrade(quantity, stock, price, Trade.Type.BUY);
    }

    public static Trade sell(int quantity, Stock stock, double price) {
        return buildTrade(quantity, stock, price, Trade.Type.SELL);
    }

    private static Trade buildTrade(int quantity, Stock stock,
                                    double price, Trade.Type buy) {
        Trade trade = new Trade();
        trade.setQuantity(quantity);
        trade.setType(buy);
        trade.setStock(stock);
        trade.setPrice(price);
        return trade;
    }

    public static double at(double price) {
        return price;
    }

    public static Stock stock(String symbol, String market) {
        Stock stock = new Stock();
        stock.setSymbol(symbol);
```

Создаем заявку для заданного клиента

Добавляем все сделки в заявку

Создаем сделку для покупки акций

Создаем сделку для продажи акций

Фиктивный метод для указания цены за одну торгуемую акцию

Создаем торгуемый тип акций

```
        stock.setMarket(market);
        return stock;
    }

    public static String on(String market) {
        return market;
    }
}
```

Фиктивный метод для указания рынка, на котором торгуются данные акции

Еще одно преимущество данного подхода, по сравнению с другими методами, — вложение различных функций отражает иерархическую структуру объектов предметной области (заявка включает одну заявку или более, каждая из которых в этом примере относится к отдельному виду акций).

К сожалению, у этого паттерна тоже есть недостатки. Возможно, вы обратили внимание, что получившийся в итоге DSL требует использования огромного числа скобок. Более того, список передаваемых в статические методы аргументов жестко задан заранее. Если в объектах предметной области есть необязательные поля, придется реализовать различные перегруженные версии методов, которые позволили бы опускать отсутствующие параметры. Наконец, смысл различных аргументов определяется скорее по их позициям, а не по названиям. Последнюю из упомянутых проблем можно смягчить за счет создания нескольких фиктивных методов, аналогично тому, как мы сделали с методами `at()` и `on()` в классе `NestedFunctionOrderBuilder`, единственная задача которых состояла в прояснении смысла аргумента.

Два показанных выше паттерна DSL не требуют применения лямбда-выражений. В следующем разделе мы покажем третий способ, использующий появившиеся в Java 8 возможности функционального программирования.

10.3.3. Задание последовательности функций с помощью лямбда-выражений

В следующем паттерне DSL участвует последовательность функций, описанных с помощью лямбда-выражений. Реализация DSL подобным образом для нашей привычной предметной области торговли акциями позволяет описать заявку так, как показано в листинге 10.9.

Листинг 10.9. Создание заявки на продажу/покупку акций путем задания последовательности функций

```
Order order = order( o -> {
    o.forCustomer( "BigBank" );
    o.buy( t -> {
        t.quantity( 80 );
        t.price( 125.00 );
        t.stock( s -> {
            s.symbol( "IBM" );
            s.market( "NYSE" );
        } );
    });
    o.sell( t -> {
```

```
        t.quantity( 50 );
        t.price( 375.00 );
        t.stock( s -> {
            s.symbol( "GOOGLE" );
            s.market( "NASDAQ" );
        } );
    });
} );
```

Для реализации подобного подхода необходимо разработать несколько построителей, принимающих в качестве параметров лямбда-выражения, и заполнить модель предметной области путем их выполнения. В этих построителях хранится промежуточное состояние создаваемых объектов, аналогично вышеприведенной DSL-реализации со связыванием методов цепочкой. Как и в паттерне связывания цепочкой, построитель верхнего уровня создает заявку, но принимает в качестве параметров объекты типа `Consumer`, чтобы пользователь DSL мог реализовать их с помощью лямбда-выражений. В листинге 10.10 приведен код, необходимый для реализации этого подхода.

Листинг 10.10. Построитель заявки, предоставляющий DSL для задания последовательности функций

```
public class LambdaOrderBuilder {

    private Order order = new Order();

    public static Order order(Consumer<LambdaOrderBuilder> consumer) {
        LambdaOrderBuilder builder = new LambdaOrderBuilder();
        consumer.accept(builder);
        return builder.order;
    }

    public void forCustomer(String customer) {
        order.setCustomer(customer);
    }

    public void buy(Consumer<TradeBuilder> consumer) {
        trade(consumer, Trade.Type.BUY);
    }

    public void sell(Consumer<TradeBuilder> consumer) {
        trade(consumer, Trade.Type.SELL);
    }

    private void trade(Consumer<TradeBuilder> consumer, Trade.Type type) {
        TradeBuilder builder = new TradeBuilder();
        builder.trade.setType(type);
        consumer.accept(builder);
        order.addTrade(builder.trade);
    }
}
```

Обернутая этим построителем заявка

Выполняет переданное в построитель заявки лямбда-выражение

Возвращает заявку, заполненную путем выполнения объекта Consumer для типа OrderBuilder

Задает разместившего заявку клиента

Потребляет объект TradeBuilder для создания сделки покупки акций

Потребляет объект TradeBuilder для создания сделки продажи акций

Выполняет переданное в TradeBuilder лямбда-выражение

Добавляет в заявку сделку, заполненную путем выполнения объекта Consumer для типа TradeBuilder

Методы `buy()` и `sell()` построителя заявки принимают два лямбда-выражения типа `Consumer<TradeBuilder>`. При выполнении эти методы заполняют сделку покупки/продажи, вот так:

```
public class TradeBuilder {

    private Trade trade = new Trade();

    public void quantity(int quantity) {
        trade.setQuantity( quantity );
    }

    public void price(double price) {
        trade.setPrice( price );
    }

    public void stock(Consumer<StockBuilder> consumer) {
        StockBuilder builder = new StockBuilder();
        consumer.accept(builder);
        trade.setStock(builder.stock);
    }
}
```

Наконец, объект `TradeBuilder` принимает на входе `Consumer` третьего построителя, задача которого состоит в задании торгуемых акций:

```
public class StockBuilder {
    private Stock stock = new Stock();

    public void symbol(String symbol) {
        stock.setSymbol( symbol );
    }

    public void market(String market) {
        stock.setMarket( market );
    }
}
```

Достоинство этого паттерна состоит в сочетании двух положительных свойств предыдущих типов DSL. Аналогично паттерну связывания методов цепочкой он позволяет описывать заявку на сделку текучим образом. Помимо этого, аналогично стилю вложения функций иерархическая структура объектов предметной области отражается уровнями вложенности различных лямбда-выражений.

К сожалению, такой подход требует большого объема вспомогательного кода, а использованию самого DSL мешает синтаксический шум лямбда-выражений Java 8.

Выбор между этими тремя стилями DSL — дело вкуса. Для того чтобы выбрать лучше всего подходящий для конкретной предметной области предметно-ориентированный язык, требуется также определенный опыт. Кроме того, можно сочетать два или более из этих стилей в одном DSL, как вы увидите в следующем подразделе.

10.3.4. Собираем все воедино

Как вы видели выше, у каждого из этих трех паттернов DSL есть свои «за» и «против», но ничто не мешает вам использовать их все вместе в одном DSL. Можно разработать DSL, с помощью которого вы сможете описать заявку на торговлю акциями так, как показано в листинге 10.11.

Листинг 10.11. Создание заявки на торговлю акциями с применением нескольких паттернов DSL

```
Order order =
        forCustomer( "BigBank",
                     buy( t -> t.quantity( 80 )
                               .stock( "IBM" )
                               .on( "NYSE" )
                               .at( 125.00 )),
                     sell( t -> t.quantity( 50 )
                                .stock( "GOOGLE" )
                                .on( "NASDAQ" )
                                .at( 125.00 )) );
```

Связывание цепочкой методов в теле лямбда-выражения, заполняющего объект сделки

Вложенная функция, предназначенная для задания атрибутов заявки верхнего уровня

Лямбда-выражение для создания отдельной сделки

В этом примере паттерн вложения функций сочетается с подходом использования лямбда-выражений. Каждая из сделок создается объектом `Consumer` для типа `TradeBuilder`, реализуемым лямбда-выражением, как показано в листинге 10.12.

Листинг 10.12. Построитель заявки, который предоставляет DSL, сочетающий несколько стилей

```
public class MixedBuilder {
    public static Order forCustomer(String customer,
                                    TradeBuilder... builders) {
        Order order = new Order();
        order.setCustomer(customer);
        Stream.of(builders).forEach(b -> order.addTrade(b.trade));
        return order;
    }
    public static TradeBuilder buy(Consumer<TradeBuilder> consumer) {
        return buildTrade(consumer, Trade.Type.BUY);
    }
    public static TradeBuilder sell(Consumer<TradeBuilder> consumer) {
        return buildTrade(consumer, Trade.Type.SELL);
    }
    private static TradeBuilder buildTrade(Consumer<TradeBuilder> consumer,
                                           Trade.Type buy) {
        TradeBuilder builder = new TradeBuilder();
        builder.trade.setType(buy);
        consumer.accept(builder);
        return builder;
    }
}
```

Наконец, вспомогательный класс `TradeBuilder`, используемый внутри класса `StockBuilder` (его реализация приведена сразу после этого абзаца), обеспечивает

текучий API, который реализует паттерн связывания методов цепочкой. Если выбрать такой вариант, можно записать тело лямбда-выражения, с помощью которого будет заполняться сделка, наиболее компактным образом:

```
public class TradeBuilder {
    private Trade trade = new Trade();

    public TradeBuilder quantity(int quantity) {
        trade.setQuantity(quantity);
        return this;
    }

    public TradeBuilder at(double price) {
        trade.setPrice(price);
        return this;
    }

    public StockBuilder stock(String symbol) {
        return new StockBuilder(this, trade, symbol);
    }
}

public class StockBuilder {
    private final TradeBuilder builder;
    private final Trade trade;
    private final Stock stock = new Stock();

    private StockBuilder(TradeBuilder builder, Trade trade, String symbol){
        this.builder = builder;
        this.trade = trade;
        stock.setSymbol(symbol);
    }
    public TradeBuilder on(String market) {
        stock.setMarket(market);
        trade.setStock(stock);
        return builder;
    }
}
```

Листинг 10.12 — пример получения удобочитаемого DSL при сочетании трех обсуждавшихся в этой главе паттернов DSL. Он позволяет воспользоваться достоинствами различных стилей DSL, но есть и небольшой недостаток: полученный в итоге DSL оказывается не столь однородным, как получившийся в результате применения единого подхода, так что пользователям такого DSL понадобится больше времени для его изучения.

До сих пор мы задействовали лямбда-выражения, но, как демонстрируют интерфейсы `Comparator` и `Stream`, с помощью ссылок на методы можно еще более повысить удобочитаемость многих DSL. Мы продемонстрируем это в следующем подразделе на практическом примере применения ссылок на методы в модели предметной области торговли акциями.

10.3.5. Использование ссылок на методы в DSL

Здесь мы попытаемся добавить в модель предметной области торговли акциями еще один простой функциональный элемент. Он предназначен для вычисления окончательной суммы заявки после добавления (или не добавления в случае нулевого значения) налогов к чистой сумме заявки, как показано в листинге 10.13.

Листинг 10.13. Налоги, которые могут применяться к чистой сумме заявки

```
public class Tax {
    public static double regional(double value) {
        return value * 1.1;
    }
    public static double general(double value) {
        return value * 1.3;
    }
    public static double surcharge(double value) {
        return value * 1.05;
    }
}
```

Простейший способ реализации подобного калькулятора налогов — воспользоваться статическим методом, принимающим в качестве параметра заявку и по одному булеву флагу для каждого из потенциальных налогов (листинг 10.14).

Листинг 10.14. Применение налогов к чистой сумме заявки с помощью набора булевых флагов

```
public static double calculate(Order order, boolean useRegional,
                               boolean useGeneral, boolean useSurcharge) {
    double value = order.getValue();
    if (useRegional) value = Tax.regional(value);
    if (useGeneral) value = Tax.general(value);
    if (useSurcharge) value = Tax.surcharge(value);
    return value;
}
```

Окончательную сумму заявки после учета местного налога и добавочного налога (но не общего налога) можно вычислить следующим образом:

```
double value = calculate(order, true, false, true);
```

Проблема с удобочитаемостью этой реализации очевидна: запомнить последовательность булевых переменных и понять, какие налоги были применены, а какие — нет, очень непросто. Традиционный способ решения этой проблемы — реализация класса `TaxCalculator`, предоставляющего минимальный DSL для текучей установки значений булевых флагов одного за другим, как показано в листинге 10.15.

Листинг 10.15. Калькулятор налогов, задающий применяемые налоги текучим образом

```
public class TaxCalculator {
    private boolean useRegional;
    private boolean useGeneral;
```

```
    private boolean useSurcharge;

    public TaxCalculator withTaxRegional() {
        useRegional = true;
        return this;
    }

    public TaxCalculator withTaxGeneral() {
        useGeneral= true;
        return this;
    }

    public TaxCalculator withTaxSurcharge() {
        useSurcharge = true;
        return this;
    }

    public double calculate(Order order) {
        return calculate(order, useRegional, useGeneral, useSurcharge);
    }
}
```

При использовании этого `TaxCalculator` становится очевидно, что нужно применить к чистой сумме заявки местный налог и добавочный налог:

```
double value = new TaxCalculator().withTaxRegional()
                                  .withTaxSurcharge()
                                  .calculate(order);
```

Основная проблема этого решения — недостаточно лаконичный код. Оно плохо масштабируется, поскольку для каждого налога в предметной области требуется дополнительное булево поле и дополнительный метод. Благодаря функциональным возможностям Java можно достичь того же результата (в смысле удобочитаемости) гораздо более лаконичным и гибким способом. Чтобы продемонстрировать это, мы переделаем `TaxCalculator` так, как показано в листинге 10.16.

Для такого программного решения достаточно только одного поля: функции, которая за один проход прибавляет к чистой сумме заявки все налоги, заданные объектом `TaxCalculator`. Исходное значение этой функции — тождественная функция. В данный момент еще не было прибавлено никаких налогов, так что окончательная сумма заявки равна чистой сумме. При добавлении нового налога с помощью метода `with()` он комбинируется с текущей функцией вычисления налога. Таким образом, все добавляемые налоги охватываются одной функцией. Наконец, при передаче заявки в метод `calculate()` функция вычисления налога, полученная в результате композиции различных заданных налогов, применяется к чистой сумме заявки. Этот переработанный `TaxCalculator` можно использовать следующим образом:

```
double value = new TaxCalculator().with(Tax::regional)
                                  .with(Tax::surcharge)
                                  .calculate(order);
```

Листинг 10.16. Калькулятор налогов, сочетающий функции для применяемых налогов

```
public class TaxCalculator {
   public DoubleUnaryOperator taxFunction = d -> d;

    public TaxCalculator with(DoubleUnaryOperator f) {
        taxFunction = taxFunction.andThen(f);
        return this;
    }

    public double calculate(Order order) {
        return taxFunction.applyAsDouble(order.getValue());
    }
}
```

Функция, вычисляющая все применяемые к сумме заявки налоги

Получаем новую функцию вычисления налога путем композиции текущей функции с функцией, переданной в качестве аргумента

Вычисляем итоговую сумму заявки путем применения функции вычисления налога к чистой сумме заявки

Возвращаем this, благодаря чему становится возможным текучее присоединение к цепочке последующих функций вычисления налогов

Код в случае подобного решения, использующего ссылки на методы, компактен, и его легко читать. Он также гибок в смысле возможности немедленного, без каких-либо модификаций, применения добавляемых в класс `Tax` новых функций вычисления налогов благодаря нашему функциональному классу `TaxCalculator`.

Мы обсудили различные способы реализации DSL в Java 8 и последующих версиях. Интересно было бы теперь посмотреть, как эти стратегии применялись в популярных инструментах и фреймворках Java.

10.4. Реальные примеры DSL Java 8

В разделе 10.3 мы рассказали вам о трех полезных паттернах для разработки DSL на языке Java, а также об их достоинствах и недостатках. В табл. 10.1 приведена краткая сводка всей этой информации.

Таблица 10.1. Достоинства и недостатки паттернов DSL

Паттерн	Достоинства	Недостатки
Связывание методов цепочкой	• Названия методов могут выступать в роли именованных аргументов. • Хорошо подходит для необязательных параметров. • Пользователя DSL можно заставить вызывать методы в заранее заданном порядке. • Статические методы не используются или почти не используются. • Минимально возможный уровень синтаксического шума	• Очень пространная реализация. • Для связывания построителей требуется специальный код. • Иерархия объектов предметной области определяется только отступами

Паттерн	Достоинства	Недостатки
Вложенные функции	• Наиболее лаконичная реализация. • Вложенность функций отражает иерархию объектов предметной области	• Активно используются статические методы. • Аргументы определяются по их позиции, а не по названию. • Для использования необязательных параметров требуется перегрузка методов
Задание последовательности функций с помощью лямбда-выражений	• Хорошо подходит для необязательных параметров. • Статические методы не используются или почти не используются. • Вложенность лямбда-выражений отражает иерархию объектов предметной области. • Для построителей не требуется связующий код	• Очень пространная реализация. • Больший объем синтаксического шума в DSL из-за лямбда-выражений

Пора объединить полученные знания и посмотреть, как эти паттерны применяются в трех известных библиотеках Java: в утилите объектно-реляционного отображения, фреймворке разработки на основе поведения и утилите, которая реализует корпоративные паттерны интеграции.

10.4.1. jOOQ

SQL — один из самых распространенных и широко используемых DSL, поэтому неудивительно, что существует библиотека Java, предоставляющая удобный DSL для написания и выполнения SQL-запросов. jOOQ — внутренний DSL, реализующий SQL в виде встроенного в Java типобезопасного языка. Генератор исходного кода восстанавливает схему базы данных, благодаря чему компилятор Java может проверять типы в сложных SQL-операторах. В результате этого процесса восстановления генерируется информация, позволяющая исследовать схему базы данных. В качестве простого примера приведем следующий SQL-запрос:

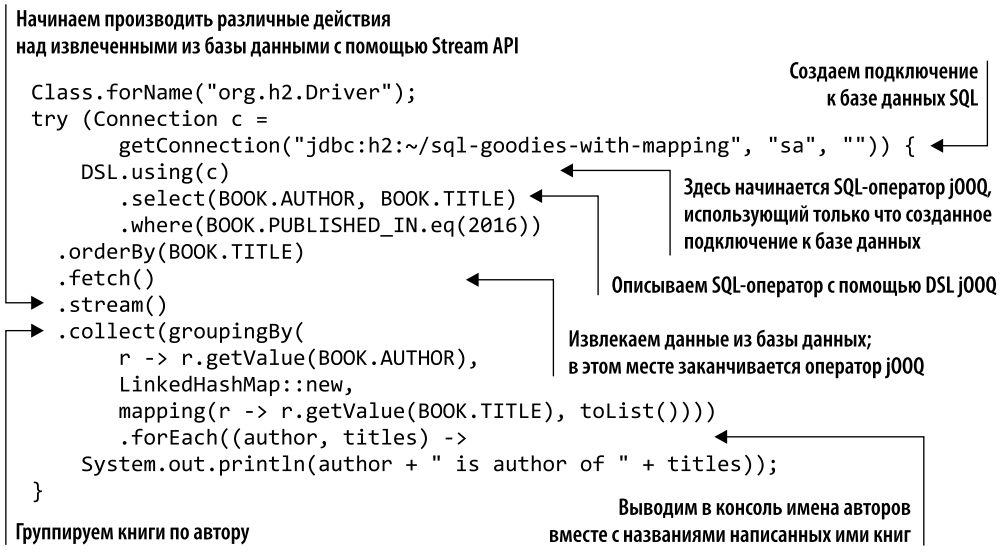
```
SELECT * FROM BOOK
WHERE BOOK.PUBLISHED_IN = 2016
ORDER BY BOOK.TITLE
```

Его можно переписать с помощью DSL jOOQ в таком виде:

```
create.selectFrom(BOOK)
      .where(BOOK.PUBLISHED_IN.eq(2016))
      .orderBy(BOOK.TITLE)
```

Еще одна приятная возможность DSL jOOQ — его можно использовать вместе со Stream API. Благодаря этой возможности над полученными в результате SQL-запроса данными можно выполнять операции в оперативной памяти, как показано в листинге 10.17.

Листинг 10.17. Выбор книг из базы данных с помощью DSL jOOQ




```
Class.forName("org.h2.Driver");
try (Connection c =
        getConnection("jdbc:h2:~/sql-goodies-with-mapping", "sa", "")) {
    DSL.using(c)
        .select(BOOK.AUTHOR, BOOK.TITLE)
        .where(BOOK.PUBLISHED_IN.eq(2016))
  .orderBy(BOOK.TITLE)
  .fetch()
  .stream()
  .collect(groupingBy(
        r -> r.getValue(BOOK.AUTHOR),
        LinkedHashMap::new,
        mapping(r -> r.getValue(BOOK.TITLE), toList())))
        .forEach((author, titles) ->
    System.out.println(author + " is author of " + titles));
}
```

Очевидно, что основной паттерн DSL, выбранный для реализации DSL jOOQ, — связывание методов цепочкой. На самом деле различные особенности этого паттерна (возможность использования необязательных параметров и навязывание вызова определенных методов в заранее заданном порядке) абсолютно необходимы для имитации синтаксиса корректного SQL-запроса. Благодаря этим возможностям, а также низкому уровню синтаксического шума паттерн связывания методов цепочкой отлично подходит для нужд jOOQ.

10.4.2. Cucumber

Разработка на основе поведения (behavior-driven development, BDD) — расширение концепции разработки через тестирование (test-driven development), использующее простой предметно-ориентированный скриптовый язык, который состоит из структурированных операторов, описывающих различные бизнес-сценарии. Cucumber, как и другие фреймворки BDD, преобразует эти операторы в выполняемые тестовые сценарии. Полученные в результате использования этой методики сценарии можно задействовать в качестве и исполняемых тестов, и критериев приемлемости заданной функциональной бизнес-возможности. В BDD основные усилия разработчиков сосредотачиваются на поставке приоритетных, проверенных бизнес-возможностей и сокращении разрыва между экспертами по предметной области и программистами за счет общей для тех и других бизнес-терминологии.

Поясним эти абстрактные понятия на практическом примере использования Cucumber — на примере BDD-утилиты, благодаря которой разработчики могут писать бизнес-сценарии на обычном естественном (в данном случае английском) языке. Описать простой бизнес-сценарий с помощью Cucumber можно следующим образом:

```
Feature: Buy stock
  Scenario: Buy 10 IBM stocks
```

```
Given the price of a "IBM" stock is 125$
When I buy 10 "IBM"
Then the order value should be 1250$
```

В Cucumber применяется нотация, которая делится на три части: описание предварительных условий (`Given`), фактическое обращение к тестируемым объектам предметной области (`When`) и контрольные утверждения для проверки результатов тестового сценария (`Then`).

Тестовый сценарий пишется на внешнем DSL с ограниченным числом ключевых слов и возможностью написания операторов в произвольном формате. Эти операторы сопоставляются с регулярными выражениями, которые захватывают переменные тестового сценария и передают их в методы, реализующие сам тест. На основе модели предметной области для торговли акциями из раздела 10.3 можно разработать тестовый сценарий Cucumber, чтобы проверить правильность вычисления суммы заявки на продажу/покупку акций, как показано в листинге 10.18.

Листинг 10.18. Реализация тестового сценария с помощью аннотаций Cucumber

```
public class BuyStocksSteps {
    private Map<String, Integer> stockUnitPrices = new HashMap<>();
    private Order order = new Order();

    @Given("^the price of a \"(.*?)\" stock is (\\d+)\\$$")
    public void setUnitPrice(String stockName, int unitPrice) {
        stockUnitValues.put(stockName, unitPrice);
    }

    @When("^I buy (\\d+) \"(.*?)\"$")
    public void buyStocks(int quantity, String stockName) {
        Trade trade = new Trade();
        trade.setType(Trade.Type.BUY);
        Stock stock = new Stock();
        stock.setSymbol(stockName);
        trade.setStock(stock);
        trade.setPrice(stockUnitPrices.get(stockName));
        trade.setQuantity(quantity);
        order.addTrade(trade);
    }

    @Then("^the order value should be (\\d+)\\$$")
    public void checkOrderValue(int expectedValue) {
        assertEquals(expectedValue, order.getValue());
    }
}
```

Задаем предварительное условие данного сценария: цену за одну акцию

Сохраняем цену за одну акцию

Задаем действия, применяемые к тестируемой модели предметной области

Заполняем модель предметной области соответствующим образом

Задаем ожидаемый результат сценария

Проверяем контрольные утверждения теста

Появление в Java 8 лямбда-выражений позволило Cucumber выработать альтернативный синтаксис без использования аннотаций с помощью методов, принимающих два аргумента: регулярное выражение, ранее содержавшееся в аргументе аннотации, и лямбда-выражение, реализующее тестовый метод. При использовании второго варианта нотации тестовый сценарий можно переписать вот так:

```
public class BuyStocksSteps implements cucumber.api.java8.En {
    private Map<String, Integer> stockUnitPrices = new HashMap<>();
```

```
    private Order order = new Order();
    public BuyStocksSteps() {
        Given("^the price of a \"(.*?)\" stock is (\\d+)\\$$",
              (String stockName, int unitPrice) -> {
                  stockUnitValues.put(stockName, unitPrice);
        });
        // ... Лямбда-выражения When и Then опущены для краткости
    }
}
```

Лаконичность — очевидное преимущество этого альтернативного синтаксиса. В частности, благодаря замене тестовых методов анонимными лямбда-выражениями вы освобождаетесь от необходимости придумывать осмысленные названия методов (которые редко повышают удобочитаемость тестового сценария).

DSL Cucumber очень прост, но он демонстрирует возможности эффективного сочетания внешнего DSL с внутренним и опять же показывает, что благодаря лямбда-выражениям можно писать более компактный и удобочитаемый код.

10.4.3. Фреймворк Spring Integration

Фреймворк *Spring Integration* расширяет основанную на внедрении зависимостей модель программирования Spring, обеспечивая поддержку известных корпоративных паттернов интеграции[1]. Основные задачи фреймворка Spring Integration — обеспечить простую модель для реализации сложных решений по корпоративной интеграции и способствовать внедрению асинхронной, ориентированной на обмен сообщениями архитектуры.

Благодаря фреймворку Spring Integration в основанных на Spring приложениях становятся возможными упрощенные удаленное взаимодействие, обмен сообщениями и распределение задач. Эти возможности доступны и через полнофункциональный, текучий DSL, который представляет собой отнюдь не просто «синтаксический сахар» поверх обычных файлов конфигурации Spring XML.

Фреймворк Spring Integration реализует все наиболее распространенные паттерны, необходимые для приложений, ориентированных на обмен сообщениями, в том числе каналы, конечные точки, средства опроса и перехватчики каналов. Для повышения удобочитаемости конечные точки выражаются в DSL «глаголами», а интеграционные процессы формируются путем композиции этих конечных точек в один или несколько потоков сообщений. Листинг 10.19 демонстрирует на простом, но завершенном примере, как работает Spring Integration.

Здесь метод `myFlow()` создает интеграционный поток с помощью DSL Spring Integration. Он использует текучий построитель, предоставляемый классом `IntegrationFlows`, который реализует паттерн связывания методов цепочкой. В данном случае полученный поток опрашивает источник сообщений через фиксированные промежутки времени, в результате чего получается последовательность целочисленных значений. Отфильтровываем четные и преобразуем их в строковые значения,

[1] Более подробную информацию можно найти в книге: *Hohpe G., Woolf B.* Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions. — Addison-Wesley, 2004.

а затем отправляем результат в выходной канал в стиле, напоминающем нативный Stream API Java 8. Этот API позволяет отправлять сообщение любому компоненту в интеграционном потоке, если известно название его `inputChannel`. Если интеграционный поток начинается с прямого канала[1], а не с источника сообщений, то можно описать поток интеграции с помощью лямбда-выражения, вот так:

```
@Bean
public IntegrationFlow myFlow() {
    return flow -> flow.filter((Integer p) -> p % 2 == 0)
                       .transform(Object::toString)
                       .handle(System.out::println);
}
```

Листинг 10.19. Настройка потока сообщений Spring Integration с помощью DSL Spring Integration

```
@Configuration
@EnableIntegration
public class MyConfiguration {

    @Bean
    public MessageSource<?> integerMessageSource() {
        MethodInvokingMessageSource source =
                new MethodInvokingMessageSource();
        source.setObject(new AtomicInteger());
        source.setMethodName("getAndIncrement");
        return source;
    }

    @Bean
    public DirectChannel inputChannel() {
        return new DirectChannel();
    }

    @Bean
    public IntegrationFlow myFlow() {
        return IntegrationFlows
                    .from(this.integerMessageSource(),
                          c -> c.poller(Pollers.fixedRate(10)))
                    .channel(this.inputChannel())
                    .filter((Integer p) -> p % 2 == 0)
                    .transform(Object::toString)
                    .channel(MessageChannels.queue("queueChannel"))
                    .get();
    }
}
```

Создаем новый источник сообщений, увеличивающий при каждом вызове значение объекта AtomicInteger на единицу

Канал для передачи поступающих от источника сообщений данных

Начинаем создание интеграционного потока с помощью построителя, следующего паттерну связывания методов цепочкой

Использует вышеупомянутый источник сообщений в качестве источника для этого интеграционного потока

Опрашиваем указанный источник сообщений для удаления из очереди передаваемых им данных

Отфильтровываем только четные числа

Указываем в качестве канала вывода для данного интеграционного потока канал queueChannel

Завершаем построение интеграционного потока и возвращаем его

Преобразуем извлеченные из источника сообщений целочисленные значения (Integer) в строковые (String)

[1] См. https://docs.spring.io/spring-integration/api/org/springframework/integration/channel/DirectChannel.html. — *Примеч. пер.*

Как видите, чаще всего в DSL Spring Integration используется паттерн связывания методов цепочкой. Он отлично подходит для основной задачи построителя интеграционного потока: создания потока для передачи сообщений и преобразований данных. Как показано в этом последнем примере, для построения высокоуровневого объекта (а в некоторых случаях и для внутренних, более сложных аргументов методов) также предусмотрено задание последовательности функций с помощью лямбда-выражений.

Резюме

- Основная задача DSL — заполнение разрыва между разработчиками и специалистами по предметной области. Редко когда человек, пишущий код для реализации бизнес-логики приложения, обладает глубокими познаниями в бизнес-сфере, в которой данное приложение будет использоваться. Написание этой бизнес-логики на языке, понятном неразработчикам, не делает специалистов по предметной области программистами, но позволяет им читать и проверять логику.
- Два основных типа DSL — *внутренний* (реализуется на том же языке программирования, что и приложение, в котором этот DSL будет использоваться) и *внешний* (применяется другой, специально созданный для этого язык). Разработка внутреннего DSL требует меньше усилий, но его синтаксис ограничивается базовым языком. Реализация внешнего DSL сложнее, но и степень его гибкости выше.
- Можно также создать многоязычный DSL, воспользовавшись еще одним языком программирования, работающим на DSL, например Scala или Groovy. Эти языки зачастую гибче и лаконичнее Java. Интеграция их с Java, однако, усложняет процесс сборки, а взаимодействие с Java вряд ли окажется идеальным.
- Вследствие излишней «многословности» и негибкого синтаксиса Java не лучший язык программирования для создания внутренних DSL, но появление лямбда-выражений и ссылок на методы в Java 8 существенно улучшило положение дел.
- В нативном API современного языка Java уже есть несколько малых DSL. Эти DSL, подобно DSL в классах `Stream` и `Collectors`, удобны и полезны, в частности, для сортировки, фильтрации, преобразования и группировки коллекций данных.
- Основные три паттерна, применяемые при реализации DSL на языке Java, — это связывание методов цепочкой, вложенные функции и задание последовательности функций. У каждого из них есть достоинства и недостатки, но при желании можно сочетать все три паттерна в одном DSL, чтобы воспользоваться сильными сторонами всех трех подходов.
- Множество фреймворков и библиотек Java можно использовать посредством DSL. В этой главе мы рассмотрели три из них: jOOQ — утилиту объектно-реляционного отображения; Cucumber — BDD-фреймворк; Spring Integration — расширение фреймворка Spring, реализующее корпоративные паттерны интеграции.

Часть IV

Java на каждый день

Четвертая часть этой книги посвящена различным новым возможностям Java 8 и Java 9, касающимся упрощения написания кода и повышения его надежности. Мы начнем с двух API, появившихся в Java 8.

В главе 11 описывается класс `java.util.Optional`, позволяющий проектировать лучшие API, а заодно и снизить число исключений, связанных с пустыми указателями.

Глава 12 посвящена изучению нового Date and Time API (дат и времени), который существенно лучше подверженных частым ошибкам предыдущих API для работы с датой/временем.

Далее мы расскажем вам о расширениях Java 8 и 9, предназначенных для написания больших систем и обеспечения возможности их развития.

Из главы 13 вы узнаете, что такое методы с реализацией по умолчанию, как развивать с их помощью API с сохранением совместимости; познакомитесь с некоторыми паттернами и правилами эффективного использования методов с реализацией по умолчанию на практике.

Глава 14 написана специально для второго издания, в ней изучается система модулей Java — масштабное новшество Java 9, позволяющее разбивать громадные системы на хорошо документированные модули вместо «беспорядочного набора пакетов».

11 Класс Optional как лучшая альтернатива null

В этой главе

- Чем плохи пустые ссылки и почему их следует избегать.
- От `null` к `Optional`: переписываем модель предметной области `null`-безопасным образом.
- Начинаем использовать опционалы: очищаем код от проверок на `null`.
- Различные способы чтения значений, которые могут содержаться в опционалах.
- Пересматриваем программу с учетом потенциально отсутствующих значений.

Поднимите руку, если за свою карьеру Java-разработчика вы хоть раз получали исключение `NullPointerException`. Оставьте ее поднятой, если это исключение — наиболее частое из встречавшихся вам. К сожалению, мы не видим вас сейчас, но очень вероятно, что ваша рука поднята. Мы также подозреваем, что вы можете думать что-то вроде: «Да, согласен. Исключения `NullPointerException` — головная боль для любого Java-разработчика, неважно, новичка или эксперта. Но поделать с ними все равно ничего нельзя, это цена, которую мы платим за использование такой удобной и, вероятно, неизбежной конструкции, как пустые ссылки». Это общее мнение в мире (императивного) программирования; тем не менее, возможно, это не вся правда, а скорее глубоко укоренившееся предубеждение.

Британский специалист в области компьютерных наук Тони Хоар (Tony Hoare), создавший пустые ссылки еще в 1965 году, при разработке языка ALGOL W, одного из первых типизированных языков программирования с записями, память под которые выделялась в куче, позднее признался, что сделал это «просто из-за легкости реализации». Хотя он хотел гарантировать «полную безопасность использования

всех ссылок с автоматической проверкой их компилятором», но решил сделать исключение для пустых ссылок, поскольку думал, что это самый удобный способ смоделировать *отсутствие значения*. Много лет спустя он пожалел об этом решении, назвав его «моя ошибка на миллиард долларов». Мы все видели результаты этого решения. Например, мы можем проверять поле объекта, чтобы определить, представляет ли оно одно из двух возможных значений, лишь для того, чтобы обнаружить, что проверяем не объект, а нулевой указатель, и тут же получить это надоедливое исключение `NullPointerException`.

На самом деле Хоар, возможно, даже недооценил масштаб затрат на исправление миллионами разработчиков ошибок, вызванных пустыми ссылками за последние 50 лет. И действительно, абсолютное большинство созданных за последние десятилетия языков программирования[1], включая Java, основывается на том же самом проектном решении, возможно, по причинам совместимости с более старыми языками или (что более вероятно), как сказал Хоар, «просто из-за легкости реализации». Мы начнем с демонстрации простого примера проблем, возникающих при использовании `null`.

11.1. Как смоделировать отсутствие значения

Представьте себе, что у вас есть приведенная в листинге 11.1 вложенная структура объектов для владельца автомобиля, купившего автостраховку.

Листинг 11.1. Модель данных Person/Car/Insurance

```
public class Person {
    private Car car;
    public Car getCar() { return car; }
}
public class Car {
    private Insurance insurance;
    public Insurance getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

Как вы думаете, в чем проблема следующего кода?

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

Этот код выглядит вполне разумно, но у многих людей нет автомобилей, так какой же в этом случае будет результат вызова метода `getCar`? Зачастую (и совершенно напрасно) возвращают пустую ссылку, чтобы указать на отсутствие значения (в данном случае чтобы указать на отсутствие машины). В результате вызов метода `getInsurance`

[1] Среди заслуживающих внимания исключений — большинство типизированных функциональных языков, таких как Haskell и ML. Они включают алгебраические типы данных, позволяющие лаконично выражать типы данных вместе с явным указанием, следует ли включать особые значения, такие как null.

вернет страховку пустой ссылки, что приведет к генерации `NullPointerException` во время выполнения и остановку программы. Но это еще не все. А что, если объект `person` был равен `null`? Что, если метод `getInsurance` тоже вернул `null`?

11.1.1. Снижение количества исключений NullPointerException с помощью проверки на безопасность

Как избежать неожиданных `NullPointerException`? Обычно можно добавить проверки на `null` везде, где нужно (а иногда, превышая требования безопасного программирования, и там, где не нужно), причем зачастую в различных стилях. Первая наша попытка написать метод, который бы предотвращал генерацию `NullPointerException`, показана в листинге 11.2.

Листинг 11.2. Null-безопасный код, попытка 1: глубокие сомнения

```
public String getCarInsuranceName(Person person) {
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                  return insurance.getName();
            }
        }
    }
    return "Unknown";
}
```

Каждая проверка на null повышает вложенность оставшейся части цепочки вызовов

Этот метод выполняет проверку на `null` при каждом разыменовании переменной, возвращая строковое значение `"Unknown"`, если хоть одна из переменных, встречавшихся в этой цепочке разыменования, представляет собой пустое значение. Единственное исключение из данного правила — мы не проверяем на `null` название страховой компании, поскольку *знаем*, что (как и у любой другой компании) у нее обязано быть название. Обратите внимание, что этой последней проверки нам удается избежать только за счет знаний предметной области, но этот факт не отражен в Java-классах, моделирующих наши данные.

Приведенный в листинге 11.2 метод мы описали как «глубокие сомнения», поскольку в нем заметен повторяющийся паттерн: каждый раз в случае сомнений, не равна ли переменная `null`, приходится добавлять еще один вложенный блок `if`, повышая тем самым уровень отступов кода. Очевидно, что эта методика плохо масштабируется и снижает удобочитаемость, так что лучше попробовать другое решение. Во избежание приведенной проблемы пойдем другим путем, как показано в листинге 11.3.

Во второй попытке мы пробуем избежать глубокого вложения блоков `if` с помощью другой стратегии: всякий раз, когда мы натыкаемся на равную `null` переменную, возвращаем строковое значение `"Unknown"`. Но это решение тоже далеко от идеала; теперь метод насчитывает четыре различные точки выхода, что сильно усложняет его сопровождение. Вдобавок возвращаемое в случае `null` значение по умолчанию — строка `"Unknown"` — повторяется в трех местах и (мы надеемся) не содержит орфографических ошибок! Во избежание этого можно, конечно, вынести повторяющуюся строку в константу.

Листинг 11.3. Null-безопасный код, попытка 2: слишком много выходов

```
public String getCarInsuranceName(Person person) {
    if (person == null) {
        return "Unknown";
    }
    Car car = person.getCar();
    if (car == null) {
        return "Unknown";
    }
    Insurance insurance = car.getInsurance();
    if (insurance == null) {
        return "Unknown";
    }
    return insurance.getName();
}
```

Более того, этот процесс подвержен ошибкам. Что, если вы забудете проверить, равно ли `null` одно из свойств? В этой главе мы приведем аргументы в пользу того, что использование `null` для представления отсутствия значения — принципиально ошибочный подход. Требуется лучший способ моделирования отсутствия и наличия значения.

11.1.2. Проблемы, возникающие с null

Если подвести итог вышесказанного, использование пустых ссылок в Java приводит к следующим, как теоретическим, так и практическим проблемам.

- *Служит источником ошибок.* `NullPointerException` — наиболее распространенное (с огромным отрывом) исключение в Java.
- *«Раздувает» код.* Ухудшает удобочитаемость из-за необходимости заполнять код проверками на `null`, зачастую глубоко вложенными.
- *Бессмысленно.* У него отсутствует какой-либо семантический смысл, в частности, такой подход к моделированию отсутствия значения в языке со статической типизацией принципиально ошибочен.
- *Нарушает идеологию языка Java.* Java всегда скрывает указатели от разработчиков, за одним исключением: нулевой указатель.
- *Создает брешь в системе типов.* `null` не включает никакого типа или другой информации, так что его можно присвоить любому типу ссылки. Это может приводить к проблемам при передаче `null` в другую часть системы, где отсутствует информация о том, чем изначально должен быть этот `null`.

В качестве основы для других допустимых решений в следующем подразделе мы вкратце рассмотрим возможности, предлагаемые другими языками программирования.

11.1.3. Альтернативы null в других языках программирования

В последние годы такие языки, как Groovy, сумели обойти данную проблему за счет появления *оператора безопасного вызова* (`?.`), предназначенного для безопасной

работы со значениями, потенциально равными null. Чтобы разобраться, как этот процесс осуществляется на практике, рассмотрим следующий код на языке Groovy. В нем извлекается название страховой компании, в которой заданный человек застраховал машину:

```
def carInsuranceName = person?.car?.insurance?.name
```

Вам должно быть ясно, что делает этот код. У человека может не быть машины, для моделирования чего мы присваиваем null ссылке car объекта person. Аналогично машина может оказаться незастрахованной. Оператор безопасного вызова языка Groovy позволяет безопасно работать с потенциально пустыми ссылками, без генерации исключения NullPointerException, передавая пустую ссылку далее по цепочке вызовов и возвращая null, если любое значение из цепочки равно null.

Аналогичную возможность предлагалось внедрить в Java 7, но затем было принято решение этого не делать. Однако, как ни странно, оператор безопасного вызова не так уж нужен в Java. Первая мысль любого Java-разработчика при столкновении с NullPointerException — быстро исправить проблему, добавив оператор if для проверки значения на null перед вызовом его метода. Решение проблемы подобным образом, без учета того, допустим ли null в этой конкретной ситуации для вашего алгоритма или модели данных, приводит не к исправлению, а к скрытию ошибки. И впоследствии для очередного разработчика (вероятно, вас самого через неделю или месяц) найти и исправить эту ошибку будет гораздо сложнее. Фактически при этом вы просто сметаете мусор под ковер. null-безопасный оператор разыменования языка Groovy — всего лишь большая и более мощная метла, с помощью которой можно делать подобные глупости, особо не волнуясь о последствиях.

Другие функциональные языки программирования, например Haskell и Scala, смотрят на эту проблему иначе. В Haskell есть тип Maybe, по сути инкапсулирующий опциональное значение. Объект типа Maybe может содержать значение заданного типа или ничего не содержать. В Haskell отсутствует понятие пустой ссылки. В Scala для инкапсуляции наличия или отсутствия значения типа T предусмотрена схожая логическая структура Option[T], которую мы обсудим в главе 20. При этом необходимо явным образом проверять, присутствует ли значение, с помощью операций типа Option, который обеспечивает «проверки на null». Теперь уже невозможно «забыть проверить на null», поскольку проверки требует сама система типов.

Ладно, мы немного отклонились от темы, и все это звучит довольно абстрактно. Наверное, вам интересно, что в этом смысле предлагает Java 8. Вдохновившись идеей опционального значения, создатели Java 8 ввели в ее состав новый класс, java.util.Optional<T>! В этой главе мы покажем, в чем состоит преимущество использования его для моделирования потенциально отсутствующих значений вместо присвоения им пустой ссылки. Мы также поясним, почему такой переход от null к Optional требует от программиста пересмотра идеологии работы с опциональными значениями в модели предметной области. Наконец, мы изучим возможности этого нового класса Optional и приведем несколько практических примеров его эффективного использования. В итоге вы научитесь проектировать улучшенные API, в которых пользователю уже из сигнатуры метода понятно, возможно ли здесь опциональное значение.

11.2. Знакомство с классом Optional

В Java 8 под влиянием языков Haskell и Scala появился новый класс `java.util.Optional<T>` для инкапсуляции опционального значения. Например, если вы знаете, что человек может владеть машиной, а может и не владеть, то не следует объявлять переменную `car` в классе `Person` с типом `Car` и присваивать ей пустую ссылку, если у человека нет машины; вместо этого ее тип должен быть `Optional<Car>`, как показано на рис. 11.1.

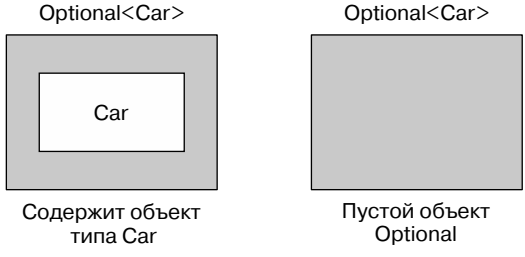


Рис. 11.1. Опционал Car

При наличии значения класс `Optional` служит адаптером для него. И наоборот, отсутствие значения моделируется с помощью пустого опционала, возвращаемого методом `Optional.empty`. Этот статический фабричный метод возвращает специальный экземпляр-одиночку класса `Optional`. Наверное, вам интересно, в чем состоит различие пустой ссылки и `Optional.empty()`. Семантически их можно считать одним и тем же, но на практике между ними существуют колоссальные различия. Попытка разыменования `null` неизбежно приводит к `NullPointerException`, а `Optional.empty()` — корректный, работоспособный объект типа `Optional`, к которому можно удобно обращаться. Скоро вы увидите, как именно.

Важное практическое смысловое отличие в использовании объектов `Optional` вместо `null`: объявление переменной типа `Optional<Car>` ясно говорит, что в этом месте допускается пустое значение. И наоборот, всегда используя тип `Car` и, возможно, иногда присваивая переменным этого типа пустые ссылки, вы подразумеваете, что полагаетесь только на свои знания предметной области, чтобы понять, относится ли `null` к области определения заданной переменной.

С учетом этого можно переделать исходную модель из листинга 11.1. Используем класс `Optional`, как показано в листинге 11.4.

Отметим, что использование класса `Optional` обогащает семантику модели. Поле типа `Optional<Car>` в классе `Person` и поле типа `Optional<Insurance>` в классе `Car` отражают тот факт, что у человека *может быть* машина, а *может и не быть*, равно как машина *может быть* застрахована, а *может и не быть*.

В то же время объявление названия страховой компании как `String`, а не `Optional<String>` явно говорит о том, что у страховой компании должно быть название. Таким образом, вы точно знаете, что получите `NullPointerException` при разыменовании названия страховой компании; добавлять проверку на `null` не нужно, ведь это только скрыло бы проблему. У страховой компании должно быть

название, так что, если вам попадется компания без названия, нужно выяснить, что не так с данными, а не добавлять фрагмент кода для скрытия этого обстоятельства.

Листинг 11.4. Переопределение модели данных Person/Car/Insurance с помощью класса Optional

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}
public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}
public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

У человека может не быть машины, так что мы объявляем это поле как Optional

Машина может оказаться незастрахованной, так что мы объявляем это поле как Optional

У страховой компании должно быть название

Благодаря последовательному использованию опциональных значений создается четкое различие между значением, которое заведомо может отсутствовать, и значением, которое отсутствует из-за ошибки в алгоритме или проблемы в данных. Важно отметить, что класс `Optional` не предназначен для замены всех пустых ссылок до единой. Его задача — помочь спроектировать более понятные API, чтобы по сигнатуре метода можно было понять, может ли там встретиться опциональное значение. Система типов Java заставляет распаковывать опционал для обработки отсутствия значения.

11.3. Паттерны для внедрения в базу кода опционалов

Пока все отлично; мы разобрались, как использовать опционалы в типах для уточнения модели предметной области, и продемонстрировали преимущества этого по сравнению с представлением отсутствующих значений с помощью пустых ссылок. Поговорим теперь об использовании обернутых в опционалы значений.

11.3.1. Создание объектов Optional

Прежде чем начать работать с `Optional`, нужно научиться создавать объекты-опционалы. Это можно сделать несколькими способами.

Пустой опционал

Как упоминалось выше, пустой опционал можно создать с помощью статического фабричного метода `Optional.empty`:

```
Optional<Car> optCar = Optional.empty();
```

Опционал из непустого значения

Можно также создать опционал из непустого значения с применением статического фабричного метода `Optional.of`:

```
Optional<Car> optCar = Optional.of(car);
```

Если объект `car` был равен `null`, то исключение `NullPointerException` будет сгенерировано сразу же (вместо скрытой ошибки, которая проявится потом, когда вы попытаетесь обратиться к свойствам `car`).

Опционал из null

Наконец, с помощью статического фабричного метода `Optional.ofNullable` можно создать объект `Optional`, который может содержать пустое значение:

```
Optional<Car> optCar = Optional.ofNullable(car);
```

Если объект `car` был равен `null`, то полученный в результате объект-опционал `optCar` будет пустым.

Наверное, вы думаете, что теперь мы займемся получением значения опционала. Для этого служит метод `get`, и мы поговорим о нем немного позже. Но метод `get` генерирует исключение, если опционал пуст, так что неупорядоченное его применение фактически приводит к тем же проблемам сопровождения, что и использование `null`. Поэтому мы сначала рассмотрим способы использования опциональных значений без явного получения, в духе аналогичных операций над потоками данных.

11.3.2. Извлечение и преобразование значений опционалов с помощью метода map

Извлечение информации из объекта — распространенный способ. Например, пусть нужно извлечь название страховой компании. Прежде чем извлекать его, необходимо проверить, что `insurance` не равно `null`:

```
String name = null;
if(insurance != null){
    name = insurance.getName();
}
```

Для этого способа класс `Optional` поддерживает метод `map`, работающий следующим образом (начиная с этого места, мы будем использовать модель, приведенную в листинге 11.4):

```
Optional<Insurance> optInsurance = Optional.ofNullable(insurance);
Optional<String> name = optInsurance.map(Insurance::getName);
```

Этот метод, по существу, схож с методом `map` интерфейса `Stream`, который вы видели в главах 4 и 5. Операция `map` применяет указанную функцию к каждому из элементов потока данных. Объект `Optional` можно рассматривать как частный случай коллекции данных, содержащей не более одного элемента. Если объект `Optional` хранит значение, то переданная `map` в качестве аргумента функция выполняет его преобразование. Если же объект `Optional` пуст, то ничего не происходит. Это сходство иллюстрирует рис. 11.2, который показывает, что происходит при передаче функции, преобразующей квадрат в треугольник, методам `map` потока квадратов и опционала для квадрата.

Идея выглядит многообещающей, но как с ее помощью переписать в безопасном виде код из листинга 11.1, который связывает цепочкой несколько вызовов методов?

```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

Ответ: воспользоваться еще одним методом `Optional` — `flatMap`.

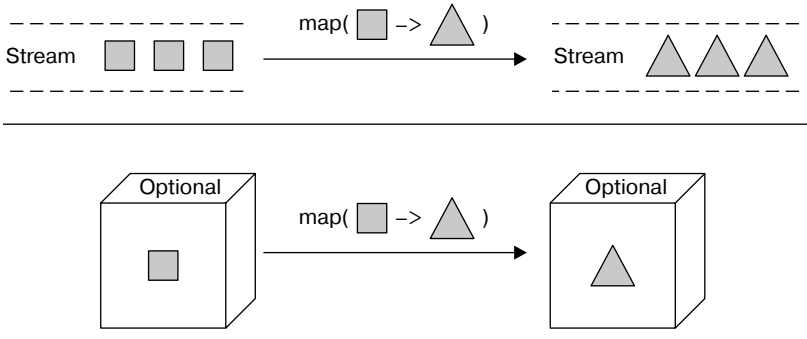


Рис. 11.2. Сравнение методов map интерфейса Stream и класса Optional

11.3.3. Связывание цепочкой объектов Optional с помощью метода flatMap

Раз вы уже знаете, как использовать `map`, то вашей первой мыслью, возможно, будет переписать код с помощью этого метода, вот так:

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name =
    optPerson.map(Person::getCar)
             .map(Car::getInsurance)
             .map(Insurance::getName);
```

К сожалению, этот код не скомпилируется. Почему? Тип переменной `optPerson` — `Optional<Person>`, так что вызывать для нее метод `map` вполне допустимо. Но метод `getCar` возвращает объект типа `Optional<Car>` (как видно из листинга 11.4), то есть результат операции `map` — объект типа `Optional<Optional<Car>>`. В итоге вызов метода `getInsurance` некорректен, поскольку самый внешний опционал содержит в качестве значения другой опционал, а метод `getInsurance`, конечно, такого не поддерживает. Получившаяся вложенная структура опционалов приведена на рис. 11.3.

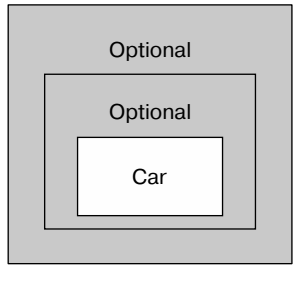


Рис. 11.3. Двухуровневый опционал

Как решить эту проблему? Можно снова обратиться к методу `flatMap`, встречавшемуся нам при работе с потоками данных. В случае потоков данных метод `flatMap` принимает в качестве аргумента функцию и возвращает другой поток данных. В результате применения этой функции к каждому из элементов потока данных получается поток потоков. Но `flatMap` заменяет каждый из сгенерированных потоков его содержимым. Другими словами, все сгенерированные функцией отдельные потоки данных сливаются (схлопываются) в единый поток. В данном случае нам требуется как раз нечто подобное: схлопнуть двухуровневый опционал в один.

Аналогично рис. 11.2 для метода `map` рис. 11.4 демонстрирует схожесть методов `flatMap` интерфейса `Stream` и класса `Optional`.

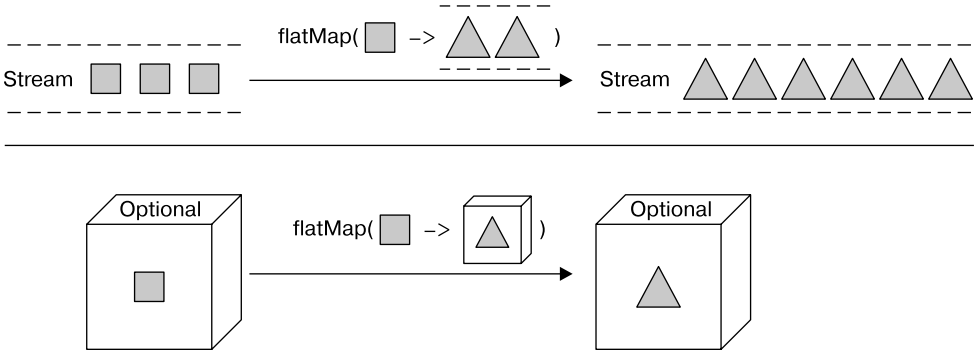


Рис. 11.4. Сравнение методов flatMap интерфейса Stream и класса Optional

На этом рисунке функция, передаваемая методу `flatMap` потока, преобразует каждый из квадратов в другой поток, содержащий два треугольника. Далее результат простого метода `map` оказался бы потоком, содержащим три других потока, по два треугольника каждый, но метод `flatMap` схлопывает этот двухуровневый поток в один поток, включающий шесть треугольников. Аналогично передаваемая методу `flatMap` опционала функция преобразует содержащийся в исходном опционале квадрат в опционал с треугольником. При передаче этой функции методу `map` получился бы опционал, содержащий другой опционал, который, в свою очередь, включает треугольник, метод `flatMap` схлопывает этот двухуровневый опционал в один опционал с треугольником.

Поиск названия компании, застраховавшей машину, с помощью опционалов

Теперь, изучив теоретическую часть методов `map` и `flatMap` класса `Optional`, можно применить знания на практике. Неуклюжие попытки из листингов 11.2 и 11.3 можно переписать с помощью опциональной модели данных из листинга 11.4 следующим образом.

Листинг 11.5. Поиск названия компании-страховщика с помощью опционалов

```
public String getCarInsuranceName(Optional<Person> person) {
    return person.flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

Значение по умолчанию, на случай, если полученный в результате опционал пуст

Сравнение листинга 11.5 с двумя предыдущими примерами демонстрирует преимущества использования опционалов при работе с потенциально отсутствующими значениями. На этот раз мы получаем требуемое с помощью понятного оператора, а не усложняем код условным ветвлением.

В отношении реализации следует сразу отметить изменение сигнатуры метода `getCarInsuranceName` из листингов 11.2 и 11.3. В ней теперь явным образом указано, что возможна передача в этот метод несуществующего человека, например, при извлечении его из базы данных по идентификатору, когда необходимо смоделировать то, что человек с таким идентификатором может отсутствовать в базе. Это дополнительное требование моделируется изменением типа аргумента нашего метода с `Person` на `Optional<Person>`.

Опять же такой подход позволяет явным образом задать с помощью системы типов некие неявные знания предметной области: первейшая задача любого языка, даже языка программирования, состоит в передаче информации. Объявление метода, принимающего опционал в качестве аргумента или возвращающего опционал в качестве результата, свидетельствует для ваших сослуживцев — и всех будущих пользователей метода, — что он может принимать пустое значение или возвращать пустое значение.

Цепочка разыменования Person/Car/Insurance с использованием опционалов

Начиная с этого `Optional<Person>`, разыменование объектов `Car` из `Person`, `Insurance` из `Car` и содержащей название страховой компании `String` выполняется путем сочетания методов `map` и `flatMap`, о котором мы говорили ранее в данной главе. Рисунок 11.5 иллюстрирует этот конвейер операций.

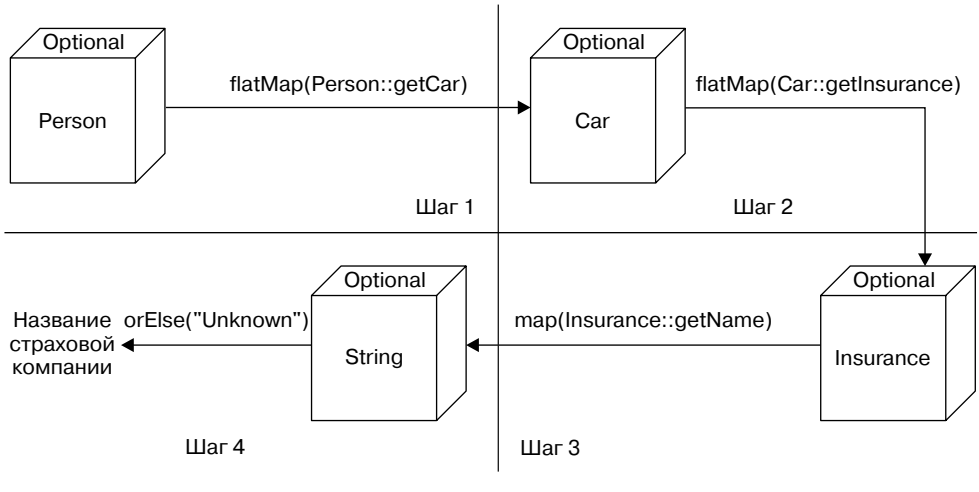


Рис. 11.5. Цепочка разыменования Person/Car/Insurance с использованием опционалов

Здесь мы начинаем с опционала, служащего адаптером для `Person`, вызывая для него `flatMap(Person::getCar)`. Как мы говорили, этот вызов можно рассматривать

как выполняемый за два шага. На первом шаге к содержащемуся внутри опционала объекту `Person` применяется функция преобразования. В данном случае она задается с помощью ссылки на метод и приводит к вызову метода `getCar` объекта `Person`. А поскольку этот метод возвращает `Optional<Car>`, то содержащийся внутри опционала объект `Person` преобразуется в экземпляр данного типа, в результате чего мы получаем двухуровневый опционал, схлопываемый далее в качестве составной части операции `flatMap`. С теоретической точки зрения такую операцию схлопывания можно считать операцией объединения двух вложенных опционалов, возвращающей пустой опционал, если хотя бы один из них пуст. В реальности же при вызове `flatMap` для пустого опционала ничего не меняется и возвращается этот пустой опционал. И наоборот, если опционал обертывает объект `Person`, к этому объекту применяется переданная методу `flatMap` функция. А поскольку получаемое в результате применения этой функции значение является опционалом, метод `flatMap` может вернуть его в неизменном виде.

Второй шаг аналогичен первому — здесь `Optional<Car>` преобразуется в `Optional<Insurance>`. На третьем шаге `Optional<Insurance>` преобразуется в `Optional<String>`, поскольку метод `Insurance.getName()` возвращает объект `String`. В этом случае `flatMap` не нужен.

На данном этапе полученный опционал будет пуст, если хотя бы один из методов в цепочке вызовов вернул пустой опционал, в противном же случае он будет содержать искомое название страховой компании. Как же прочитать это значение? Мы получили в результате объект `Optional<String>`, который может содержать или не содержать название страховой компании. В листинге 11.5 мы воспользовались еще одним методом, `orElse`, позволяющим указать значение по умолчанию на случай, если опционал окажется пуст. Существует множество методов, позволяющих задать действие по умолчанию или распаковать опционал. В следующем разделе мы рассмотрим их во всех подробностях.

Опционалы в модели предметной области и почему они не сериализуемы

В листинге 11.4 мы показали, как использовать опционалы в модели предметной области для того, чтобы отметить специальным типом значения, которые могут отсутствовать или оставаться неопределенными. Создатели класса `Optional`, впрочем, разрабатывали его исходя из других предпосылок и в расчете на другой сценарий применения. В частности, архитектор языка Java Брайан Гётц (Brian Goetz) четко заявил, что цель `Optional` состоит только в поддержке возможности возврата опционального значения.

Поскольку класс `Optional` не предназначался для использования в качестве типа полей классов, то он не реализует интерфейс `Serializable`. Вследствие этого применение класса `Optional` в модели предметной области может привести к проблемам в приложениях, основанных на утилитах или фреймворках, которые требуют сериализуемой модели. Тем не менее нам кажется, что мы достаточно убедительно продемонстрировали, почему имеет смысл использовать опционалы в качестве типов для предметной области, особенно при необходимости обхода графа потенциально

отсутствующих объектов. Если же вам обязательно нужна сериализуемая модель предметной области, рекомендуем вам по крайней мере создать метод, который бы позволял обращаться к любому потенциально отсутствующему значению как опционалу. См. следующий пример:

```
public class Person {
    private Car car;
    public Optional<Car> getCarAsOptional() {
        return Optional.ofNullable(car);
    }
}
```

11.3.4. Операции над потоком опционалов

С помощью метода `stream()` класса `Optional`, появившегося в Java 9, можно преобразовать опционал со значением в поток данных, содержащий только это значение, или пустой опционал в такой же пустой поток. Это может, в частности, пригодиться при распространенном сценарии, когда необходимо преобразовать поток опционалов в другой поток, содержащий лишь значения из непустых опционалов исходного потока. В этом подразделе мы покажем еще на одном реальном примере, для чего может понадобиться поток опционалов и как выполнить эту операцию.

Пример из листинга 11.6 основан на модели предметной области `Person/Car/Insurance`, описанной в листинге 11.4. Пусть вам нужно реализовать метод, в который передается объект `List<Person>`, возвращающий объект `Set<String>`. Этот объект содержит все неповторяющиеся названия страховых компаний, популярных среди владельцев автомобилей из списка.

Листинг 11.6. Поиск неповторяющихся названий страховых компаний, используемых людьми из заданного списка

Преобразуем список людей в поток объектов Optional<Car>, содержащий их машины

С помощью операции FlatMap преобразуем каждый из объектов Optional<Car> в соответствующий Optional<Insurance>

```
public Set<String> getCarInsuranceNames(List<Person> persons) {
    return persons.stream()
                  .map(Person::getCar)
                  .map(optCar -> optCar.flatMap(Car::getInsurance))
                  .map(optIns -> optIns.map(Insurance::getName))
                  .flatMap(Optional::stream)
                  .collect(toSet());
}
```

Отображаем каждый из объектов Optional<Insurance> в содержащий соответствующее название объект Optional<String>

Преобразуем Stream<Optional<String>> в Stream<String>, содержащий только присутствующие в потоке названия

Собираем итоговые строковые значения в множество, чтобы оставить только неповторяющиеся значения

Зачастую в результате манипуляций над элементами потоков получаются длинные цепочки преобразований, фильтров и других операций, но в нашем случае все усложняется еще и тем, что каждый из элементов обернут в объект `Optional`. Напомним, что мы моделируем наличие/отсутствие у человека (`Person`) автомобиля за счет того, что метод `getCar()` класса `Person` возвращает не просто `Car`, а `Optional<Car>`. Так что после первого преобразования с помощью `map` получается `Stream<Optional<Car>>`. Далее с помощью двух последовательных операций `map` мы преобразуем каждый `Optional<Car>` в `Optional<Insurance>`, а затем и в `Optional<String>`, как делали в листинге 11.5 для отдельных элементов (а не потоков).

По окончании этих трех преобразований мы получаем `Stream<Optional<String>>`, в котором часть опционалов пуста, поскольку у некоторых людей нет машины или потому, что их машина не застрахована. С применением опционалов эти операции являются совершенно `null`-безопасными даже в случае пропущенных значений, но возникает проблема: как избавиться от пустых опционалов и распаковать содержащиеся в остальных значения, прежде чем собирать результаты в `Set`. Конечно, можно достичь этого с помощью операции `filter` с последующей `map`:

```
Stream<Optional<String>> stream = ...
Set<String> result = stream.filter(Optional::isPresent)
                           .map(Optional::get)
                           .collect(toSet());
```

Однако, как показано в листинге 11.6, можно добиться того же результата за одну операцию вместо двух, воспользовавшись методом `stream()` класса `Optional`. В самом деле этот метод преобразует опционал в поток, не содержащий элементов или содержащий один элемент, в зависимости от пустоты преобразуемого опционала. Поэтому ссылку на данный метод можно рассматривать как функцию отображения отдельного элемента потока в другой поток с последующей передачей вызванному для исходного потока методу `FlatMap`. Как вы уже знаете, при этом каждый из элементов преобразуется в поток, а затем двухуровневый поток потоков схлопывается в одноуровневый. Благодаря такой уловке можно за один шаг распаковать содержащие значение опционалы и пропустить пустые.

11.3.5. Действия по умолчанию и распаковка опционалов

В подразделе 11.3.3 мы решили прочитать значение опционала с помощью метода `orElse`, позволяющего указать значение по умолчанию, возвращаемое в случае пустого опционала. Класс `Optional` предоставляет несколько методов экземпляра для чтения хранимых в экземплярах `Optional` значений.

- Простейший, но в то же время наименее безопасный из этих методов — `get()`. Он возвращает упакованное значение при его наличии или генерирует исключение `NoSuchElementException`. Так что использовать этот метод не стоит, разве что вы уверены, что опционал не пуст. Кроме того, нельзя сказать, что он чем-то лучше вложенных проверок на `null`.
- Метод `orElse(T other)` применяется в листинге 11.5, и, как мы отмечали, в нем можно указать значение по умолчанию на случай, если опционал не содержит значения.

- Метод `orElseGet(Supplier<? extends T> other)` — аналог метода `orElse(T other)` с отложенным выполнением. Поставщик `other` вызывается, только когда опционал не содержит значения. Этот метод следует применять, если создание значения по умолчанию занимает много времени, чтобы повысить производительность, или когда вызывать поставщик нужно только при пустом опционале (в этом случае `orElseGet` насущно необходим).
- Метод `or(Supplier<? extends Optional<? extends T>> supplier)` напоминает `orElseGet`, но не распаковывает значение, если оно содержится внутри опционала. Фактически этот метод (появившийся в Java 9) ничего не делает и просто возвращает объект `Optional` в неизменном виде, если тот содержит значение, и возвращает отложенным образом другой опционал, если исходный был пуст.
- Метод `orElseThrow(Supplier<? extends X> exceptionSupplier)` напоминает `get` тем, что тоже генерирует исключение, если исходный опционал пуст, но позволяет выбрать тип генерируемого исключения.
- Метод `ifPresent(Consumer<? super T> consumer)` выполняет передаваемое в качестве аргумента действие, если в опционале содержится значение; иначе никаких действий не производится.

В Java 9 появился еще один метод экземпляра: `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`. Он отличается от `ifPresent` дополнительным параметром типа `Runnable`, позволяющим выполнять какое-либо действие, если опционал пуст.

11.3.6. Сочетание двух опционалов

Теперь представьте себе, что у вас есть метод, который принимает на входе `Person` и `Car`, запрашивает какие-либо внешние сервисы и воплощает сложную бизнес-логику поиска страховой компании с наиболее выгодным ценовым предложением для них:

```
public Insurance findCheapestInsurance(Person person, Car car) {
    // Выполняет запросы к предоставляемым различными страховыми компаниями
    // сервисам и сравнивает полученные данные
    return cheapestCompany;
}
```

Допустим также, что вы хотели бы разработать `null`-безопасную версию этого метода, принимающую в качестве аргументов два опционала и возвращающую объект `Optional<Insurance>`, который будет пуст, если пусто хотя бы одно из переданных значений. Класс `Optional` предоставляет метод `isPresent`, возвращающий `true`, если опционал содержит значение, так что наш первый вариант реализации нужного метода будет выглядеть таким образом:

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
                           Optional<Person> person, Optional<Car> car) {
    if (person.isPresent() && car.isPresent()) {
        return Optional.of(findCheapestInsurance(person.get(), car.get()));
    } else {
        return Optional.empty();
    }
}
```

Преимущество этого метода в том, что по его сигнатуре ясно видно: любое из передаваемых в него значений `Person` и `Car` может отсутствовать и поэтому он может не возвращать значения. К сожалению, его реализация слишком сильно напоминает проверки на `null`, которые вы могли бы написать для метода, принимающего аргументы `Person` и `Car`, притом что оба могут быть равны `null`. Существует ли лучший способ реализации этого метода на основе возможностей класса `Optional`? Не пожалейте несколько минут на контрольное задание 11.1 и попробуйте найти красивое решение этой задачи.

Контрольное задание 11.1. Комбинирование двух опционалов без их распаковки

С помощью методов `map` и `flatMap`, о которых мы рассказали в этом разделе, перепишите реализацию вышеприведенного метода `nullSafeFindCheapestInsurance()` в виде одного оператора.

Ответ: реализовать этот метод в виде одного оператора, без каких-либо условных конструкций вроде тернарных операторов можно следующим образом:

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
                            Optional<Person> person, Optional<Car> car) {
    return person.flatMap(p -> car.map(c -> findCheapestInsurance(p, c)));
}
```

Здесь мы вызываем `flatMap` для первого из опционалов, так что если он пуст, то переданное в него лямбда-выражение не будет выполнено и будет возвращен пустой опционал. И наоборот, если `Person` существует, то используется `flatMap` в качестве входного значения для функции, возвращающей необходимый для метода `flatMap` объект `Optional<Insurance>`. В теле этой функции вызывается `map` для второго опционала, так что, если он не содержит никакого `Car`, она вернет пустой опционал, как и весь метод `nullSafeFindCheapestInsurance` в целом. Наконец, при наличии `Person` и `Car` передаваемое аргументом в метод `map` лямбда-выражение может безопасно вызвать исходный метод `findCheapestInsurance` с ними в качестве аргументов.

Сходство между классом `Optional` и интерфейсом `Stream` не ограничивается методами `map` и `flatMap`. Еще один метод, `filter`, ведет себя очень похожим образом в обоих классах, и мы поговорим об этом далее.

11.3.7. Отбрасывание определенных значений с помощью метода filter

Нередко бывает нужно вызвать метод для объекта, чтобы проверить значение какого-либо его свойства. Например, проверить, что название страховой компании равно `CambridgeInsurance`. Чтобы сделать это безопасным образом, необходимо сначала проверить, не равна ли `null` ссылка, указывающая на объект `Insurance`, и лишь затем вызывать метод `getName`, вот так:

```
Insurance insurance = ...;
if(insurance != null && "CambridgeInsurance".equals(insurance.getName())){
```

```
  System.out.println("ok");
}
```

Этот пример можно переписать с помощью метода `filter` объекта `Optional`:

```
Optional<Insurance> optInsurance = ...;
optInsurance.filter(insurance ->
                        "CambridgeInsurance".equals(insurance.getName()))
            .ifPresent(x -> System.out.println("ok"));
```

Метод `filter` принимает в качестве аргумента предикат. Если в объекте `Optional` присутствует значение, которое к тому же соответствует предикату, то метод `filter` возвращает его; в противном случае он возвращает пустой объект `Optional`. Если вспомнить, что опционал можно рассматривать как поток, содержащий не более одного элемента, то поведение метода вполне понятно. Если опционал уже пуст, метод ничего не делает; в противном же случае предикат применяется к хранящемуся в опционале значению. Если в результате возвращается `true`, то опционал возвращается в неизменном виде; в противном случае значение фильтруется, оно отбрасывается и опционал остается пустым. Вы можете проверить, насколько хорошо понимаете принципы работы метода `filter`, выполнив контрольное задание 11.2.

Контрольное задание 11.2. Фильтрация опционалов

Пусть в классе `Person` нашей модели `Person/Car/Insurance` есть еще метод `getAge`, предназначенный для получения возраста человека. Измените метод `getCarInsuranceName` из листинга 11.5 с помощью сигнатуры:

```
public String getCarInsuranceName(Optional<Person> person, int minAge)
```

так, чтобы название страховой компании возвращалось, *только* если возраст человека превышает значение аргумента `minAge`.

Ответ: для фильтрации `Optional<Person>`, чтобы исключить людей, чей возраст меньше аргумента `minAge`, можно закодировать данное условие в передаваемом в метод `filter` предикате следующим образом:

```
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

В следующем разделе мы обсудим оставшиеся функциональные возможности класса `Optional` и приведем дополнительные практические примеры различных способов рефакторинга кода для работы с отсутствующими значениями.

В табл. 11.1 приведен краткий список методов класса `Optional`.

Таблица 11.1. Методы класса Optional

Метод	Описание
empty	Возвращает пустой экземпляр Optional
filter	При наличии значения и соответствии его заданному предикату возвращает текущий опционал, если нет — возвращает пустой
flatMap	При наличии значения возвращает опционал, получающийся в результате применения к нему указанной функции отображения; в противном случае возвращает пустой опционал
get	При наличии возвращает значение, содержащееся в данном опционале; в противном случае генерирует исключение NoSuchElementException
ifPresent	При наличии значения вызывает специальный потребитель, передавая ему это значение; иначе ничего не делает
ifPresentOrElse	При наличии значения производит указанное действие с ним в качестве входных данных; в противном случае выполняет другое действие без входных данных
isPresent	При наличии значения возвращает true; иначе — false
map	При наличии значения применяет к нему заданную функцию отображения
of	Возвращает опционал — адаптер для заданного значения или генерирует NullPointerException, если это значение равно null
ofNullable	Возвращает опционал — адаптер для заданного значения или пустой опционал, если это значение равно null
or	При наличии значения возвращает текущий опционал; в противном случае выдает другой опционал, возвращаемый указанной функцией
orElse	При наличии значения возвращает его; иначе возвращает заданное значение по умолчанию
orElseGet	При наличии значения возвращает его; в противном случае возвращает значение, которое выдает заданный поставщик
orElseThrow	При наличии значения возвращает его; иначе генерирует исключение, создаваемое заданным поставщиком
stream	При наличии значения возвращает включающий только его поток данных; в противном случае возвращает пустой поток данных

11.4. Примеры использования опционалов на практике

Как вы уже знаете, эффективное использование нового класса `Optional` требует полного пересмотра принципов работы с отсутствующими значениями. Этот пересмотр касается не только написания вашего кода, но и (что, вероятно, важнее) взаимодействия с нативными API Java.

На самом деле, вероятно, многие из этих API выглядели бы совсем иначе, если бы на момент их создания существовал класс `Optional`. Из соображений обратной совместимости старые API Java нельзя поменять так, чтобы использовать все возможности опционалов, но не все потеряно. Эту проблему можно решить или по крайней мере обойти, добавив в код небольшие вспомогательные методы для использования возможностей опционалов. Мы покажем на нескольких практических примерах, как это сделать.

11.4.1. Обертывание потенциально пустого значения в опционал

Существующие API Java почти всегда возвращают `null` для указания на отсутствие требуемого значения или невозможность его получения по каким-либо причинам. Метод `get` класса `Map` возвращает `null` в качестве значения, если в данном ассоциативном массиве отсутствует соответствие для указанного ключа. Но по перечисленным выше причинам в большинстве подобных случаев лучше было бы, если бы эти методы возвращали опционал. Поменять сигнатуры методов невозможно, но можно легко обернуть возвращаемое ими значение в опционал. Продолжим рассмотрение примера с `Map`. Обращение к значению с ключом `key` ассоциативного массива `Map<String, Object>` с помощью оператора:

```
Object value = map.get("key");
```

приведет к возврату `null`, если в ассоциативном массиве нет значения, соответствующего строковому `"key"`. Мы в силах усовершенствовать этот код, обернув возвращаемое методом `get` значение в опционал. Так что можно воспользоваться или уродливым условным оператором `if-then-else`, только усложняющим код, или методом `Optional.ofNullable`:

```
Optional<Object> value = Optional.ofNullable(map.get("key"));
```

Его можно использовать всегда, когда требуется безопасно преобразовать в опционал значение, которое может оказаться равным `null`.

11.4.2. Исключения или опционалы?

Вместо того чтобы возвращать `null` при отсутствии значения, в API Java часто прибегают к генерации исключения. Типичный пример — преобразование `String` в `int` с помощью статического метода `Integer.parseInt(String)`. Этот метод генерирует исключение `NumberFormatException`, если объект `String` не содержит корректного целочисленного значения. Опять же получается, что код сообщает пользователю о некорректном аргументе, если объект `String` не является целочисленным значением, с тем единственным отличием, что вместо проверки на `null` условием `if` приходится использовать блок `try/catch`.

Можно также смоделировать некорректное значение, возникшее вследствие не поддающегося преобразованию объекта `String`, с помощью пустого опционала, так что лучше, чтобы `parseInt` возвращал опционал. Изменить существующий метод Java невозможно, но ничто не мешает вам реализовать для него адаптер — крошечный вспомогательный метод, возвращающий требуемый опционал, как показано в листинге 11.7.

Листинг 11.7. Преобразование String в Integer с возвратом опционала

```
public static Optional<Integer> stringToInt(String s) {
    try {
        return Optional.of(Integer.parseInt(s));
    } catch (NumberFormatException e) {
        return Optional.empty();
    }
}
```

Если данную строку можно преобразовать в целое число, возвращаем содержащий ее опционал

В противном случае возвращаем пустой опционал

Предлагаем вам собрать несколько схожих методов во вспомогательный класс и назвать его, скажем, `OptionalUtility`. С этого момента вы всегда сможете преобразовать `String` в `Optional<Integer>` благодаря методу `OptionalUtility.stringToInt`. Можете забыть про инкапсулированную в него неуклюжую логику `try/catch`.

11.4.3. Версии опционалов для простых типов данных и почему их лучше не использовать

Заметим, что, как и у потоков данных, у опционалов есть версии для простых типов данных — `OptionalInt`, `OptionalLong` и `OptionalDouble`, — так что метод из листинга 11.7 мог возвращать `OptionalInt` вместо `Optional<Integer>`. В главе 5 мы рекомендовали вам использовать по возможности версии потоков для простых типов данных (особенно когда они могут состоять из огромного количества элементов) из соображений быстродействия, но такое обоснование неприменимо в данном случае, ведь опционалы содержат не более одного значения.

Мы не рекомендуем вам использовать версии опционалов для простых типов данных, поскольку у них отсутствуют методы `map`, `flatMap` и `filter` — наиболее полезные методы класса `Optional`. Более того, опционалы, как это бывает и с потоками данных, несовместимы со своими аналогами для простых типов данных, так что если бы метод из листинга 11.7 возвращал `OptionalInt`, его нельзя было бы передать в виде ссылки в метод `flatMap` другого опционала.

11.4.4. Собираем все воедино

В этом подразделе мы продемонстрируем, как применить показанные выше методы совместно в интересном сценарии. Пусть мы передаем в программу настройки в виде объектов класса `Properties`. Для этого примера и проверки дальнейшего кода мы создадим несколько `Properties`, вот так:

```
Properties props = new Properties();
props.setProperty("a", "5");
props.setProperty("b", "true");
props.setProperty("c", "-3");
```

Допустим также, что наша программа должна читать значение из этих `Properties`, рассматривая его как длительность в секундах. Поскольку длительность должна быть положительным (> 0) числом, нам нужен метод с сигнатурой следующего вида:

```
public int readDuration(Properties props, String name)
```

чтобы, если значение заданного свойства представляет собой объект `String`, соответствующий положительному числу, метод возвращал данное число, а во всех остальных случаях возвращал `0`. Уточним это требование, формализовав его в нескольких операторах контроля JUnit:

```
assertEquals(5, readDuration(param, "a"));
assertEquals(0, readDuration(param, "b"));
assertEquals(0, readDuration(param, "c"));
assertEquals(0, readDuration(param, "d"));
```

Эти операторы контроля отражают изначальное требование: метод `readDuration` возвращает `5` для свойства `"a"`, поскольку значение этого свойства представляет собой объект `String`, который можно преобразовать в положительное число; `0` для `"b"`, поскольку это не число; `0` для `"c"`, поскольку это число, но отрицательное; `0` для `"d"`, поскольку свойства с таким названием не существует. Попробуем реализовать в императивном стиле метод, который удовлетворял бы этому требованию, как показано в листинге 11.8.

Листинг 11.8. Императивное чтение длительности из свойства

```
public int readDuration(Properties props, String name) {
    String value = props.getProperty(name);
    if (value != null) {
        try {
            int i = Integer.parseInt(value);
            if (i > 0) {
                return i;
            }
        } catch (NumberFormatException nfe) { }
    }
    return 0;
}
```

Проверяем, существует ли свойство с требуемым названием

Пытаемся преобразовать строковое свойство в число

Проверяем, больше ли нуля получившееся число

Возвращаем 0, если хоть одно из условий не соблюдается

Как и следовало ожидать, получившаяся реализация, включающая несколько вложенных условных операторов (как `if`, так и `try/catch`), весьма запутанна и неудобочитаема. Потратьте несколько минут на контрольное задание 11.3 и попытайтесь получить тот же результат, применив знания, усвоенные в этой главе.

Контрольное задание 11.3. Чтение длительности из свойства с помощью опционала

Попытайтесь переписать императивный метод из листинга 11.8 в виде одного текучего оператора, применив возможности класса `Optional` и вспомогательного метода из листинга 11.7.

Ответ: поскольку возвращаемое методом `Properties.getProperty(String)` значение равно `null`, то, если требуемого свойства не существует, было бы удобно преобразовать это значение в опционал с помощью фабричного метода `ofNullable`. Далее можно преобразовать `Optional<String>` в `Optional<Integer>`, передав его методу `flatMap` ссылку на разработанный в листинге 11.7 метод `OptionalUtility.stringToInt`. Наконец, можно легко отбросить отрицательные числа с помощью метода `filter`. Таким образом, если хоть одна из операций возвращает пустой опционал, метод вернет `0`, указанный в качестве значения по умолчанию в методе `orElse`. В противном случае он вернет содержащееся в опционале положительное число. Это реализуется следующим образом:

```
public int readDuration(Properties props, String name) {
    return Optional.ofNullable(props.getProperty(name))
```

```
                    .flatMap(OptionalUtility::stringToInt)
                    .filter(i -> i > 0)
                    .orElse(0);
}
```

Обратите внимание на схожесть стиля при использовании потоков данных и опционалов; и те и другие напоминают запросы базы данных, в которых несколько операций связывается цепочкой.

Резюме

- Исходно пустые ссылки появились в языках программирования как средство уведомления об отсутствии значения.
- Для моделирования наличия или отсутствия значения в Java 8 появился класс `java.util.Optional<T>`.
- Объекты класса `Optional` можно создавать с помощью статических фабричных методов `Optional.empty`, `Optional.of` и `Optional.ofNullable`.
- Класс `Optional` поддерживает множество методов, таких как `map`, `flatMap` и `filter`, которые, по своей сути, похожи на методы потока данных.
- Использование класса `Optional` заставляет разработчика явным образом распаковывать опционал для обработки варианта отсутствия значения; в результате код оказывается защищенным от непредусмотренных `NullPointerException`.
- Класс `Optional` помогает разработчикам проектировать улучшенные API, в которых пользователю понятно из сигнатуры метода, стоит ли ожидать опциональное значение.

12 Новый API для работы с датой и временем

В этой главе

- Зачем нужна новая библиотека для работы с датой/временем, появившаяся в Java 8.
- Представления даты и времени для компьютера и человека.
- Описание отрезка времени.
- Операции с датами, форматирование и синтаксический разбор дат.
- Работа с различными часовыми поясами и системами летоисчисления.

API Java включает множество удобных компонентов для создания сложных приложений. К сожалению, API Java неидеален. Полагаем, большинство опытных Java-разработчиков согласятся, что поддержка операций с датой/временем до Java 8 была далека от идеала. Впрочем, не волнуйтесь, в Java 8 появился совершенно новый API для работы с датой/временем.

Поддержка работы с датой/временем в языке Java 1.0 ограничивалась классом `java.util.Date`. Несмотря на название, класс служил для задания не дат, а моментов времени с точностью до миллисекунд. Но еще больше удобству использования этого класса вредили некоторые странные проектные решения, например выбор точек отсчета: отсчет лет начинался с 1900, а месяцев — с 0. Например, для даты выхода Java 9 — 21 сентября 2017 года — нужно создать следующий экземпляр `Date`:

```
Date date = new Date(117, 8, 21);
```

При выводе этой даты в консоль мы получим:

```
Thu Sep 21 00:00:00 CET 2017
```

Не слишком интуитивно понятный вид, не правда ли? Более того, даже возвращаемое методом `toString` строковое представление класса `Date` может вводить

в заблуждение. Оно также включает время по умолчанию виртуальной машины Java, в нашем случае CET (центральноевропейское время). Фактически класс `Date` просто берет часовой пояс JVM по умолчанию!

Проблемы и ограничения класса `Date` были понятны сразу после выхода Java 1.0, но точно так же было ясно, что их нельзя решить, не нарушив обратную совместимость. Поэтому в Java 1.1 многие методы класса `Date` были объявлены устаревшими, а сам этот класс заменен классом `java.util.Calendar`. К сожалению, у класса `Calendar` были аналогичные проблемы и конструктивные недостатки, в результате чего код оказывался подверженным ошибкам. Месяцы также начинались с индекса 0 (но по крайней мере в `Calendar` избавились от отсчета лет с 1900). Однако наличие *одновременно* классов `Date` и `Calendar` внесло еще большую неразбериху в умы разработчиков, не понимавших, какой использовать. Кроме того, такие функции, как `DateFormat`, предназначенные для форматирования и синтаксического разбора дат/времени не зависящим от языка способом, работали только для класса `Date`.

Свои проблемы привнесла и `DateFormat`. Например, она не типобезопасна, то есть попытка синтаксического разбора даты двумя потоками выполнения одновременно с использованием одного форматера может привести к непредсказуемым результатам.

Наконец, и `Date` и `Calendar` — изменяемые классы. А как можно изменить 21 сентября 2017 года на 25 октября? Подобные проектные решения вполне могут привести к настоящему кошмару при сопровождении программы, как вы узнаете из главы 18, посвященной функциональному программированию.

Все эти изъяны и противоречия стали причиной популярности сторонних библиотек для работы с датой/временем, например Joda-Time. Поэтому компания Oracle решила обеспечить качественную поддержку даты и времени в нативном API Java. В результате в пакет `java.time` Java 8 были включены многие возможности Joda-Time.

В этой главе мы рассмотрим возможности, появившиеся в новом API для работы с датой/временем. Начнем с простейших сценариев, таких как создание значений для дат и времени, подходящих для использования как людьми, так и машинами. Затем мы постепенно перейдем к более продвинутым функциям нового API: рассмотрим различные операции над объектами даты и времени, сделаем их синтаксический разбор и вывод в консоль, а также научимся работать с различными часовыми поясами и календарями.

12.1. Классы LocalDate, LocalTime, LocalDateTime, Instant, Duration и Period

Начнем с создания простых дат и интервалов времени. Пакет `java.time` включает множество новых классов для этой цели: `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Duration` и `Period`.

12.1.1. Работа с классами LocalDate и LocalTime

Вероятно, первым классом, с которым вы столкнетесь при использовании нового API для работы с датой/временем, будет `LocalDate`. Экземпляр этого класса представляет собой неизменяемый объект, соответствующий простой дате без времени. В частности, он не содержит никакой информации о часовом поясе.

Создать экземпляр `LocalDate` можно с помощью одного из статических фабричных методов `of`. У экземпляра `LocalDate` есть множество методов для чтения наиболее используемых его значений (год, месяц, день недели и т. д.), как показано в листинге 12.1.

Листинг 12.1. Создание экземпляра LocalDate и чтение содержащихся в нем значений

```
LocalDate date = LocalDate.of(2017, 9, 21); ◄— 2017-09-21
int year = date.getYear();            ◄— 2017
Month month = date.getMonth();        ◄— SEPTEMBER
int day = date.getDayOfMonth();       ◄— 21
DayOfWeek dow = date.getDayOfWeek();  ◄— THURSDAY
int len = date.lengthOfMonth();       ◄— 30 (дней в сентябре)
boolean leap = date.isLeapYear();     ◄— false (год не високосный)
```

Можно также получить текущую дату от системных часов с помощью фабричного метода `now`:

```
LocalDate today = LocalDate.now();
```

У всех остальных классов даты/времени, которые мы будем обсуждать в оставшейся части главы, есть аналогичные фабричные методы. Можно получить ту же информацию, передав `TemporalField` методу `get`. `TemporalField` представляет собой интерфейс, который описывает способ получения значения конкретного поля временно*го* объекта. Этот интерфейс реализует перечисляемый тип `ChronoField`, элементы которого удобно использовать в методе `get` (листинг 12.2).

Листинг 12.2. Чтение значений LocalDate с помощью TemporalField

```
int year = date.get(ChronoField.YEAR);
int month = date.get(ChronoField.MONTH_OF_YEAR);
int day = date.get(ChronoField.DAY_OF_MONTH);
```

Можно также воспользоваться встроенными методами `getYear()`, `getMonthValue()` и `getDayOfMonth()` для получения этой же информации более удобочитаемым способом:

```
int year = date.getYear();
int month = date.getMonthValue();
int day = date.getDayOfMonth();
```

Аналогично для представления времени дня (например, `13:45:20`) предусмотрен класс `LocalTime`. Для создания экземпляров `LocalTime` служат два перегруженных статических фабричных метода с названием `of`. Первый из них принимает на входе часы и минуты, а второй — еще и секунды. Подобно классу `LocalDate`, у `LocalTime` тоже есть методы-геттеры для доступа к значениям (листинг 12.3).

Листинг 12.3. Создание экземпляра LocalTime и чтение содержащихся в нем значений

```
LocalTime time = LocalTime.of(13, 45, 20); ◄— 13:45:20
int hour = time.getHour();          ◄— 13
int minute = time.getMinute();      ◄— 45
int second = time.getSecond();      ◄— 20
```

Создать как `LocalDate`, так и `LocalTime` можно путем синтаксического разбора соответствующих строковых значений. Для этого используйте их статические методы `parse`:

```
LocalDate date = LocalDate.parse("2017-09-21");
LocalTime time = LocalTime.parse("13:45:20");
```

В метод `parse` можно также передать объект `DateTimeFormatter`. Экземпляр этого класса задает способ форматирования объекта даты и/или времени. Он задуман как замена старого класса `java.util.DateFormat`, упоминавшегося выше. В подразделе 12.2.2 мы покажем подробнее, как использовать класс `DateTimeFormatter`. Отметим также, что оба этих метода `parse` могут генерировать исключение `DateTimeParseException` — расширение `RuntimeException`, — если переданный в качестве аргумента объект `String` не является корректными датой/временем.

12.1.2. Сочетание даты и времени

Составной класс `LocalDateTime` сочетает `LocalDate` и `LocalTime`. Он представляет как дату, так и время без часового пояса, и его экземпляры можно создавать непосредственно или сочетанием даты и времени, как показано в листинге 12.4.

Листинг 12.4. Создание экземпляра LocalDateTime непосредственно или при сочетании даты и времени

```
// 2017-09-21T13:45:20
LocalDateTime dt1 = LocalDateTime.of(2017, Month.SEPTEMBER, 21, 13, 45, 20);
LocalDateTime dt2 = LocalDateTime.of(date, time);
LocalDateTime dt3 = date.atTime(13, 45, 20);
LocalDateTime dt4 = date.atTime(time);
LocalDateTime dt5 = time.atDate(date);
```

Отметим, что можно также создавать объекты `LocalDateTime` путем передачи времени объекту `LocalDate` или даты — объекту `LocalTime`, используя их методы `atTime` и `atDate` соответственно. Можно также извлекать компоненты `LocalDate` или `LocalTime` из объекта `LocalDateTime` с помощью методов `toLocalDate` и `toLocalTime`:

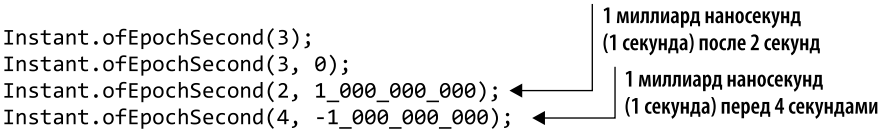
```
LocalDate date1 = dt1.toLocalDate();   ◄—— 2017-09-21
LocalTime time1 = dt1.toLocalTime(); ◄—— 13:45:20
```

12.1.3. Класс Instant: дата и время для компьютеров

Люди привыкли думать о дате и времени в терминах недель, дней, часов и минут. Однако для компьютера такое представление неудобно. С точки зрения машины наиболее естественный формат времени — одно большое число, представляющее собой точку на непрерывной временной шкале. Этот подход воплощен в новом классе `java.time.Instant`, отражающем количество секунд, прошедших с момента начала отсчета времени операционной системы Unix, которое по соглашению равно 1 января 1970 года UTC.

Создать экземпляр этого класса можно, передав значение в секундах его статическому фабричному методу `ofEpochSecond`. Кроме того, класс `Instant` поддерживает время с точностью до наносекунд. Дополнительная перегруженная версия метода

ofEpochSecond принимает второй аргумент — сдвиг в наносекундах по отношению к прошедшему числу секунд из первого аргумента. Эта перегруженная версия при необходимости внесет поправку, чтобы сохраняемое количество наносекунд находилось между 0 и 999 999 999. В результате следующие вызовы фабричного метода ofEpochSecond возвращают один и тот же объект Instant:



Аналогично LocalDate и другим удобочитаемым для человека классам даты и времени класс Instant содержит еще один статический фабричный метод now, который позволяет получить метку даты/времени для текущего момента. Важно еще раз отметить, что класс Instant предназначен исключительно для машинной обработки. Он содержит число секунд и наносекунд и не поддерживает никаких осмысленных для человека единиц времени. Оператор вида:

```
int day = Instant.now().get(ChronoField.DAY_OF_MONTH);
```

приведет к генерации примерно такого исключения:

```
java.time.temporal.UnsupportedTemporalTypeException: Unsupported field:
       DayOfMonth
```

Но для работы с объектами Instant можно использовать классы Duration и Period, которые мы рассмотрим далее.

12.1.4. Описание продолжительности и промежутков времени

Все встречавшиеся нам до сих пор классы реализуют интерфейс Temporal, который описывает чтение и операции над значениями объектов, моделирующих обобщенные моменты времени. Мы показали вам несколько способов создания различных экземпляров Temporal. Следующий логичный шаг — создание объекта для длительности промежутка между двумя временны́ми объектами. Именно для этого и предназначен статический фабричный метод between класса Duration. Задать длительность промежутка времени между двумя объектами LocalTime, двумя LocalDateTime или двумя Instant можно следующим образом:

```
Duration d1 = Duration.between(time1, time2);
Duration d1 = Duration.between(dateTime1, dateTime2);
Duration d2 = Duration.between(instant1, instant2);
```

Поскольку классы LocalDateTime и Instant предназначены для различных целей (один для применения людьми, а второй — машинами), то смешивать их нельзя. Если вы попытаетесь задать длительность промежутка между ними, то просто получите DateTimeException. Более того, поскольку класс Duration отражает время, измеряемое в секундах и наносекундах, то передавать LocalDate в объект between нельзя.

Если нужно смоделировать время, выраженное в годах, месяцах и днях, можно воспользоваться классом `Period`. С помощью фабричного метода `between` этого класса можно вычислить разницу между двумя объектами `LocalDate`:

```
Period tenDays = Period.between(LocalDate.of(2017, 9, 11),
                                LocalDate.of(2017, 9, 21));
```

Наконец, как показано в листинге 12.5, в классах `Duration` и `Period` есть и другие удобные фабричные методы для непосредственного создания их экземпляров, без задания в виде разницы двух временны́х объектов.

Листинг 12.5. Создание объектов Duration и Period

```
Duration threeMinutes = Duration.ofMinutes(3);
Duration threeMinutes = Duration.of(3, ChronoUnit.MINUTES);
Period tenDays = Period.ofDays(10);
Period threeWeeks = Period.ofWeeks(3);
Period twoYearsSixMonthsOneDay = Period.of(2, 6, 1);
```

В классах `Duration` и `Period` есть множество схожих методов (табл. 12.1).

Таблица 12.1. Общие методы классов, служащих для представления промежутков времени

Метод	Статический	Описание
between	Да	Создает интервал между двумя моментами времени
from	Да	Создает интервал времени на основе временного объекта
of	Да	Создает экземпляр интервала времени на основе его составных частей
parse	Да	Создает экземпляр интервала времени на основе объекта String
addTo	Нет	Создает копию данного интервала времени, сложив его с указанным временным объектом
get	Нет	Читает часть значения интервала времени
isNegative	Нет	Проверяет, не отрицателен ли данный интервал, исключая 0
isZero	Нет	Проверяет, не нулевой ли длины данный интервал времени
minus	Нет	Создает копию интервала времени, вычитая из него указанное время
multipliedBy	Нет	Создает копию интервала времени, умножив ее на заданное скалярное значение
negated	Нет	Создает копию интервала времени с инвертированной длительностью
plus	Нет	Создает копию интервала времени, добавив к ней указанное время
subtractFrom	Нет	Вычитает интервал времени из указанного временного объекта

Все обсуждавшиеся до сих пор классы были неизменяемыми — прекрасное проектное решение, которое ведет к более функциональному стилю программирования, обеспечивает потокобезопасность и сохраняет согласованность модели предметной области. Тем не менее новый API для работы с датой и временем предоставляет несколько удобных методов для создания модифицированных версий этих объектов. Например, вам может понадобиться добавить три дня к существующему экземпляру `LocalDate`, и уже в следующем разделе мы обсудим, как это сделать. Мы также разберемся, как создать форматер даты/времени по заданному шаблону, например `dd/MM/yyyy`, или даже программно и как использовать его для синтаксического разбора и вывода в консоль даты.

12.2. Операции над датами, синтаксический разбор и форматирование дат

Самый простой способ создать модифицированную версию существующего объекта `LocalDate` — изменить один из его атрибутов с помощью одного из его методов `withAttribute`. Обратите внимание, что все эти методы возвращают новый объект с измененным атрибутом, как показано в листинге 12.6. Они не меняют исходный объект!

Листинг 12.6. Абсолютное изменение атрибутов объекта LocalDate

```
LocalDate date1 = LocalDate.of(2017, 9, 21); ◄—— 2017-09-21
LocalDate date2 = date1.withYear(2011);      ◄—— 2011-09-21
LocalDate date3 = date2.withDayOfMonth(25);  ◄—— 2011-09-25
LocalDate date4 = date3.with(ChronoField.MONTH_OF_YEAR, 2); ◄—— 2011-02-25
```

Можно сделать то же самое с помощью обобщенного метода `with`, принимающего объект `TemporalField` в качестве первого аргумента, как в последнем операторе листинга 12.6. Этот последний метод `with` дополняет метод `get`, использовавшийся в листинге 12.2. Оба метода объявлены в интерфейсе `Temporal`, который реализован всеми классами API даты и времени, например `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant`. Точнее говоря, методы `get` и `with` позволяют соответственно читать и модифицировать[1] поля объектов `Temporal`. Если требуемое поле не поддерживается данным объектом `Temporal` (например, `ChronoField.MONTH_OF_YEAR` объектом `Instant` или `ChronoField.NANO_OF_SECOND` объектом `LocalDate`), то генерируется исключение `UnsupportedTemporalTypeException`.

Существует даже возможность декларативного манипулирования объектом `LocalDate`. Например, можно прибавлять к нему или вычитать из него заданное время (листинг 12.7).

Листинг 12.7. Относительное изменение атрибутов объекта LocalDate

```
LocalDate date1 = LocalDate.of(2017, 9, 21);   ◄—— 2017-09-21
LocalDate date2 = date1.plusWeeks(1);          ◄—— 2017-09-28
LocalDate date3 = date2.minusYears(6);         ◄—— 2011-09-28
LocalDate date4 = date3.plus(6, ChronoUnit.MONTHS); ◄—— 2012-03-28
```

Подобно тому, что мы рассказывали про методы `with` и `get`, в интерфейсе `Temporal` объявлен метод `plus` (который можно видеть в последнем операторе листинга 12.7) и аналогичный метод `minus`. С их помощью можно сдвигать объект `Temporal` на определенный промежуток времени вперед и назад, определяемый указанным количеством[2] заданных единиц измерения времени[3] (`TemporalUnit`). Перечисляемый тип `ChronoUnit` представляет собой удобную реализацию интерфейса `TemporalUnit`.

---

[1] Помните, что эти методы with не модифицируют существующий объект Temporal, а создают копию, в которой изменено соответствующее поле. Такое процесс называют функциональным обновлением (см. главу 19).

[2] Первый аргумент. — *Примеч. пер.*

[3] Второй аргумент. — *Примеч. пер.*

Как вы могли предположить, у всех классов для работы с датой/временем, отражающих момент времени (таких как `LocalDate`, `LocalTime`, `LocalDateTime` и `Instant`), есть множество общих методов[1]. Их краткая сводка приведена в табл. 12.2.

Таблица 12.2. Часто используемые для работы с датой/временем методы классов, служащих для представления отдельных моментов времени

Метод	Статический	Описание
from	Да	Создает экземпляр данного класса на основе переданного временного объекта
now	Да	Создает временной объект на основе информации системных часов
of	Да	Создает экземпляр временного объекта на основе его составных частей
parse	Да	Создает экземпляр временного объекта на основе объекта String
atOffset	Нет	Дополняет временной объект смещением от времени по Гринвичу, точнее UTC
atZone	Нет	Дополняет временной объект часовым поясом
format	Нет	Преобразует временной объект в String с помощью заданного форматера (в классе Instant этот метод отсутствует)
get	Нет	Читает часть значения временного объекта
minus	Нет	Создает копию временного объекта, из которой вычтено заданное время
plus	Нет	Создает копию временного объекта, к которой прибавлено заданное время
with	Нет	Создает копию временного объекта, часть значения которой изменена

Проверьте полученные знания о работе с датами, выполнив контрольное задание 12.1.

Контрольное задание 12.1. Операции над LocalDate

Каким будет значение указанной переменной для даты после следующих манипуляций?

```
LocalDate date = LocalDate.of(2014, 3, 18);
date = date.with(ChronoField.MONTH_OF_YEAR, 9);
date = date.plusYears(2).minusDays(10);
date.withYear(2011);
```

Ответ:

```
2016-09-08
```

Как вы видели, можно производить как абсолютные, так и относительные операции над датами. Можно также объединить несколько операций в один оператор, поскольку каждое изменение приводит к созданию нового объекта `LocalDate` и каждая последующая операция выполняется над объектом, созданным в результате предыдущей. Наконец, работа последнего оператора из этого фрагмента кода никакого видимого эффекта не дает, поскольку, как обычно, он создает новый экземпляр `LocalDate`, но мы не присваиваем это новое значение какой-либо переменной.

[1] Автор допускает небольшую неточность: по понятным причинам в классе LocalDate нет методов atOffset и atZone, а в классе LocalTime — метода atZone. — *Примеч. пер.*

12.2.1. Работа с классом TemporalAdjusters

Все операции над датами, которые мы видели до сих пор, были относительно просты. Иногда, впрочем, приходится выполнять более сложные операции, например корректировать дату, сдвигая ее на следующее воскресенье, следующий будний день или последний день месяца. В подобных случаях можно передать в перегруженную версию метода `with` объект `TemporalAdjuster`, что обеспечит более тонкий выбор необходимых операций над заданной датой. API даты и времени включает встроенные варианты `TemporalAdjuster` для большинства распространенных сценариев использования. Обратиться к ним можно с помощью статических фабричных методов из класса `TemporalAdjusters` (листинг 12.8).

Листинг 12.8. Использование встроенных объектов TemporalAdjuster

```
import static java.time.temporal.TemporalAdjusters.*;
LocalDate date1 = LocalDate.of(2014, 3, 18);          ← 2014-03-18
LocalDate date2 = date1.with(nextOrSame(DayOfWeek.SUNDAY)); ← 2014-03-23
LocalDate date3 = date2.with(lastDayOfMonth());       ← 2014-03-31
```

В табл. 12.3 приводится список объектов `TemporalAdjuster`, которые можно создать с помощью этих фабричных методов.

Как вы можете видеть, класс `TemporalAdjusters` позволяет выполнять более сложные операции над датами, при этом код по-прежнему ясно отражает решаемую задачу. Более того, если вы не нашли подходящий встроенный `TemporalAdjuster`, то можете относительно легко создать собственную пользовательскую реализацию. Интерфейс `TemporalAdjuster` фактически объявляет только один метод (что делает его функциональным интерфейсом), описанный так, как показано в листинге 12.9.

Таблица 12.3. Фабричные методы класса TemporalAdjusters

Метод	Описание
dayOfWeekInMonth	Создает новую дату в пределах того же месяца с соответствующим днем недели по порядку[1] (отрицательные числа отсчитываются в обратном направлении, с конца месяца)
firstDayOfMonth	Создает новую дату, установленную на первый день текущего месяца
firstDayOfNextMonth	Создает новую дату, установленную на первый день следующего месяца
firstDayOfNextYear	Создает новую дату, установленную на первый день следующего года
firstDayOfYear	Создает новую дату, установленную на первый день текущего года
firstInMonth	Создает новую дату в пределах того же месяца, соответствующую первому из указанных дней недели
lastDayOfMonth	Создает новую дату, установленную на последний день текущего месяца
lastDayOfNextMonth	Создает новую дату, установленную на последний день следующего месяца
lastDayOfNextYear	Создает новую дату, установленную на последний день следующего года
lastDayOfYear	Создает новую дату, установленную на последний день текущего года
lastInMonth	Создает новую дату в пределах того же месяца, соответствующую последнему из указанных дней недели

[1] Например, dayOfWeekInMonth(2, DayOfWeek.TUESDAY) задает второй вторник данного месяца. — *Примеч. пер.*

Метод	Описание
next previous	Создает новую дату, установленную на первое появление заданного дня недели после/до корректируемой даты
nextOrSame previousOrSame	Создает новую дату, установленную на первое появление заданного дня недели после/до корректируемой даты, за исключением совпадения дня недели, в случае чего возвращает тот же объект

Листинг 12.9. Интерфейс TemporalAdjuster

```
@FunctionalInterface
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

Этот пример означает, что реализация интерфейса `TemporalAdjuster` должна определять преобразование одного объекта `Temporal` в другой `Temporal`. `TemporalAdjuster` можно считать аналогом `UnaryOperator<Temporal>`. Потратьте несколько минут, чтобы применить на практике полученные знания и реализовать собственный `TemporalAdjuster` в контрольном задании 12.2.

Контрольное задание 12.2. Реализация пользовательского TemporalAdjuster

Разработайте класс `NextWorkingDay`, реализующий интерфейс `TemporalAdjuster`, который бы сдвигал дату на один день, пропуская субботы и воскресенья. Оператор:

```
date = date.with(new NextWorkingDay());
```

должен сдвигать дату на следующий день, если это день от понедельника до пятницы, или на следующий понедельник, если это суббота или воскресенье.

Ответ: реализовать класс `NextWorkingDay` можно таким образом:

```
public class NextWorkingDay implements TemporalAdjuster {
    @Override
    public Temporal adjustInto(Temporal temporal) {
        DayOfWeek dow =
                DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
        int dayToAdd = 1;
        if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;
        else if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;
        return temporal.plus(dayToAdd, ChronoUnit.DAYS);
    }
}
```

Получаем текущий день недели

По умолчанию добавляем один день

Но если сегодня пятница, то добавляем три дня

А если сегодня суббота — добавляем два дня

Возвращаем измененную дату, к которой добавлено нужное число дней

По умолчанию эта реализация `TemporalAdjuster` сдвигает дату вперед на один день, разве что сегодня пятница или суббота, в случае чего он сдвигает дату на три или два

дня соответственно. Отметим, что, поскольку `TemporalAdjuster` — функциональный интерфейс, поведение этого корректора даты можно передавать лямбда-выражением:

```
date = date.with(temporal -> {
        DayOfWeek dow =
               DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
    int dayToAdd = 1;
    if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;
   else if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;
   return temporal.plus(dayToAdd, ChronoUnit.DAYS);
});
```

Такая операция над датой может пригодиться во многих местах кода, поэтому имеет смысл инкапсулировать ее в виде класса, как мы здесь и сделали. Так следует поступать со всеми часто встречающимися операциями из вашего кода. В итоге у вас соберется маленькая библиотека корректоров, которые вы и ваша команда смогут переиспользовать в базе кода.

Чтобы описать данный `TemporalAdjuster` с помощью лямбда-выражения, лучше воспользоваться статическим фабричным методом `ofDateAdjuster` класса `TemporalAdjusters`, принимающим `UnaryOperator<LocalDate>` в качестве аргумента, вот так:

```
TemporalAdjuster nextWorkingDay = TemporalAdjusters.ofDateAdjuster(
    temporal -> {
       DayOfWeek dow =
                DayOfWeek.of(temporal.get(ChronoField.DAY_OF_WEEK));
        int dayToAdd = 1;
        if (dow == DayOfWeek.FRIDAY) dayToAdd = 3;
        else if (dow == DayOfWeek.SATURDAY) dayToAdd = 2;
        return temporal.plus(dayToAdd, ChronoUnit.DAYS);
    });
date = date.with(nextWorkingDay);
```

Еще одна часто встречающаяся операция над объектами даты и времени — вывод их в консоль в различных форматах в соответствии со спецификой предметной области. И наоборот, часто бывает нужно преобразовать строковые значения в этих форматах в объекты дат. В следующем подразделе мы продемонстрируем механизмы, предлагаемые новым API даты и времени для решения этих задач.

12.2.2. Вывод в консоль и синтаксический разбор объектов даты/времени

Форматирование и синтаксический разбор — важные для работы с датой и временем возможности. Именно для этого и предназначен новый пакет `java.time.format`. Главный класс пакета — `DateTimeFormatter`. Простейший способ создать форматер — воспользоваться его статическим фабричным методом и константами. Константы вида `BASIC_ISO_DATE` и `ISO_LOCAL_DATE` представляют собой заранее определенные экземпляры класса `DateTimeFormatter`. Любой `DateTimeFormatter` можно задействовать для создания объекта `String`, соответствующего заданным дате или времени

в нужном формате. Далее для примера мы создаем строковое значение с помощью двух различных форматеров:

```
LocalDate date = LocalDate.of(2014, 3, 18);
String s1 = date.format(DateTimeFormatter.BASIC_ISO_DATE);  <-- 20140318
String s2 = date.format(DateTimeFormatter.ISO_LOCAL_DATE);  <-- 2014-03-18
```

Для воссоздания объекта даты можно выполнить синтаксический разбор даты/времени в нужном формате. Для этого можно воспользоваться фабричным методом `parse`, который есть во всех классах API даты/времени, служащих для представления момента времени или интервала:

```
LocalDate date1 = LocalDate.parse("20140318",
                                  DateTimeFormatter.BASIC_ISO_DATE);
LocalDate date2 = LocalDate.parse("2014-03-18",
                                  DateTimeFormatter.ISO_LOCAL_DATE);
```

По сравнению со старым классом `java.util.DateFormat` все экземпляры `DateTimeFormatter` потокобезопасны. А значит, можно создавать форматеры-одиночки вроде тех, что описываются константами `DateTimeFormatter`, и одновременно использовать их в нескольких потоках выполнения.

В листинге 12.10 показан пример статического фабричного метода из класса `DateTimeFormatter`, с помощью которого можно создать форматер на основе заданного шаблона.

Листинг 12.10. Создание объекта DateTimeFormatter на основе заданного шаблона

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate date1 = LocalDate.of(2014, 3, 18);
String formattedDate = date1.format(formatter);
LocalDate date2 = LocalDate.parse(formattedDate, formatter);
```

В этом фрагменте кода метод `format` класса `LocalDate` генерирует объект `String`, представляющий дату с заданным шаблоном. Далее статический метод `parse` воссоздает ту же дату путем синтаксического разбора этого объекта `String` с применением того же форматера. У метода `ofPattern` также есть перегруженная версия, с помощью которой можно создать форматер для заданных региональных настроек (определяются классом `Locale`), как показано в листинге 12.11.

Листинг 12.11. Создание локализованного DateTimeFormatter

```
DateTimeFormatter italianFormatter =
              DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.ITALIAN);
LocalDate date1 = LocalDate.of(2014, 3, 18);
String formattedDate = date.format(italianFormatter); // 18. marzo 2014
LocalDate date2 = LocalDate.parse(formattedDate, italianFormatter);
```

Наконец, если вам нужно еще больше возможностей тонкой настройки, благодаря классу `DateTimeFormatterBuilder` можно пошагово описывать сложные форматеры, используя понятные методы. Он также позволяет выполнять синтаксический разбор с учетом регистра, опережающий синтаксический разбор (когда синтаксический анализатор применяет эвристические алгоритмы для истолкования тех входных

данных, которые не соответствуют в точности заданному формату), добавлять пробелы и включать в форматер необязательные разделы. Например, с помощью `DateTimeFormatterBuilder` можно создать программным образом использовавшийся в листинге 12.11 `italianFormatter` (листинг 12.12).

Листинг 12.12. Формирование объекта DateTimeFormatterBuilder

```
DateTimeFormatter italianFormatter = new DateTimeFormatterBuilder()
        .appendText(ChronoField.DAY_OF_MONTH)
        .appendLiteral(". ")
        .appendText(ChronoField.MONTH_OF_YEAR)
        .appendLiteral(" ")
        .appendText(ChronoField.YEAR)
        .parseCaseInsensitive()
        .toFormatter(Locale.ITALIAN);
```

Итак, вы научились создавать, форматировать объекты для отдельных моментов времени, а также временны́х интервалов, выполнять их синтаксический разбор и разные операции над ними. Однако мы еще не сталкивались с такими нюансами обработки дат и времени, как различные часовые пояса или системы летоисчисления. В следующем разделе мы рассмотрим решение этих вопросов с помощью нового API даты/времени.

12.3. Различные часовые пояса и системы летоисчисления

Ни один из вышеупомянутых классов не содержал какой-либо информации о часовых поясах. Работа с часовыми поясами — еще одна важная задача, решение которой сильно упростилось благодаря новому API даты/времени. Старый класс `java.util.TimeZone` был заменен новым `java.time.ZoneId`, намного лучше скрывающим от разработчика сложности, связанные с часовыми поясами, например проблемы перехода на летнее время (Daylight Saving Time, DST). Как и другие классы API даты/времени, он неизменяем.

12.3.1. Использование часовых поясов

Часовой пояс (time zone) — это набор правил, соответствующих региону с одним местным временем. Существуют экземпляры класса `ZoneRules` примерно для 40 часовых поясов. Получить правила для конкретного часового пояса можно, вызвав метод `getRules()` класса `ZoneId`. Конкретный `ZoneId` определяется идентификатором региона, как показано в следующем примере:

```
ZoneId romeZone = ZoneId.of("Europe/Rome");
```

Формат всех идентификаторов регионов имеет вид `"{область}/{город}"`, где используется предоставляемый администрацией адресного пространства Интернета (Internet Assigned Numbers Authority, IANA) список возможных географических местоположений. Можно также преобразовать старый объект `TimeZone` в `ZoneId` с помощью нового метода `toZoneId`:

```
ZoneId zoneId = TimeZone.getDefault().toZoneId();
```

Путем сочетания объекта `ZoneId` с `LocalDate`, `LocalDateTime` или `Instant` можно преобразовать его в объект `ZonedDateTime`, соответствующий моменту времени относительно выбранного часового пояса (листинг 12.13).

Листинг 12.13. Добавление часового пояса к моменту времени

```
LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
ZonedDateTime zdt1 = date.atStartOfDay(romeZone);
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
ZonedDateTime zdt2 = dateTime.atZone(romeZone);
Instant instant = Instant.now();
ZonedDateTime zdt3 = instant.atZone(romeZone);
```

На рис. 12.1 показаны компоненты `ZonedDateTime` для иллюстрации различий между `LocalDate`, `LocalTime`, `LocalDateTime` и `ZoneId`.

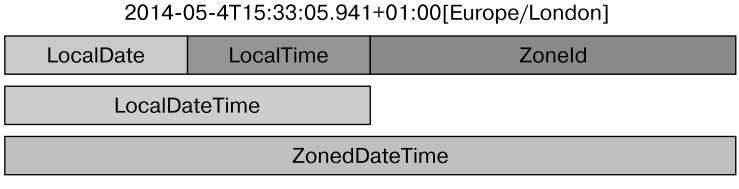


Рис. 12.1. Разбираемся с ZonedDateTime

Можно также преобразовать `LocalDateTime` в `Instant` с помощью экземпляра `ZoneId`:

```
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
Instant instantFromDateTime = dateTime.toInstant(romeZone);
```

Возможно и обратное преобразование:

```
Instant instant = Instant.now();
LocalDateTime timeFromInstant = LocalDateTime.ofInstant(instant, romeZone);
```

Отметим полезность `Instant` для работы с унаследованным кодом, в котором применяется класс `Date`. Для упрощения взаимодействия между устаревшим API и новым API даты/времени в класс `Date` было добавлено два метода: `toInstant()` и статический метод `fromInstant()`[1].

12.3.2. Фиксированное смещение от UTC/времени по Гринвичу

Часовые пояса также часто указывают в виде фиксированного смещения от UTC/времени по Гринвичу. Например, это можно выразить так: «Нью-йоркское время отстает на пять часов от лондонского». В подобных случаях можно воспользоваться классом `ZoneOffset`, производным от `ZoneId`, который служит для выражения разницы между временем данного региона и нулевым (гринвичским) меридианом (Лондон):

```
ZoneOffset newYorkOffset = ZoneOffset.of("-05:00");
```

[1] Скорее всего, авторы ошибаются. Этот метод называется from, его сигнатура имеет следующий вид: public static Date from(Instant instant). — *Примеч. пер.*

Смещение `-05:00` действительно соответствует североамериканскому восточному времени (Eastern Standard Time, EST). Но учтите, однако, что при подобном задании `ZoneOffset` нет возможности управлять переходом на летнее/зимнее время, поэтому в большинстве случаев данный способ не рекомендуется. А поскольку `ZoneOffset` является также `ZoneId`, то его можно использовать так, как показано в листинге 12.13. Можно также создать объект типа `OffsetDateTime`, представляющий дату/время со смещением от UTC/времени по Гринвичу в системе летоисчисления ISO-8601:

```
LocalDateTime dateTime = LocalDateTime.of(2014, Month.MARCH, 18, 13, 45);
OffsetDateTime dateTimeInNewYork = OffsetDateTime.of(date, newYorkOffset);
```

Еще одна продвинутая возможность нового API даты/времени — поддержка систем летоисчисления, отличных от ISO.

12.3.3. Использование альтернативных систем летоисчисления

Система летоисчисления ISO-8601 — де-факто стандартная система летоисчисления, принятая во всем мире. Но в Java 8 появилась поддержка еще четырех систем летоисчисления. Для каждой из них был создан отдельный класс дат: `ThaiBuddhistDate`, `MinguoDate`, `JapaneseDate` и `HijrahDate`. Все они, а также класс `LocalDate`, реализуют интерфейс `ChronoLocalDate`, предназначенный для моделирования дат в произвольном летоисчислении. Экземпляры этих классов можно создавать из объектов `LocalDate`. Вообще говоря, можно создать и любой другой экземпляр `Temporal` с помощью их статических фабричных методов `from`:

```
LocalDate date = LocalDate.of(2014, Month.MARCH, 18);
JapaneseDate japaneseDate = JapaneseDate.from(date);
```

Или можно создать явным образом систему летоисчисления для конкретных региональных настроек и экземпляр даты для этих региональных настроек. В новом API даты/времени систему летоисчисления моделирует интерфейс `Chronology`, экземпляр которого можно получить с помощью его статического фабричного метода `ofLocale`:

```
Chronology japaneseChronology = Chronology.ofLocale(Locale.JAPAN);
ChronoLocalDate now = japaneseChronology.dateNow();
```

Создатели API даты/времени рекомендуют использовать в большинстве случаев `LocalDate` вместо `ChronoLocalDate`, поскольку код может включать допущения, которые, увы, для других систем летоисчисления несправедливы. В числе подобных допущений может быть то, что в месяце не более 31 дня, в году 12 месяцев или даже что число месяцев в году фиксировано. Поэтому мы рекомендуем везде в приложениях использовать `LocalDate`, включая хранение, обработку и истолкование бизнес-правил, а `ChronoLocalDate` применять только при локализации входных/выходных данных программы.

Исламская система летоисчисления. Одна из добавленных в Java 8 новых систем летоисчисления (за нее отвечает класс `HijrahDate`), вероятно, наиболее сложная, поскольку существуют различные ее варианты. Календарь Хиджры основывается на лунных месяцах. Существует множество вариантов того, как определить, начался ли

новый месяц, например, по моменту наблюдения в небе молодой Луны в произвольной точке мира или сначала в Саудовской Аравии. Для выбора желаемого варианта служит метод `withVariant`. В качестве стандартного в классе Java 8 используется саудовский вариант Умм аль-Кура.

Приведенный в следующем коде пример показывает, как вывести начальную и конечную даты месяца Рамадан для текущего года по исламскому календарю в виде дат по ISO:

```
HijrahDate ramadanDate =
    HijrahDate.now().with(ChronoField.DAY_OF_MONTH, 1)
                    .with(ChronoField.MONTH_OF_YEAR, 9);
System.out.println("Ramadan starts on " +
                   IsoChronology.INSTANCE.date(ramadanDate) +
                   " and ends on " +
                   IsoChronology.INSTANCE.date(
                       ramadanDate.with(
                           TemporalAdjusters.lastDayOfMonth())));
```

Получаем текущую дату по календарю Хиджры; затем сдвигаем ее до первого дня месяца Рамадан (девятого по счету)

Рамадан 1438 начался 2017-05-26 и закончился 2017-06-24

IsoChronology.INSTANCE — статический экземпляр класса IsoChronology

Резюме

- Старый класс `java.util.Date` и все остальные классы, использовавшиеся для моделирования дат и времени в версиях Java до 8, отличаются несогласованностью и имеют множество конструктивных недостатков, в числе которых изменяемость и неудачно выбранные смещения, значения по умолчанию и названия.
- Все объекты даты и времени нового API даты/времени — неизменяемые.
- Этот новый API включает два различных представления даты/времени с учетом различных потребностей людей и машин.
- Можно производить как абсолютные, так и относительные операции над объектами даты и времени, причем результаты их всегда представляют собой новый экземпляр, в то время как старый остается неизменным.
- С помощью различных `TemporalAdjuster` можно выполнять и более сложные операции над датами, чем простое изменение их частей, а также описывать и осуществлять пользовательские преобразования дат.
- Для вывода в консоль и синтаксического разбора объектов даты/времени в определенном формате можно описывать форматеры. Их можно создавать на основе шаблона или программно, причем они все типобезопасны.
- Существует возможность представления часовых поясов, относящихся к конкретному региону/местоположению, в виде фиксированного смещения от UTC/времени по Гринвичу и применять их к объектам даты/времени для локализации.
- Можно использовать системы летоисчисления, отличные от стандартной ISO-8601.

13 Методы с реализацией по умолчанию

В этой главе

- Что такое методы с реализацией по умолчанию.
- Сохранение совместимости при эволюции API.
- Примеры использования методов с реализацией по умолчанию.
- Правила разрешения.

Интерфейсы Java традиционно группируют взаимосвязанные методы в контракт. Любой (не абстрактный) класс, реализующий такой интерфейс, *обязан* реализовать все описанные в интерфейсе методы или унаследовать реализацию от суперкласса. Но это требование приводит к проблемам, если создатели библиотеки захотят модифицировать интерфейс, добавив в него новый метод. И действительно, уже существующие конкретные классы (которые, возможно, находятся вне контроля создателей интерфейса) нужно модифицировать, чтобы отразить изменения интерфейса. Особенно эта ситуация осложняется появлением в существующих интерфейсах Java 8 множества новых методов, например метода `sort` интерфейса `List`, который мы использовали в предыдущих главах. Только представьте себе недовольство всех специалистов, занимающихся сопровождением альтернативных фреймворков вроде Guava и Apache Commons, которым нужно будет модифицировать все классы, реализующие интерфейс `List`, и добавить в них реализацию метода `sort`!

Но не волнуйтесь. В Java 8 появился новый механизм для решения указанной проблемы. Вы удивитесь, но, начиная с Java 8, существует два способа объявления методов с реализацией в интерфейсах. Во-первых, Java 8 разрешает объявление

внутри интерфейсов *статических методов*. Во-вторых, в Java 8 появилась новая возможность — *методы с реализацией по умолчанию* (default methods), с помощью которых можно задать реализацию по умолчанию для методов интерфейса. Другими словами, теперь в интерфейсе может быть приведена конкретная реализация метода. В результате, если уже существующие классы, реализующие данный интерфейс, не описывают явным образом реализации метода, то автоматически наследуют реализацию по умолчанию. Мы постоянно использовали в этой книге несколько методов с реализацией по умолчанию. Два примера: метод `sort` интерфейса `List` и метод `stream` интерфейса `Collection`.

Метод `sort` интерфейса `List`, который мы встречали в главе 1, появился только в Java 8, и его объявление выглядит следующим образом:

```
default void sort(Comparator<? super E> c){
    Collections.sort(this, c);
}
```

Обратите внимание на новый модификатор `default` перед типом возвращаемого значения. Именно с его помощью объявляется, что у метода есть реализация по умолчанию. В данном случае метод `sort` для выполнения сортировки вызывает метод `Collections.sort`. Сортировка списка сводится к непосредственному вызову нового метода:

```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 6);
numbers.sort(Comparator.naturalOrder());
```

sort — метод с реализацией по умолчанию в интерфейсе List

В этом коде есть и еще кое-что новое для нас. Обратите внимание на вызов метода `Comparator.naturalOrder()`. Этот новый статический метод интерфейса `Comparator` возвращает компаратор, предназначенный для сортировки элементов в естественном порядке (обычная буквенно-цифровая сортировка). Метод `stream` интерфейса `Collection`, с которым мы встречались в главе 4, выглядит следующим образом:

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

Здесь метод `stream`, регулярно использовавшийся нами в предыдущих главах для обработки коллекций, возвращает поток данных, вызывая метод `StreamSupport.stream`. Обратите внимание на то, как в теле метода `stream` вызывается `spliterator()` — также один из методов с реализацией по умолчанию интерфейса `Collection`.

Ух ты! Так что, интерфейсы теперь напоминают абстрактные классы? И да и нет; между ними есть принципиальные отличия, о которых мы расскажем в этой главе. И главное, почему вас вообще должны волновать методы с реализацией по умолчанию? Основные их пользователи — создатели библиотек. Как мы расскажем далее, методы с реализацией по умолчанию были изначально предназначены для сохранения совместимости при развитии библиотек, таких как API Java (рис. 13.1).

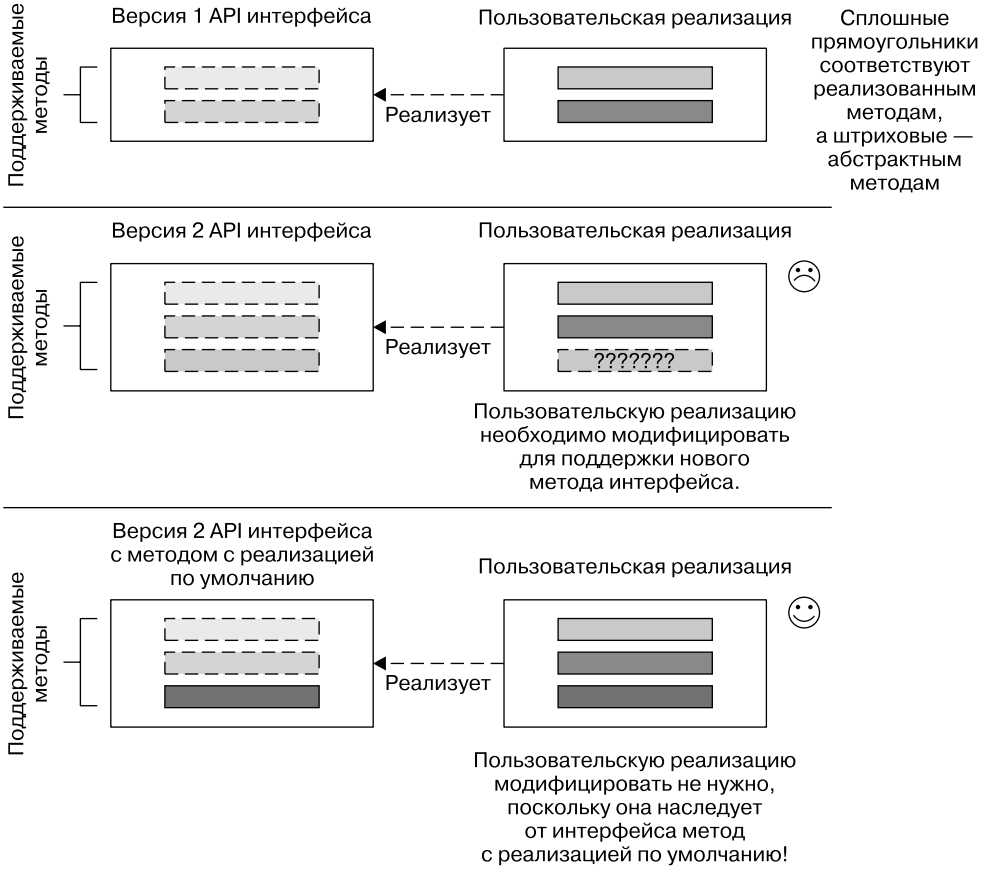


Рис. 13.1. Добавление метода в интерфейс

Если вкратце, добавление метода в интерфейс может стать источником множества проблем; оно требует добавления реализации метода в уже существующие классы, которые реализуют этот интерфейс. Если за интерфейс и все его реализации отвечаете только вы, то ничего страшного. Но зачастую это не так, поэтому и возникла потребность в методах с реализацией по умолчанию, которые бы позволили классам автоматически наследовать реализацию от интерфейса.

Для создателей библиотек эта глава важна тем, что с помощью методов с реализацией по умолчанию можно развивать интерфейсы без модификации существующих реализаций. Кроме того, как мы объясним далее в главе, методы с реализацией по умолчанию помогают структурировать программы, предоставляя гибкий механизм множественного наследования поведения; класс может наследовать методы с реализацией по умолчанию нескольких интерфейсов. Следовательно, такие методы могут быть интересны не только создателям библиотек.

Статические методы и интерфейсы

В Java часто используется следующий паттерн: описание интерфейса вместе со вспомогательным классом-спутником, в котором описывается множество статических методов для работы с экземплярами этого интерфейса. Например, `Collections` — класс-спутник для работы с объектами `Collection`. Теперь, когда статические методы допускается описывать в интерфейсах, подобные вспомогательные классы не нужны и их статические методы можно перенести в интерфейс. Подобные классы-спутники оставлены в API Java лишь для обратной совместимости.

Кратко обрисуем структуру этой главы. Во-первых, мы рассмотрим сценарий развития API и проблемы, которые могут при этом возникнуть. Затем мы расскажем, что такое методы с реализацией по умолчанию, и обсудим, как с их помощью можно решить проблемы, возникающие при этом сценарии. Далее мы покажем вам, как создать собственные методы с реализацией по умолчанию и воплотить в Java своеобразное множественное наследование. В заключение мы приведем немного технической информации о разрешении неоднозначностей компилятором Java при наследовании нескольких методов с реализацией по умолчанию с одинаковой сигнатурой.

13.1. Эволюция API

Чтобы понять, почему так сложно развивать API после их публикации, попробуйте поставить себя на место создателя популярной графической библиотеки для Java. Пусть эта ваша библиотека включает интерфейс `Resizable`, в котором описано множество методов, необходимых для простых фигур с изменяемым размером: `setHeight`, `setWidth`, `getHeight`, `getWidth` и `setAbsoluteSize`. Кроме того, она включает несколько готовых реализаций, например `Square` и `Rectangle`. А поскольку библиотека столь популярна, то некоторые пользователи создали на основе вашего интерфейса `Resizable` свои собственные интересные реализации, например `Ellipse`.

Через несколько месяцев после выпуска API вы понимаете, что в `Resizable` не хватает некоторых возможностей. Например, было бы неплохо добавить в интерфейс метод `setRelativeSize` — он бы принимал в качестве аргумента коэффициент увеличения, на основе которого будет изменяться размер фигуры. Да, вы можете добавить в `Resizable` метод `setRelativeSize` и актуализировать реализации `Square` и `Rectangle`. Но постойте! Как же насчет тех пользователей, которые создали свои реализации интерфейса `Resizable`? К сожалению, к коду их классов, реализующих `Resizable`, у вас доступа нет и поменять их вы не можете. Это та же проблема, с которой сталкиваются создатели библиотеки Java при попытке развития API Java.

В следующем подразделе мы подробно рассмотрим пример, демонстрирующий последствия модификации уже опубликованного интерфейса.

13.1.1. Версия 1 API

Первая версия интерфейса `Resizable` включает следующие методы:

```
public interface Resizable extends Drawable{
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
}
```

Пользовательская реализация

Один из самых преданных ваших пользователей решил создать свою собственную реализацию `Resizable` под названием `Ellipse`:

```
public class Ellipse implements Resizable {
    ...
}
```

Он создал игру, в которой выполняются действия над различными фигурами типа `Resizable` (включая его собственный подтип `Ellipse`):

```
public class Game{
    public static void main(String...args){
        List<Resizable> resizableShapes =
            Arrays.asList(new Square(), new Rectangle(), new Ellipse());
        Utils.paint(resizableShapes);
    }
}
public class Utils{
    public static void paint(List<Resizable> l){
        l.forEach(r -> {
                        r.setAbsoluteSize(42, 42);
                        r.draw();
                      });
    }
}
```

Список фигур с изменяемым размером

Вызов для каждой из фигур метода setAbsoluteSize

13.1.2. Версия 2 API

За несколько месяцев эксплуатации вашей библиотеки вы получили множество просьб изменить реализации интерфейса `Resizable` — `Square` и `Rectangle`, добавив в них метод `setRelativeSize`. Вы публикуете версию 2 вашего API, показанную ниже и проиллюстрированную на рис. 13.2.

```
public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    void setRelativeSize(int wFactor, int hFactor);
}
```

Добавление в версию 2 API нового метода

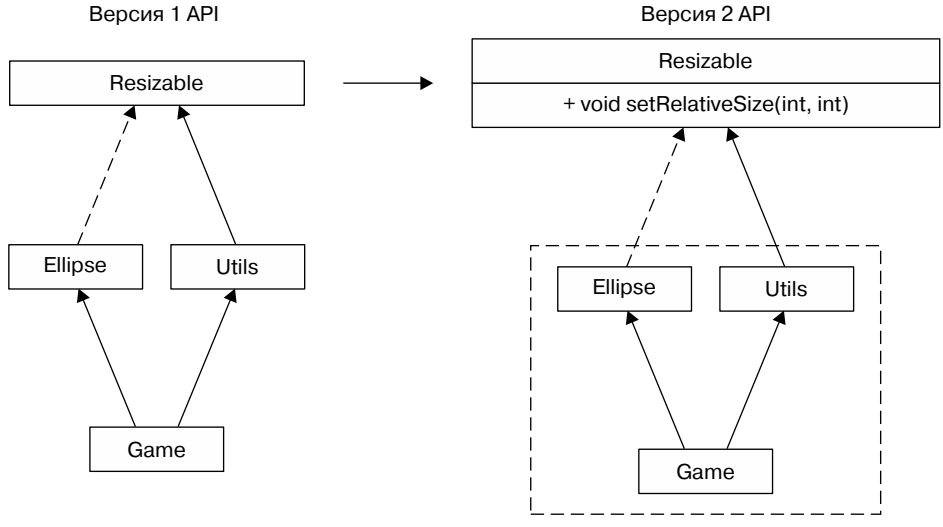


Рис. 13.2. Развитие API путем добавления метода в интерфейс Resizable. Повторная компиляция приложения приводит к ошибкам, поскольку оно зависит от интерфейса Resizable

Возникающие у ваших пользователей проблемы

Эта модификация интерфейса `Resizable` вызывает определенные проблемы. Во-первых, интерфейс теперь требует реализации метода `setRelativeSize`, но в созданной вашим пользователем реализации `Ellipse` он отсутствует. Добавление нового метода в интерфейс сохраняет *совместимость на уровне двоичного кода* (binary compatible), то есть существующие реализации `class`-файлов могут работать и без реализации нового метода, если не пытаться их перекомпилировать. В данном случае игра попрежнему будет работать (если не перекомпилировать ее), несмотря на добавление в интерфейс `Resizable` нового метода `setRelativeSize`. Однако пользователь может модифицировать метод `Utils.paint` игры так, что в нем будет задействован метод `setRelativeSize`, поскольку метод `paint` ожидает в качестве аргумента список объектов `Resizable`. Тогда при передаче объекта `Ellipse` во время выполнения будет генерироваться исключение, поскольку метод `setRelativeSize` не реализован:

```
Exception in thread "main" java.lang.AbstractMethodError:
     lambdasinaction.chap9.Ellipse.setRelativeSize(II)V
```

Во-вторых, если пользователь попытается перекомпилировать полностью все свое приложение (включая `Ellipse`), то получит следующую ошибку компиляции:

```
lambdasinaction/chap9/Ellipse.java:6: error: Ellipse is not abstract and
    does not override abstract method setRelativeSize(int,int) in Resizable
```

Таким образом, обновление уже опубликованного API приводит к обратным несовместимостям. Именно поэтому развитие существующих API, таких как официальный Collection API Java, вызывает проблемы у пользователей этих API. Можно

найти альтернативные развитию API решения, но это не лучший вариант. Например, можно создать отдельную версию API и сопровождать как старую, так и новую версии, но этот вариант неудобен по нескольким причинам. Во-первых, для вас, как создателя библиотеки, увеличивается объем работ. Во-вторых, возможно, пользователям придется задействовать в своей базе кода обе версии вашего API, что отрицательно скажется на объеме используемой памяти и времени загрузки из-за увеличения количества `class`-файлов в их проектах.

В этом случае на помощь спешат методы с реализацией по умолчанию. Благодаря им создатели библиотеки могут развивать свои API, не нарушая работы существующего кода, поскольку классы, реализующие модифицированный интерфейс, автоматически наследуют реализацию по умолчанию.

Различные виды совместимости: на уровне двоичного кода, исходного кода и поведения

При внесении изменений в программы на языке Java возможны три вида совместимости: на уровне двоичного кода, исходного кода и поведения (см. https://blogs.oracle.com/darcy/entry/kinds_of_compatibility). Как вы видели, добавление метода в интерфейс сохраняет совместимость на уровне двоичного кода, но приводит к ошибке компилятора в случае перекомпиляции реализующего интерфейс класса. Не помешает знать об остальных видах совместимости, так что мы подробно рассмотрим их в этой врезке.

Совместимость на уровне двоичного кода (binary compatibility) означает, что работающие без ошибок существующие библиотеки и далее будут подключаться (то есть проходить этапы проверки, подготовки и разрешения) без ошибок и после внесения изменений. Например, совместимость на уровне двоичного кода сохраняется при добавлении метода в интерфейс, поскольку если не вызывать этот метод, то существующие методы могут по-прежнему работать без проблем.

В простейшем варианте *совместимость на уровне исходного кода* (source compatibility) означает, что программа продолжает успешно компилироваться после внесения изменений в связанный с ней код. При добавлении метода в интерфейс совместимость на уровне исходного кода отсутствует; существующие реализации не скомпилируются, поскольку в них должен быть реализован новый метод.

Наконец, *совместимость на уровне поведения* (behavioral compatibility) означает, что программа, запущенная с теми же входными данными, после внесения изменений продемонстрирует то же поведение. Добавление метода в интерфейс совместимо на уровне поведения, поскольку этот метод в программе никогда не вызывается (или переопределяется в реализации).

13.2. Коротко о методах с реализацией по умолчанию

Вы уже видели, что добавление методов в опубликованные API может разрушать существующие реализации. *Методы с реализацией по умолчанию* (default methods) — новая возможность Java 8, предназначенная для сохранения совместимости

при развитии API. Теперь интерфейсы могут включать сигнатуры методов, для которых не обязательна реализация в классе, реализующем данный интерфейс. Кто же их реализует? Недостающие тела этих методов включаются в сам интерфейс (поэтому и говорится о реализациях по умолчанию), а не в реализующий класс.

Как понять, что метод относится к методам с реализацией по умолчанию? Очень просто: его описание начинается с модификатора `default` и содержит тело, как у объявленных в классе методов. Например, для библиотеки работы с коллекциями можно описать интерфейс `Sized` с одним абстрактным методом `size` и методом с реализацией по умолчанию `isEmpty`:

```
public interface Sized {
    int size();
    default boolean isEmpty() {    ◄—— Метод с реализацией по умолчанию
        return size() == 0;
    }
}
```

Теперь все классы, реализующие интерфейс `Sized`, будут автоматически наследовать реализацию метода `isEmpty`. Следовательно, добавление в интерфейс метода с реализацией по умолчанию не приводит к несовместимости на уровне исходного кода.

Вернемся к исходному примеру графической библиотеки Java и вашей игры. Для сохранения совместимости при развитии библиотеки (так, что пользователям не нужно будет модифицировать все реализующие `Resizable` классы) вам нужно воспользоваться методом с реализацией по умолчанию и задать реализацию по умолчанию для метода `setRelativeSize`:

```
default void setRelativeSize(int wFactor, int hFactor){
    setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
}
```

Означает ли наличие методов с реализацией в интерфейсах, что в Java наконец-то появилось множественное наследование? Что будет, если реализующий класс также описывает метод с такой же сигнатурой? Или методы с реализацией по умолчанию можно переопределять? Не задумывайтесь пока об этих проблемах, для их решения существует несколько правил и механизмов, которые мы подробнее обсудим в разделе 13.4.

Наверное, вы догадываетесь, что методы с реализацией по умолчанию широко применяются в API Java 8. В начале этой главы вы видели, что у активно использовавшегося нами в предыдущих главах метода `stream` из интерфейса `Collection` есть реализация по умолчанию. Во многих функциональных интерфейсах, с которыми мы познакомили вас в главе 3, например `Predicate`, `Function` и `Comparator`, появились новые методы с реализацией по умолчанию, такие как `Predicate.and` и `Function.andThen` (напоминаем, что функциональный интерфейс содержит ровно один абстрактный метод; методы с реализацией по умолчанию — не абстрактные).

Абстрактные классы и интерфейсы в Java 8

В чем разница между абстрактным классом и интерфейсом? Ведь и те и другие могут включать абстрактные методы и методы с телом.

Во-первых, класс может расширять только один абстрактный класс, а реализовывать — *несколько* интерфейсов.

Во-вторых, абстрактный класс может хранить в переменных (полях) экземпляра совместно используемое состояние. У интерфейсов не может быть переменных экземпляра.

Проверьте на практике свои знания методов с реализацией по умолчанию и выполните контрольное задание 13.1.

Контрольное задание 13.1. removeIf

Для этого контрольного задания представьте себе, что вы один из создателей языка и API Java. Вы получаете множество просьб создать метод `removeIf` для `ArrayList`, `TreeSet`, `LinkedList` и всех остальных типов коллекций. Метод `removeIf` должен удалять из коллекции все соответствующие определенному предикату элементы. Ваша задача в этом контрольном задании — найти оптимальный способ расширения Collection API новым методом.

Ответ: какой способ расширения будет наиболее разрушительным для Collection API? Можно скопировать и вставить реализацию метода `removeIf` в каждый конкретный класс Collection API, но это был бы плевок в лицо сообществу Java-разработчиков. Что еще можно предпринять? Ну, все классы коллекций в Java реализуют интерфейс `java.util.Collection`. Прекрасно; можем ли мы добавить в него метод? Да. Как вы уже знаете, с помощью методов с реализацией по умолчанию можно добавлять реализации в интерфейсы с сохранением совместимости на уровне исходного кода. При этом все реализующие интерфейс `Collection` классы (в том числе пользовательские, не включенные в Collection API) смогут задействовать эту реализацию `removeIf`. В виде кода это решение для `removeIf` выглядит следующим образом (по существу, практически совпадая с реализацией из официального Collection API Java 8) и представляет собой метод с реализацией по умолчанию в интерфейсе `Collection`:

```
default boolean removeIf(Predicate<? super E> filter) {
    boolean removed = false;
    Iterator<E> each = iterator();
    while(each.hasNext()) {
        if(filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

13.3. Примеры использования методов с реализацией по умолчанию

Мы уже видели, что методы с реализацией по умолчанию могут быть полезны для сохранения совместимости при развитии библиотеки. Где еще они могут пригодиться? Вы можете создавать свои интерфейсы с методами с реализацией по умолчанию. Это может понадобиться в двух сценариях, которые мы обсудим в двух следующих подразделах.

13.3.1. Необязательные методы

Вероятно, вам приходилось сталкиваться с реализующими интерфейс классами, в которых реализации отдельных методов были пустыми. Возьмем, например, интерфейс `Iterator`, в котором описаны методы `hasNext` и `next`, а также метод `remove`. До Java 8 последний часто игнорировали, поскольку пользователям эта возможность была не нужна. В результате во многих классах, реализующих интерфейс `Iterator`, реализация `remove` была пустой, что приводило к ненужному стереотипному коду.

С помощью методов с реализацией по умолчанию можно решить проблему, так что пустая реализация в конкретных классах будет не нужна. В Java 8 реализация по умолчанию для метода `remove` в интерфейсе `Iterator` выглядит следующим образом:

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    default void remove() {
        throw new UnsupportedOperationException();
    }
}
```

В результате сокращается количество стереотипного кода. Реализующим интерфейс `Iterator` классам больше не требуется объявлять пустой метод `remove`, поскольку теперь у него есть реализация по умолчанию.

13.3.2. Множественное наследование поведения

Благодаря методам с реализацией по умолчанию стало возможным нечто ранее недоступное и изящное: *множественное наследование поведения* (multiple inheritance of behavior) — возможность класса переиспользовать код из нескольких источников (рис. 13.3).

Напомним, что класс в языке Java может наследовать только один другой класс, но классы всегда могли реализовывать несколько интерфейсов. В качестве подтверждения приведем описание класса `ArrayList` в API Java:

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable,
               Serializable {
}
```

Наследует один класс

Реализует четыре интерфейса

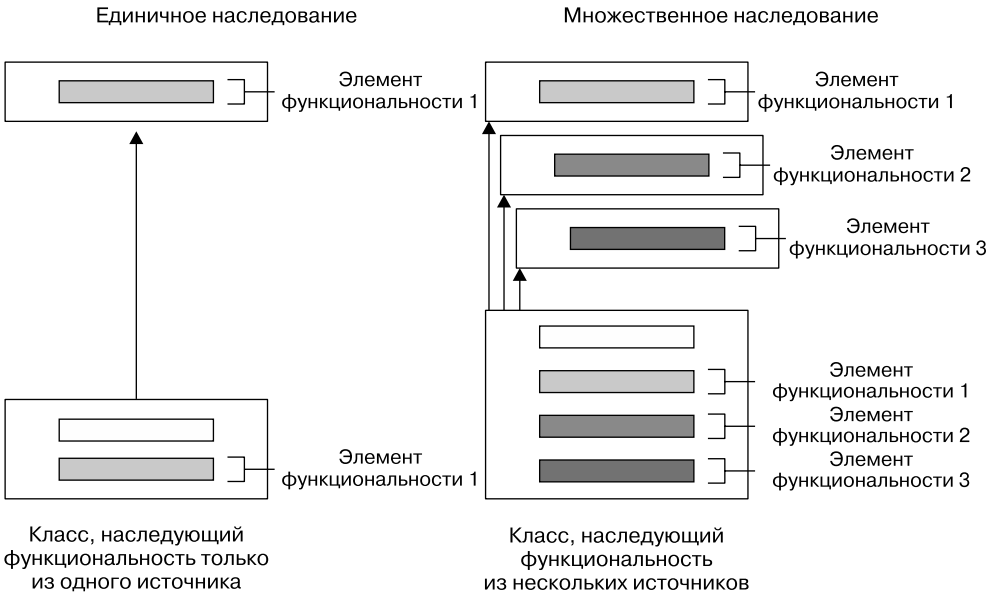


Рис. 13.3. Единичное и множественное наследование

Множественное наследование типов

В коде выше класс `ArrayList` расширяет один класс и непосредственно реализует четыре интерфейса. В результате `ArrayList` представляет собой прямой *подтип* семи типов: `AbstractList`, `List`, `RandomAccess`, `Cloneable`, `Serializable`, `Iterable` и `Collection`. В некотором смысле это уже множественное наследование типов. А поскольку у методов интерфейса в Java 8 теперь могут быть реализации, то класс может наследовать поведение (код реализации) от нескольких интерфейсов. Далее мы рассмотрим пример, который продемонстрирует, как извлечь выгоду из этой возможности. Минимальный размер и взаимная независимость интерфейсов существенно повышают степень переиспользования и композиции поведения внутри базы кода.

Интерфейсы минимального размера с взаимно независимой функциональностью

Пусть для создаваемой игры вам нужно описать несколько фигур с различными характеристиками. У одних из них должна быть возможность изменения размера, но не вращения; а у других — возможность вращения и перемещения, но не изменения размера. Как здесь переиспользовать код по максимуму?

Можно начать с описания отдельного интерфейса `Rotatable`, включающего два абстрактных метода: `setRotationAngle` и `getRotationAngle`. В этом интерфейсе также объявлен метод с реализацией по умолчанию `rotateBy`, реализация которого основана на `setRotationAngle` и `getRotationAngle`:

```
public interface Rotatable {
    void setRotationAngle(int angleInDegrees);
    int getRotationAngle();
    default void rotateBy(int angleInDegrees){
        setRotationAngle((getRotationAngle () + angleInDegrees) % 360);
    }
}
```

Реализация по умолчанию для метода rotateBy

Эта методика в чем-то родственна паттерну проектирования «Шаблонный метод», в котором основной каркас алгоритма выражается в виде других методов, требующих реализации.

Теперь любой класс, реализующий интерфейс `Rotatable`, должен будет задать реализацию методов `setRotationAngle` и `getRotationAngle`, но без каких-либо усилий со стороны разработчика унаследует реализацию по умолчанию метода `rotateBy`.

Аналогичным образом можно описать два вышеупомянутых интерфейса: `Moveable` и `Resizable`, которые содержат реализации по умолчанию. Вот код для интерфейса `Moveable`:

```
public interface Moveable {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
    default void moveHorizontally(int distance){
        setX(getX() + distance);
    }
    default void moveVertically(int distance){
        setY(getY() + distance);
    }
}
```

А вот код для интерфейса `Resizable`:

```
public interface Resizable {
    int getWidth();
    int getHeight();
    void setWidth(int width);
    void setHeight(int height);
    void setAbsoluteSize(int width, int height);
    default void setRelativeSize(int wFactor, int hFactor){
        setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
    }
}
```

Сочетание интерфейсов

Вы можете создавать различные конкретные классы для своей игры, комбинируя вышеупомянутые интерфейсы. Например, монстры могут перемещаться, вращаться и менять размер:

```
public class Monster implements Rotatable, Moveable, Resizable {
...
}
```

Здесь необходимо задать реализации только для всех абстрактных методов, но не для методов с реализацией по умолчанию

Класс `Monster` автоматически наследует методы с реализацией по умолчанию от интерфейсов `Rotatable`, `Moveable` и `Resizable`. В данном случае `Monster` наследует реализации методов `rotateBy`, `moveHorizontally`, `moveVertically` и `setRelativeSize`.

Теперь можно вызывать непосредственно методы:

```
Monster m = new Monster();
m.rotateBy(180);
m.moveVertically(10);
```

Внутри конструктора задаются координаты, высота, ширина и угол по умолчанию

Вызываем метод rotateBy из интерфейса Rotatable

Вызываем метод moveVertically из интерфейса Moveable

Предположим, вам нужно объявить еще один класс объектов, которые могут перемещаться, вращаться, но не менять размер. Копировать и вставлять код не нужно, можно переиспользовать реализации по умолчанию из интерфейсов `Moveable` и `Rotatable`, как показано ниже.

```
public class Sun implements Moveable, Rotatable {
...
}
```

Здесь необходимо задать реализации только для всех абстрактных методов, но не для методов с реализацией по умолчанию

На рис. 13.4 приведена UML-диаграмма этого сценария.

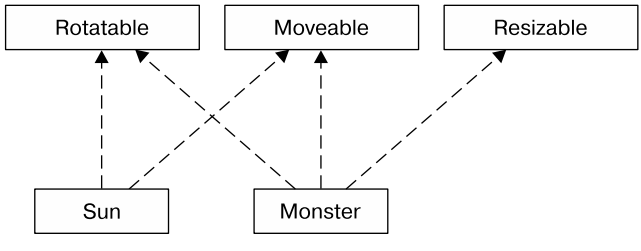


Рис. 13.4. Сочетание нескольких вариантов поведения

Вот еще одно преимущество использования простых интерфейсов с реализациями по умолчанию, подобных вышеприведенным. Пусть требуется модифицировать реализацию метода `moveVertically`, чтобы повысить его быстродействие. Вы можете поменять ее непосредственно в интерфейсе `Moveable`, и все реализующие его классы автоматически унаследуют код (конечно, если они не реализуют этот метод сами)!

Вред наследования

Не стоит использовать наследование как единственный метод переиспользования кода. Например, наследовать класс с сотней методов и полей ради переиспользования одного-единственного метода — плохая идея, это только приводит к ненужному усложнению. Лучше воспользоваться *делегированием*: создать метод, который будет напрямую

вызывать нужный метод класса через переменную — член класса. Поэтому иногда можно встретить классы, намеренно объявленные с модификатором `final`: их нельзя наследовать во избежание реализации упомянутого антипаттерна или вмешательства в их базовое поведение. Например, как `final` объявлен класс `String`, поскольку вмешательство в подобную фундаментальную функциональность явно нежелательно.

То же самое относится к интерфейсам, включающим методы с реализацией по умолчанию. Путем минимализации интерфейсов повышается уровень композиции поведения, ведь появляется возможность выбора только нужных реализаций.

Как видите, методы с реализацией по умолчанию полезны во многих сценариях. Но вот вам немного пищи для размышлений. Что, если класс реализует два интерфейса с одинаковой сигнатурой метода с реализацией по умолчанию? Какой метод класс может использовать? Мы обсудим этот вопрос в следующем разделе.

13.4. Правила разрешения конфликтов

Как вы знаете, в языке Java класс может расширять только один родительский класс, но реализовывать несколько интерфейсов. С появлением методов с реализацией по умолчанию в Java 8 стало реальным наследование классом нескольких методов с одинаковой сигнатурой. Какую из версий метода при этом следует использовать? Подобные конфликты на практике встречаются редко, но на случай их возникновения все равно необходимы правила разрешения. В этом разделе мы расскажем, как компилятор Java разрешает подобные потенциальные конфликты. Наша задача — ответить на вопрос вида: «Какой из методов `hello` вызывает класс `C` в нижеприведенном коде?» Отметим, что примеры предназначены для обсуждения гипотетических сценариев, вовсе не обязательно, что такие ситуации будут часто встречаться на практике:

```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}
public class C implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

Что будет выведено в консоль?

Кроме того, наверное, вы слышали о проблеме ромбовидного наследования в языке C++, когда класс может наследовать два метода с одной сигнатурой. Какой же будет выбран? Java 8 включает правила разрешения и для этой проблемы. Читайте дальше!

13.4.1. Три важных правила разрешения неоднозначностей

При наследовании классом метода с одной сигнатурой из нескольких мест (других классов или интерфейсов) работают три правила.

1. У классов всегда преимущество. Объявление метода в классе или суперклассе имеет приоритет перед объявлением любого метода с реализацией по умолчанию.
2. Если предыдущее правило не позволяет разрешить неоднозначность, то преимущество — у производных интерфейсов: используется метод с той же сигнатурой из наиболее конкретного интерфейса из числа содержащих реализацию по умолчанию (если интерфейс `B` расширяет интерфейс `A`, то `B` — более конкретный, чем `A`).
3. Наконец, если неоднозначность сохраняется, то класс, наследующий метод из нескольких интерфейсов, должен явным образом выбрать используемую реализацию метода по умолчанию, переопределив его и явно вызвав нужный метод.

Мы обещаем, что вам нужно знать только эти правила, больше ничего! В следующем подразделе рассмотрим несколько примеров.

13.4.2. Преимущество — у самого конкретного интерфейса из числа содержащих реализацию по умолчанию

Здесь мы вернемся к примеру из начала этого раздела, где класс `C` реализует как интерфейс `B`, так и интерфейс `A`, в каждом из которых описан метод с реализацией по умолчанию `hello`. Помимо этого, интерфейс `B` — производный от интерфейса `A`. UML-диаграмма для такого сценария приведена на рис. 13.5.

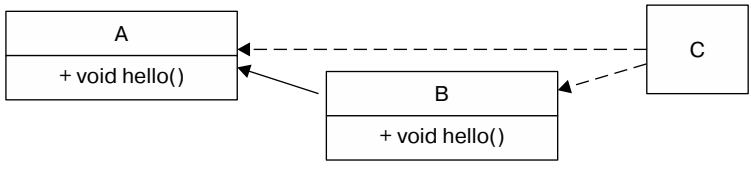


Рис. 13.5. Преимущество — у самого конкретного из содержащих реализацию по умолчанию интерфейсов

Какое из объявлений метода `hello` выберет компилятор? Правило 2 гласит, что выбирается метод из самого конкретного из содержащих реализацию по умолчанию интерфейсов. А поскольку интерфейс `B` конкретнее интерфейса `A`, то будет выбран метод `hello` из интерфейса `B`. Следовательно, программа выведет в консоль `"Hello from B"`.

Рассмотрим теперь случай, когда C также наследует класс D (рис. 13.6):

```
public class D implements A{ }
public class C extends D implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

Что будет выведено в консоль?

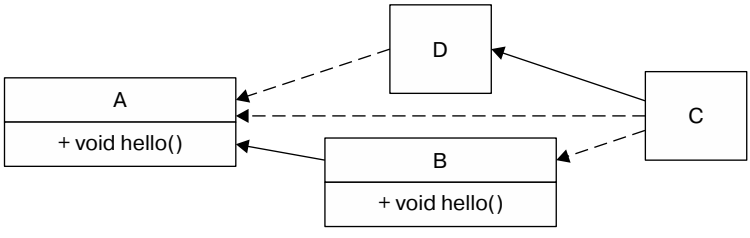


Рис. 13.6. Наследование класса с одновременной реализацией двух интерфейсов

Правило 1 гласит, что приоритет — у объявления метода в классе. Но класс D не переопределяет метод hello, он реализует интерфейс A. Следовательно, его метод с реализацией по умолчанию получен от интерфейса A. Правило 2 гласит, что если нет метода из класса или суперкласса, то выбирается метод самого конкретного из содержащих реализацию по умолчанию интерфейсов. Компилятор, следовательно, выбирает между методом hello из интерфейса A и методом hello из интерфейса B. А поскольку интерфейс B конкретнее интерфейса A, то программа опять выведет в консоль "Hello from B".

Чтобы проверить себя на понимание правил разрешения неоднозначностей, попробуйте выполнить контрольное задание 13.2.

Контрольное задание 13.2. Не забывайте о правилах разрешения неоднозначностей

Для этого контрольного задания воспользуемся предыдущим примером, за исключением того, что теперь класс D переопределяет метод hello из интерфейса A. Как вы думаете, что будет выведено в консоль?

```
public class D implements A{
    void hello(){
        System.out.println("Hello from D");
    }
}
public class C extends D implements B, A {
    public static void main(String... args) {
        new C().hello();
    }
}
```

Ответ: программа выведет в консоль `"Hello from D"`, поскольку приоритет в данном случае у объявления метода из суперкласса, как гласит правило 1.

Заметим также, что если бы класс `D` был объявлен следующим образом:

```
public abstract class D implements A {
    public abstract void hello();
}
```

то в классе `C` пришлось бы реализовать метод `hello`, хотя в иерархии классов и присутствует реализация по умолчанию.

13.4.3. Конфликты и разрешение неоднозначностей явным образом

В примерах, которые мы рассмотрели до сих пор, можно было разобраться с помощью первых двух правил разрешения неоднозначностей. Теперь предположим, что интерфейс `B` более не расширяет `A` (рис. 13.7).

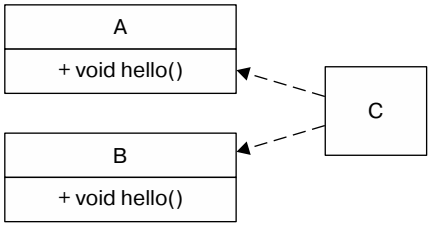


Рис. 13.7. Реализует два интерфейса

```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B {
    default void hello() {
        System.out.println("Hello from B");
    }
}
public class C implements B, A { }
```

Правило 2 здесь нам не поможет, поскольку ни один из интерфейсов не является более конкретным. Оба метода из `A` и `B` подходят. Соответственно, компилятор Java выдает ошибку компиляции, так как не знает, какой из них подходит лучше: `"Error: class C inherits unrelated defaults for hello() from types B and A."`.

Разрешение конфликта

Вариантов, как разрешить конфликт между двумя возможными подходящими методами, немного: вам придется явным образом указать, какое из объявлений метода вы

хотели бы использовать в C. Для этого можно переопределить метод hello в классе C, а затем в его теле явным образом вызвать нужный метод. В Java 8 появился новый синтаксис X.super.m(...), где X — суперинтерфейс, метод которого m нужно вызвать. Например, код, при котором в классе C будет использоваться метод с реализацией по умолчанию из B, выглядит вот так:

```
public class C implements B, A {
    void hello(){
        B.super.hello();
    }
}
```

Явным образом указываем, что нужно вызвать метод из интерфейса B

А теперь попробуйте разобраться в хитром сценарии из контрольного задания 13.3.

Контрольное задание 13.3. Почти такая же сигнатура

Для целей этого контрольного задания предположим, что интерфейсы A и B объявлены следующим образом:

```
public interface A{
    default Number getNumber(){
        return 10;
    }
}
public interface B{
    default Integer getNumber(){
        return 42;
    }
}
```

А класс C объявлен вот так:

```
public class C implements B, A {
    public static void main(String... args) {
        System.out.println(new C().getNumber());
    }
}
```

Что выведет в консоль эта программа?

Ответ: класс C не может определить, какой из методов — из интерфейса A или B — более конкретен. Поэтому класс C не скомпилируется.

13.4.4. Проблема ромбовидного наследования

Наконец, рассмотрим сценарий, от которого бросает в дрожь разработчиков на C++:

```
public interface A{
    default void hello(){
        System.out.println("Hello from A");
    }
```

```
}
public interface B extends A { }
public interface C extends A { }
public class D implements B, C {
    public static void main(String... args) {
        new D().hello();
    }
}
```

Что будет выведено в консоль?

Рисунок 13.8 демонстрирует UML-диаграмму для этого сценария. Проблема называется *проблемой ромбовидного наследования* (diamond problem), поскольку диаграмма напоминает ромб. Какое из объявлений метода с реализацией по умолчанию унаследует `D`: из интерфейса `B` или интерфейса `C`? Выбирать здесь особо не из чего, поскольку метод с реализацией по умолчанию объявлен только в `A`. А поскольку этот интерфейс представляет собой суперинтерфейс `D`, то в консоль будет выведено `"Hello from A"`.

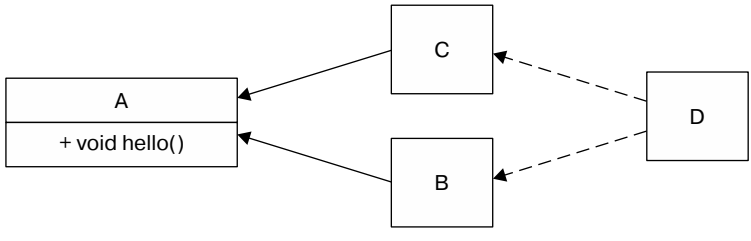


Рис. 13.8. Проблема ромбовидного наследования

Что же произойдет, если в интерфейсе `B` также будет объявлен метод с реализацией по умолчанию `hello` с такой же сигнатурой? Правило 2 гласит, что выбирается метод из самого конкретного из содержащих реализацию по умолчанию интерфейсов. А поскольку интерфейс `B` более конкретен, чем `A`, то выбирается объявление метода с реализацией по умолчанию из `B`. Если же метод с реализацией по умолчанию `hello` с одинаковой сигнатурой объявлен и в `B`, и в `C`, возникнет конфликт, который придется разрешить явным образом, как мы показывали выше.

Кстати, вам, наверное, интересно, что будет, если добавить абстрактный (без реализации по умолчанию) метод `hello` в интерфейс `C` (в интерфейсах `A` и `B` никаких методов по-прежнему нет):

```
public interface C extends A {
    void hello();
}
```

У нового абстрактного метода `hello` из интерфейса `C` есть приоритет над методом `hello` с реализацией по умолчанию из интерфейса `A` в силу большей конкретности интерфейса `C`. Следовательно, в классе `D` должна быть явная реализация метода `hello`, иначе программа не скомпилируется.

Проблема ромбовидного наследования в языке C++

Проблема ромбовидного наследования в языке C++ еще более запутанна. Во-первых, в C++ разрешено множественное наследование классов. По умолчанию, если класс `D` наследует классы `B` и `C`, а оба класса `B` и `C` наследуют класс `A`, то у класса `D` есть доступ к копии объекта `B` и копии объекта `C`. В результате приходится явным образом уточнять, откуда брать методы интерфейса `A`: из интерфейса `B` или `C`? Кроме того, у классов есть состояние, так что изменение переменных — членов класса из `B` не отражается на копии объекта `C`.

Как видите, механизм разрешения неоднозначностей для методов с реализацией по умолчанию при наследовании классом нескольких методов с одной сигнатурой прост. Для разрешения всех конфликтов достаточно методично следовать трем правилам.

1. У явного объявления метода в классе или суперклассе есть приоритет над любым объявлением метода с реализацией по умолчанию.
2. Иначе из числа методов с одинаковой сигнатурой выбирается метод из наиболее конкретного содержащего реализацию по умолчанию интерфейса.
3. Наконец, если конфликт сохраняется, необходимо явным образом переопределить методы с реализацией по умолчанию и выбрать, какой из них должен использовать ваш класс.

Резюме

- Интерфейсы в Java 8 могут содержать код реализации в методах с реализацией по умолчанию и статических методах.
- Методы с реализацией по умолчанию начинаются с ключевого слова `default` и содержат тело, как обычные методы класса.
- Добавление абстрактного метода в опубликованный интерфейс может привести к несовместимости на уровне исходного кода.
- Методы с реализацией по умолчанию полезны для создателей библиотек тем, что позволяют сохранять обратную совместимость при развитии API.
- Методы с реализацией по умолчанию можно применять для создания необязательных методов и множественного наследования поведения.
- Правила разрешения неоднозначностей предназначены для разрешения конфликтов при наследовании классом нескольких методов с реализацией по умолчанию с одинаковой сигнатурой.
- У объявления метода в классе или суперклассе есть приоритет над любым объявлением метода с реализацией по умолчанию. Иначе из числа методов с одинаковой сигнатурой выбирается метод из наиболее конкретного содержащего реализацию по умолчанию интерфейса.
- Если два метода одинаково конкретны, класс должен явным образом переопределить метод, чтобы указать, какой из них вызывать.

14 Система модулей Java

В этой главе

- Причины, приведшие к появлению в Java системы модулей.
- Общая структура: объявления модулей, директивы `requires` и `exports`.
- Автоматические модули для унаследованных Java-архивов (JAR).
- Модуляризация и библиотека JDK.
- Модули и сборки Maven.
- Краткая сводка директив модулей, помимо простейших `requires` и `exports`.

Основная и наиболее обсуждаемая из появившихся в Java 9 новых возможностей — система модулей, выросшая из проекта Jigsaw. Ее разработка заняла почти десятилетие, что хорошо демонстрирует как важность такого дополнения, так и сложности, с которыми столкнулась команда разработчиков языка Java при его реализации. В этой главе мы расскажем вам, почему разработчикам важно знать, что такое система модулей, а также опишем, для чего предназначена новая система модулей Java и как извлечь из нее пользу.

Отметим, что система модулей Java — непростая тема, заслуживающая отдельной книги. Мы рекомендуем книгу *The Java Module System* Николаи Парлога (Nicolai Parlog), выпущенную издательством Manning Publications (https://www.manning.com/books/the-java-module-system).

В этой главе мы умышленно ограничимся общей картиной, чтобы вам стала понятна основная мотивация появления системы модулей, и приведем краткий обзор работы с модулями Java.

14.1. Движущая сила появления системы модулей Java: размышления о ПО

Прежде чем углубиться в систему модулей Java, не помешает разобраться в предпосылках и мотивации ее появления, чтобы понять, какие цели ставили перед собой создатели языка Java. Что такое модульность? Какие задачи должна решать система модулей? В этой книге мы потратили немало времени на обсуждение новых возможностей языка Java, чтобы научиться писать код, который бы яснее отражал постановку задачи, а значит, был понятнее и проще в сопровождении. Впрочем, это низкоуровневая формулировка задачи. В конечном счете на высоком уровне (уровне архитектуры приложения) задача состоит в создании удобного для анализа программного проекта, что повышает производительность разработчика при внесении изменений в базу кода. В следующих подразделах мы уделим особое внимание двум принципам проектирования, с помощью которых можно создавать более удобное для анализа программное обеспечение: *разделение ответственности* и *скрытие информации*.

14.1.1. Разделение ответственности

Разделение ответственности (separation of concerns, SoC) — принцип, поощряющий разбиение компьютерной программы на отдельные функциональные возможности. Допустим, вам нужно разработать бухгалтерское приложение для синтаксического разбора информации о расходах в различных форматах, их анализа и выдачи пользователю сводных отчетов. При разделении ответственности синтаксический разбор, анализ и формирование отчетов разбиваются на отдельные части, называемые модулями, — взаимосвязанные, но практически не перекрывающиеся части кода. Другими словами, модули группируют классы, позволяя выражать отношения видимости между классами приложения.

Наверное, вы скажете: «Но ведь за группировку классов уже отвечают пакеты Java». Вы правы, но модули Java 9 позволяют более точно управлять видимостью одних классов другими и дают возможность проверки этого во время выполнения. По сути, пакеты Java не обеспечивают модульной организации кода.

Принцип SoC полезен с архитектурной точки зрения (например, при реализации паттерна «Модель — представление — контроллер») и при низкоуровневом подходе (например, отделении бизнес-логики от механизмов восстановления). У него есть следующие преимущества:

- позволяет работать над отдельными частями программы независимо, таким образом способствуя работе в команде;
- облегчает переиспользование отдельных частей;
- упрощает сопровождение системы в целом.

14.1.2. Скрытие информации

Скрытие информации (information hiding) — принцип, поощряющий скрытие нюансов реализации. Чем этот принцип важен? Требования к создаваемому про-

граммному обеспечению часто меняются. Благодаря скрытию нюансов реализации можно снизить вероятность того, что небольшое локальное изменение приведет к необходимости лавинообразных изменений в других частях программы. Другими словами, этот принцип очень полезен для управления кодом и его защиты. Наверняка вы часто слышали слово *«инкапсуляция»*, означающее, что определенная часть кода настолько хорошо изолирована от остальных частей приложения, что изменение ее внутренней реализации на них не повлияет. В Java можно с помощью ключевого слова `private` заставить компилятор проверять правильность инкапсуляции *компонентов внутри класса*. Но до появления Java 9 не существовало языковой конструкции, чтобы компилятор мог проверить, что классы и пакеты можно использовать только для тех целей, которые предполагались.

14.1.3. Программное обеспечение на языке Java

Эти два принципа — основа любого хорошо спроектированного программного обеспечения. Как они соответствуют возможностям языка Java? Java — объектно-ориентированный язык программирования, так что мы работаем с классами и интерфейсами. *Модульная организация* кода достигается за счет группировки вместе пакетов, классов и интерфейсов, отвечающих за решение одной задачи. На практике анализировать сам исходный код не очень удобно. Поэтому для анализа программного обеспечения подойдут такие инструменты, как UML-диаграммы (проще говоря, прямоугольники и стрелки), позволяющие наглядно представить зависимости между частями кода. На рис. 14.1 показана UML-диаграмма приложения, предназначенного для управления профилем пользователя, которое было разбито на три конкретные зоны ответственности.

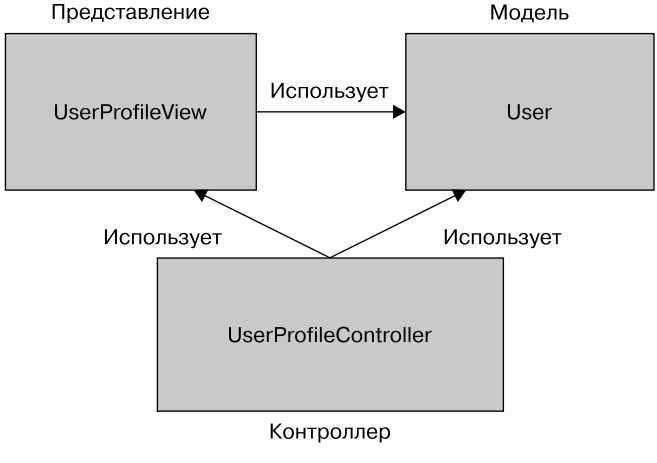


Рис. 14.1. Три отдельные зоны ответственности и зависимости между ними

А как насчет скрытия информации? Вы уже знакомы с модификаторами видимости в Java, служащими для контроля доступа к методам, полям и классам: `public`,

`private`, `protected`, или с доступом в пределах пакета. Впрочем, как мы объясним в следующем разделе, подобного уровня модульности зачастую недостаточно, так что может понадобиться объявить метод `public`, даже если он не должен быть доступным для конечных пользователей. В начале существования Java, когда приложения и цепочки зависимостей были относительно малы, это не было большой проблемой. Но сейчас, когда размеры многих приложений Java очень велики, эта проблема стала намного актуальнее. И действительно, если вы видите в классе поле или метод, помеченный как `public`, то, наверное, считаете себя вправе им воспользоваться (не правда ли?), хотя создатель класса мог считать их предназначенными для своего личного использования в нескольких своих классах!

Теперь, когда вам понятны преимущества модульной организации, у вас может возникнуть вопрос: к каким изменениям в Java приводит ее поддержка. Мы расскажем об этом в следующем разделе.

14.2. Причины создания системы модулей Java

Из этого раздела вы узнаете, почему для языка и компилятора Java была спроектирована новая система модулей. Во-первых, мы поговорим про ограничения модульной организации до Java 9. Далее мы расскажем подноготную библиотеки JDK и объясним, почему так важно было модуляризовать ее.

14.2.1. Ограничения модульной организации

К сожалению, встроенная поддержка модуляризации программных проектов в языке Java до версии 9 была несколько ограничена. В Java было три возможных уровня группировки кода: классы, пакеты и JAR-файлы, а также всегда существовала поддержка модификаторов доступа и инкапсуляции для классов. На уровне пакетов и JAR-файлов, однако, дела с инкапсуляцией обстояли намного хуже.

Ограниченное управление видимостью

Как обсуждалось в предыдущем разделе, в языке Java существуют модификаторы для поддержки скрытия информации. В числе этих модификаторов видимости: `public` (общедоступный), `protected` (защищенный), с доступом в пределах пакета, и `private` (приватный). Но как насчет управления видимостью *между* пакетами? Большинство приложений включает несколько пакетов, предназначенных для группировки различных классов, но возможностей управления видимостью между пакетами немного. Если требуется, чтобы классы и интерфейсы из одного пакета были видимы в другом, придется объявить их как `public`. В результате эти классы и интерфейсы оказываются доступными для всех кого угодно. Вы могли часто наблюдать один из типичных случаев этой проблемы: пакеты-спутники с реализациями по умолчанию, названия которых включают строку `"impl"`. В подобном случае невозможно предотвратить применение таких внутренних реализаций пользователями, поскольку код внутри пакета объявлен `public`. В итоге развитие кода становится невозможным без серьезных изменений, когда предназначенный для внутреннего использования код применяется программистами для «вр*е*менного» решения ка-

ких-либо задач, а затем становится неотъемлемой частью системы. Что еще хуже, такая ситуация ужасна с точки зрения безопасности, поскольку чем больше кода оказывается доступным, тем обширнее поверхность атаки.

Путь к классам

Ранее в этой главе мы обсуждали преимущества программного обеспечения, написанного так, чтобы упростить его поддержку и понимание — другими словами, его анализ. Мы также говорили о разделении ответственности и моделировании зависимостей между модулями. К сожалению, языку Java изначально недоставало поддержки этих идей, в контексте компоновки и запуска приложения. Разработчикам приходится поставлять все скомпилированные классы в виде одного неструктурированного JAR-файла, доступного по пути к классам[1]. В дальнейшем JVM может динамически находить и загружать классы в соответствии с путями к классам по мере необходимости.

К сожалению, у такого сочетания пути к классам и JAR-файлов есть несколько недостатков.

Во-первых, путь к классам никак не отражает различия версий класса. Например, невозможно указать, что класс `JSONParser` из библиотеки синтаксического разбора должен быть версии 1.0 или 2.0, так что невозможно предвидеть, что произойдет, если две версии одной библиотеки окажутся доступными по пути к классам. Такое часто встречается в больших приложениях, поскольку разные компоненты приложения могут использовать различные версии одних и тех же библиотек.

Во-вторых, путь к классам не поддерживает явные зависимости; все классы внутри различных JAR-файлов сливаются в единое множество классов в пути к ним. Другими словами, путь к классам не дает возможности явно выражать зависимость одного JAR-файла от набора содержащихся внутри другого JAR-файла классов. Из-за этого становится сложнее анализировать путь к классам и отвечать на вопросы такого плана.

- Чего-то не хватает?
- Имеют ли место какие-то конфликты?

Решить эту проблему помогают системы сборки, такие как Maven и Cradle. Однако до Java 9 ни в Java, ни в JVM не было какой-либо поддержки внешних зависимостей. Этот клубок проблем часто называют «преисподней JAR-файлов» (JAR Hell) или «преисподней пути к классам» (Class Path Hell). Из-за этих проблем разработчикам часто приходится методом проб и ошибок добавлять и удалять расположенные по пути к классам `class`-файлы в надежде, что JVM выполнит приложение и не сгенерирует исключение времени выполнения вроде `ClassNotFoundException`. Желательно обнаруживать подобные проблемы на ранних стадиях процесса разработки. Согласованное применение системы модулей Java 9 позволяет обнаруживать все такие ошибки во время компиляции.

Впрочем, инкапсуляция и «преисподняя пути к классам» — проблемы не только для архитектуры вашего приложения. Как насчет самого JDK?

---

[1] Такое написание (class path) используется в документации Java, но соответствующий аргумент программ часто называется classpath.

14.2.2. Монолитный JDK

Пакет инструментов Java-разработчика (Java Development Kit, JDK) — коллекция инструментов для создания и запуска Java-программ. Наиболее важные из уже знакомых вам утилит — `javac` для компиляции и `java` для загрузки и запуска Java-приложений, а также библиотека JDK, обеспечивающая, в частности, поддержку ввода/вывода, операций с коллекциями и потоками данных. Первая его версия была выпущена в 1996 году. Важно отдавать себе отчет, что JDK, как и любое программное обеспечение, с тех пор значительно увеличился в размере. Множество технологий были в него добавлены, а затем признаны устаревшими. Хороший пример — CORBA. Неважно, используете ли вы CORBA в своих приложениях; классы CORBA поставляются в комплекте с JDK. Такая ситуация в особенности проблематична для мобильных и облачных приложений, которым обычно не требуются все имеющиеся в JDK компоненты.

Как выйти из этого положения в масштабах всей экосистемы? Шагом вперед стало появление в Java 8 *сжатых профилей* (compact profile). В зависимости от того, какие части библиотеки JDK вас интересовали, вы могли воспользоваться одним из трех профилей, отличавшихся различным объемом потребляемой памяти. Сжатые профили, впрочем, были лишь кратковременным решением. Многие внутренние API JDK не предназначены для всеобщего применения. К сожалению, из-за того, что язык Java не обеспечивает хорошей инкапсуляции, эти API используются повсеместно. Например, класс `sun.misc.Unsafe` задействуется несколькими библиотеками (включая Spring, Netty и Mockito), хотя должен был быть доступен лишь во внутренних компонентах JDK. В результате очень непросто развивать эти API так, чтобы не создать какой-нибудь несовместимости.

Все эти проблемы стали причиной разработки системы модулей Java, вполне подходящей и для модуляризации самого JDK. Если вкратце, понадобились новые средства структурирования, чтобы разработчик мог выбирать нужные ему части JDK, анализировать путь к классам и обеспечивать усиленную инкапсуляцию с целью развития платформы.

14.2.3. Сравнение с OSGi

В этом подразделе мы сравним модули Java 9 с OSGi. Если вы не слышали про OSGi, можете просто пропустить этот подраздел.

До появления в Java 9 модулей, основанных на проекте Jigsaw, в языке уже существовала полнофункциональная система модулей под названием OSGi, хотя формально она и не была частью платформы Java. Инициатива доступа к открытым сервисам (Open Service Gateway initiative, OSGi) возникла в 2000 году и до появления Java 9 фактически служила стандартом реализации модульных приложений для JVM.

На деле OSGi и новая система модулей Java 9 вовсе не несовместимы; они вполне могут сосуществовать в одном приложении. Фактически их возможности пересекаются лишь частично. OSGi охватывает намного более широкую сферу и предоставляет множество отсутствующих в Jigsaw возможностей.

Модули OSGi называются *комплектами* (bundles) и выполняются внутри конкретного фреймворка OSGi. Существует несколько сертифицированных реализа-

ций фреймворков OSGi, но чаще всего применяются два: Apache Felix и Equinox (используемый также для работы Eclipse IDE). При работе внутри OSGi-фреймворка можно удаленно устанавливать, запускать, останавливать и удалять отдельные комплекты без перезагрузки. Другими словами, OSGi описывает четкий жизненный цикл комплектов, включающий перечисленные в табл. 14.1 состояния.

Таблица 14.1. Состояния комплектов в OSGi

Состояние комплекта	Описание
INSTALLED	Комплект был успешно установлен
RESOLVED	Все необходимые для комплекта Java-классы доступны
STARTING	Выполнение комплекта было запущено, а метод BundleActivator.start вызван, но еще ничего не вернул
ACTIVE	Комплект был успешно переведен в активное состояние и выполняется в данный момент
STOPPING	Комплект находится в процессе останова, метод BundleActivator.stop был вызван, но еще ничего не вернул
UNINSTALLED	Комплект был удален. Его нельзя перевести в какое-либо другое состояние

Вероятно, основное преимущество OSGi над Jigsaw — возможность «горячей» замены различных частей приложения без необходимости его перезапуска. Комплект задается с помощью текстового файла, описывающего требуемые для работы комплекта внешние пакеты, а также экспортируемые комплектом внутренние пакеты, которые затем становятся доступными для других комплектов.

Еще одна интересная особенность OSGi — возможность установки во фреймворк различных версий одного комплекта одновременно. Система модулей Java 9 не поддерживает контроля версий, поскольку в Jigsaw на одно приложение до сих пор приходится только один загрузчик классов, в то время как в OSGi для загрузки каждого из комплектов отводится свой собственный загрузчик классов.

14.3. Модули Java: общая картина

В Java 9 появилась новая организационная единица структуры программы — *модуль*. Модуль определяется с помощью нового ключевого слова[1] `module`, за которым следуют его название и тело. Подобные *дескрипторы модулей*[2] располагаются

[1] Формально служащие для описания модулей идентификаторы, такие как module, requires и export, представляют собой ключевые слова с ограниченной областью видимости (restricted keywords). В контексте, где допустимы модули, они интерпретируются как ключевые слова, а в прочих местах программы их по-прежнему (для обратной совместимости) можно использовать в качестве идентификаторов.

[2] Формально текстовая форма модуля называется объявлением модуля (module declaration), а его дескриптором (module descriptor) называют двоичное представление в файле module-info.class.

в специальном файле `module-info.java`, который после компиляции превращается в `module-info.class`. Тело дескриптора модуля состоит из специальных выражений, важнейшие из которых — `requires` и `exports`. Первое из них задает другие модули, необходимые вашему для работы, а `exports` определяет, что из вашего модуля должно быть доступно для использования другими модулями. Мы расскажем вам об этих выражениях подробнее в последующих разделах.

Дескриптор модуля описывает и инкапсулирует один или несколько пакетов (и обычно располагается в том же каталоге, что и эти пакеты), но в простых случаях он экспортирует (делает видимым) только один из них.

Базовая структура дескриптора модуля Java приведена на рис. 14.2.

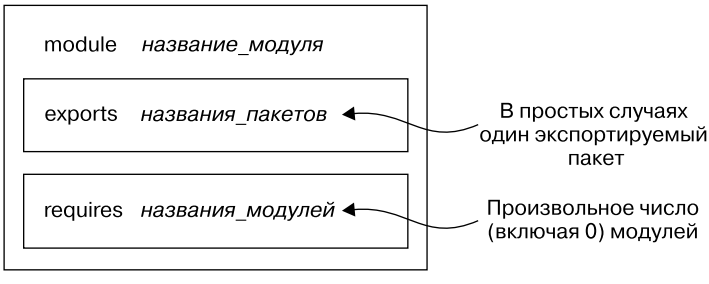


Рис. 14.2. Базовая структура дескриптора модуля Java (module-info.java)

Части `requires` и `exports` модуля удобно рассматривать соответственно как выступы и углубления пазла (от которых и получил, вероятно, свое рабочее название проект Jigsaw[1]). На рис. 14.3 приведен пример с несколькими модулями.

При использовании таких утилит, как Maven, за большую часть нюансов описаний модулей отвечает IDE, от пользователя они скрыты.

В следующем разделе мы изучим вышеупомянутые понятия подробнее, на конкретных примерах.

14.4. Разработка приложения с помощью системы модулей Java

Здесь мы приведем обзор системы модулей Java 9 на примере создания простого модульного приложения с нуля. Вы узнаете, как структурировать, скомпоновать и запустить простое модульное приложение. Мы не станем углубляться во все вопросы, но продемонстрируем общую картину, чтобы при необходимости вы смогли самостоятельно разобраться в теме.

14.4.1. Подготовка приложения

Для начала работы с системой модулей Java нам понадобится проект, для которого мы будем писать код. Возможно, вы часто путешествуете, покупаете много продуктов

[1] От англ. выражения jigsaw puzzle, соответствующего русскому «пазл». — *Примеч. пер.*

и ходите пить кофе с друзьями, так что вам приходится часто иметь дело с чеками. Домашняя бухгалтерия никому радости не доставляет. Чтобы облегчить себе жизнь, вы решили написать приложение для контроля расходов. В число задач этого приложения входят:

- чтение списка расходов из файла или URL;
- синтаксический разбор строковых представлений этих расходов;
- подсчет статистики;
- отображение удобных сводных показателей;
- обеспечение согласования запуска и завершения этих задач.

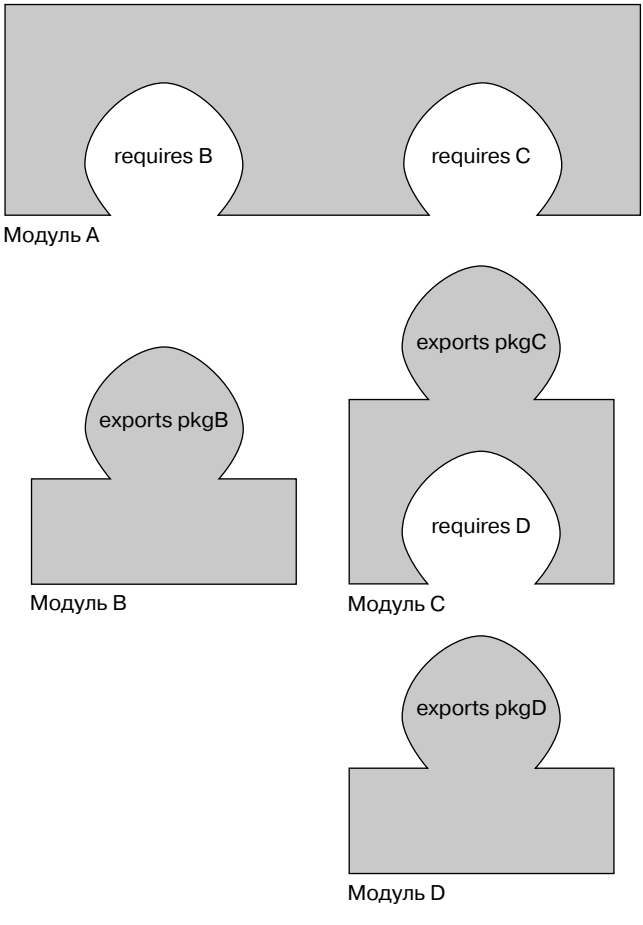


Рис. 14.3. Пример Java-системы из четырех модулей (A, B, C, D) в стиле пазла. Для модуля A нужны модули B и C, поэтому ему требуется доступ к пакетам pkgB и pkgC (экспортируемым модулями B и C соответственно). Аналогично модуль C может использовать пакет pkgD, а модуль B — нет

Для моделирования концептов этого приложения необходимо описать несколько различных классов. Во-первых, интерфейс `Reader` — для чтения сериализованной информации о расходах из источника. У него будет несколько реализаций, например `HttpReader` и `FileReader`, в зависимости от источника данных. Нам также понадобится интерфейс `Parser` для десериализации JSON-объектов в объекты предметной области `Expense`, чтобы работать с ними в Java-приложении. Наконец, нам нужен класс `SummaryCalculator`, который будет вычислять сводные показатели на основе списка объектов `Expense` и возвращать объекты `SummaryStatistics`.

А теперь, создав проект, нужно разобраться, как модуляризовать его с помощью системы модулей Java. Проект явно включает несколько зон ответственности, которые следует отделить друг от друга:

- чтение данных из различных источников (`Reader`, `HttpReader`, `FileReader`);
- синтаксический разбор данных в различных форматах (`Parser`, `JSONParser`, `ExpenseJSONParser`);
- представление объектов предметной области (`Expense`);
- вычисление и возврат сводных показателей (`SummaryCalculator`, `SummaryStatistics`);
- согласование различных зон ответственности (`ExpensesApplication`).

Из соображений большей понятности для читателя мы выбрали мелкоблочный подход. Каждая зона ответственности группируется в отдельный модуль (позднее мы обсудим подробно схему именования модулей):

- `expenses.readers`;
- `expenses.readers.http`;
- `expenses.readers.file`;
- `expenses.parsers`;
- `expenses.parsers.json`;
- `expenses.model`;
- `expenses.statistics`;
- `expenses.application`.

В этом простом приложении мы воспользуемся мелкоблочным разбиением для иллюстрации различных частей системы модулей. На практике подобный мелкоблочный подход для простого проекта привел бы к высоким предварительным затратам при относительно небольшой выгоде в виде должной инкапсуляции небольших частей проекта. По мере роста проекта и добавления новых внутренних реализаций, впрочем, преимущества инкапсуляции и упрощения анализа становятся более явными. Вышеприведенный список можно считать списком пакетов, в зависимости от границ приложения. Модуль объединяет последовательность пакетов. Например, в одном из модулей могут находиться относящиеся к конкретной реализации пакеты, которые нежелательно делать видимыми для остальных

модулей. Например, модуль `expenses.statistics` может содержать несколько пакетов для различных реализаций экспериментальных методов вычисления сводных показателей. В дальнейшем вы сможете выбрать, какие из этих пакетов сделать доступными для пользователей.

14.4.2. Мелкоблочная и крупноблочная модуляризация

При модульной организации системы можно выбрать масштаб разбиения. В случае наиболее мелкоблочной схемы свой модуль имеется у каждого из пакетов (как в предыдущем разделе); при наиболее крупноблочной схеме все пакеты системы содержатся в одном модуле. Как упоминалось ранее, первая из этих схем дает весьма ограниченные преимущества при значительном повышении затрат на проектирование, а при второй теряются все достоинства модульной организации. Оптимальный вариант: умеренное разбиение системы на модули при регулярном пересмотре, чтобы сохранить достаточную (для удобства анализа и модификации развивающегося программного проекта) степень модуляризации.

Короче говоря, модульная организация — то, что не дает проекту «заржаветь».

14.4.3. Основы системы модулей Java

Начнем с простого модульного приложения, включающего лишь один модуль, предназначенный для основного приложения. Структура каталогов проекта имеет следующий вид, где каждый уровень соответствует каталогу:

```
|—  expenses.application
  |—  module-info.java
  |—  com
    |—  example
      |—  expenses
        |—  application
          |—  ExpensesApplication.java
```

Наверное, вы обратили внимание в структуре проекта на загадочный файл `module-info.java`. Как мы уже объясняли ранее в главе, это дескриптор модуля, который должен располагаться в корневом каталоге иерархии файлов исходного кода модуля, задавая зависимости модуля и те пакеты, которые необходимо сделать видимыми.

Для нашего приложения подсчета расходов файл `module-info.java` содержит поименованное пустое описание модуля, поскольку тот не зависит ни от каких модулей и не предоставляет свою функциональность другим модулям. О более продвинутых возможностях мы расскажем вам позднее, начиная с раздела 14.5. Содержимое файла `module-info.java` выглядит следующим образом:

```
module expenses.application {

}
```

Как запустить модульное приложение? Рассмотрим несколько низкоуровневых команд. Этот код автоматически создается IDE и системой сборки, но понимание

происходящего никогда не помешает. Выполните следующие команды, находясь в каталоге исходного кода вашего проекта:

```
javac module-info.java
    com/example/expenses/application/ExpensesApplication.java -d target

jar cvfe expenses-application.jar
    com.example.expenses.application.ExpensesApplication -C target
```

При этом в консоли вы увидите примерно такой вывод, показывающий включенные в сгенерированный JAR-файл (`expenses-application.jar`) каталоги и классы:

```
added manifest
added module-info: module-info.class
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/example/(in = 0) (out= 0)(stored 0%)
adding: com/example/expenses/(in = 0) (out= 0)(stored 0%)
adding: com/example/expenses/application/(in = 0) (out= 0)(stored 0%)
adding: com/example/expenses/application/ExpensesApplication.class(in = 456)
    (out= 306)(deflated 32%)
```

Наконец, запустите сгенерированный JAR-файл как модульное приложение:

```
java --module-path expenses-application.jar \
     --module expenses/com.example.expenses.application.ExpensesApplication
```

Наверное, первые два этапа, представляющие собой обычный способ компоновки Java-приложения в JAR-файл, вам хорошо знакомы. Новое здесь только включение в этап компиляции файла `module-info.java`.

В утилите `java`, выполняющей файлы `.class` Java, появились два новых параметра:

- `--module-path` — задает доступные для загрузки модули. Он отличается от аргумента `--classpath`, который обеспечивает доступность `class`-файлов;
- `--module` — позволяет указать основные запускаемые модуль и класс.

Объявление модуля не содержит строки с его версией. Система модулей Java 9 не была предназначена для решения задачи выбора версии, так что контроль версий не поддерживается. Такое проектное решение объясняется тем, что эту функцию должны выполнять системы сборки и приложения-контейнеры.

14.5. Работа с несколькими модулями

Теперь, когда вы уже знаете, как подготовить простое приложение с одним модулем, можете приступить к немного более реалистическому примеру с несколькими модулями. Нам нужно, чтобы наше приложение читало информацию о расходах из источника. Для этого мы создадим инкапсулирующий эти обязанности новый модуль `expenses.readers`. Взаимодействие между модулями `expenses.application` и `expenses.readers` задается с помощью выражений `requires` и `exports` Java 9.

14.5.1. Выражение exports

Объявление модуля `expenses.readers` может выглядеть, например, следующим образом (не задумывайтесь пока насчет синтаксиса и прочих нюансов, мы вернемся к этим вопросам позднее):

```
module expenses.readers {

    exports com.example.expenses.readers;
    exports com.example.expenses.readers.file;
    exports com.example.expenses.readers.http;
}
```

Это названия пакетов, не модулей

Новое здесь только выражение `exports`, с помощью которого общедоступные (`public`) типы из конкретных пакетов делаются видимыми для других модулей. По умолчанию все инкапсулируется внутри модуля. Система модулей использует подход типа «белый список», который помогает добиться высокой степени инкапсуляции за счет требования явного указания, что нужно сделать доступным для применения другими модулями. Этот подход предотвращает случайный экспорт каких-либо предназначенных для внутреннего использования возможностей, которыми спустя многие годы могут воспользоваться хакеры для взлома вашей системы.

Структура каталогов двухмодульной версии проекта выглядит следующим образом:

```
|-  expenses.application
  |-  module-info.java
  |-  com
    |-  example
      |-  expenses
        |-  application
          |-  ExpensesApplication.java

|-  expenses.readers
  |-  module-info.java
  |-  com
    |-  example
      |-  expenses
        |-  readers
          |-  Reader.java
        |-  file
          |-  FileReader.java
        |-  http
          |-  HttpReader.java
```

14.5.1. Выражение requires

Можно было также написать в файле `module-info.java` следующее:

```
module expenses.readers {
    requires java.base;
```

Это название модуля, а не пакета

```
    exports com.example.expenses.readers;
    exports com.example.expenses.readers.file;
    exports com.example.expenses.readers.http;
}
```

Это название пакета, не модуля

Новый для нас здесь элемент — выражение `requires`, с помощью которого можно указать, от каких модулей зависит данный. По умолчанию все модули зависят от модуля платформы `java.base`, включающего основные пакеты Java, в том числе `net`, `io` и `util`. Данный модуль всегда подгружается по умолчанию, так что указывать это явным образом не нужно (подобно тому как `"class Foo { … }"` в языке Java эквивалентно `"class Foo extends Object { … }"`).

Выражение `requires` полезно в случаях, когда нужно импортировать еще какие-то модули, кроме `java.base`.

Комбинирование выражений `requires` и `exports` обеспечивает возможность более продвинутого контроля доступа к классам в Java 9. В табл. 14.2 приведена краткая сводка различий в видимости при различных модификаторах доступа до и после Java 9.

Таблица 14.2. Java 9 предоставляет больше возможностей контроля видимости классов

Видимость классов	До Java 9	С появлением Java 9
Все классы общедоступны (public) для всех	✓✓	✓✓ (сочетание выражений exports и requires)
Общедоступно (public) ограниченное количество классов	✗✗	✓✓ (сочетание выражений exports и requires)
Классы общедоступны (public) только внутри одного модуля	✗✗	✓ (без выражения exports)
Защищенная (protected)	✓✓	✓✓
На уровне пакета	✓✓	✓✓
Приватная (private)	✓✓	✓✓

14.5.3. Именование

На этом этапе не помешает рассказать про соглашение об именах модулей. Мы воспользовались сокращенным вариантом (например, `expenses.application`), чтобы не смешивать понятия модулей и пакетов (модуль может экспортировать несколько пакетов). Однако рекомендуется опираться на иное соглашение.

Oracle рекомендует использовать для модулей то же соглашение об обратном порядке именования интернет-доменов (например, `com.iteratrlearning.training`), что и для пакетов. Более того, названия модулей должны соответствовать их основному экспортируемому пакету API, который также должен следовать этому соглашению. Если в модуле такой пакет отсутствует или по иным причинам ему требуется название, не соответствующее ни одному из его экспортируемых пакетов, то оно должно начинаться с обратной формы интернет-домена его автора.

Теперь, когда вы разобрались в подготовке проекта с несколькими модулями, поговорим о том, как его скомпоновать и запустить. Мы обсудим этот вопрос в следующем разделе.

14.6. Компиляция и компоновка

Ознакомившись с подготовкой проекта и объявлением модуля, вы уже готовы посмотреть на компиляцию проекта с помощью системы сборки, такой как Maven. В этом разделе мы предполагаем, что вы знакомы с Maven — одной из самых распространенных утилит сборки в экосистеме Java. Еще одна популярная система сборки — Gradle, и мы рекомендуем вам с ней тоже познакомиться, если вы о ней еще не слышали.

Во-первых, необходимо создать для каждого модуля файл `pom.xml`. Фактически можно компилировать модули отдельно, они ведут себя в этом смысле как отдельные проекты. Потребуется также добавить файл `pom.xml` для родительского модуля всех модулей, чтобы согласовать сборку проекта в целом. Общая структура после этого выглядит следующим образом:

```
|–  pom.xml
|–  expenses.application
  |–  pom.xml
  |–  src
    |–  main
      |–  java
        |–  module-info.java
        |–  com
          |–  example
            |–  expenses
              |–  application
                |–  ExpensesApplication.java
|–  expenses.readers
  |–  pom.xml
  |–  src
    |–  main
      |–  java
        |–  module-info.java
        |–  com
          |–  example
            |–  expenses
              |–  readers
                |–  Reader.java
              |–  file
                |–  FileReader.java
              |–  http
                |–  HttpReader.java
```

Обратите внимание на три новых файла `pom.xml` и структуру Maven каталогов проекта. Дескриптор модуля (файл `module-info.java`) должен располагаться в каталоге `src/main/java`. Maven обеспечит использование утилитой `javac` соответствующего пути к исходному коду модуля.

Файл `pom.xml` для проекта `expenses.readers` выглядит следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>expenses.readers</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.example</groupId>
        <artifactId>expenses</artifactId>
        <version>1.0</version>
    </parent>
</project>
```

Важно отметить, что в этом коде явным образом упоминается родительский модуль для упрощения процесса сборки с `artifactId`, равным `expenses`. Как вы вскоре увидите, родительский модуль необходимо описать в файле `pom.xml`.

Далее следует задать файл `pom.xml` для модуля `expenses.application`. Он аналогичен предыдущему, но в него придется добавить зависимость от проекта `expenses.readers`, поскольку содержащиеся в нем классы и интерфейсы нужны для компиляции класса `ExpensesApplication`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>expenses.application</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>com.example</groupId>
        <artifactId>expenses</artifactId>
        <version>1.0</version>
    </parent>

    <dependencies>
        <dependency>
                <groupId>com.example</groupId>
                <artifactId>expenses.readers</artifactId>
                <version>1.0</version>
        </dependency>
    </dependencies>

</project>
```

После создания файлов `pom.xml` для двух модулей, `expenses.application` и `expenses.readers`, можно создать глобальный файл `pom.xml` для общего управления процессом сборки. Maven поддерживает проекты с несколькими модулями Maven

благодаря специальному XML-элементу `<module>`, который ссылается на `artifactId` дочерних модулей. Вот полное описание со ссылками на два дочерних модуля — `expenses.application` и `expenses.readers`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>expenses</artifactId>
    <packaging>pom</packaging>
    <version>1.0</version>

    <modules>
    <module>expenses.application</module>
        <module>expenses.readers</module>
    </modules>

    <build>
        <pluginManagement>
            <plugins>
            <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.7.0</version>
                <configuration>
                        <source>9</source>
                        <target>9</target>
                </configuration>
            </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

Поздравляем! Теперь можете выполнить команду `mvn clean package`, чтобы сгенерировать JAR-файлы для модулей проекта. Эта команда генерирует два JAR-файла:

```
./expenses.application/target/expenses.application-1.0.jar
./expenses.readers/target/expenses.readers-1.0.jar
```

Для запуска нашего модульного приложения необходимо включить эти два JAR-файла в путь к модулям вот так:

```
java --module-path \
 ./expenses.application/target/expenses.application-1.0.jar:\
 ./expenses.readers/target/expenses.readers-1.0.jar \
    --module \
 expenses.application/com.example.expenses.application.ExpensesApplication
```

Пока что мы занимались созданными нами модулями и показали, как воспользоваться выражением `requires` для ссылки на `java.base`. Настоящие приложения, однако, зависят от внешних модулей и библиотек. Как это все работает и что будет,

если в старую библиотеку не добавили явный файл `module-info.java`? В следующем разделе мы ответим на эти вопросы, познакомив вас с автоматическими модулями.

14.7. Автоматические модули

Возможно, вы сочтете реализацию вашего `HttpReader` слишком низкоуровневой и захотите вместо нее воспользоваться специализированной библиотекой, например `httpclient` из проекта Apache. Каким образом включить эту библиотеку в проект? Вы уже знаете про выражение `requires`, так что можно попробовать добавить ее в файл `module-info.java` для проекта `expenses.readers`. Выполните команду `mvn clean package`, чтобы увидеть, что получится. К сожалению, результаты неутешительны:

```
[ERROR] module not found: httpclient
```

Эта ошибка возникла потому, что вы забыли обновить `pom.xml` и указать там нужную зависимость. При сборке проекта, включающего файл `module-info.java` плагин компилятора Maven добавляет все зависимости в путь к модулям, в результате чего соответствующие JAR-файлы скачиваются и опознаются в проекте:

```
<dependencies>
    <dependency>
        <groupId>org.apache.httpcomponents</groupId>
        <artifactId>httpclient</artifactId>
        <version>4.5.3</version>
    </dependency>
</dependencies>
```

Теперь при выполнении команды `mvn clean package` проект будет собран без ошибок. Впрочем, обратите внимание на интересный нюанс: библиотека `httpclient` — это не Java-модуль. Это внешняя библиотека, которую мы хотели бы использовать в качестве модуля, но она пока не была модуляризована. Java преобразует подходящие JAR-файлы в так называемые автоматические модули. Любой JAR-файл без `module-info`, расположенный в соответствии с путями к модулям, становится автоматическим модулем. Автоматические модули неявно экспортируют все свои пакеты. Название для автоматического модуля также формируется автоматически, на основе названия JAR-файла. Существует несколько способов формирования этого названия, простейший из которых — воспользоваться командой `jar` с аргументом `--describe-module`:

```
jar --file=./expenses.readers/target/dependency/httpclient-4.5.3.jar \
    --describe-module
httpclient@4.5.3 automatic
```

В данном случае название — `httpclient`.

Последний шаг — запуск приложения и добавление JAR-файла `httpclient` в путь к модулям:

```
java --module-path \
 ./expenses.application/target/expenses.application-1.0.jar:\
 ./expenses.readers/target/expenses.readers-1.0.jar \
 ./expenses.readers/target/dependency/httpclient-4.5.3.jar \
```

```
   --module \
 expenses.application/com.example.expenses.application.ExpensesApplication
```

ПРИМЕЧАНИЕ

Существует проект (https://github.com/moditect/moditect), нацеленный на улучшение поддержки системы модулей Java 9 в Maven, например на автоматическую генерацию файлов module-info.

14.8. Объявление модуля и выражения

Система модулей Java — весьма объемная тема. Мы рекомендуем вам прочитать отдельную книгу по этой теме, если вы хотите узнать больше. Тем не менее в этом разделе мы приведем краткий обзор других ключевых слов языка объявления модулей, чтобы дать вам представление об имеющихся возможностях.

Как вы знаете из предыдущих разделов, модули объявляются с помощью директивы `module`. Вот объявление модуля `com.iteratrlearning.application`:

```
module com.iteratrlearning.application {

}
```

А что же указывается внутри объявления модуля? Вы уже знаете про выражения `requires` и `exports`, но существуют и другие, в том числе `requires ... transitive`, `exports ... to`, `open`, `opens`, `uses` и `provides`.

14.8.1. Выражение requires

Выражение `requires` позволяет указывать, что модуль зависит от другого модуля, как во время компиляции, так и в процессе выполнения. Например, модуль `com.iteratrlearning.application` зависит от модуля `com.iteratrlearning.ui`:

```
module com.iteratrlearning.application {
    requires com.iteratrlearning.ui;
}
```

В результате модуль `com.iteratrlearning.application` сможет использовать только экспортируемые модулем `com.iteratrlearning.ui` `public`-типы.

14.8.2. Выражение exports

Выражение `exports` делает объявленные как `public` типы из конкретных пакетов доступными для применения в других модулях. По умолчанию никакие пакеты не экспортируются. Благодаря явному указанию экспортируемых пакетов инкапсуляция становится намного строже. В следующем примере экспортируются пакеты `com.iteratrlearning.ui.panels` и `com.iteratrlearning.ui.widgets` (обратите внимание, что выражение `exports` принимает в качестве аргумента *название пакета*, а выражение `requires` — *название модуля*, несмотря на схожие схемы наименования).

```
module com.iteratrlearning.ui {
    requires com.iteratrlearning.core;
```

```
    exports com.iteratrlearning.ui.panels;
    exports com.iteratrlearning.ui.widgets;
}
```

14.8.3. Выражение requires transitive

Существует возможность указать, что модуль может использовать объявленные как `public` типы, указанные в выражении `requires` другого модуля. Например, можно поменять выражение `requires` внутри модуля `com.iteratrlearning.ui` на `requires transitive`:

```
module com.iteratrlearning.ui {
    requires transitive com.iteratrlearning.core;

    exports com.iteratrlearning.ui.panels;
    exports com.iteratrlearning.ui.widgets;
}

module com.iteratrlearning.application {
    requires com.iteratrlearning.ui;
}
```

В результате модуль `com.iteratrlearning.application` получит доступ к общедоступным типам, экспортируемым модулем `com.iteratrlearning.core`. Транзитивность полезна тогда, когда указанный в `requires` модуль (в данном случае `com.iteratrlearning.ui`) возвращает типы из другого модуля, указанного в его собственном выражении `requires` (`com.iteratrlearning.core`). Было бы весьма неудобно заново писать `requires com.iteratrlearning.core` в модуле `com.iteratrlearning.application`. Эту проблему решает выражение `requires transitive`. Теперь любой модуль, зависящий от `com.iteratrlearning.ui`, автоматически читает модуль `com.iteratrlearning.core`.

14.8.4. Выражение exports ... to

Возможности контроля видимости простираются и дальше, вы можете ограничить круг пользователей конкретного экспорта конструкцией `exports to`. Как вы видели в подразделе 14.8.2, можно ограничить круг пользователей пакета `com.iteratrlearning.ui.widgets` модулем `com.iteratrlearning.ui.widgetuser`, изменив его объявление следующим образом:

```
module com.iteratrlearning.ui {
    requires com.iteratrlearning.core;

    exports com.iteratrlearning.ui.panels;
    exports com.iteratrlearning.ui.widgets to
       com.iteratrlearning.ui.widgetuser;
}
```

14.8.5. Выражения open и opens

Объявление модуля с квалификатором `open` открывает другим модулям рефлексивный доступ ко всем его пакетам. Квалификатор `open` не влияет на видимость модуля, только разрешает рефлексивный доступ, как показано в следующем примере:

```
open module com.iteratrlearning.ui {

}
```

До появления Java 9 можно было считывать приватные состояния объектов с помощью рефлексии. Другими словами, настоящей инкапсуляции не существовало. Эта возможность часто применялась в утилитах объектно-реляционного отображения (object-relational mapping, ORM), таких как Hibernate, для прямого доступа и модификации состояния. В Java 9 рефлексия более не разрешена по умолчанию. Выражение `open` из предыдущего кода позволяет разрешать такое поведение при необходимости.

Вместо того чтобы открывать для рефлексии весь модуль, можно воспользоваться выражением `opens` в объявлении модуля, чтобы выборочно открыть только нужные пакеты. Можно также воспользоваться квалификатором `to` в варианте `open ... to`, чтобы ограничить модули, которым разрешен рефлексивный доступ, аналогично тому, как `exports ... to` ограничивает модули, которым разрешено делать `requires` экспортированного пакета.

14.8.6. uses и provides

Если вы знакомы с сервисами и классом `ServiceLoader`, то благодаря системе модулей Java можете описать модуль как поставщик сервиса с помощью выражения `provides` или потребитель сервиса с помощью выражения `uses`. Однако это весьма непростая тема, которая выходит далеко за рамки данной главы. Если вас интересуют вопросы сочетания модулей и загрузчиков сервисов, мы рекомендуем вам обратиться к такому всеобъемлющему источнику информации, как *The Java Module System* Николаи Парлога, упомянутому выше в этой главе.

14.9. Пример побольше и дополнительные источники информации

Чтобы попробовать на вкус, что такое система модулей Java, взгляните на следующий пример, взятый из документации компании Oracle по языку Java. Он включает объявление модуля с использованием большинства обсуждавшихся в этой главе возможностей. Мы хотим с его помощью не напугать вас (абсолютное большинство операторов модуля — простые `exports` и `requires`), а просто продемонстрировать некоторые более продвинутые возможности:

```
module com.example.foo {
    requires com.example.foo.http;
    requires java.logging;
    requires transitive com.example.foo.network;

    exports com.example.foo.bar;
    exports com.example.foo.internal to com.example.foo.probe;

    opens com.example.foo.quux;
    opens com.example.foo.internal to com.example.foo.network,
                                      com.example.foo.probe;
```

```
    uses com.example.foo.spi.Intf;
    provides com.example.foo.spi.Intf with com.example.foo.Impl;
}
```

В этой главе мы обсудили, для чего понадобилась новая система модулей Java, и вкратце рассмотрели ее основные возможности. Мы не охватили многие ее функции, в том числе загрузчики сервисов, дополнительные выражения для описания модулей и такие утилиты для работы с модулями, как `jdeps` и `jlink`. Если вы разработчик приложений на платформе Java EE, то при миграции приложений на Java 9 учтите, что несколько относящихся к Java EE пакетов не загружаются по умолчанию в модуляризованной виртуальной машине Java 9. Например, классы API JAXB сейчас считаются API Java EE и отсутствуют в пути к классам по умолчанию в Java SE 9. Для сохранения совместимости необходимо явным образом добавить нужные вам модули, указав параметр командной строки `--add-modules`. Например, для добавления `java.xml.bind` следует указать `--add-modules java.xml.bind`.

Как мы уже отмечали ранее, полноценное рассмотрение системы модулей Java потребовало бы целой книги, а не одной-единственной главы. Более подробное обсуждение всех ее нюансов вы можете найти в книге *The Java Module System* Николаи Парлога.

Резюме

- Разделение ответственности и скрытие информации — два важных принципа, с помощью которых можно создавать легко анализируемое программное обеспечение.
- До появления Java 9 модульность кода обеспечивалась с помощью пакетов, классов и интерфейсов со своими конкретными зонами ответственности, но возможностей этих элементов недостаточно для эффективной инкапсуляции.
- Проблема «преисподней пути к классам» сильно усложняет анализ зависимостей приложения.
- До появления Java 9 JDK был монолитным, что приводило к высокой стоимости разработки и ограничению возможностей развития.
- В Java 9 появилась новая система модулей, в которой файл `module-info.java` задает название модуля, его зависимости (`requires`) и общедоступные API (`exports`).
- Выражение `requires` позволяет задавать зависимости модуля от других модулей.
- Выражение `exports` делает объявленные как `public` типы конкретных пакетов модуля доступными для других модулей.
- Рекомендуемое соглашение о наименовании модуля соответствует соглашению об обратном порядке наименования интернет-доменов.
- Все JAR-файлы, расположенные по пути к модулям, у которых отсутствует файл `module-info`, становятся автоматическими модулями.
- Автоматические модули неявно экспортируют все свои пакеты.
- Система сборки Maven поддерживает приложения, структурированные с помощью системы модулей Java 9.

Часть V

Расширенная конкурентность в языке Java

Пятая часть данной книги исследует способы структурирования конкурентных программ на языке Java — более продвинутые, чем простая параллельная обработка потоков данных, с которой вы уже познакомились в главах 6 и 7. Опять же ничто остальное в книге не зависит от информации из этой части, так что можете смело пропустить ее, если эти идеи вам (пока что) не нужны.

Глава 15 появилась во втором издании книги и описывает идею асинхронных API «с высоты птичьего полета» — включая понятие фьючерса и протокол публикации-подписки, лежащий в основе реактивного программирования и включенный в Flow API (API потоков сообщений/событий) Java 9.

Глава 16 посвящена классу `CompletableFuture`, с помощью которого можно выражать сложные асинхронные вычисления декларативным образом — аналогично Stream API для синхронных вычислений.

Глава 17 тоже появилась только во втором издании, в ней подробно изучается Flow API Java 9 с упором на пригодный для практического использования реактивный код.

15 Основные концепции класса CompletableFuture и реактивное программирование

В этой главе

- Потоки выполнения, фьючерсы и силы, побуждающие Java поддерживать более развитые API конкурентности.
- Асинхронные API.
- Конкурентные вычисления наглядно, в виде блоков и каналов.
- Использование комбинаторов `CompletableFuture` для динамического соединения блоков.
- Протокол публикации/подписки — основа Flow API для реактивного программирования на языке Java 9.
- Реактивное программирование и реактивные системы.

В последние годы наметились две тенденции, которые заставляют разработчиков переосмысливать способы написания программного обеспечения. Первая из них связана с аппаратным обеспечением, на котором работают приложения, а вторая — со структурой приложений (особенно со способом их взаимодействия). В главе 7 мы обсудили влияние тенденций в аппаратном обеспечении и отметили, что со времени появления многоядерных процессоров их максимально полное использование — наиболее эффективный способ ускорения работы приложений. Как

вы видели, можно разбивать большие задачи и выполнять полученные подзадачи параллельно друг с другом. Вы также научились делать это проще и эффективнее (по сравнению с непосредственной работой с потоками выполнения) с помощью фреймворка ветвления-объединения (появился в Java 7) и параллельных потоков данных (появились в Java 8).

Вторая тенденция отражает растущую доступность интернет-сервисов и их использование приложениями. Например, в последние несколько лет все большую популярность набирает микросервисная архитектура, при которой приложение не является монолитным, а делится на меньшие сервисы. Согласование работы таких меньших сервисов приводит к росту обмена сетевыми сообщениями. Аналогично к множеству интернет-сервисов можно обращаться через общедоступные API, созданные такими компаниями, как Google (информация о местоположении), Facebook (социальная информация) и Twitter (новости). Сегодня редко можно встретить сайт или сетевое приложение, которое бы работало совершенно изолированно. Гораздо чаще попадаются гибридные веб-приложения, получающие контент из нескольких источников и агрегирующие его для упрощения жизни пользователей.

Допустим, вам нужно создать сайт для сбора и агрегирования тональностей высказываний в соцсетях по заданной тематике для ваших друзей из Франции. Для этого можно воспользоваться API Facebook или Twitter для поиска наиболее популярных комментариев на эту тему на различных языках и ранжирования наиболее релевантных с помощью внутренних алгоритмов. Далее можно использовать Google Translate для перевода этих комментариев на французский или Google Maps для определения географического местоположения их авторов, агрегирования всей этой информации и отображения ее на сайте.

Конечно, если какие-либо из этих внешних сетевых сервисов реагируют медленно, то лучше выдавать пользователям лишь часть результатов, например показывать текстовые результаты рядом с общей картой с вопросительным знаком на ней, вместо того чтобы демонстрировать пустой экран, пока картографический сервис не ответит или не истечет время ожидания. Рисунок 15.1 демонстрирует взаимодействие подобного гибридного приложения с удаленными сервисами.

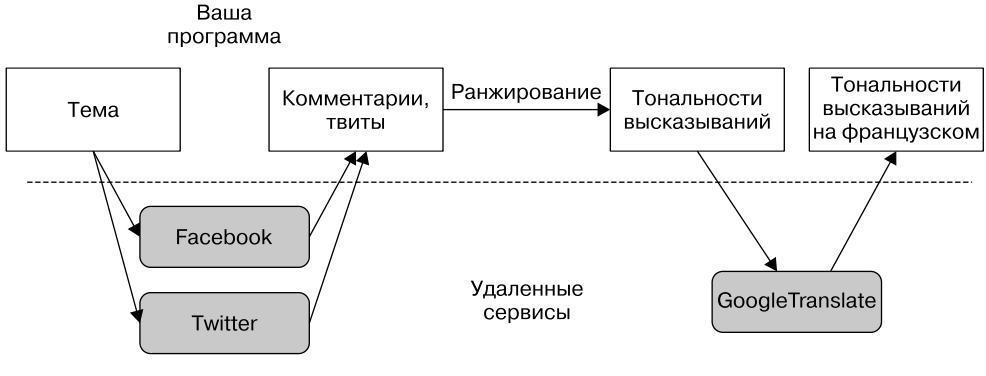


Рис. 15.1. Типичное гибридное приложение

Для реализации подобных приложений необходимо обращаться через Интернет к нескольким веб-сервисам. Но при этом нежелательно блокировать вычисления и тратить миллиарды драгоценных тактов процессора на ожидание ответа от этих сервисов. Например, какой смысл ждать ответа от Facebook перед обработкой поступающих из Twitter данных?

Эта ситуация — обратная сторона многозадачного программирования. Фреймворк ветвления-объединения и параллельные потоки данных, обсуждавшиеся в главе 7, очень полезны для реализации параллелизма; они разбивают задачу на множество подзадач и выполняют их параллельно на различных ядрах, процессорах или даже машинах.

И напротив, в случае конкурентности, а не параллелизма или когда главная цель состоит в выполнении нескольких слабо связанных задач на одних и тех же процессорах (причем так, чтобы их ядра оставались как можно более загруженными с целью максимизации пропускной способности приложения) желательно избегать блокировки потоков выполнения и напрасной траты вычислительных ресурсов во время ожидания (возможно, довольно длительного) ответа от удаленного сервиса или базы данных.

Java предлагает два основных набора инструментов для подобных случаев. Во-первых, как вы увидите в главах 16 и 17, это интерфейс `Future`, в особенности его реализация `CompletableFuture` из Java 8, — с его помощью часто можно создавать простые и эффективные решения (глава 16). А позднее, в Java 9, появилось реактивное программирование, основанное на идее так называемого протокола публикации/подписки через Flow API, которое дает возможность реализации более продвинутых подходов программирования (глава 17).

Рисунок 15.2 иллюстрирует различия между конкурентностью и параллелизмом. Конкурентность — программное свойство (перекрывающееся выполнение), возможное даже на одноядерной машине, а параллелизм — свойство выполняющего программу аппаратного обеспечения (одновременное выполнение).

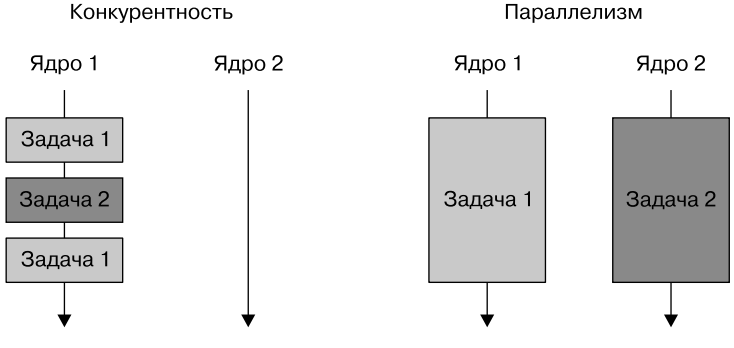


Рис. 15.2. Конкурентность и параллелизм

Почти вся глава посвящена изложению идей, лежащих в основе новых API Java — CompletableFuture и Flow API.

Мы начнем с рассказа об эволюции конкурентности в Java, включая потоки выполнения и абстракции более высокого уровня, в том числе пулы потоков выполнения и фьючерсы (раздел 15.1). Отметим, что в главе 7 мы имели дело в основном с параллелизмом в реализующих циклы программах. В разделе 15.2 мы поговорим об оптимальном использовании конкурентности для вызовов методов. В разделе 15.3 отдельные части программы схематически рассматриваются в виде блоков, взаимодействующих через каналы связи. В разделах 15.4 и 15.5 мы рассмотрим класс `CompletableFuture` и принципы реактивного программирования в Java 8 и 9. Наконец, в разделе 15.6 мы объясним, в чем разница между реактивной системой и реактивным программированием.

Указания читателю

В этой главе будет мало реального кода на языке Java. Тем читателям, кого интересует лишь код, мы рекомендуем сразу перейти к главам 16 и 17. С другой стороны, общеизвестно, что реализующий непривычные идеи код обычно малопонятен. Поэтому для пояснения глобальных идей, таких как лежащий в основе Flow API протокол публикации/подписки, мы воспользуемся простыми функциями и приведем поясняющие схемы.

Мы поясним большинство концепций на живых примерах, демонстрирующих использование различных конкурентных возможностей Java для вычисления выражений типа `f(x)+g(x)` с последующим возвратом, или выводом в консоль, результатов при условии, что `f(x)` и `g(x)` — «долгоиграющие» вычисления.

15.1. Эволюция поддержки конкурентности в Java

Поддержка конкурентного программирования в Java существенно выросла, отражая в основном изменения аппаратного обеспечения, программных систем и концепций программирования за последние 20 лет. Далее мы приведем краткую историю ее развития, чтобы вам стали понятнее причины появления новых возможностей и их роль в программировании и проектировании систем.

В языке Java с самого начала были блокировки (с указанием `synchronized` для классов и методов), реализации `Runnable` и потоков выполнения (класс `Thread`). В 2004 году в Java 5 появился пакет `java.util.concurrent`. Он поддерживал более широкие возможности выражения конкурентности, в частности интерфейс `ExecutorService`[1] (отделявший отправку задачи на выполнение от выполнения потока), а также интерфейсы `Callable<T>` и `Future<T>` с более высокоуровневыми и возвращающими результат вариантами `Runnable` и `Thread`. Кроме того, этот пакет использовал обобщенные типы (также появившиеся в Java 5). Объ-

[1] Интерфейс ExecutorService расширяет интерфейс Executor, добавляя метод submit для запуска Callable; в интерфейсе Executor был лишь метод execute для Runnable.

екты `ExecutorService` пригодны для выполнения экземпляров как `Runnable`, так и `Callable`. Эти возможности упростили параллельное программирование на многоядерных процессорах. Откровенно говоря, никому не нравится работать непосредственно с потоками выполнения!

В более поздних версиях Java поддержка конкурентности продолжила расширяться по мере увеличения спроса на нее со стороны разработчиков, которые нуждались в возможностях эффективного программирования многоядерных процессоров. Как вы уже видели в главе 7, в Java 7 был добавлен класс `java.util.concurrent.RecursiveTask` для поддержки реализации алгоритмов типа «разделяй и властвуй» на основе ветвления-объединения, а в Java 8 прибавилась поддержка потоков данных и их параллельной обработки (на основе только что появившейся поддержки лямбда-выражений).

Еще более обогатила возможности конкурентности Java поддержка *композиции* фьючерсов (посредством реализации класса `CompletableFuture` интерфейса `Future`, см. раздел 15.4 и главу 16) и Java 9 с его явной поддержкой распределенного асинхронного программирования. Эти API служат ментальной моделью и инструментарием для создания гибридных веб-приложений, упоминавшихся во вступлении к этой главе. Например, к их числу можно отнести приложение, работающее за счет обращения к различным веб-сервисам и объединяющее полученную от них информацию в режиме реального времени для пользователя или другого веб-сервиса. Такой процесс называется *реактивным программированием* (reactive programming), и Java 9 поддерживает его с помощью *протокола публикации/подписки* (publish-subscribe protocol) (определяется интерфейсом `java.util.concurrent.Flow`; см. раздел 15.5 и главу 17). Основная идея класса `CompletableFuture` и интерфейса `java.util.concurrent.Flow` состоит в предоставлении разработчику языковых конструкций для конкурентного выполнения при любой возможности не зависящих друг от друга задач, причем с максимально полным использованием параллелизма (многоядерных процессоров или нескольких машин).

15.1.1. Потоки выполнения и высокоуровневые абстракции

О потоках выполнения и процессах обычно рассказывают в курсе операционных систем (ОС). Одноядерные процессоры могут работать с несколькими пользователями за счет того, что ОС выделяет каждому из пользователей по процессу. Операционная система выделяет этим процессам отдельные виртуальные адресные пространства, так что каждому пользователю кажется, что он единственный. Операционная система углубляет эту иллюзию за счет разделения CPU между процессами. Процесс может запросить у ОС выделения ему одного или нескольких *потоков выполнения* (threads) — процессов, совместно использующих адресное пространство их родительского процесса и выполняющих за счет этого задачи сообща и конкурентно.

В многоядерной среде, например в однопользовательском ноутбуке с одним пользовательским процессом, программа может полностью задействовать вычислительные мощности компьютера лишь с помощью потоков выполнения. Каждое ядро может использоваться для одного или нескольких процессов либо потоков выполнения, но если программа не использует потоки выполнения, то задействует лишь одно ядро.

Более того, если процессор четырехъядерный и все ядра непрерывно выполняют полезную работу, то программа будет теоретически выполняться в четыре раза быстрее (накладные расходы, конечно, немного ухудшают этот показатель). При заданном массиве чисел размером 1 000 000, содержащем количество правильных ответов студентов, сравните программу:

```
long sum = 0;
for (int i = 0; i < 1_000_000; i++) {
    sum += stats[i];
}
```

отлично работавшую в одном потоке выполнения во времена одноядерных процессоров, и версию с четырьмя потоками выполнения, первый из которых выполняет такой код:

```
long sum0 = 0;
for (int i = 0; i < 250_000; i++) {
    sum0 += stats[i];
}
```

а четвертый выполняет следующий код:

```
long sum3 = 0;
for (int i = 750_000; i < 1_000_000; i++) {
    sum3 += stats[i];
}
```

Эти четыре потока дополняет главная программа, запускающая их по очереди (`.start()` в Java), ожидающая их завершения (`.join()`), а затем вычисляющая:

```
sum = sum0 + … + sum3;
```

Проблема в том, что делать это для каждого цикла утомительно и чревато ошибками. Кроме того, что делать для кода, отличного от цикла?

В главе 7 мы показали, как достичь подобного параллелизма с помощью потоков данных Java, написав минимальное количество кода, благодаря использованию внутренней итерации вместо внешней (явных циклов):

```
sum = Arrays.stream(stats).parallel().sum();
```

Основной вывод из этого: итерация с помощью параллельных потоков данных — концепция более высокого уровня по сравнению с явным применением потоков выполнения. Другими словами, такое применение потоков данных *абстрагирует* этот паттерн использования потоков выполнения. Такое абстрагирование аналогично паттерну проектирования, но преимущество состоит в том, что основные сложности реализации скрыты внутри библиотеки, а не представлены в виде стереотипного кода. В главе 7 также рассказывалось об использовании поддержки классом `java.util.concurrent.RecursiveTask` абстракции ветвления-объединения потоков выполнения в Java 7 для распараллеливания алгоритмов на основе принципа «разделяй и властвуй», например, с целью реализации более высокоуровневого способа суммирования массива на многоядерной машине.

Прежде чем изучать другие абстракции для потоков выполнения, мы обратимся к идеям (из Java 5) `ExecutorService` и пулов потоков выполнения, лежащим в основе этих дальнейших абстракций.

15.1.2. Исполнители и пулы потоков выполнения

В качестве высокоуровневой абстракции, позволяющей использовать все возможности потоков выполнения, Java 5 предоставляет фреймворк `Executor` и пулы потоков выполнения, с помощью которых Java-программисты могут расцеплять отправку задач на выполнение с собственно их выполнением.

Проблемы, возникающие при использовании потоков выполнения

Потоки выполнения Java обращаются напрямую к потокам выполнения операционной системы. Проблема состоит в больших затратах на создание и уничтожение (включая взаимодействие с таблицей страниц) потоков выполнения ОС, а также в ограниченности их количества. Превышение допустимого количества потоков выполнения операционной системы с большой вероятностью приведет к загадочному сбою Java-приложения, так что постарайтесь не создавать новые потоки выполнения без уничтожения существующих.

Количество потоков выполнения операционной системы (и Java) существенно превышает количество аппаратных потоков выполнения[1], чтобы все аппаратные потоки выполнения были заняты выполнением кода даже в том случае, когда часть потоков ОС блокирована или приостановлена. В качестве примера можно привести серверный процессор 2016 года Intel Core i7-6900K с восемью ядрами, каждое из которых — с двумя потоками симметричной мультипроцессорной обработки (SMP). В результате получается 16 потоков выполнения, причем в сервере может быть несколько таких процессоров со, скажем, 64 аппаратными потоками выполнения суммарно. И наоборот, в ноутбуке может быть лишь один или два аппаратных потока выполнения, так что работа переносимых программ не должна основываться на допущениях относительно количества имеющихся аппаратных потоков выполнения. С другой стороны, оптимальное количество потоков выполнения Java для заданной программы зависит от количества доступных аппаратных ядер!

Пулы потоков выполнения и их преимущества

Интерфейс `ExecutorService` языка Java предоставляет интерфейс для отправки задач на выполнение с получением результатов позднее. В ожидаемой реализации применяется пул потоков выполнения, который можно создать с помощью одного из фабричных методов, например `newFixedThreadPool`:

```
static ExecutorService newFixedThreadPool(int nThreads)
```

[1] Мы бы написали здесь «ядер», но в таких CPU, как Intel i7-6900K, на одно ядро приходится несколько аппаратных потоков выполнения, чтобы CPU мог выполнять полезные инструкции даже во время небольших задержек, например при отсутствии затребованных данных в кэше.

Этот метод создает `ExecutorService`, содержащий `n` потоков выполнения (часто называемых потоками-исполнителями (worker threads)), и сохраняет их в пуле потоков, из которого по принципу «кто первым поступил, тот первым и обслуживается» берутся свободные потоки для выполнения задач. По завершении выполнения задач потоки возвращаются в пул потоков. В итоге (и это замечательно) можно легко передать в пул на выполнение тысячи задач, при этом количество запущенных в данный момент задач будет соответствовать возможностям аппаратного обеспечения. Имеются различные возможности настройки, включая размер очереди, стратегию отклонения и различные приоритеты для разных задач.

Обратите внимание на терминологию: программист отправляет на выполнение *задачу* (task) (объект типа `Runnable` или `Callable`), которую выполняет *поток выполнения* (thread).

Пулы потоков выполнения и их недостатки

Пулы потоков выполнения практически всем отличаются в лучшую сторону от явной работы с потоками выполнения, но обратите внимание на два нюанса.

- Пул с k потоками выполнения способен конкурентно выполнять лишь k задач. Любые задачи, отправленные на выполнение сверх этого количества, попадают в очередь, и поток выполнения им будет выделен лишь по завершении одной из существующих задач. Такая ситуация обычно вполне допустима, поскольку позволяет отправлять на выполнение множество задач и не бояться случайно создать слишком большое число потоков выполнения. Но нужно остерегаться задач приостановленных либо ожидающих ввода/вывода или сетевого соединения. В контексте блокирующего ввода/вывода эти задачи только занимают потоки-исполнители, не выполняя во время ожидания никакой полезной работы. Попробуйте при четырех аппаратных потоках выполнения и пуле на пять потоков отправить в него 20 задач (рис. 15.3). Можно ожидать, что задачи будут выполняться параллельно, пока не закончатся все 20. Но представьте себе, что первые три из отправленных в пул задач приостановлены или ждут ввода/вывода. Тогда для оставшихся 15 задач доступны только два потока выполнения, так что пропускная способность окажется на половину меньше той, которую вы ожидали (и получили бы при пуле на восемь потоков выполнения). Возможно даже возникновение взаимных блокировок в пуле, если отправленные в него ранее или уже выполняемые задачи должны ждать поступления последующих задач — типичный сценарий для фьючерсов.

  Из этого можно сделать вывод, что необходимо избегать отправки в пул потенциально блокирующих задач, но в уже существующих системах это не всегда возможно.

- Java обычно ждет завершения всех потоков выполнения, прежде чем выполнить возврат из метода `main`, чтобы не прерывать выполнения потоков с потенциально жизненно важным кодом. Следовательно, на практике важно и рекомендуется завершать работу всех пулов перед выходом из программы (поскольку пото-

ки-исполнители этого пула будут созданы, но не завершены, ожидая отправки в пул очередной задачи). На практике часто применяется «долгоиграющий» `ExecutorService`, управляющий постоянно запущенным интернет-сервисом.

Для контроля такого поведения в Java предусмотрен метод `Thread.setDaemon`, который мы обсудим в следующем подразделе.

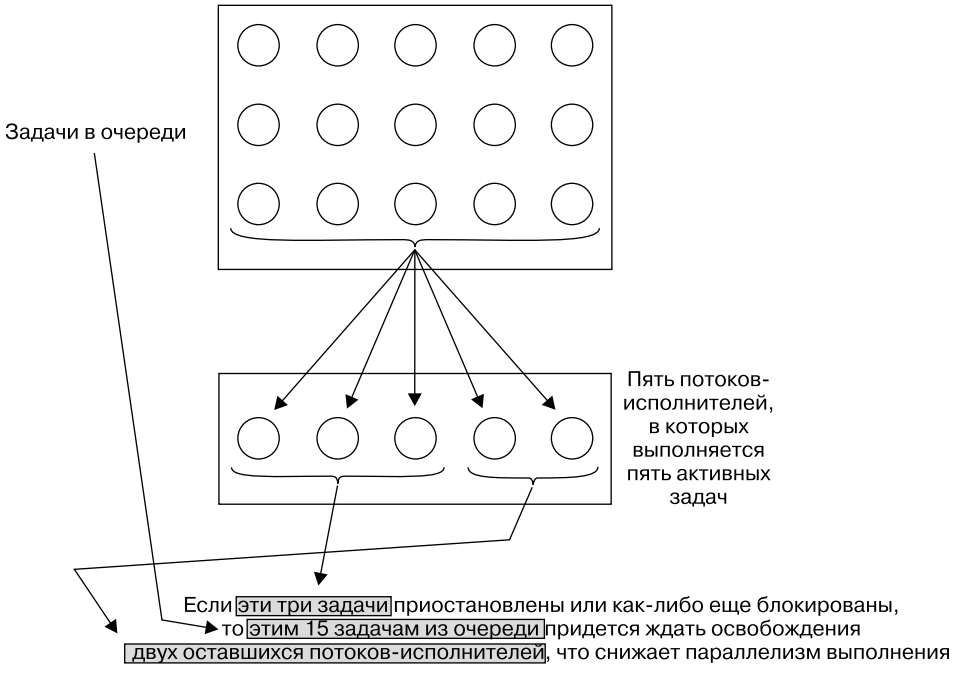


Рис. 15.3. Приостановленные задачи снижают пропускную способность пулов потоков выполнения

15.1.3. Другие абстракции потоков выполнения: (не) вложенность в вызовы методов

Чтобы пояснить суть отличий форм конкурентности из данной главы от тех, о которых говорилось в главе 7 (параллельная обработка потоков и фреймворк ветвления-объединения), нужно отметить одну особенность использовавшихся в главе 7 форм. А именно: при запуске любой задачи (или потока выполнения) изнутри вызова метода возврат из этого вызова возможен лишь по завершении этой задачи (потока). Другими словами, создание потока выполнения и соответствующая операция `join()` прекрасно укладываются в рамки вложенности вызовов методов (вызов/возврат). Эта идея — так называемое *строгое ветвление-объединение* (strict fork/join) — проиллюстрирована на рис. 15.4.

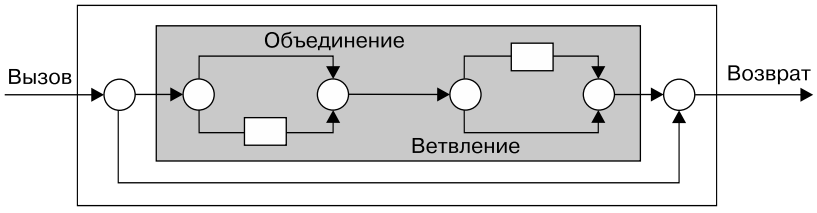


Рис. 15.4. Строгое ветвление-объединение. Стрелки соответствуют потокам выполнения, окружности — ветвлениям и объединениям, а прямоугольники отражают вызовы методов и возвраты из них

Можно без особой опаски воспользоваться более мягкой формой ветвления-объединения, при которой порождаемая задача выходит из рамок внутреннего вызова метода и объединение происходит во внешнем вызове, так что с точки зрения предоставляемого пользователю интерфейса вызов кажется обычным[1] (рис. 15.5).

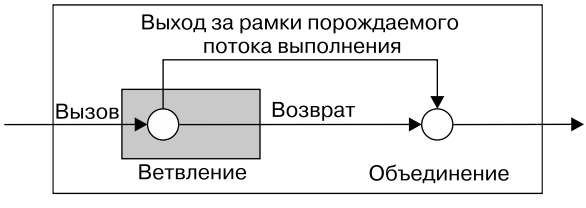


Рис. 15.5. Смягченный вариант ветвления-объединения

В этой главе мы сосредоточим наше внимание на более многообещающих формах конкурентности, в которых создаваемые при вызове пользователем метода потоки выполнения (или порождаемые задачи) могут просуществовать дольше самого вызова (рис. 15.6).

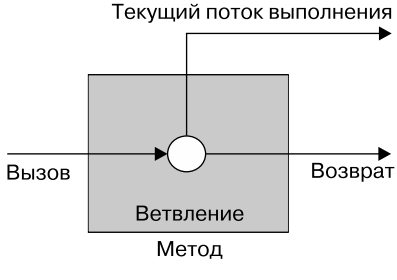


Рис. 15.6. Асинхронный метод

Подобные методы часто называют асинхронными, особенно когда текущая порожденная задача продолжает выполнять полезную для вызывающей стороны рабо-

[1] См. главу 18, в которой мы поговорим о возможности создания интерфейса без сторонних эффектов для метода с внутренними сторонними эффектами!

ту. Начиная с раздела 15.2, мы изучим, как в Java 8 и 9 извлечь выгоду из подобных методов, но предварительно обсудим возможные риски.

- Текущий поток выполнения работает конкурентно с кодом, следующим за вызовом метода, а значит, программировать его нужно осторожно, чтобы не привести к состоянию гонки по данным.
- Что будет, если возврат из метода `main()` произойдет до завершения текущего потока выполнения? Существует два возможных варианта, причем оба вряд ли вас устроят.
  - Прежде чем выйти из приложения, ждать завершения всех подобных невыполненных потоков.
  - Прервать выполнение всех невыполненных потоков, после чего выйти из приложения.

Первое из указанных решений несет риск того, что приложение никогда не завершится из-за забытого потока выполнения, — для пользователя это выглядит фатальным сбоем. При втором мы рискуем прервать последовательность операций записи на диск, из-за чего внешние данные могут остаться в несогласованном состоянии. Во избежание обеих проблем убедитесь, что в вашей программе отслеживаются все создаваемые потоки выполнения и перед выходом (включая завершение работы каких-либо пулов потоков выполнения) для них производится операция `join`.

С помощью вызова метода `setDaemon()` можно помечать потоки выполнения Java как *потоки-демоны* (daemon[1]) и прочие потоки. Выполнение потоков-демонов прерывается при выходе из программы (поэтому их удобно использовать для тех сервисов, которые не оставляют диск в несогласованном состоянии); но программа будет ожидать после возврата из метода `main()` завершения всех методов — не демонов.

15.1.4. Что нам нужно от потоков выполнения

Нам нужна возможность структурировать программу так, чтобы при необходимости воспользоваться параллелизмом в ней нашлось достаточно задач для всех доступных аппаратных потоков выполнения. Иначе говоря, следует обеспечить наличие в программе множества маленьких задач (но не слишком маленьких, учитывая затраты на переключение задач). В главе 7 мы показали вам, как делать это для циклов и алгоритмов типа «разделяй и властвуй», с помощью параллельной обработки потоков данных и ветвления-объединения. В оставшейся же части данной главы (а также главах 16 и 17) вы увидите, как сделать это в отношении вызовов методов без написания безумного количества стереотипного кода для работы с потоками выполнения.

---

[1] Этимологически английские слова daemon и demon происходят от одного греческого слова, но daemon означает преимущественно доброго духа, а demon — злого. В Unix применили слово daemon для вычислительных нужд, называя им системные сервисы, например, sshd — процесс (поток выполнения), который слушает на предмет входящих ssh-соединений (к сожалению, в русскоязычной компьютерной литературе этот нюанс традиционно игнорируется и для подобных сервисов используется слово «демон»). — *Примеч. пер.*

15.2. Синхронные и асинхронные API

Из главы 7 вы знаете, что благодаря Stream API Java 8 можно использовать возможности параллельного аппаратного обеспечения. Это происходит в два этапа. Во-первых, нужно заменить внешнюю итерацию (явные циклы `for`) внутренней (методы интерфейсов `Stream`). После этого можно применить к потоку данных метод `parallel()` для параллельной обработки его элементов с помощью библиотеки времени выполнения Java, а не переписывать все циклы и использовать сложные операции создания потоков выполнения. Дополнительное преимущество такого способа — программист может только гадать, сколько будет доступно потоков на момент выполнения цикла, а среда выполнения «знает» это наверняка.

Извлечь пользу из параллелизма можно и в отличных от циклов случаях. В основе этой главы и глав 16 и 17 лежит важное усовершенствование Java — асинхронные API.

Рассмотрим «живой» пример задачи, где суммируются результаты обращения к методам `f` и `g` со следующими сигнатурами:

```
int f(int x);
int g(int x);
```

Чтобы подчеркнуть, что эти методы возвращают результаты при фактическом возврате из них (в смысле, который скоро станет вам понятен), мы будем называть такие API *синхронными* (synchronous). Например, к такому API можно обратиться с помощью следующего фрагмента кода, который вызывает оба этих метода и выводит в консоль сумму возвращаемых ими результатов:

```
int y = f(x);
int z = g(x);
System.out.println(y + z);
```

Теперь представьте себе, что методы `f` и `g` выполняются весьма длительное время (например, могут решать задачу математической оптимизации, скажем градиентный спуск, но в главах 16 и 17 мы рассмотрим более реалистичные примеры интернет-запросов). Вообще говоря, компилятор Java не может никак оптимизировать такой код, поскольку методы `f` и `g` могут взаимодействовать неясными для компилятора способами. Но если вы точно знаете, что методы `f` и `g` не взаимодействуют между собой, или это взаимодействие вас не волнует, то имеет смысл выполнять их в различных ядрах CPU, в результате чего общее время выполнения будет равно не сумме времен выполнения методов `f` и `g`, а максимальному из них. Все, что нужно сделать, — запустить вызовы методов `f` и `g` в отдельных потоках выполнения. Это прекрасная идея, но она сильно усложняет[1] вышеприведенный простой код:

```
class ThreadExample {

    public static void main(String[] args) throws InterruptedException {
```

[1] Часть сложностей связана с переносом результатов обратно из потока выполнения. В лямбда-выражениях и внутренних классах могут использоваться только терминальные внешние объектные переменные, но основная проблема все равно состоит в явных операциях над потоками выполнения.

```
        int x = 1337;
        Result result = new Result();

        Thread t1 = new Thread(() -> { result.left = f(x); } );
        Thread t2 = new Thread(() -> { result.right = g(x); });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(result.left + result.right);
    }

    private static class Result {
        private int left;
        private int right;
    }
}
```

Этот код можно немного упростить за счет применения API класса `Future` вместо `Runnable`. Если предварительно создать пул потоков выполнения (типа `ExecutorService`), можно написать следующее:

```
public class ExecutorServiceExample {
    public static void main(String[] args)
        throws ExecutionException, InterruptedException {

        int x = 1337;
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        Future<Integer> y = executorService.submit(() -> f(x));
        Future<Integer> z = executorService.submit(() -> g(x));
        System.out.println(y.get() + z.get());

        executorService.shutdown();
    } }
```

но этот код все равно переполнен стереотипным кодом, включающим явные вызовы `submit`.

Нужно найти более оптимальный способ выражения этой идеи, подобно тому, как внутренняя итерация потоков данных позволяет избежать использования синтаксиса создания потоков выполнения, для распараллеливания внешней итерации.

Решение этой задачи включает переход к *асинхронному* API[1]. Вместо того чтобы метод возвращал результат в момент фактического возврата выполнения на вы-

[1] Синхронные API также называются блокирующими, поскольку фактический возврат результата откладывается до момента его готовности (наиболее ясно это видно на примере вызова операций ввода/вывода), в то время как асинхронные API позволяют реализовать естественным образом неблокирующий ввод/вывод (когда обращение к API просто запускает операцию ввода/вывода, без ожидания результата, при условии, что под рукой есть библиотека, например Netty, поддерживающая неблокирующие операции ввода/вывода).

зывающую сторону (синхронно), фактический возврат выполнения допускается до генерации результата, как показано на рис. 15.6. Таким образом, можно вызывать метод `f` и следующий за ним код (в данном случае метод `g`) параллельно. Для достижения такого параллелизма можно воспользоваться двумя способами, каждый из которых требует изменения сигнатур *обоих* методов `f` и `g`.

Первый способ заключается в более продвинутом использовании фьючерсов Java. Фьючерсы появились в Java 5 и получили развитие в классе `CompletableFuture`, обеспечившем возможности их сочетания. Мы расскажем об этом в разделе 15.4 и подробно изучим API Java на примере работающего кода в главе 16. Второй способ — перейти на программирование в реактивном стиле с помощью интерфейсов `java.util.concurrent.Flow` из Java 9, с применением протокола публикации/подписки, о котором мы расскажем вам в разделе 15.5 и работу которого продемонстрируем на практическом примере в главе 17.

Как же эти способы влияют на сигнатуры методов `f` и `g`?

15.2.1. Фьючерсный API

При этом варианте сигнатуры методов `f` и `g` необходимо изменить следующим образом:

```
Future<Integer> f(int x);
Future<Integer> g(int x);
```

а их вызовы — вот так:

```
Future<Integer> y = f(x);
Future<Integer> z = g(x);
System.out.println(y.get() + z.get());
```

Идея в том, что метод `f` возвращает фьючерс, содержащий задачу, которая продолжает вычислять исходное его тело, но возврат из `f` происходит максимально быстро после вызова. Метод `g` тоже возвращает фьючерс, а третья строка кода благодаря вызову `get()` ожидает завершения обоих фьючерсов и суммирует возвращаемые ими результаты.

В данном случае мы могли не менять API и вызов метода `g` без ущерба параллелизму — просто воспользоваться фьючерсом для метода `f`. Но в больших программах этого не стоит делать по двум причинам.

- Прочие варианты использования метода `g` могут потребовать фьючерсной его версии, так что единообразный стиль API предпочтительнее.
- Чтобы параллельное аппаратное обеспечение могло выполнять ваши программы как можно быстрее, лучше увеличить количество задач, одновременно уменьшив их размер (в разумных пределах).

15.2.2. Реактивный API

Основная идея второго варианта: воспользоваться программированием в стиле обратных вызовов, изменив сигнатуры методов `f` и `g` вот так:

```
void f(int x, IntConsumer dealWithResult);
```

Такой вариант поначалу может привести вас в недоумение. Как же метод `f` будет работать, если он не возвращает никакого значения? Разгадка в том, что вместо этого мы передаем в метод `f` функцию обратного вызова[1] (лямбда-выражение) в качестве дополнительного аргумента и внутри тела `f` порождается задача, вызывающая это лямбда-выражение с уже готовым результатом. То есть не нужно возвращать значение с помощью оператора `return`. Опять же возврат из метода `f` происходит сразу же после порождения задачи для вычисления его тела, что приводит к коду следующего вида:

```
public class CallbackStyleExample {
    public static void main(String[] args) {

        int x = 1337;
        Result result = new Result();

        f(x, (int y) -> {
            result.left = y;
            System.out.println((result.left + result.right));
        } );

        g(x, (int z) -> {
            result.right = z;
            System.out.println((result.left + result.right));
        });

    }
}
```

Ой, но это же не совсем то! Прежде чем вывести правильный результат (сумму результатов вызовов методов `f` и `g`), он выводит в консоль самое первое из полученных значений (причем иногда выводит сумму дважды, ведь никакой блокировки нет и оба операнда операции `+` могут оказаться обновленными до выполнения любого из вызовов `println`). Существует два ответа на это замечание.

- Можно добиться исходного поведения программы, перед вызовом `println` проверив с помощью оператора `if-then-else`, что обе функции обратного вызова уже были вызваны, например, путем подсчета их вызовов с помощью соответствующей блокировки.
- Подобный API в реактивном стиле предназначен для реагирования на последовательность событий, а не на отдельные результаты, для чего лучше подойдут фьючерсы.

Обратите внимание, что при таком реактивном стиле программирования методы `f` и `g` могут вызывать свою функцию обратного вызова `dealWithResult` по нескольку

[1] Некоторые авторы под термином «функция обратного вызова» (callback) подразумевают любое лямбда-выражение или ссылку на метод, передаваемые как аргумент в метод, например аргументы методов Stream.filter и Stream.map. Мы будем называть так только те лямбда-выражения и ссылки на методы, которые можно вызвать после возврата из метода.

раз. Исходным версиям методов `f` и `g` приходилось использовать оператор `return`, выполняемый однократно. Аналогично завершить выполнение фьючерса тоже можно лишь один раз, и его результат будет доступен методу `get()`. В определенном смысле асинхронные реактивные API открывают возможности для естественных последовательностей значений (которые мы далее сравним с потоками данных), в то время как фьючерсный API соответствует одноразовой структуре данных.

В разделе 15.5 мы вернемся к этому основополагающему примеру для моделирования вызова электронной таблицы, содержащей формулу вида `=C1+C2`.

Вы можете возразить, что оба варианта усложняют код. В некоторой степени это утверждение справедливо; ни один из этих вариантов API не следует использовать для всех методов подряд. Но при таких API код все равно оказывается проще (и содержит более высокоуровневые конструкции), чем при явных операциях с потоками выполнения. Кроме того, осмотрительное применение таких API для вызовов методов, которые а) приводят к «долгоиграющим» вычислениям (скажем, дольше нескольких миллисекунд) или б) ждут сетевого соединения или ввода данных пользователем, может существенно повысить эффективность работы приложения. В случае «а» они позволяют повысить быстродействие программы без явного повсеместного засорения программы потоками выполнения. В случае «б» дополнительное преимущество состоит в том, что базовая программа может эффективно использовать потоки выполнения без возникновения «заторов». Мы вернемся к последнему вопросу в следующем разделе.

15.2.3. Приостановка и другие блокирующие операции вредны

При взаимодействии с человеком или приложением, для которого важна ограниченность темпа происходящего, для программы будет естественным воспользоваться методом `sleep()`. Однако приостановленный поток выполнения все равно занимает ресурсы системы. Это не играет роли при наличии всего нескольких потоков выполнения, но важно, когда их много, причем большинство приостановлены (см. обсуждение в подразделе 15.2.1 и рис. 15.3).

Запомните, что приостановленные задачи в пуле потоков выполнения расходуют ресурсы, блокируя запуск других задач (остановить задачи, уже распределенные по потокам, они не могут, поскольку их выполнение планирует операционная система).

Приводить к заторам среди имеющихся потоков выполнения в пуле, конечно, может не только приостановка задач. Все блокирующие операции способны на это. Блокирующие операции делятся на две разновидности: ожидающие выполнения другой задачей каких-либо действий (пример: вызов метода `get()` для фьючерса) и ожидающие внешних взаимодействий, например чтения из сети, с сервера базы данных или устройств с интерфейсом пользователя (клавиатур).

Что можно сделать? Один, довольно авторитарный, метод — никогда не выполнять блокировку внутри задачи или по крайней мере делать это лишь в исключительных случаях (иногда все-таки приходится посмотреть правде в глаза, см. подраздел 15.2.4). Лучше будет разбить задачу на две части: «до» и «после» — и попросить Java запланировать выполнение второй из них на момент, когда ничего не будет блокироваться.

Сравните код А, реализованный в виде одной задачи:

```
work1();
Thread.sleep(10000); // Приостановка на 10 секунд
work2();
```

с кодом Б:

```
public class ScheduledExecutorServiceExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduledExecutorService
            = Executors.newScheduledThreadPool(1);

        work1();
        scheduledExecutorService.schedule(
    ScheduledExecutorServiceExample::work2, 10, TimeUnit.SECONDS);

        scheduledExecutorService.shutdown();
    }

    public static void work1(){
        System.out.println("Hello from Work1!");
    }

    public static void work2(){
        System.out.println("Hello from Work2!");
    }
}
```

Для work2() запланировано выполнение отдельной задачи через 10 секунд после завершения work1()

Представим себе, как обе эти задачи будут выполняться в пуле потоков выполнения.

Рассмотрим выполнение кода А. Сначала он попадает в очередь на выполнение в пуле потоков выполнения, а позднее начинает выполняться. Однако на полпути вызов блокируется из-за приостановки, и поток-исполнитель целых 10 секунд оказывается занят ничегонеделанием. Затем, перед завершением, выполняется `work2()` и поток-исполнитель освобождается. Код Б выполняет `work1()` и затем завершает выполнение — но только после того, как помещает в очередь задачу на выполнение `work2()` через 10 секунд.

Код Б лучше, но почему? Код А и код Б делают одно и то же. Различие в том, что код А во время приостановки занимает драгоценный поток выполнения, а код Б вместо приостановки отправляет в очередь на выполнение отдельную задачу (что не требует потока выполнения и занимает всего несколько байтов памяти).

Следует всегда учитывать этот эффект при создании задач. Задачи занимают ценные ресурсы после начала выполнения, так что нужно стараться, чтобы они активно выполнялись вплоть до окончания, а затем освобождать их ресурсы. Вместо блокировки задача должна завершать выполнение после отправки на выполнение дополнительной задачи, которая бы доделала нужную работу.

Там, где возможно, эти рекомендации относятся и к операциям ввода/вывода. Вместо традиционного блокирующего чтения задача должна инициировать не-

блокирующий вызов метода для начала чтения, а затем завершаться. При этом она отправит запрос библиотеке времени выполнения запланировать запуск дополнительной задачи по завершении чтения.

Может показаться, что этот паттерн проектирования приводит к большому объему трудночитаемого кода. Но интерфейс `CompletableFuture` языка Java (раздел 15.4 и глава 16) абстрагирует подобный код внутри библиотеки времени выполнения благодаря использованию комбинаторов вместо применения явных блокирующих операций `get()` к фьючерсам, как мы обсуждали ранее.

В качестве последнего замечания скажем, что в случае неограниченного количества потоков выполнения и их «дешевизны» в смысле ресурсов эффективность кода А и кода Б была бы одинаковой. Но это не так, так что код Б — оптимальный вариант в случае нескольких задач, которые потенциально могут быть приостановлены или иным образом блокировать выполнение.

15.2.4. Смотрим правде в глаза

Если вы хотите при разработке новой системы воспользоваться возможностями параллельного аппаратного обеспечения, вероятно, лучше всего проектировать ее в расчете на много маленьких конкурентных задач, чтобы все потенциально блокирующие операции были реализованы с помощью асинхронных вызовов. Но в принцип проектирования «все асинхронно» вмешивается жестокая действительность (помните: лучшее — враг хорошего). Начиная с версии 1.4 (2002 год), в Java были неблокирующие примитивы ввода/вывода (пакет `java.nio`), относительно запутанные и малоизвестные широким кругам разработчиков. Мы рекомендуем вам подходить к делу прагматично, выявлять ситуации, в которых могут оказаться полезны продвинутые API конкурентности, и использовать их, не пытаясь сделать все API асинхронными.

Полезно будет также взглянуть на более новые библиотеки, например Netty (https://netty.io/), — они обеспечивают единообразные блокирующие/неблокирующие API для сетевых серверов.

15.2.5. Исключения и асинхронные API

Как в основанном на фьючерсах, так и в реактивном варианте асинхронного API основное тело вызываемого метода выполняется в отдельном потоке, а выполнение вызывающего метода, вероятно, выйдет за область видимости всех охватывающих вызов обработчиков исключений. Очевидно, что при нестандартном поведении, которое может привести к исключению, должно выполняться какое-либо другое действие. Вопрос: какое? В реализации фьючерсов классом `CompletableFuture` API предоставляет возможность получения информации об исключении во время выполнения метода `get()`, а также обеспечивает восстановление после исключений с помощью таких методов, как `exceptionally()`, о котором мы поговорим в главе 16.

Для использования реактивных асинхронных API в интерфейс необходимо включить дополнительную функцию обратного вызова, вызываемую вместо генерации исключения, подобно тому, как уже существующая функция обратного вызова вы-

зывается вместо выполнения оператора `return`. Для этого включите в реактивный API несколько функций обратного вызова, как показано в следующем примере:

```
void f(int x, Consumer<Integer> dealWithResult,
              Consumer<Throwable> dealWithException);
```

Теперь тело `f` может выполнить такой оператор:

```
dealWithException(e);
```

Вместо того чтобы указывать несколько функций обратного вызова отдельно, можно сделать их методами одного объекта. В Flow API Java 9, например, несколько функций обратного вызова обертываются в один объект (интерфейса `Subscriber<T>`, содержащего четыре метода, рассматриваемых как функции обратного вызова). Вот три из них:

```
void    onComplete()
void    onError(Throwable throwable)
void    onNext(T item)
```

Существуют отдельные функции обратного вызова для индикации наличия очередного значения (`onNext`), генерации исключения при попытке получить значение (`onError`) и для указания, что больше не будет сгенерировано значений (и исключений) (`onComplete`). В предыдущем примере API для метода `f` мог бы выглядеть так:

```
void f(int x, Subscriber<Integer> s);
```

а метод `f` теперь мог бы сообщать про исключение, представленное объектом `Throwable t`, выполняя:

```
s.onError(t);
```

Сравните этот API, содержащий несколько функций обратного вызова, с чтением чисел из файла или при вводе с клавиатуры. Если рассматривать такое устройство в качестве генератора, а не просто пассивной структуры данных, можно сказать, что оно генерирует последовательность элементов «Число» или «Какой-то испорченный элемент вместо числа» и, наконец, уведомление «Больше символов не осталось (конец файла)».

Такие вызовы обычно называют сообщениями, или *событиями* (events). Например, говорят, что в результате чтения файла были сгенерированы числовые события 3, 7 и 42, за которыми следовало испорченное событие, после которого — числовое событие 2, а затем — событие конца файла.

Рассматривая эти события как часть API, важно понимать, что API ничего не говорит об относительной их упорядоченности (что часто называется *канальным протоколом*). На практике протокол описывается в сопутствующей документации фразами вида «После события `onComplete` никаких событий не генерируется».

15.3. Модель блоков и каналов

Зачастую удобнее всего проектировать и обдумывать конкурентные системы графически. Это методика с применением так называемой *модели блоков и каналов* (box-and-channel model). Рассмотрим простой пример с целыми числами, обобщающий

предыдущий пример вычисления `f(x) + g(x)`. Пусть нам теперь нужно вызвать метод/функцию `p` с аргументом `x`, передать результат функциям `q1` и `q2`, вызвать метод/функцию `r` с результатами этих двух вызовов в качестве аргументов, после чего вывести в консоль результат. Для упрощения пояснений мы не будем проводить различий между методом `m` класса `C` и соответствующей ему функцией `C::m`. Визуально задача проста, как показано на рис. 15.7.

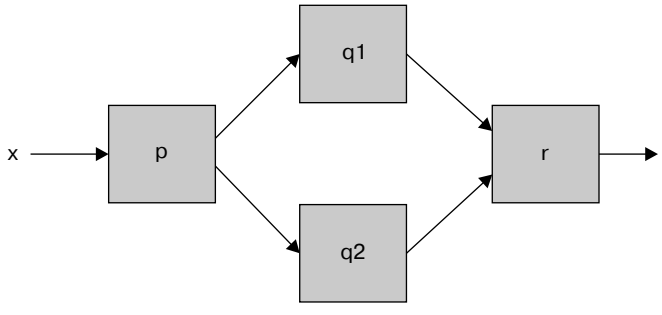


Рис. 15.7. Простая схема блоков и каналов

Рассмотрим два способа реализации схемы на рис. 15.7 в виде кода на языке Java и разберем возникающие при этом проблемы. Первый способ:

```
int t = p(x);
System.out.println( r(q1(t), q2(t)) );
```

Этот код выглядит простым, но Java выполняет вызовы `q1` и `q2` по очереди, чего мы как раз и хотим избежать при попытке использовать аппаратный параллелизм.

Еще один способ применения фьючерсов для параллельного вычисления `f` и `g`:

```
int t = p(x);
Future<Integer> a1 = executorService.submit(() -> q1(t));
Future<Integer> a2 = executorService.submit(() -> q2(t));
System.out.println( r(a1.get(),a2.get()));
```

Примечание: мы не обертываем `p` и `r` во фьючерсы в этом примере из-за формы схемы блоков и каналов. Метод/функцию `p` необходимо выполнить перед всем остальным, а метод/функцию `r` — после всего остального. Другое дело, если мы поменяем пример на вариант:

```
System.out.println( r(q1(t), q2(t)) + s(x) );
```

в котором для максимизации конкурентности нужно обернуть все пять функций (`p`, `q1`, `q2`, `r` и `s`) во фьючерсы.

Такое решение хорошо работает, если общий уровень конкурентности в системе невелик. Но что будет в большой системе с множеством отдельных схем блоков и каналов, причем если некоторые из блоков, в свою очередь, используют собственные блоки и каналы? В такой ситуации многие задачи могут ожидать (вызывая `get()`) завершения выполнения фьючерса, что, как обсуждалось в подразделе 15.1.2, может

привести к недостаточному использованию аппаратного параллелизма или даже взаимной блокировке. Более того, зачастую нелегко разобраться в структуре подобных масштабных систем настолько, чтобы понять, сколько задач окажутся в состоянии ожидания `get()`. В Java 8 для решения этой проблемы появились *комбинаторы* (combinators) (класс `CompletableFuture`; см. подробности в разделе 15.4). Вы уже видели, что можно применить к двум функциям методы `compose()` и `andThen()` для получения третьей функции (см. главу 3). Если, например, метод `add1` прибавляет 1 к целому числу, а метод `dble` увеличивает целое число в два раза, можно написать:

```
Function<Integer, Integer> myfun = add1.andThen(dble);
```

для создания объекта `Function`, который бы увеличивал свой аргумент вдвое и добавлял 2 к полученному результату. Но схемы блоков и каналов можно запрограммировать и напрямую, с использованием комбинаторов. Схему на рис. 15.7 можно лаконично реализовать с помощью функций `Function` `p`, `q1`, `q2` и `BiFunction` `r`:

```
p.thenBoth(q1,q2).thenCombine(r)
```

К сожалению, ни `thenBoth`, ни `thenCombine` не включены в классы `Function` и `BiFunction` именно в такой форме.

Из следующего раздела вы увидите, как комбинаторы применимы в отношении класса `CompletableFuture` и как они избавляют задачи от необходимости ждать, используя `get()`.

Прежде чем завершить данный раздел, мы хотели бы еще раз подчеркнуть: модель блоков и каналов помогает структурировать мысли и код. Она значительно повышает уровень абстракции при создании большой системы. Рисунки блоков (или комбинаторы в программе) выражают необходимые вычисления, которые в дальнейшем выполняются, причем, вероятно, эффективнее, чем при ручном их программировании. Применять комбинаторы подобным образом можно не только для математических функций, но и для фьючерсов и реактивных потоков данных. В разделе 17.3 мы обобщим эти схемы блоков и каналов и познакомим вас с «мраморными» диаграммами, в которых на каждом из каналов показаны разноцветные шарики (отражающие сообщения). Модель блоков и каналов также помогает взглянуть на ситуацию с другой стороны: дать возможность комбинаторам незаметно выполнять работу, вместо того чтобы реализовать логику конкурентности непосредственно. Это напоминает то, как потоки данных Java 8 позволили комбинаторам незаметно выполнять работу вместо написания программистом кода для явной итерации по данным.

15.4. Реализация конкурентности с помощью класса CompletableFuture и комбинаторов

Одна из проблем с интерфейсом `Future` — то, что он поощряет представление и структурирование задач конкурентного программирования в виде фьючерсов. Исторически, однако, фьючерсы были способны лишь на несколько действий, если не считать реализаций `FutureTask`: создание фьючерса для заданных вычислений, его запуск, ожидание его завершения и т. д. В последующих версиях Java их поддержка

стала более организованной (в качестве примера можно привести обсуждавшийся в главе 7 класс `RecursiveTask`).

Вклад Java 8 заключался в возможности композиции фьючерсов с помощью реализации `CompletableFuture` интерфейса `Future`. Так почему же эта реализация называется `CompletableFuture`, а не, скажем, `ComposableFuture`? Ну, обыкновенный фьючерс обычно создается на основе экземпляра `Callable`, который выполняется, а результат затем можно получить с помощью вызова `get()`. Класс `CompletableFuture` позволяет создать фьючерс, не передавая в него никакого кода для выполнения, а метод `complete()` дает возможность укомплектовать его значением (отсюда и название метода), к которому затем сможет обратиться метод `get()`. Для конкурентного суммирования `f(x)` и `g(x)` можно написать код:

```
public class CFComplete {

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;

        CompletableFuture<Integer> a = new CompletableFuture<>();
        executorService.submit(() -> a.complete(f(x)));

        int b = g(x);
        System.out.println(a.get() + b);

        executorService.shutdown();
    }
}
```

или:

```
public class CFComplete {

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;

        CompletableFuture<Integer> a = new CompletableFuture<>();
        executorService.submit(() -> b.complete(g(x)));
        int a = f(x);
        System.out.println(a + b.get());

        executorService.shutdown();
    }
}
```

Обратите внимание, что и тот и другой код могут впустую расходовать вычислительные ресурсы (см. подраздел 15.2.3) за счет блокировки потока выполнения в ожидании `get` — первый вариант в том случае, если дольше выполняется `f(x)`, а второй — если дольше выполняется `g(x)`. Благодаря классу `CompletableFuture` можно избежать подобной ситуации.

Теперь попробуйте выполнить контрольное задание.

Контрольное задание 15.1

Прежде чем продолжить чтение, задумайтесь, как бы вы написали задачи для оптимального использования фьючерсов при следующем сценарии: два потока выполнения активны во время выполнения `f(x)` и `g(x)` и один поток выполнения начинается после завершения (включая оператор `return`) первого.

Ответ: лучше использовать одну задачу для выполнения `f(x)`, вторую задачу — для выполнения `g(x)` и третью (новую или одну из уже существующих) — для вычисления их суммы, причем третья не должна запускаться до завершения первых двух. Как решить такую задачу на Java?

Решение состоит в применении идеи композиции фьючерсов.

Во-первых, освежим в вашей памяти операции композиции, с которыми вы уже встречались дважды в этой книге. Операции композиции — весьма многообещающий принцип структурирования программ. Он используется во множестве других языков программирования, но в Java заработал по-настоящему лишь после добавления в Java 8 лямбда-выражений. Один из примеров применения принципа композиции — композиция операций над потоком данных, как в следующем коде:

```
myStream.map(…).filter(…).sum()
```

Еще один пример: применение таких методов, как `compose()` и `andThen()`, к двум объектам типа `Function` для получения еще одного объекта `Function` (см. раздел 15.5).

Благодаря этому принципу появляется новый, более оптимальный способ сложения результатов двух вычислений — с помощью метода `thenCombine` из класса `CompletableFuture<T>`. Не задумывайтесь сейчас о нюансах; мы обсудим этот вопрос обстоятельнее в главе 16. Сигнатура метода `thenCombine` выглядит следующим образом (немного упрощенная, чтобы не загромождать код обобщенными типами и джокерными символами):

```
CompletableFuture<V> thenCombine(CompletableFuture<U> other,
                                 BiFunction<T, U, V> fn)
```

Этот метод принимает на входе два значения типа `CompletableFuture` (с типами результата `T` и `U`) и создает новое (с типом результата `V`). По завершении первых двух он применяет к обоим результатам `fn` и завершает выполнение полученного фьючерса без блокирования. Предыдущий код можно переписать в следующем виде:

```
public class CFCombine {

    public static void main(String[] args) throws ExecutionException,
      InterruptedException {

        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;
```

```
        CompletableFuture<Integer> a = new CompletableFuture<>();
        CompletableFuture<Integer> b = new CompletableFuture<>();
        CompletableFuture<Integer> c = a.thenCombine(b, (y, z)-> y + z);
        executorService.submit(() -> a.complete(f(x)));
        executorService.submit(() -> b.complete(g(x)));

        System.out.println(c.get());
        executorService.shutdown();
    }
}
```

Строка с `thenCombine` критически важна: без какой-либо информации о выполняемых во фьючерсах `a` и `b` вычислительных операциях она создает новую вычислительную операцию, запланированную на выполнение в пуле потоков по окончании обеих из первых двух. Осуществление в потоке выполнения третьей вычислительной операции, `c`, не разрешается до тех пор, пока не завершатся первые две (вместо запуска ее раньше с последующей блокировкой). В итоге нет никакой операции ожидания, вызывавшей проблемы в предыдущих двух версиях кода. В тех версиях, если реализуемое во фьючерсе вычисление завершалось вторым, два потока выполнения в пуле оставались активными, хотя нам нужен был только один! На рис. 15.8 эта ситуация проиллюстрирована схематически. В обеих ранних версиях вычисление `y+z` производилось в том же фиксированном потоке выполнения, который вычислял `f(x)` или `g(x)`, — возможно, после некоторого ожидания. Напротив, при использовании `thenCombine` выполнение операции суммирования запланировано по завершении вычисления обеих операций — `f(x)` и `g(x)`.

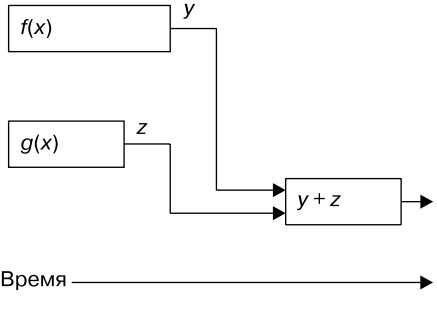


Рис. 15.8. Временная диаграмма с тремя операциями вычисления: f(x) и g(x) и суммирования их результатов

Для ясности уточним, что существует много кода, в котором нет смысла волноваться о нескольких заблокированных в ожидании `get()` потоках выполнения, так что задействовать фьючерсы из версий Java, предшествующих восьмой, — достаточно разумный вариант. В некоторых ситуациях, впрочем, может понадобиться большое количество фьючерсов (например, при обработке множества запросов к сервисам). Тогда наилучшим решением будет воспользоваться классом `CompletableFuture` и его комбинаторами во избежание блокирующих обращений к `get()` и потери из-за этого параллелизма или возникновения взаимных блокировок.

15.5. Публикация/подписка и реактивное программирование

Ментальная модель для интерфейса `Future` и класса `CompletableFuture` заключается в выполнении вычислений независимо друг от друга и конкурентно. Результат фьючерса можно получить с помощью `get()` по завершении вычисления. Таким образом, фьючерсы являются *одноразовыми* и выполняют код строго однократно.

Напротив, ментальная модель реактивного программирования представляет собой фьючерсоподобный объект, выдающий со временем несколько результатов. Рассмотрим два примера, начиная с объекта-термометра. Такой объект должен выдавать результаты многократно, передавая значение температуры каждые несколько секунд. Еще один пример: объект, который представляет собой компонент-прослушиватель веб-сервера. Он должен дожидаться появления в сети HTTP-запроса и передавать из него данные. После этого объекты термометр и прослушиватель возвращаются в состояние измерения температуры или прослушивания, прежде чем выдать дальнейшие результаты.

Отметим два нюанса. Главное: эти примеры аналогичны применению фьючерсов, но отличаются тем, что могут производить вычисления (выдавать результаты) несколько раз, а не один. Еще один нюанс: во втором примере предыдущие результаты могут иметь не меньшее значение, чем последующие, а от термометра большинству пользователей нужна только текущая температура. Но почему такой стиль программирования называется *реактивным*? Дело в том, что другая часть программы может захотеть *отреагировать* на сообщение о низкой температуре (например, включить обогреватель).

Наверное, вам кажется, что вышеприведенная идея описывает просто поток данных. Если ваша программа естественным образом укладывается в модель потока данных, возможно, он будет оптимальной реализацией. Однако, вообще говоря, парадигма реактивного программирования позволяет лучше выражать логику. Потреблять конкретный поток данных Java может только одна завершающая операция. Как мы упоминали в разделе 15.3, в парадигме потоков данных сложно выразить потокоподобные операции, в которых последовательность значений разбивается на два конвейера обработки (фактически ветвление) или происходит обработка с последующим соединением элементов из двух отдельных потоков данных (фактически объединение). Конвейеры обработки в потоках данных линейны.

Java 9 моделирует реактивное программирование с помощью интерфейсов из пакета `java.util.concurrent.Flow` и реализует так называемую модель (или протокол) публикации/подписки. Мы расскажем подробнее о Flow API Java 9 в главе 17, а здесь приведем лишь краткий обзор. Рассмотрим основные его понятия.

- *Издатель* (publisher), на которого может подписываться *подписчик* (subscriber).
- Соединение, называемое *подпиской* (subscription).
- *Сообщения* (messages), называемые также *событиями* (events), которые передаются через соединение.

Рисунок 15.9 наглядно иллюстрирует эту идею, подписки изображены на нем в виде каналов, а издатели и подписчики — в виде отверстий на блоках. На одного издателя может подписаться несколько компонентов, один компонент может публиковать несколько отдельных потоков данных, и один компонент может подписаться на несколько издателей. В следующем подразделе мы последовательно покажем вам, как это работает, в терминологии интерфейса `Flow` Java 9.

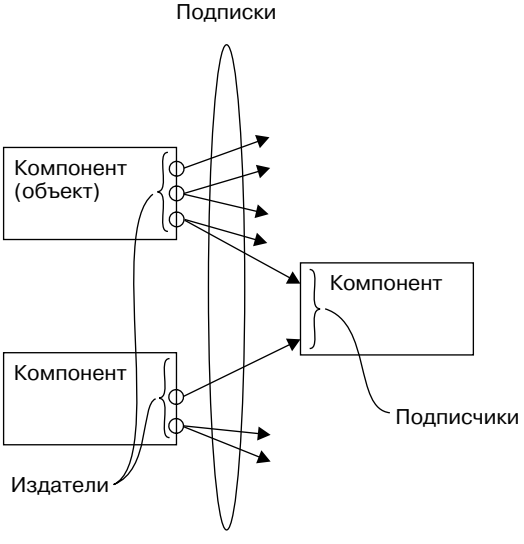


Рис. 15.9. Модель публикации/подписки

15.5.1. Пример использования для суммирования двух потоков сообщений

Простой, но показательный пример публикации/подписки — объединение событий из двух источников информации и публикация их для чтения пользователем. На первый взгляд этот процесс может показаться запутанным, но именно это и делает, по существу, ячейка с формулой в электронной таблице. Смоделируем ячейку электронной таблицы C3, содержащую формулу `"=C1+C2"`. При каждом обновлении ячейки `C1` или `C2` (либо пользователем, либо на основе содержащейся в ней еще одной формулы) `C3` также обновляется соответствующим образом. В следующем коде предполагается, что единственная доступная операция — сложение значений ячеек.

Во-первых, смоделируем понятие ячейки со значением:

```
private class SimpleCell {
    private int value = 0;
    private String name;

    public SimpleCell(String name) {
        this.name = name;
    }
}
```

Пока код прост, и мы можем инициализировать несколько ячеек, вот так:

```
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");
```

Но как указать, что при изменении значения ячейки `C1` или `C2` ячейка `C3` должна просуммировать эти два значения? Нам нужно придумать способ подписать `C3` на события ячеек `C1` и `C2`. Для этого понадобится интерфейс `Publisher<T>`, выглядящий, по существу, вот так:

```
interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```

Он принимает в качестве аргумента подписчика, подходящего для взаимодействия. Интерфейс `Subscriber<T>` включает простой метод `onNext`, принимающий эту информацию в виде аргумента, для которого потом можно создать конкретную реализацию:

```
interface Subscriber<T> {
    void onNext(T t);
}
```

Как свести воедино эти два понятия? Наверное, вы уже поняли, что ячейка фактически представляет собой как издателя (другие ячейки могут подписываться на ее события), так и подписчика (реагирует на события других ячеек). Реализация класса для ячейки получается вот такой:

```
private class SimpleCell implements Publisher<Integer>, Subscriber<Integer> {
    private int value = 0;
    private String name;
    private List<Subscriber> subscribers = new ArrayList<>();

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

    @Override
    public void subscribe(Subscriber<? super Integer> subscriber) {
        subscribers.add(subscriber);
    }

    private void notifyAllSubscribers() {
        subscribers.forEach(subscriber -> subscriber.onNext(this.value));
    }

    @Override
    public void onNext(Integer newValue) {
        this.value = newValue;
        System.out.println(this.name + ":" + this.value);
        notifyAllSubscribers();
    }
}
```

Этот метод оповещает всех подписчиков о новом значении

Реагирует на новое значение ячейки, на которую подписан, обновляя свое значение

Выводит в консоль значение, но потенциально может и визуализировать обновленную ячейку в пользовательском интерфейсе

Оповещает всех подписчиков о новом значении

Попробуем простой пример:

```
Simplecell c3 = new SimpleCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3);

c1.onNext(10); // Обновляет значение ячейки C1, делая его равным 10
c2.onNext(20); // Обновляет значение ячейки C2, делая его равным 20
```

Поскольку ячейка `C3` напрямую подписана на `C1`, этот код выведет в консоль следующее:

```
C1:10
C3:10
C2:20
```

Как теперь реализовать поведение `"C3=C1+C2"`? Нам нужен отдельный класс для хранения двух операндов арифметической операции (правого и левого):

```
public class ArithmeticCell extends SimpleCell {

    private int left;
    private int right;

    public ArithmeticCell(String name) {
        super(name);
    }

    public void setLeft(int left) {
        this.left = left;
        onNext(left + this.right);
    }

    public void setRight(int right) {
        this.right = right;
        onNext(right + this.left);
    }
}
```

Теперь можно попробовать более реалистичный пример:

```
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c1.onNext(10); // Обновляет значение ячейки C1, делая его равным 10
c2.onNext(20); // Обновляет значение ячейки C2, делая его равным 20
c1.onNext(15); // Обновляет значение ячейки C1, делая его равным 15
```

Результаты выполнения:

```
C1:10
C3:10
C2:20
C3:30
C1:15
C3:35
```

Если взглянуть на выведенные результаты, можно заметить, что при изменении значения ячейки `C1` на `15`, ячейка `C3` сразу реагирует и ее значение обновляется. Формат взаимодействия «издатель — подписчик» удобен возможностью нарисовать граф издателей и подписчиков. Например, создадим еще одну ячейку, `C5`, зависящую от `C3` и `C4` (`"C5=C3+C4"`):

```
ArithmeticCell c5 = new ArithmeticCell("C5");
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c4 = new SimpleCell("C4");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c3.subscribe(c5::setLeft);
c4.subscribe(c5::setRight);
```

Теперь можно производить различные обновления нашей электронной таблицы:

```
c1.onNext(10); // обновляет значение ячейки C1, делая его равным 10
c2.onNext(20); // обновляет значение ячейки C2, делая его равным 20
c1.onNext(15); // обновляет значение ячейки C1, делая его равным 15
c4.onNext(1); // обновляет значение ячейки C4, делая его равным 1
c4.onNext(3); // обновляет значение ячейки C4, делая его равным 3
```

Эти действия приводят к следующему выводу в консоль:

```
C1:10
C3:10
C5:10
C2:20
C3:30
C5:30
C1:15
C3:35
C5:35
C4:1
C5:36
C4:3
C5:38
```

В конце значение ячейки `C5` равно `38`, поскольку `C1` равно `15`, `C2` равно `20`, а `C4` равно `3`.

Терминология

Поскольку данные движутся в направлении от издателя (генератора) к подписчику (потребителю), разработчики часто используют такие термины, как *«ближе по конвейеру»* (upstream) и *«далее по конвейеру»* (downstream). В предыдущих примерах кода данные `newValue`, полученные расположенными ранее по конвейеру методами `onNext()`, передаются с помощью вызова `notifyAllSubscribers()` в расположенный далее по конвейеру вызов `onNext()`.

Такова основная идея публикации/подписки. Мы опустили несколько нюансов, часть из которых служит просто для украшения кода, но один (противодавление) настолько важен, что мы обсудим его отдельно в следующем разделе.

Прежде всего разберем самое простое. Как мы отметили в разделе 15.2, на практике при программировании потоков сообщений может потребоваться оповещать не только о событиях `onNext`, так что подписчики (прослушиватели) должны определить методы `onError` и `onComplete`, чтобы издатель мог сигнализировать об исключениях и завершении потоков движения данных. Например, если термометр был заменен и больше никаких значений через `onNext` не сгенерирует. Фактический интерфейс `Subscriber` из Flow API Java 9 поддерживает методы `onError` и `onComplete`. В частности, благодаря им возможности протокола публикации/подписки шире, чем у обычного паттерна «Наблюдатель».

Две простые, но жизненно важные идеи, существенно усложняющие интерфейсы потоков сообщений/событий, — давление и противодавление потока сообщений. Может показаться, что это мелочи, но на самом деле они критически важны для обеспечения загруженности потоков выполнения. Предположим, что наш термометр, ранее сообщавший температуру каждые несколько секунд, заменили новым, лучшим, сообщающим температуру каждую миллисекунду. Сможет ли программа реагировать на эти события достаточно быстро, не произойдет ли переполнение какого-либо буфера и не будет ли фатального сбоя программы (вспомните, какие проблемы возникают в пулах потоков выполнения при большом количестве задач в случае блокировок более чем в нескольких задачах)? Представьте также, что вы подписались на издателя, отвечающего за доставку всех СМС в ваш телефон. Такая подписка будет отлично работать на новеньком телефоне, в котором пока лишь несколько СМС, но что будет через несколько лет, когда число сообщений достигнет тысяч, причем все они могут потенциально быть отправленными через вызовы `onNext` менее чем за секунду. Такую ситуацию часто называют *давлением* потока сообщений (pressure).

Теперь представьте себе вертикальную трубу, содержащую записанные на шары сообщения. Необходим какой-либо вид, говоря техническим языком, *противодавления* (backpressure), например механизм ограничения числа добавляемых в эту колонну шаров. Противодавление реализовано в Flow API Java 9 с помощью метода `request()` (в новом интерфейсе `Subscription`), который запрашивает у издателя отправку очередного (-ых) элемента (-ов), вместо отправки элементов с неограниченной частотой (модель извлечения вместо модели проталкивания элементов). Мы обратимся к этому вопросу в следующем подразделе.

15.5.2. Противодавление

Мы показали вам, как передать объекту `Publisher` объект типа `Subscriber` (содержащий методы `onNext`, `onError` и `OnComplete`), который издатель может при необходимости вызывать. Этот объект передает информацию от издателя подписчику. Нам нужно ограничить темпы отправки этой информации посредством противодавления (управления потоком событий), что требует отправки информации от подписчика (`Subscriber`) издателю (`Publisher`). Проблема в том, что у издателя может быть несколько подписчиков, а противодавление должно влиять только на конкретное соединение между двумя объектами. В Flow API Java 9 интерфейс `Subscriber` содержит четвертый метод:

```
void onSubscribe(Subscription subscription);
```

вызываемый при отправке первого события по каналу, организованному между издателем и подписчиком. Объект `Subscription` содержит методы, с помощью которых подписчик может взаимодействовать с издателем:

```
interface Subscription {
    void   cancel();
    void   request(long n);
}
```

Обратите внимание на обычную для функций обратного вызова видимость происходящего «задом наперед». Издатель создает подписку и передает ее подписчику, который может далее вызывать ее методы для передачи информации от подписчика обратно издателю.

15.5.3. Простой вариант реализации противодавления на практике

Чтобы соединение публикации/подписки могло обрабатывать события по одному, необходимо произвести следующие изменения.

- Обеспечить локальное хранение в `Subscriber` передаваемого через метод `OnSubscribe` объекта `Subscription`, например, в поле `subscription`.
- Сделать так, чтобы последним действием методов `onSubscribe`, `onNext` и, возможно, `onError` был запрос следующего события (только одного события, чтобы не перегрузить подписчика) с помощью вызова `channel.request(1)`.
- Изменить интерфейс `Publisher` так, чтобы (в данном примере) метод `notifyAllSubscribers` отправлял событие `onNext` или `onError` только по тем каналам, по которым пришел запрос. Обычно издатель создает новый объект `Subscription` для каждого подписчика, чтобы каждый из них мог обрабатывать данные с подходящей ему скоростью.

Хотя этот процесс выглядит очень просто, для реализации противодавления нужно определиться с несколькими нюансами.

- Отправлять ли события нескольким подписчикам с минимальной (среди них) скоростью или поддерживать отдельную очередь пока еще не отправленных данных для каждого подписчика?

- Что произойдет, если размер этих очередей увеличится слишком сильно?
- Игнорировать ли событие, если подписчик не готов его получить?

Выбор зависит от сущности отправляемых данных. Потеря одного показания температуры из последовательности вряд ли существенна, а вот потеря денег на банковском счету — еще как!

Описанную идею часто называют реактивным противодавлением на основе извлечения событий (reactive pull-based backpressure). *Реактивным* и *основанным на извлечении* — потому, что дает возможность подписчику извлекать (запрашивать) из издателя дополнительную информацию посредством событий (реактивность). В результате получается механизм противодавления.

15.6. Реактивные системы и реактивное программирование

В среде разработчиков и специалистов по компьютерным наукам все чаще можно услышать термины «реактивные системы» и «реактивное программирование», так что важно понимать, что они выражают разные идеи.

Реактивная система (reactive system) — это программа, архитектура которой позволяет ей реагировать на изменения среды, где она выполняется. Необходимые для реактивных систем свойства описаны в Манифесте реактивности (http://www.reactivemanifesto.org/ru) (см. главу 17). Три из этих свойств: быстрота реакции (responsiveness), отказоустойчивость (resiliency) и адаптивность (elasticity).

Быстрота реакции (responsiveness) означает, что реактивная система способна реагировать на вводимые данные в режиме реального времени, а не откладывать выполнение простого запроса по причине занятости большим заданием для другого пользователя. *Отказоустойчивая* (resilient) система обычно не завершает аварийно работу из-за сбоя одного компонента; нерабочая сетевая ссылка не должна влиять на запросы, в которых она не используется, а запросы к компоненту, который не реагирует, можно перенаправить другому компоненту. *Адаптивность* (elasticity) системы означает способность ее приспосабливаться к изменениям нагрузки и продолжать эффективно работать. Подобно тому, как можно динамически перераспределить официантов в баре между разносом напитков и еды, чтобы клиенты ждали в обоих случаях примерно одинаковое время, можно подстроить и число потоков-исполнителей для различных программных сервисов, чтобы ни один из них не простаивал, а все очереди продолжали обрабатываться.

Разумеется, добиться таких свойств можно по-разному, но основной подход — *реактивный* стиль *программирования*, который в языке Java основан на интерфейсах из пакета `java.util.concurrent.Flow`. Архитектура этих интерфейсов отражает четвертое, последнее из свойств, упомянутых в Манифесте реактивности: ориентированность на обмен сообщениями. Внутренние API *ориентированных на обмен сообщениями* (message-driven) систем основываются на модели блоков и каналов, компоненты в них ожидают входных данных, которые затем обрабатываются, а результаты отправляются в виде сообщений другим компонентам, благодаря чему система становится быстрореагирующей.

Резюме

- Поддержка конкурентности в Java существенно выросла и продолжает развиваться. Пулы потоков выполнения — полезная возможность, но они могут вызывать проблемы в случае большого числа потенциально блокирующих задач.
- Асинхронность методов (возврат из них до завершения выполнения всей работы) обеспечивает дополнительные возможности параллелизма, помимо параллелизма, используемого в целях оптимизации циклов.
- Для визуализации асинхронных систем можно использовать модель блоков и каналов.
- И класс `CompletableFuture` Java 8, и Flow API Java 9 могут реализовывать диаграммы блоков и каналов.
- Класс `CompletableFuture` служит для выражения однократных асинхронных вычислений. Для композиции асинхронных вычислений без риска блокировки (присущего традиционным способам применения фьючерсов) можно воспользоваться комбинаторами.
- В основе Flow API лежит протокол публикации/подписки, включающий противодавление. Он формирует фундамент реактивного программирования в Java.
- С помощью реактивного программирования можно реализовывать реактивные системы.

16 Класс CompletableFuture: композиция асинхронных компонентов

В этой главе

- Создание асинхронных вычислительных операций и извлечение их результатов.
- Повышение пропускной способности за счет использования неблокирующих операций.
- Проектирование и реализация асинхронного API.
- Асинхронное потребление синхронного API.
- Конвейерная обработка и слияние двух или более асинхронных операций.
- Как реагировать на завершение асинхронной операции.

В главе 15 мы обсуждали современный контекст конкурентности: доступность нескольких обрабатывающих ресурсов (ядер процессоров и т. п.), из которых программы должны извлекать максимальную выгоду, причем с помощью высокоуровневых операций (вместо загромождения программ плохо структурированными и неудобными для сопровождения операциями с потоками выполнения). Мы отметили, что параллельные потоки данных и параллелизм ветвления-объединения дают возможность выражать параллелизм языковыми конструкциями более высокого уровня в программах, обходящих в цикле коллекции, и в тех, где используется методика «разделяй и властвуй». Но дополнительные возможности параллельного выполнения кода предлагают и вызовы методов. В Java 8 и 9 появились два API специально для

этой цели: `CompletableFuture` и парадигма реактивного программирования. В этой главе мы на практических примерах кода объясним, какие новые инструменты появляются в арсенале разработчика благодаря реализации `CompletableFuture` интерфейса `Future` Java 8. Кроме того, обсудим дополнения, появившиеся в Java 9.

16.1. Простые применения фьючерсов

Интерфейс `Future` появился в Java 5 для моделирования результата, который будет доступен в какой-то момент будущего. Например, результат запроса к удаленному сервису не становится доступным сразу же. Интерфейс `Future` моделирует асинхронную операцию вычисления и позволяет ссылаться на результат, который окажется доступен по ее завершении. Запуск потенциально длительного действия внутри фьючерса дает возможность вызывающему потоку продолжать выполнять полезные действия, а не ожидать результата данной операции. Можно провести аналогию этого процесса со сдачей вещей в химчистку. Химчистка выдает чек, по которому вы сможете узнать, когда вещи будут готовы (фьючерс), а в промежутке вы можете заниматься другими делами. Еще одно преимущество фьючерсов — работать с ними удобнее, чем с более низкоуровневыми потоками выполнения. Для работы с фьючерсом обычно достаточно обернуть «долгоиграющую» операцию в объект `Callable` и передать ее на выполнение `ExecutorService`. В листинге 16.1 приведен пример кода, написанного до Java 8.

Листинг 16.1. Выполнение «долгоиграющей» операции асинхронно во фьючерсе

Передача задачи на выполнение объекту ExecutorService

Создание объекта ExecutorService для передачи задач на выполнение в пул потоков

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<Double> future = executor.submit(new Callable<Double>() {
        public Double call() {
            return doSomeLongComputation();
        }});
doSomethingElse();
try {
    Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
    // Вычислительная операция вернула исключение
} catch (InterruptedException ie) {
    // Выполнение текущего потока было прервано во время ожидания
} catch (TimeoutException te) {
    // Время ожидания истекло до завершения выполнения фьючерса
}
```

Выполняем долгоиграющую операцию асинхронно в отдельном потоке выполнения

Извлекаем результат асинхронной операции, с блокировкой, если он еще недоступен, но ждем не более 1 секунды

Выполняем что-то другое во время асинхронной операции

Как показано на рис. 16.1, благодаря такому стилю программирования можно выполнять другие задачи во время конкурентного выполнения «долгоиграющей» операции в отдельном потоке, предоставленном `ExecutorService`. А затем, когда уже не осталось никакой работы, которую можно сделать без результатов этой асин-

хронной операции, извлечь их из фьючерса, вызвав его метод `get`. Он немедленно возвращает результат операции, если она уже завершена, или блокирует поток выполнения в ожидании появления результатов.

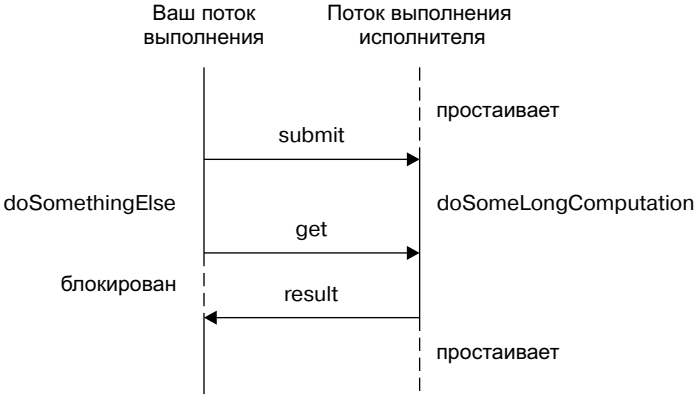


Рис. 16.1. Применение фьючерсов для асинхронного выполнения «долгоиграющих» операций

Обратите внимание на проблему с этим сценарием. Что будет, если «долгоиграющая» операция никогда не вернет результат? В таком случае лучше воспользоваться версией метода `get` с двумя аргументами, принимающего на входе время ожидания (вместе с единицей времени) — максимальное время, которое ваш поток выполнения готов ждать результата фьючерса (подобно тому, как показано в листинге 16.1). Версия метода `get` без аргументов будет ждать результата бесконечно.

16.1.1. Разбираемся в работе фьючерсов и их ограничениях

Этот первый маленький пример демонстрирует, что в интерфейсе `Future` есть методы для проверки того, завершилась ли асинхронная операция (метод `isDone`), а также проверки ожидания ее завершения и извлечения ее результата. Но этих возможностей недостаточно для написания лаконичного конкурентного кода. Например, выражать зависимости между результатами фьючерсов достаточно непросто. Описательно легко сформулировать: «Когда результат этой “долгоиграющей” вычислительной операции будет доступен, отправьте его в другую “долгоиграющую” вычислительную операцию, а когда она завершится, объедините ее результат с результатом еще одного запроса». Совсем другой вопрос — как реализовать такую инструкцию с помощью имеющихся в интерфейсе `Future` операций. Именно поэтому в реализации не помешают дополнительные декларативные возможности, в том числе следующие.

- Комбинация двух асинхронных вычислительных операций, как независимых, так и таких, когда вторая зависит от результатов первой.
- Ожидание завершения всех выполняемых набором фьючерсов задач.
- Ожидание завершения лишь самой быстрой задачи в наборе фьючерсов (может пригодиться, например, когда фьючерсы пытаются вычислить одно и то же значение различными способами) и извлечение ее результата.

- Завершение выполнения фьючерса программным способом (то есть путем задания вручную результата асинхронной операции).
- Реагирование на завершение выполнения фьючерса (то есть уведомление о завершении с возможностью выполнения дальнейших действий с его результатом вместо блокировки в ожидании результата).

Далее в главе мы расскажем, как стало возможным делать все это декларативно благодаря классу `CompletableFuture` (реализующему интерфейс `Future`). Архитектуры потоков данных и завершаемых фьючерсов следуют одному паттерну, поскольку оба применяют лямбда-выражения и используют конвейерную обработку. Поэтому можно сказать, что завершаемый фьючерс по отношению к обычному фьючерсу — то же, что и поток данных по отношению к коллекции.

16.1.2. Построение асинхронного приложения с помощью завершаемых фьючерсов

Для изучения возможностей класса `CompletableFuture` мы поэтапно разработаем в этом подразделе приложение для поиска самой низкой цены заданного товара или сервиса в нескольких интернет-магазинах. Попутно вы получите несколько важных навыков.

- Научитесь предоставлять клиентам асинхронный API (полезно для владельцев интернет-магазинов).
- Делать ваш код, как потребитель синхронного API, неблокирующим. Вы узнаете, как создать конвейер из двух последовательных асинхронных операций, объединив их в одну асинхронную вычислительную операцию. Это может понадобиться, например, когда интернет-магазин вместе с исходной ценой желаемого товара выдает скидочный код. Приходится обращаться к стороннему сервису скидок для выяснения процентного значения скидки из данного кода, чтобы вычислить фактическую стоимость товара.
- Реактивно обрабатывать события завершения асинхронных операций и с помощью этого постоянно обновлять наиболее выгодную цену для желаемого товара в приложении по мере возврата каждым интернет-магазином цены, а не ждать информацию о ценах от всех магазинов. Этот навык позволяет избежать ситуации, в которой пользователь постоянно видит белый экран, так как сервер одного из интернет-магазинов не работает.

Синхронные и асинхронные API

Выражение *«синхронный API»* (synchronous API) — просто синоним для традиционного вызова метода: метод вызывается, вызывающая сторона ожидает его выполнения, метод возвращает результат и вызывающая сторона продолжает работу на основе полученного значения. Даже если вызывающая и вызываемая стороны выполняются в различных потоках выполнения, вызывающая сторона все равно ждет завершения выполнения вызываемой. Такая ситуация стала причиной возникновения термина *«блокирующий вызов»* (blocking call). Напротив, в асинхронном API возврат из метода происходит сразу же

(или по крайней мере до завершения его вычисления), а оставшиеся вычисления поручаются работающему асинхронно с вызывающей стороной потоку выполнения — отсюда название *«неблокирующий вызов»* (nonblocking call). Оставшиеся вычисления передают значение вызывающей стороне с помощью обратного вызова или вызова еще одного метода типа «ждать, пока вычисления не будут завершены». Подобный стиль вычислений часто применяется при программировании систем ввода/вывода: инициируется обращение к диску, происходящее асинхронно с другими вычислениями, а когда никакой полезной работы, которую нужно сделать, не остается, происходит ожидание загрузки дисковых блоков в оперативную память. Отметим, что терминами «блокирующий» и «неблокирующий» обычно оперируют применительно к конкретным реализациям ввода/вывода операционной системой. Однако эти термины часто используют для синхронных и асинхронных операций даже вне контекста ввода/вывода.

16.2. Реализация асинхронного API

Начнем реализацию приложения для поиска наилучшей цены с описания API, который должен предоставлять каждый из интернет-магазинов. Прежде всего интернет-магазин объявляет метод, возвращающий цену товара по его названию:

```
public class Shop {
    public double getPrice(String product) {
        // будущая реализация
    }
}
```

Внутри реализации этого метода может выполняться запрос к базе данных магазина, а также, вероятно, другие занимающие много времени задачи, например обращение к внешним сервисам (поставщикам этого магазина или сервисам рекламных скидок от производителей товаров). Для имитации подобного «долгоиграющего» выполнения метода далее в главе мы будем использовать метод `delay`, добавляющий искусственную задержку на 1 секунду, как описано в листинге 16.2.

Листинг 16.2. Метод для имитации секундной задержки

```
public static void delay() {
    try {
        Thread.sleep(1000L);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Для задач этой главы можно моделировать метод `getPrice` путем вызова метода `delay` с последующим возвратом случайного значения цены, как показано в листинге 16.3. Код, возвращающий сгенерированную случайную цену, выглядит «косты-

лем»; элемент случайности вносится в цену на основе числовых значений первых букв (получаемых с помощью `charAt`) названия товара.

Листинг 16.3. Включаем имитацию секундной задержки в метод getPrice

```
public double getPrice(String product) {
    return calculatePrice(product);
}
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

Из этого кода следует, что, когда потребитель этого API (в данном случае приложение поиска наилучшей цены) вызывает данный метод, он блокируется, а затем бездействует на протяжении 1 секунды, ожидая синхронного завершения метода. Это неприемлемо, особенно учитывая то, что приложение поиска наилучшей цены должно повторять такую операцию для всех магазинов в сети. В последующих разделах данной главы мы покажем вам, как решить эту проблему с помощью потребления синхронного API асинхронно. Но пока, чтобы научиться проектировать асинхронные API, вы продолжите притворяться, что находитесь с другой стороны баррикады. Вы будете играть роль умудренного опытом владельца магазина, который отдает себе отчет в том, насколько этот синхронный API неудобен его пользователям, и хочет переписать его в асинхронном виде, чтобы облегчить их жизнь.

16.2.1. Преобразование синхронного метода в асинхронный

Для достижения этой цели необходимо сначала преобразовать метод `getPrice` в `getPriceAsync` и изменить тип возвращаемого им значения, вот так:

```
public Future<Double> getPriceAsync(String product) { ... }
```

Как мы уже упоминали в начале главы, интерфейс `java.util.concurrent.Future` появился в Java 5 и был предназначен для представления результата асинхронной вычислительной операции (то есть такой, что вызывающий поток выполнения может работать без блокировки). Фьючерс — это своего рода дескриптор для пока еще недоступного значения, которое тем не менее можно будет извлечь с помощью вызова метода `get` после окончания его вычисления. В результате возможен немедленный возврат из метода `getPriceAsync`, благодаря чему у вызывающего потока выполнения появляется возможность производить в промежутке другие полезные вычисления. Класс `CompletableFuture` Java 8 предоставляет множество возможностей простой реализации этого метода, как показано в листинге 16.4.

Листинг 16.4. Реализация метода getPriceAsync

Создание объекта CompletableFuture — контейнера для результата вычисления

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
```

```
        new Thread( () -> {
            double price = calculatePrice(product);
            futurePrice.complete(price);
        }).start();
        return futurePrice;
    }
```

Задаем для фьючерса значение, возвращенное «долгоиграющей» операцией, когда оно станет доступно

Возвращаем фьючерс, не ожидая завершения вычисления содержащегося в нем результата

Производим вычисления асинхронно в другом потоке выполнения

Здесь мы создали экземпляр `CompletableFuture`, представляющий асинхронную вычислительную операцию и содержащий ее результат (когда он становится доступен). Далее мы производим ответвление отдельного потока выполнения, который и вычислит цену, и возвращаем экземпляр `Future`, не ожидая окончания этого «долгоиграющего» вычисления. Когда цена интересующего нас товара наконец-то становится доступной, можно завершить выполнение `CompletableFuture`, воспользовавшись его методом `complete` для задания значения. Эта его функциональная возможность и объясняет название данной реализации интерфейса `Future` в Java 8. Клиент API может его вызвать так, как показано в листинге 16.5.

Листинг 16.5. Использование асинхронного API

```
Shop shop = new Shop("BestShop");
long start = System.nanoTime();
Future<Double> futurePrice = shop.getPriceAsync("my favorite product");
long invocationTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Invocation returned after " + invocationTime
                                                 + " msecs");
// Выполняем дополнительные задачи, например запросы к другим магазинам,
doSomethingElse();
// пока вычисляется цена нужного товара
try {
    double price = futurePrice.get();
    System.out.printf("Price is %.2f%n", price);
} catch (Exception e) {
    throw new RuntimeException(e);
}
long retrievalTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Price returned after " + retrievalTime + " msecs");
```

Запрашиваем у магазина цену товара

Читаем цену из фьючерса или блокируем, дожидаясь, пока она не окажется доступной

Как видите, клиент запрашивает у магазина цену определенного товара. Поскольку магазин предоставляет асинхронный API, этот вызов практически немедленно возвращает фьючерс, через который клиент может позднее извлечь цену. Далее клиент может выполнять другие задачи, например запрашивать информацию у других магазинов, вместо того чтобы находиться в состоянии блокировки и ждать результата от первого магазина. Позднее, когда у клиента уже не останется работы, которую можно выполнить без этой цены товара, он сможет вызвать метод `get` фьючерса. Таким образом, клиент распакует содержащееся во фьючерсе значение (если выполнение асинхронной задачи уже завершено) или будет блокирован вплоть до момента, когда появится значение. Выводимые кодом из листинга 16.5 результаты выглядят примерно следующим образом:

```
Invocation returned after 43 msecs
Price is 123.26
Price returned after 1045 msecs
```

Как видите, возврат из вызова метода `getPriceAsync` происходит гораздо быстрее, чем завершается вычисление цены. Из раздела 16.4 вы узнаете, что у клиента есть возможность избежать риска каких-либо блокировок. Вместо них клиент просто будет оповещаться о завершении выполнения фьючерса и сможет выполнить код обратного вызова, описанный в виде лямбда-выражения или ссылки на метод, только когда будут доступны результаты вычисления. А пока мы займемся другой проблемой: обработкой ошибок, возникающих во время выполнения асинхронной задачи.

16.2.2. Обработка ошибок

Разработанный выше код работает правильно, если не возникает никаких проблем. Но что произойдет, если операция вычисления цены вернет ошибку? К сожалению, в этом случае все будет особенно плохо: исключение, сгенерированное для уведомления об ошибке, окажется ограничено рамками потока выполнения, вычисляющего цену товара, и в итоге приведет к его аварийному завершению. В итоге клиент будет бесконечно ждать результата `get` и окажется заблокированным навсегда.

Предотвратить эту проблему клиент может с помощью перегруженной версии метода `get`, принимающей в качестве параметра также и время ожидания. Использовать параметр тайм-аута для предотвращения подобных ситуаций где-либо в коде — рекомендуемая практика. При этом клиент по крайней мере не должен ждать бесконечно, а будет оповещен с помощью `TimeoutException`, когда время ожидания истечет. Но в результате у клиента не будет возможности выяснить, что привело к сбою потока выполнения, вычисляющего цену товара. Чтобы клиент мог узнать, почему у магазина не получилось предоставить ему цену запрошенного товара, необходимо транслировать вызвавшее проблему исключение внутрь объекта `CompletableFuture` с помощью его метода `completeExceptionally`. Если применить эту идею к листингу 16.4, получаем код, показанный в листинге 16.6.

Листинг 16.6. Трансляция исключения внутрь объекта CompletableFuture

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread( () -> {
                try {
                    double price = calculatePrice(product);
                    futurePrice.complete(price);
                } catch (Exception ex) {
                    futurePrice.completeExceptionally(ex);
                }
    }).start();
    return futurePrice;
}
```

При успешном вычислении цены завершаем фьючерс с этой ценой

В противном случае завершаем фьючерс исключением, передав ему вызвавшее этот сбой исключение

Теперь клиент будет оповещен с помощью объекта `ExecutionException` (принимающего параметр типа `Exception`, содержащий причину сбоя — исходное исключение, сгенерированное методом вычисления цены). Например, если метод вычисления

цены сгенерирует исключение RuntimeException, указывающее на недоступность товара, то клиент получит примерно следующее ExecutionException:

```
Exception in thread "main" java.lang.RuntimeException:
    java.util.concurrent.ExecutionException: java.lang.RuntimeException:
         product not available
    at java89inaction.chap16.AsyncShopClient.main(AsyncShopClient.java:16)
Caused by: java.util.concurrent.ExecutionException:
    java.lang.RuntimeException: product not available
    at java.base/java.util.concurrent.CompletableFuture.reportGet
      (CompletableFuture.java:395)
    at java.base/java.util.concurrent.CompletableFuture.get
      (CompletableFuture.java:1999)
    at java89inaction.chap16.AsyncShopClient.main(AsyncShopClient.java:14)
Caused by: java.lang.RuntimeException: product not available
    at java89inaction.chap16.AsyncShop.calculatePrice(AsyncShop.java:38)
    at java89inaction.chap16.AsyncShop.lambda$0(AsyncShop.java:33)
    at java.base/java.util.concurrent.CompletableFuture$AsyncSupply.run
      (CompletableFuture.java:1700)
    at java.base/java.util.concurrent.CompletableFuture$AsyncSupply.exec
      (CompletableFuture.java:1692)
    at java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:283)
    at java.base/java.util.concurrent.ForkJoinPool.runWorker
      (ForkJoinPool.java:1603)
    at java.base/java.util.concurrent.ForkJoinWorkerThread.run
      (ForkJoinWorkerThread.java:175)
```

Создание завершаемого фьючерса с помощью фабричного метода supplyAsync

До сих пор мы создавали объекты CompletableFuture и завершали их, когда нам было удобно, программным способом, но в классе CompletableFuture есть множество удобных фабричных методов, значительно упрощающих этот процесс и позволяющих писать меньше кода. Например, с помощью метода supplyAsync можно переписать метод getPriceAsync в виде одного оператора, как показано в листинге 16.7.

Листинг 16.7. Создание экземпляра завершаемого фьючерса с помощью фабричного метода supplyAsync

```
public Future<Double> getPriceAsync(String product) {
    return CompletableFuture.supplyAsync(() -> calculatePrice(product));
}
```

Метод supplyAsync принимает в качестве аргумента объект Supplier и возвращает объект CompletableFuture, выполнение которого асинхронно завершается значением, полученным при вызове этого Supplier. Этот поставщик выполняется одним из исполнителей пула ForkJoinPool, но можно указать и другой исполнитель, передав его в качестве второго аргумента перегруженной версии метода supplyAsync. Вообще говоря, передать исполнитель можно и любому другому из фабричных методов класса CompletableFuture. Мы воспользуемся данной возможностью в подразделе 16.3.4, где продемонстрируем положительное влияние на быстродействие приложения от применения подходящего для него исполнителя.

Отметим также, что объект CompletableFuture, возвращаемый в листинге 16.7 методом supplyAsync, эквивалентен созданному (и завершенному) нами вруч-

ную в листинге 16.6, а значит, он обеспечивает такую же аккуратную обработку ошибок.

В оставшейся части данной главы мы будем предполагать, что никакого контроля над реализованным классом `Shop` API у нас нет и что он предоставляет только синхронные блокирующие методы. Такая ситуация вполне обычна для потребления HTTP API какого-либо сервиса. Вы увидите, что, несмотря на это, можно выполнять запросы к нескольким магазинам асинхронно, таким образом избегая блокировок отдельных запросов и повышая производительность и пропускную способность приложения поиска лучшей цены.

16.3. Делаем код неблокирующим

Нам нужно разработать приложение для поиска наилучшей цены, и все опрашиваемые интернет-магазины должны предоставлять один и тот же синхронный API, реализованный так, как показано в начале раздела 16.2. Другими словами, у нас есть список магазинов, вот такой:

```
List<Shop> shops = List.of(new Shop("BestPrice"),
                           new Shop("LetsSaveBig"),
                           new Shop("MyFavoriteShop"),
                           new Shop("BuyItAll"));
```

Нам нужно реализовать метод со следующей сигнатурой, который по названию товара возвращал бы список строковых значений, содержащих название магазина и цену в нем на интересующий нас товар:

```
public List<String> findPrices(String product);
```

Первая идея, которая приходит в голову: воспользоваться возможностями потоков данных, о которых вы узнали из глав 4–6. Наверное, вам хочется написать что-то вроде листинга 16.8 (хорошо, если вы уже догадываетесь, что это первое решение — далеко не лучшее!).

Листинг 16.8. Реализация findPrices, где последовательно опрашиваются все магазины

```
public List<String> findPrices(String product) {
    return shops.stream()
       .map(shop -> String.format("%s price is %.2f",
                                  shop.getName(), shop.getPrice(product)))
       .collect(toList());
}
```

Это решение слишком прямолинейно. Давайте попробуем узнать с помощью данного метода цену самого желанного для вас сегодня товара: myPhone27S. В листинге 16.9 выведена также информация о длительности его работы. Это позволит нам сравнить быстродействие данной реализации с усовершенствованной версией, которую мы разработаем позднее.

Листинг 16.9. Проверяем правильность работы и определяем быстродействие метода findPrices

```
long start = System.nanoTime();
System.out.println(findPrices("myPhone27S"));
long duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Done in " + duration + " msecs");
```

Выводимые кодом из листинга 16.9 результаты выглядят так:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
is 214.13, BuyItAll price is 184.74]
Done in 4032 msecs
```

Как можно было ожидать, метод `findPrices` выполняется чуть более 4 секунд, поскольку наши четыре магазина опрашиваются последовательно, блокируя выполнение один за другим, а вычисление цены запрашиваемого товара для каждого из магазинов занимает примерно 1 секунду. Как можно улучшить этот показатель?

16.3.1. Распараллеливание запросов с помощью параллельного потока данных

После прочтения главы 7 первое усовершенствование, которое должно прийти вам на ум, — избежать последовательного выполнения вычислений, воспользовавшись параллельным потоком данных вместо последовательного, как показано в листинге 16.10.

Листинг 16.10. Распараллеливаем выполнение метода findPrices

```
public List<String> findPrices(String product) {
    return shops.parallelStream()
       .map(shop -> String.format("%s price is %.2f",
                                  shop.getName(), shop.getPrice(product)))
       .collect(toList());
}
```

Используем для параллельного извлечения цен товара для различных магазинов параллельный поток данных

Проверим, лучше ли работает эта новая версия метода `findPrices`, снова запустив код из листинга 16.9:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
    is 214.13, BuyItAll price is 184.74]
Done in 1180 msecs
```

Отлично! Несмотря на простоту, эта идея, похоже, работает отлично. Теперь все четыре магазина опрашиваются параллельно, так что выполнение кода занимает чуть более 1 секунды.

Можно ли еще улучшить этот результат? Попробуем преобразовать все синхронные обращения к магазинам в методе `findPrices` в асинхронные, воспользовавшись полученными знаниями о классе `CompletableFuture`.

16.3.2. Выполнение асинхронных запросов с помощью завершаемых фьючерсов

Ранее вы видели, как создавать объекты `CompletableFuture` с помощью фабричного метода `supplyAsync`. Воспользуемся им:

```
List<CompletableFuture<String>> priceFutures =
        shops.stream()
        .map(shop -> CompletableFuture.supplyAsync(
             () -> String.format("%s price is %.2f",
```

```
                shop.getName(), shop.getPrice(product))))
        .collect(toList());
```

При таком подходе мы получаем `List<CompletableFuture<String>>`, в котором каждый объект `CompletableFuture` в списке по завершении его вычисления содержит строковое название магазина. Но, поскольку метод `findPrices`, рефакторинг которого на основе `CompletableFuture` мы хотим провести, должен возвращать `List<String>`, мы должны дождаться завершения всех этих фьючерсов и извлечь из них значения, прежде чем вернуть список.

Чтобы добиться этого, можно применить к исходному `List<CompletableFuture<String>>` вторую операцию `map`, вызвать операцию `join` для всех фьючерсов из списка, после чего дождаться их поочередного завершения. Отметим, что смысл метода `join` из класса `CompletableFuture` совпадает со смыслом объявленного в интерфейсе `Future` метода `get`. Единственное отличие заключается в том, что метод `get` не генерирует никаких проверяемых исключений. Благодаря методу `join` можно не раздувать размер передаваемого в этот второй `map` лямбда-выражения вставкой в него блока `try/catch`. Если собрать все воедино, можно переписать метод `findPrices` так, как показано в листинге 16.11.

Листинг 16.11. Реализация метода findPrices с помощью класса CompletableFuture

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
            shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(
                         () -> shop.getName() + " price is " +
                               shop.getPrice(product)))
            .collect(Collectors.toList());
    return priceFutures.stream()
            .map(CompletableFuture::join)
            .collect(toList());
}
```

Вычисляем все цены асинхронно с помощью завершаемых фьючерсов

Ожидаем завершения всех асинхронных операций

Обратите внимание на использование двух отдельных потоковых конвейеров вместо двух последовательных операций `map` в одном конвейере потоковой обработки — и на это есть уважительная причина. В силу отложенной природы промежуточных потоковых операций при обработке потока данных в одном конвейере все запросы к различным магазинам обрабатывались бы синхронно и последовательно. Создание каждого из объектов `CompletableFuture` для опроса магазинов будет начинаться лишь после завершения вычисления предыдущего и возврата методом `join` результата данного вычисления. Этот важный нюанс проиллюстрирован на рис. 16.2.

Верхняя половина рис. 16.2 показывает, что обработка потока данных с помощью одного конвейера подразумевает параллельный порядок вычислений (указанный штриховой линией). Фактически новый `CompletableFuture` создается лишь после полного завершения вычисления предыдущего. И наоборот, на нижней половине рисунка показано, что, собрав сначала объекты `CompletableFuture` в списке (отмеченном овалом), можно запустить их все, не дожидаясь завершения.

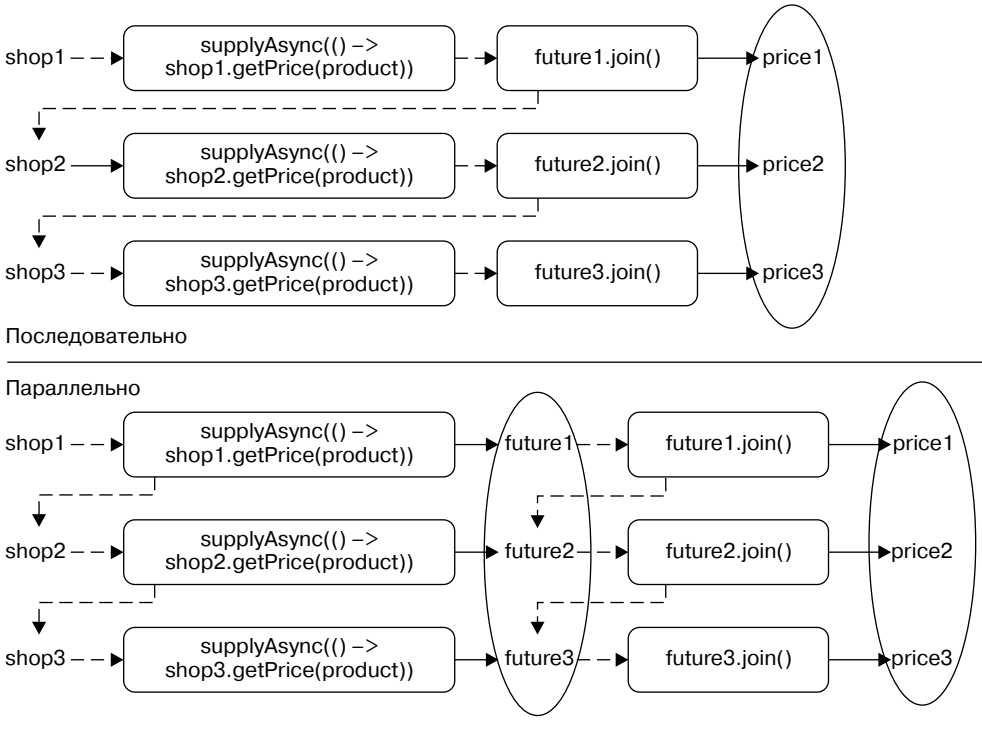


Рис. 16.2. Почему отложенная природа потоков данных приводит к последовательному выполнению вычислений и как этого избежать

При запуске кода из листинга 16.11 для определения производительности этой, третьей версии метода `findPrices` в консоль выводится следующее:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47,
     MyFavoriteShop price is 214.13, BuyItAll price is 184.74]
Done in 2005 msecs
```

Результат неутешительный, правда? Эта реализация с объектами `CompletableFuture` выполняется более 2 секунд, то есть быстрее исходной «наивной» последовательной реализации с блокировками из листинга 16.8, но почти вдвое медленнее предыдущей реализации с параллельным потоком данных. Еще более разочаровывает тот факт, что для создания версии с параллельным потоком данных потребовалось лишь тривиальное изменение последовательной версии.

Новая версия с завершаемыми фьючерсами потребовала немало труда. Было ли применение их здесь пустой тратой времени? Или мы упустили нечто важное? Прежде чем двигаться дальше, задумаемся на несколько минут и вспомним, в частности, что мы проверяем наши примеры на машине, способной на одновременную работу четырех потоков[1].

[1] Если ваша машина способна на большее (скажем, восемь потоков выполнения), то для воспроизведения демонстрируемого далее поведения вам понадобятся дополнительные магазины и параллельные процессы.

16.3.3. Поиск лучше масштабируемого решения

Версия с параллельным потоком данных работает так хорошо лишь благодаря возможности работы четырех потоков одновременно, вследствие чего для каждого магазина можно выделить по отдельному потоку выполнения. Что будет, если добавить в список просматриваемых нашим приложением для поиска наилучшей цены еще один, пятый магазин? Последовательная версия при этом, что неудивительно, выполняется более 5 секунд, как показано в следующем выводе:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop
  price is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 5025 msecs
```

Результаты работы программы, использующей последовательный поток данных

К сожалению, выполнение версии с параллельным потоком данных также занимает на целую секунду больше, чем ранее, поскольку все четыре потока выполнения (имеющиеся в пуле потоков выполнения) заняты первыми четырьмя магазинами. Пятому запросу придется дождаться завершения одной из предыдущих операций и освобождения одного из потоков выполнения:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop
  price is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 2167 msecs
```

Результаты работы программы, использующей параллельный поток данных

А как насчет версии с завершаемым фьючерсом? Попробуем ее для пяти магазинов:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop
  price is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 2006 msecs
```

Результаты работы программы, использующей объекты CompletableFuture

Версия с объектами `CompletableFuture` отрабатывает немного быстрее, чем версия с параллельным потоком данных, но скорость ее работы все равно нас не устраивает. Если попытаться запустить код с девятью магазинами, то версия с параллельным потоком данных выполнится за 3143 миллисекунды, а версия с завершаемыми фьючерсами — за 3009 миллисекунд. Версии выглядят эквивалентными: внутри обеих используется один и тот же общий пул, по умолчанию содержащий фиксированное число потоков выполнения, равное возвращаемому `Runtime.getRuntime().availableProcessors()` значению. Тем не менее у версии с объектами `CompletableFuture` есть преимущество: в отличие от API параллельных потоков данных она позволяет задать другой исполнитель для отправки задач. Можно задать такие настройки этого исполнителя и размер его пула потоков выполнения, которые оптимально подойдут для требований именно вашего приложения. В следующем разделе мы продемонстрируем, как эти улучшенные возможности настройки превращаются в улучшенное быстродействие приложения.

16.3.4. Пользовательский исполнитель

В данном случае разумно будет создать исполнитель, в пуле которого содержалось бы соответствующее фактической нагрузке приложения количество потоков. Как же выбрать правильный размер исполнителя?

Выбор размера пула потоков выполнения

В замечательной книге *Java Concurrency in Practice* (Addison-Wesley, 2006; http://jcip.net) Брайан Гётц (Brian Goetz) и его соавторы привели совет по выбору оптимального размера пула потоков выполнения. Этот совет важен, поскольку при слишком большом числе потоков в пуле они начинают конкурировать за процессорные ресурсы и оперативную память и тратить время на переключение контекста. И наоборот, если их количество слишком мало (как, вероятно, в нашем приложении), часть ядер CPU будет использоваться не на все 100 %. Гётц предлагает следующую формулу для приближенной оценки размера пула с желаемой интенсивностью использования CPU:

$$N^{потоков} = N^{CPU} * U^{CPU} * (1 + W / C).$$

В этой формуле N^{CPU} — количество ядер процессора, равное значению, возвращаемому `Runtime.getRuntime().availableProcessors()`.

- U^{CPU} — желаемая степень использования CPU (от 0 до 1).
- W / C — соотношение времени ожидания и времени вычислений.

Приложение проводит почти 99 % времени в ожидании ответов магазинов, так что соотношение W / C можно считать равным 100. Если мы хотим использовать CPU на 100 %, то пул должен содержать 400 потоков выполнения. На практике будет расточительством держать больше потоков выполнения, чем магазинов, поскольку часть потоков в пуле при этом никогда не будет использоваться. Таким образом, вам нужен исполнитель с фиксированным числом потоков выполнения, равным числу опрашиваемых магазинов, чтобы у каждого магазина был свой поток. Следует также установить верхний предел числа потоков выполнения равным 100 во избежание аварийного останова сервера в случае большего числа магазинов, как показано в листинге 16.12.

Листинг 16.12. Пользовательский исполнитель, специально подобранный для нашего приложения поиска наилучшей цены

```
private final Executor executor =
        Executors.newFixedThreadPool(Math.min(shops.size(), 100),
                                     (Runnable r) -> {
            Thread t = new Thread(r);
            t.setDaemon(true);
            return t;
        }
);
```

Используем потоки-демоны, не предотвращающие завершение выполнения программы

Создаем пул потоков выполнения с числом потоков, равным минимуму из 100 и числа магазинов

Обратите внимание, что мы создаем пул из потоков-демонов. Выйти из Java-программы или завершить ее выполнение нельзя, пока выполняется обычный поток, так что поток, оставшийся в состоянии ожидания события, которое так и не произойдет, приводит к возникновению проблем. А если отметить поток как демон, то его выполнение можно будет прервать при завершении программы. Производительность при этом не меняется. Теперь можно передать этот новый `Executor` в качестве второго аргумента фабричного метода `supplyAsync`. Помимо этого, создадим объект `CompletableFuture` для извлечения цены интересующего нас товара из заданного магазина, вот так:

```
CompletableFuture.supplyAsync(() -> shop.getName() + " price is " +
                                    shop.getPrice(product), executor);
```

После этих усовершенствований выполнение решения, использующего объекты `CompletableFuture`, занимает 1021 миллисекунду при обработке пяти магазинов и 1022 миллисекунды при обработке девяти. Эта тенденция сохраняется, пока количество магазинов не достигнет порогового значения 400, вычисленного нами выше. Данный пример демонстрирует преимущества создания исполнителя, подогнанного под требования конкретного приложения, и использования завершаемых фьючерсов для передачи ему задач на выполнение. Эта стратегия прекрасно работает практически всегда, имейте ее в виду при интенсивном использовании асинхронных операций.

Параллелизм: потоки данных или завершаемые фьючерсы?

Мы показали вам два способа выполнения потоковых операций над коллекциями: преобразование коллекции в параллельный поток данных и применение к нему таких операций, как `map`, либо проход по коллекции в цикле и порождение операций внутри завершаемых фьючерсов. Второй из этих подходов предоставляет больше возможностей контроля за счет выбора размеров пулов потоков выполнения, благодаря чему вы можете быть уверены, что выполнение программы в целом не будет блокировано, поскольку все (фиксированное количество!) потоки ожидают ввода/вывода.

Советуем вам использовать эти API вот так.

- При выполнении требующих большого объема вычислений операций без какого-либо ввода/вывода простейшей и, вероятно, наиболее эффективной реализацией будет та, что предоставляет интерфейс `Stream` (если все потоки выполнения ограничены по скорости вычислений, то не имеет смысла, чтобы их было больше, чем ядер процессора).
- Если выполняемые параллельно операции требуют ожидания ввода/вывода (в том числе сетевых соединений), решение с завершаемыми фьючерсами окажется более гибким и даст возможность подобрать число потоков выполнения, соответствующее соотношению ожидания/вычислений (W / C), как обсуждалось выше. Еще одна причина, почему не стоит использовать параллельные потоки данных при наличии ожиданий ввода/вывода в конвейере потоковой обработки, — то, что отложенная сущность потоков затрудняет анализ программы при возникновении ожиданий.

Вы уже знаете, как использовать завершаемые фьючерсы для предоставления клиентам асинхронного API и работы в качестве клиента синхронного, но медленного сервера, но мы пока выполняли в каждом из фьючерсов лишь одну «долгоиграющую» операцию. В следующем разделе мы воспользуемся завершаемыми фьючерсами для декларативной организации в виде конвейера нескольких асинхронных операций в стиле, напоминающем тот, что мы применяли для Stream API.

16.4. Конвейерная организация асинхронных задач

Пусть у всех магазинов есть единый централизованный сервис скидок. В этом сервисе используется пять кодов скидок, каждый — со своей скидкой (в процентном отношении). Мы отразим эту идею в нашей программе, описав перечислимый тип `Discount.Code`, как показано в листинге 16.13.

Листинг 16.13. Перечислимый тип с описанием кодов скидок

```
public class Discount {
    public enum Code {
        NONE(0), SILVER(5), GOLD(10), PLATINUM(15), DIAMOND(20);
        private final int percentage;
        Code(int percentage) {
            this.percentage = percentage;
        }
    }
    // Реализацию класса мы опускаем, см. листинг 16.14
}
```

Пусть также все магазины договорились изменить формат результата метода `getPrice`, и теперь он будет возвращать строковое значение в формате `НазваниеМагазина:цена:КодСкидки`. Наш пример реализации возвращает случайный `Discount.Code` и уже вычисленную случайную цену, вот так:

```
public String getPrice(String product) {
    double price = calculatePrice(product);
    Discount.Code code = Discount.Code.values()[
                            random.nextInt(Discount.Code.values().length)];
    return String.format("%s:%.2f:%s", name, price, code);
}
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

Вызов метода `getPrice` может вернуть, например, строковое значение:

```
BestPrice:123.26:GOLD
```

16.4.1. Реализация сервиса скидок

Приложение поиска наилучшей цены должно теперь получать цены из магазинов, производить синтаксический разбор полученных строковых значений и для каждого

из них запрашивать сервис скидок. В результате этого процесса получается итоговая цена с учетом скидки (фактический процент скидки конкретного скидочного кода может меняться, поэтому приходится каждый раз запрашивать информацию у сервера). Синтаксический разбор строковых значений, сгенерированных магазином, инкапсулируется в следующем классе `Quote`:

```
public class Quote {
    private final String shopName;
    private final double price;
    private final Discount.Code discountCode;
    public Quote(String shopName, double price, Discount.Code code) {
        this.shopName = shopName;
        this.price = price;
        this.discountCode = code;
    }
    public static Quote parse(String s) {
        String[] split = s.split(":");
        String shopName = split[0];
        double price = Double.parseDouble(split[1]);
        Discount.Code discountCode = Discount.Code.valueOf(split[2]);
        return new Quote(shopName, price, discountCode);
    }
    public String getShopName() { return shopName; }
    public double getPrice() { return price; }
    public Discount.Code getDiscountCode() { return discountCode; }
}
```

Получить расценки в виде экземпляра класса `Quote`, содержащего название магазина, цену без учета скидки и скидочный код можно, передав сгенерированное магазином строковое значение статическому фабричному методу `parse`.

У сервиса `Discount` есть также метод `applyDiscount`, принимающий на входе объект `Quote` и возвращающий строковое значение с ценой с учетом скидки для соответствующего магазина, как показано в листинге 16.14.

Листинг 16.14. Сервис Discount

```
public class Discount {
    public enum Code {
        // исходный код опущен ...
    }
    public static String applyDiscount(Quote quote) {
        return quote.getShopName() + " price is " +
               Discount.apply(quote.getPrice(),
                              quote.getDiscountCode());
    }
    private static double apply(double price, Code code) {
        delay();
        return format(price * (100 - code.percentage) / 100);
    }
}
```

Применяем скидочный код к исходной цене

Имитируем задержку ответа сервиса Discount

16.4.2. Использование скидочного сервиса

В силу удаленного характера сервиса `Discount` мы добавляем имитацию задержки в 1 секунду, как показано в листинге 16.14. Как и в разделе 16.3, сначала попытаемся реализовать метод `findPrices` так, чтобы он удовлетворял этим требованиям простейшим (но, увы, последовательным и синхронным) способом (листинг 16.15).

Листинг 16.15. Простейшая реализация метода findPrices, использующая сервис Discount

```
public List<String> findPrices(String product) {
    return shops.stream()
        .map(shop -> shop.getPrice(product))
        .map(Quote::parse)
        .map(Discount::applyDiscount)
        .collect(toList());
}
```

Получаем цену без учета скидки от каждого из магазинов

Преобразуем возвращаемые магазинами строковые значения в объекты Quote

Обращаемся к сервису скидок для применения скидок ко всем расценкам

Мы получили нужный результат, организовав в конвейер три операции `map` над потоком магазинов.

- Первая из них отображает каждый из магазинов в строковое значение, кодирующее цену и скидочный код запрашиваемого у этого магазина товара.
- Вторая операция выполняет синтаксический разбор этих строковых значений и преобразование каждого из них в расценки (объект `Quote`).
- Третья операция связывается с удаленным сервисом `Discount`, который вычисляет итоговую цену с учетом скидки и возвращает еще одно строковое значение с названием магазина и эту цену.

Как вы догадываетесь, быстродействие этой реализации далеко от оптимального. Попробуем все же измерить его, как обычно, с помощью нашего теста производительности:

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 10028 msecs
```

Как и ожидалось, выполнение данного кода занимает 10 секунд, поскольку 5 секунд, необходимые для последовательного опроса пяти магазинов, прибавляются к 5 секундам, которые требуются сервису скидок для применения к возвращаемым пятью магазинами ценам скидочного кода. Вы уже знаете, что этот результат можно улучшить, преобразовав поток в параллельный. Но также вам известно (из раздела 16.3), что это решение плохо масштабируется при увеличении числа опрашиваемых магазинов из-за фиксированного размера пула потоков выполнения, необходимого для работы потоков данных. И напротив, вы уже знаете, что можно более полно использовать CPU, описав пользовательского исполнителя, планирующего график выполнения задач завершаемыми фьючерсами.

16.4.3. Композиция синхронных и асинхронных операций

В этом разделе мы попытаемся реализовать метод `findPrices` в асинхронном виде, снова используя возможности класса `CompletableFuture`. В листинге 16.16 показан

соответствующий код. Не беспокойтесь, что некоторые его части могут выглядеть непривычно, мы объясним все далее.

Листинг 16.16. Реализация метода findPrices с помощью завершаемых фьючерсов

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
        shops.stream()
             .map(shop -> CompletableFuture.supplyAsync(
                                 () -> shop.getPrice(product), executor))
             .map(future -> future.thenApply(Quote::parse))
             .map(future -> future.thenCompose(quote ->
                         CompletableFuture.supplyAsync(
                           () -> Discount.applyDiscount(quote), executor)))
                .collect(toList());
    return priceFutures.stream()
            .map(CompletableFuture::join)
            .collect(toList());
}
```

Преобразуем возвращенное магазином строковое значение в объект Quote

Асинхронно получаем от каждого из магазинов цену без учета скидки

Выполняем композицию получившегося фьючерса с другой асинхронной задачей, применяя скидочный код

Ожидаем завершения всех фьючерсов в потоке данных и извлекаем их результаты

На этот раз все более запутанно, так что попробуем разобраться в происходящем поэтапно. Последовательность этих трех преобразований приведена на рис. 16.3.

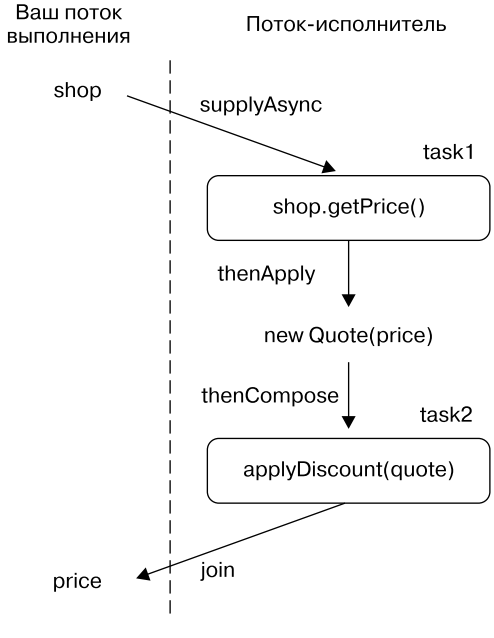


Рис. 16.3. Композиция синхронных и асинхронных операций

Мы производим те же три операции `map`, что и в синхронном решении из листинга 16.15, но делаем их асинхронными по мере необходимости благодаря возможностям класса `CompletableFuture`.

Получение цен

Первую из трех операций вы уже встречали в разнообразных примерах в этой главе; мы асинхронно запрашиваем информацию у магазина, передавая фабричному методу `supplyAsync` лямбда-выражение. В результате этого первого преобразования получаем объект `Stream<CompletableFuture<String>>`, в котором каждый объект `CompletableFuture` после завершения выполнения содержит возвращаемое соответствующим магазином строковое значение. Отметим, что для этих завершаемых фьючерсов мы задействуем пользовательский исполнитель, разработанный в листинге 16.12.

Парсинг строковых значений в расценки

Теперь нам нужно преобразовать эти строковые значения в расценки (объекты `Quote`) с помощью второго преобразования. Но эту операцию парсинга (parsing), поскольку она не требует обращения к удаленным сервисам или какого-либо ввода/вывода, можно произвести практически мгновенно и синхронно, без какой-либо задержки. Поэтому это второе преобразование мы производим с помощью вызова для сгенерированных на первом шаге завершаемых фьючерсов метода `thenApply`, которому передаем функцию, преобразующую строковое значение в экземпляр `Quote`.

Отметим, что использование метода `thenApply` не приводит к блокировке кода до момента завершения соответствующего завершаемого фьючерса. После завершения выполнения объекта `CompletableFuture` мы должны преобразовать содержащееся в нем значение с помощью переданного в метод `thenApply` лямбда-выражения, таким образом преобразуя каждый из объектов `CompletableFuture<String>` из потока данных в соответствующий объект `CompletableFuture<Quote>`. Этот процесс можно рассматривать как формирование рецепта, задающего действия, которые требуется выполнить с результатом завершаемого фьючерса, аналогично тому, как мы работали с конвейером потоковой обработки.

Композиция фьючерсов для вычисления цены с учетом скидки

Третья операция `map` включает обращение к удаленному сервису скидок для применения соответствующей скидки к исходным ценам, полученным от магазинов. Это преобразование отличается от предыдущего тем, что должно выполняться удаленно (или в данном случае имитировать удаленный вызов с помощью задержки), а поэтому желательно асинхронно.

Для этого, как и при первом вызове метода `supplyAsync` с `getPrice`, мы передаем эту операцию в виде лямбда-выражения в фабричный метод `supplyAsync`, который возвращает еще один завершаемый фьючерс. В итоге у нас оказывается две асинхронные операции, моделируемые с помощью двух завершаемых фьючерсов, которые мы хотели бы выполнить один за другим.

- Получить цену из магазина и преобразовать ее в объект `Quote`.
- Передать этот объект `Quote` сервису скидок для получения итоговой цены с учетом скидки.

Как раз для этой цели класс `CompletableFuture` API Java 8 включает метод `thenCompose`, с помощью которого можно связать цепочкой две асинхронные операции с передачей результата первой операции во вторую после его появления. Другими словами, для композиции двух завершаемых фьючерсов необходимо вызвать метод `thenCompose` для первого из них и передать ему объект `Function`. Один из аргументов этой функции — значение, возвращаемое этим первым объектом `CompletableFuture` по завершении его выполнения. Сама функция возвращает второй объект `CompletableFuture`, входными данными для вычислений которого служат результаты первого. Отметим, что при таком подходе основной поток выполнения может производить другие полезные операции (например, реагировать на события UI), пока фьючерсы получают данные о расценках от магазинов.

Собирая полученные в результате этих трех операций `map` элементы потока данных в список, получаем `List<CompletableFuture<String>>`. Наконец, дожидаемся завершения этих завершаемых фьючерсов и извлекаем их значения с помощью операции `join`, точно так же, как в листинге 16.11. Результаты выполнения новой версии метода `findPrices` выглядят следующим образом:

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 2035 msecs
```

У использовавшегося в листинге 16.16 метода `thenCompose`, как и у других методов класса `CompletableFuture`, есть вариант с суффиксом `Async` — `thenComposeAsync`. Вообще говоря, методы без суффикса `Async` выполняют задачу в том же потоке выполнения, что и предыдущую, а методы, название которых заканчивается на `Async`, всегда отправляют следующую задачу в пул потоков выполнения, так что задачи могут обрабатываться различными потоками. В данном случае результат второго завершаемого фьючерса основан на результате первого, так что для конечного результата и его приблизительных временных показателей неважно, с помощью какого из вариантов метода производится композиция двух наших завершаемых фьючерсов. Отметим, однако, что далеко не всегда понятно, какой поток выполнения используется, особенно при работе приложения, которое само управляет пулом потоков выполнения (например, Spring).

16.4.4. Сочетание двух завершаемых фьючерсов: зависимого и независимого

В листинге 16.16 мы вызывали метод `thenCompose` одного из завершаемых фьючерсов и передавали ему второй завершаемый фьючерс, требующий в качестве входных данных значение, получаемое в результате выполнения первого. В другом распространенном сценарии нужно комбинировать результаты операций, производимых двумя независимыми завершаемыми фьючерсами, причем желательно так, чтобы не ждать завершения первого перед запуском второго.

В подобных ситуациях используйте метод `thenCombine`. Он принимает в качестве второго аргумента экземпляр `BiFunction`, определяющий способ композиции результатов двух наших завершаемых фьючерсов после их появления. Подобно методу `thenCompose` у `thenCombine` есть `Async`-вариант. В данном случае использование метода `thenCombineAsync` приводит к отправке в пул потоков и последующему

асинхронному выполнению в отдельной задаче операции сочетания, задаваемой объектом `BiFunction`.

Возвращаясь к примеру данной главы, предположим, что один из магазинов возвращает цены в евро (EUR), но вы хотели бы сообщать их вашим покупателям в долларах (USD). Можно асинхронно запросить у магазина цену нужного товара *и отдельно* извлечь из удаленного сервиса обмена валют текущий курс евро к доллару. По завершении обоих запросов можно получить композицию их результатов путем умножения цены на курс обмена. При таком подходе мы получим третий завершаемый фьючерс, который завершается по появлении результатов обоих остальных завершаемых фьючерсов с последующей композицией с помощью `BiFunction` (листинг 16.17).

Листинг 16.17. Сочетание двух независимых завершаемых фьючерсов

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD)),
            (price, rate) -> price * rate
        ));
```

Создаем первую задачу для запроса цены продукта у магазина

Сочетание цены и курса обмена путем их умножения

Создаем вторую независимую задачу для извлечения курса обмена евро/доллар

Здесь выполнение операции сочетания в отдельной задаче означало бы впустую расходовать ресурсы, поскольку она представляет собой простое умножение. Так что лучше воспользоваться методом `thenCombine` вместо его асинхронного аналога `thenCombineAsync`. Рисунок 16.4 иллюстрирует выполнение задач, созданных в листинге 16.17, в различных потоках выполнения из пула и пример сочетания их результатов.

16.4.5. Сравниваем фьючерсы с завершаемыми фьючерсами

Последние два примера в листингах 16.16 и 16.17 ясно демонстрируют одно из важнейших преимуществ завершаемых фьючерсов над остальными реализациями фьючерсов, актуальными до Java 8. Завершаемые фьючерсы предоставляют декларативный API на основе лямбда-выражений. С его помощью можно легко сочетать различные синхронные и асинхронные задачи для максимально эффективного выполнения сложных операций. Чтобы лучше прочувствовать преимущества завершаемых фьючерсов в смысле удобочитаемости кода, попробуем добиться результатов листинга 16.17 с помощью чистого кода Java 7. Листинг 16.18 демонстрирует, как именно.

В листинге 16.18 мы создаем первый фьючерс, отправляя объект `Callable` в исполнитель, который запрашивает внешний сервис для выяснения курса обмена евро/доллар. Далее мы создаем второй фьючерс, извлекая цену нужного товара в заданном магазине в евро. Наконец, как и в листинге 16.17, умножаем цену на курс обмена в том же фьючерсе, в котором запрашивали у магазина цену в евро.

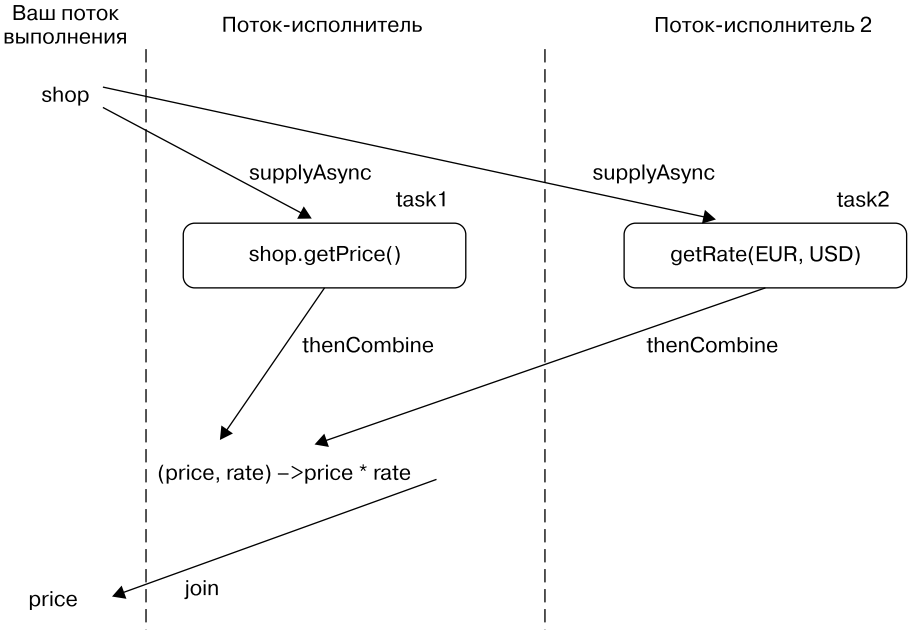


Рис. 16.4. Сочетание двух независимых асинхронных задач

Листинг 16.18. Сочетание двух фьючерсов в Java 7

```
ExecutorService executor = Executors.newCachedThreadPool();
final Future<Double> futureRate = executor.submit(new Callable<Double>() {
        public Double call() {
            return exchangeService.getRate(Money.EUR, Money.USD);
        }});
Future<Double> futurePriceInUSD = executor.submit(new Callable<Double>() {
        public Double call() {
            double priceInEUR = shop.getPrice(product);
            return priceInEUR * futureRate.get();
        }});
```

Создаем объект ExecutorService для отправки задач в пул потоков выполнения

Создаем фьючерс для извлечения курса обмена евро/доллар

Выясняем цену интересующего нас товара для заданного магазина во втором фьючерсе

Умножаем цену на курс обмена в том же фьючерсе, который использовался для нахождения цены

Отметим, что использование метода `thenCombineAsync` вместо `thenCombine` в листинге 16.17 было бы эквивалентно умножению цены на курс в третьем фьючерсе в листинге 16.18. Различие между этими двумя реализациями может показаться незначительным, поскольку мы сочетаем лишь два фьючерса.

16.4.6. Эффективное использование тайм-аутов

Как уже упоминалось в подразделе 16.2.2, имеет смысл всегда указывать длительность времени ожидания при попытке чтения вычисленного фьючерсом значения во избежание бесконечной блокировки в ожидании его вычисления. В Java 9 у завершаемых фьючерсов появилось несколько удобных методов, расширяющих возможности указания времени ожидания. Метод `orTimeout` использует `ScheduledThreadExecutor` для завершения фьючерса, генерируя `TimeoutException` по истечении заданного времени ожидания и возвращает другой завершаемый фьючерс. С помощью этого метода можно продолжить цепочку конвейера вычислений и обработать `TimeoutException` с возвратом понятного сообщения. Можно добавить время ожидания во фьючерс в листинге 16.17 так, чтобы он генерировал `TimeoutException`, если не завершится в течение 3 секунд, добавив этот метод в конец цепочки методов, как показано в листинге 16.19. Длительность времени ожидания должна, конечно, соответствовать бизнес-требованиям.

Листинг 16.19. Добавление времени ожидания в завершаемый фьючерс

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD)),
            (price, rate) -> price * rate
        ))
        .orTimeout(3, TimeUnit.SECONDS);
```

Обеспечиваем генерацию фьючерсом TimeoutException в случае, если он не завершится в течение 3 секунд. Асинхронное управление тайм-аутами было добавлено в Java 9

Иногда допустимо также использовать значение по умолчанию, если сервис временно не может достаточно быстро ответить на запрос. Например, вы могли принять решение ожидать, как в листинге 16.19, получения курса валют от сервиса обмена не дольше 1 секунды, но не генерировать исключения для прерывания вычислений в целом, если завершение запроса займет больше времени. Вместо этого можно воспользоваться заранее заданным курсом обмена. Этот второй тип тайм-аута можно добавить с помощью метода `completeOnTimeout`, также появившегося в Java 9 (листинг 16.20).

Листинг 16.20. Завершение объекта CompletableFuture с использованием значения по умолчанию по истечении времени ожидания

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD))
```

```
            .completeOnTimeout(DEFAULT_RATE, 1, TimeUnit.SECONDS),
        (price, rate) -> price * rate
    ))
    .orTimeout(3, TimeUnit.SECONDS);
```

Используем курс обмена по умолчанию, если сервис обмена не возвращает результат в течение 1 секунды

Подобно методу `orTimeout`, метод `completeOnTimeOut` возвращает объект `CompletableFuture`, так что его можно связать цепочкой с другими методами `CompletableFuture`. Резюмируем: мы задали два вида тайм-аута — один приводит к сбою вычислений в целом, если они занимают более 3 секунд, и второй — на 1 секунду — завершаемый фьючерс с заранее заданным значением, который не приводит к сбою.

Мы почти закончили приложение для поиска наилучшей цены, но не хватает еще одного элемента. Мы хотели бы показывать пользователям цены в магазинах сразу же, как только они будут доступны (как обычно делают веб-сайты по страхованию машин и сравнению цен авиабилетов), вместо того чтобы ждать завершения всех запросов цен, как раньше. В следующем разделе мы узнаем, как достичь этого путем реагирования на завершение объекта `CompletableFuture` вместо вызова для него `get` или `join` и блокировки вплоть до завершения самого `CompletableFuture`.

16.5. Реагирование на завершение CompletableFuture

Во всех примерах кода, которые встречались в данной главе, мы имитировали методы, выполняющие удаленные вызовы с секундной задержкой ответа. На практике же длительность задержки у удаленных сервисов будет совершенно непредсказуема по самым разнообразным причинам, начиная от загрузки сервера и до сетевых задержек, а также в зависимости от того, насколько важным сочтет сервер обращение вашего приложения по сравнению с приложениями, платящими больше за каждый запрос.

Поэтому, вероятно, цены интересующих вас товаров для одних магазинов окажутся доступны раньше, чем для других. В листинге 16.21 мы попробуем сымитировать этот сценарий с помощью случайной задержки длительностью от 0,5 до 2,5 секунды, для чего напишем метод `randomDelay` вместо метода `delay`, который ожидал 1 секунду.

Листинг 16.21. Метод для имитации случайной задержки длительностью от 0,5 до 2,5 секунды

```
private static final Random random = new Random();
public static void randomDelay() {
    int delay = 500 + random.nextInt(2000);
    try {
        Thread.sleep(delay);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

До сих пор мы реализовывали метод `findPrices` так, что он показывал предоставленные магазинами цены только после того, как они все стали доступны. Теперь же

мы хотим, чтобы наше приложение для поиска наилучшей цены отображало цену для заданного магазина сразу же, как только она станет доступна, а не ждало самого медленного из магазинов (что может даже привести к истечению времени ожидания). Как же реализовать такое усовершенствование?

16.5.1. Рефакторинг приложения для поиска наилучшей цены

Прежде всего мы не хотим ждать создания списка, содержащего все цены. Необходимо работать непосредственно с потоком данных завершаемых фьючерсов, в котором каждый завершаемый фьючерс выполняет последовательность требуемых для заданного магазина операций. В листинге 16.22 мы переделаем первую часть реализации из листинга 16.16 в метод `findPricesStream`, чтобы получить этот поток данных завершаемых фьючерсов.

Листинг 16.22. Переделываем метод findPrices так, чтобы он возвращал поток данных фьючерсов

```
public Stream<CompletableFuture<String>> findPricesStream(String product) {
    return shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(
                                () -> shop.getPrice(product), executor))
            .map(future -> future.thenApply(Quote::parse))
            .map(future -> future.thenCompose(quote ->
                    CompletableFuture.supplyAsync(
                        () -> Discount.applyDiscount(quote), executor)));
}
```

На этой стадии мы добавим четвертую операцию `map` для обработки потока данных, возвращаемого методом `findPricesStream`, к трем, уже выполняемым внутри этого метода. Новая операция регистрирует производимое над каждым завершаемым фьючерсом действие, которое потребляет значение завершаемого фьючерса, как только оно становится доступно. В классе `CompletableFuture` API Java 8 для этого есть метод `thenAccept`, принимающий в качестве аргумента потребитель значения для его завершения. В данном случае это значение представляет собой объект `String`, возвращаемый скидочным сервисом и содержащий название магазина вместе с ценой с учетом скидки запрошенного товара в данном магазине. Потребление этого значения состоит в единственном действии — выводе его в консоль:

```
findPricesStream("myPhone").map(f -> f.thenAccept(System.out::println));
```

Как вы уже видели в случаях методов `thenCompose` и `thenCombine`, у метода `thenAccept` есть асинхронный вариант — `thenAcceptAsync`. Этот вариант вместо непосредственного выполнения передаваемого ему потребителя в том же потоке, который завершал `CompletableFuture`, планирует его выполнение в новом потоке из пула. Но, поскольку желательно избегать лишних переключений контекста и поскольку (что более важно) нам нужно как можно быстрее реагировать на завершение `CompletableFuture`, а не ждать, когда будет доступен новый поток выполнения, этот вариант мы здесь использовать не станем.

Поскольку метод `thenAccept` определяет, как потреблять результат завершаемого фьючерса, когда он окажется доступен, он возвращает `CompletableFuture<Void>`. В результате операция `map` возвращает `Stream<CompletableFuture<Void>>`. С объектом `CompletableFuture<Void>` особо ничего не сделаешь, можно только ждать его завершения, но именно это нам и нужно. Мы хотели бы также дать самому медленному из магазинов шанс ответить и вывести в консоль полученную от него цену. Для этого мы поместим все объекты `CompletableFuture<Void>` потока данных в массив и будем ждать завершения их всех, как показано в листинге 16.23.

Листинг 16.23. Реагируем на завершение CompletableFuture

```
CompletableFuture[] futures = findPricesStream("myPhone")
        .map(f -> f.thenAccept(System.out::println))
        .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
```

Фабричный метод `allOf` принимает на входе массив объектов `CompletableFuture` и возвращает объект `CompletableFuture<Void>`, который завершается только по завершении всех переданных объектов `CompletableFuture`. С помощью вызова `join` для возвращаемого методом `allOf` завершаемого фьючерса можно легко реализовать ожидание завершения всех объектов `CompletableFuture` из исходного потока данных. Этот метод удобен для приложения поиска наилучшей цены, поскольку позволяет отобразить сообщение, например `Все магазины вернули результаты или время их ожидания истекло`, так что пользователю не придется гадать, появится ли еще какая-то информация о ценах.

В других приложениях иногда требуется дождаться завершения только одного из завершаемых фьючерсов из массива, например при обращении к двум серверам обмена валют, когда можно воспользоваться результатом того из них, который откликнется первым. В таком случае можно воспользоваться фабричным методом `anyOf`. Он принимает на входе массив объектов `CompletableFuture` и возвращает `CompletableFuture<Object>`, завершающийся с тем же значением, что и первый из завершившихся `CompletableFuture`.

16.5.2. Собираем все вместе

Как мы уже говорили в начале раздела 16.5, допустим, что во всех имитирующих удаленный вызов методах используется метод `randomDelay` из листинга 16.21, вносящий случайную задержку длительностью от 0,5 до 2,5 секунды вместо фиксированной задержки в 1 секунду. Запускаем код из листинга 16.23 после этого изменения и видим, что возвращаемые магазинами цены не появляются одновременно, как было раньше, а выводятся постепенно, по мере появления цены с учетом скидки для заданного магазина. Для большей наглядности результатов этого изменения мы слегка модифицировали код, чтобы выводить длительность вычисления каждой из цен:

```
long start = System.nanoTime();
CompletableFuture[] futures = findPricesStream("myPhone27S")
        .map(f -> f.thenAccept(
```

```
            s -> System.out.println(s + " (done in " +
                 ((System.nanoTime() - start) / 1_000_000) + " msecs)")))
        .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
System.out.println("All shops have now responded in "
                   + ((System.nanoTime() - start) / 1_000_000) + " msecs");
```

Выводимые этим кодом в консоль результаты выглядят вот так:

```
BuyItAll price is 184.74 (done in 2005 msecs)
MyFavoriteShop price is 192.72 (done in 2157 msecs)
LetsSaveBig price is 135.58 (done in 3301 msecs)
ShopEasy price is 167.28 (done in 3869 msecs)
BestPrice price is 110.93 (done in 4188 msecs)
All shops have now responded in 4188 msecs
```

Как видите, в результате случайных задержек первая цена выводится теперь более чем вдвое быстрее последней.

В главе 17 идея завершаемых фьючерсов (одноразовых, вычисление которых либо проводится до конца, либо прекращается с возвратом значения) получит обобщение в Flow API Java 9, с помощью которого можно генерировать ряд значений перед (необязательным) прекращением выполнения.

Резюме

- Выполнение относительно «долгоиграющих» операций с помощью асинхронных задач может повысить производительность и быстроту реакции приложения, особенно когда оно зависит от одного или нескольких внешних сервисов.
- Имеет смысл предоставить клиентам асинхронный API. Легко реализовать его с помощью класса `CompletableFuture`.
- Класс `CompletableFuture` предоставляет возможность трансляции и обработки ошибок, сгенерированных внутри асинхронной задачи.
- Обернув вызов синхронного API в завершаемый фьючерс, можно потреблять его асинхронно.
- Можно сочетать несколько асинхронных задач, когда они независимы друг от друга, а также в случае, когда результат одной из них служит входными данными для другой.
- Для реактивного выполнения кода по завершении фьючерса и появлении результата можно зарегистрировать обратный вызов для объекта `CompletableFuture`.
- Можно выбрать: ждать завершения или всех элементов из списка завершаемых фьючерсов, или только первого.
- В Java 9 была добавлена поддержка асинхронных тайм-аутов в завершаемых фьючерсах посредством методов `orTimeout` и `completeOnTimeout`.

17 Реактивное программирование

В этой главе

- Определение реактивного программирования и обсуждение принципов Манифеста реактивности.
- Реактивное программирование на прикладном и системном уровнях.
- Пример кода, использующего реактивные потоки данных и Flow API Java 9.
- Знакомство с RxJava — широко используемой реактивной библиотекой.
- Возможности библиотеки RxJava по преобразованию и объединению нескольких реактивных потоков данных.
- «Мраморные» диаграммы для наглядного описания операций над реактивными потоками данных.

Прежде чем разбираться подробно, что такое реактивное программирование и как оно работает, не помешает выяснить, чем обусловлен рост значимости этой парадигмы. Несколько лет тому назад самые крупные приложения насчитывали десятки серверов и гигабайты данных; время отклика несколько секунд и обслуживание с отключением серверов длительностью несколько часов считались вполне допустимыми. Сегодня ситуация стремительно меняется как минимум по трем причинам.

- *Большие данные* — обычно они измеряются в петабайтах, их объем растет ежедневно.
- *Неоднородные среды* — приложения развертываются в различных средах, начиная от мобильных устройств и заканчивая облачными кластерами с тысячами многоядерных процессоров.

- *Паттерны использования* — пользователи ожидают от приложений времени отклика порядка миллисекунд и доступности 100 % времени.

Эти изменения означают, что устаревшие архитектуры программного обеспечения не удовлетворяют современным требованиям. Особенно ярко эта ситуация проявилась сейчас, когда основным источником интернет-трафика стали мобильные устройства, и в ближайшем будущем состояние дел будет только усугубляться, когда этот поток перекроет трафик от Интернета вещей (Internet of Things, IoT).

Реактивное программирование решает эти задачи за счет возможностей асинхронной обработки и объединения потоков элементов данных, поступающих от различных систем и источников. Написанные с применением этой парадигмы приложения реагируют на элементы данных по мере их поступления, что повышает скорость реакции на действия пользователей. Более того, реактивный подход применим не только к созданию отдельного компонента или приложения, но и к координации работы множества компонентов в рамках целостной реактивной системы. Спроектированные в подобном стиле системы могут обмениваться сообщениями и маршрутизировать их в меняющихся сетевых условиях, а также обеспечивать доступность при сильной нагрузке с учетом сбоев и перебоев в обслуживании. Отметим, что, хотя разработчики обычно рассматривают свои системы и приложения как созданные из отдельных компонентов, при таком гибридном, слабо сцепленном стиле создания систем эти компоненты часто сами представляют собой целые приложения. Поэтому слова *«компонент»* и *«приложение»* в данном контексте — практически синонимы.

Возможности и преимущества реактивных приложений и систем сформулированы в Манифесте реактивности, который мы обсудим в следующем разделе.

17.1. Манифест реактивности

Манифест реактивности (The Reactive Manifesto, https://www.reactivemanifesto.org), разработанный в 2013–2014 годах Йонасом Бонером (Jonas Bonér), Дэйвом Фарли (Dave Farley), Роландом Куном (Roland Kuhn) и Мартином Томпсоном (Martin Thompson), формализует набор основных принципов создания реактивных приложений и систем. Он включает четыре отличительные особенности реактивных систем.

- *Быстрота реакции* (responsiveness). У реактивной системы короткое, а главное, предсказуемое время отклика. В результате пользователь заранее знает, чего ему ожидать. Это, в свою очередь, повышает степень доверия пользователя к приложению — вне всякого сомнения, ключевой аспект хорошего приложения.
- *Отказоустойчивость* (resiliency). Система остается доступной (реагирующей на запросы пользователей) независимо от сбоев. Манифест реактивности предлагает различные методики достижения отказоустойчивости, включая дублирование выполнения компонентов, расцепление компонентов по времени (жизненные циклы отправителя и получателя не зависят друг от друга) и рабочему простран-

ству (отправитель и получатель работают в различных процессах), а также возможность асинхронной передачи каждым из компонентов задач на выполнение другим компонентам.

- *Адаптивность* (elasticity). Еще одно обстоятельство, негативно влияющее на быстроту реакции приложений — изменение нагрузки на протяжении их жизненного цикла. Реактивные системы спроектированы так, чтобы автоматически реагировать на повышение нагрузки путем увеличения выделяемых соответствующим компонентам ресурсов.
- *Ориентированность на обмен сообщениями* (message-driven). Отказоустойчивость и адаптивность требуют четкого определения границ компонентов, из которых состоит система, для обеспечения слабого сцепления, изоляции и прозрачности размещения. Взаимодействие сквозь эти границы выполняется посредством асинхронной передачи сообщений. Такой вариант обеспечивает отказоустойчивость (за счет делегирования обработки сбоев в виде сообщений) и адаптивность (за счет мониторинга числа передаваемых сообщений и соответствующего масштабирования выделяемых ресурсов).

Рисунок 17.1 демонстрирует взаимосвязи и зависимости друг от друга этих функциональных особенностей. Эти принципы справедливы на различных уровнях, начиная от разработки внутренней архитектуры маленького приложения и до координации подобных приложений с целью построения большой системы. Впрочем, конкретные вопросы касательно уровней модульности, на которых эти идеи применяются, заслуживают отдельного обсуждения.

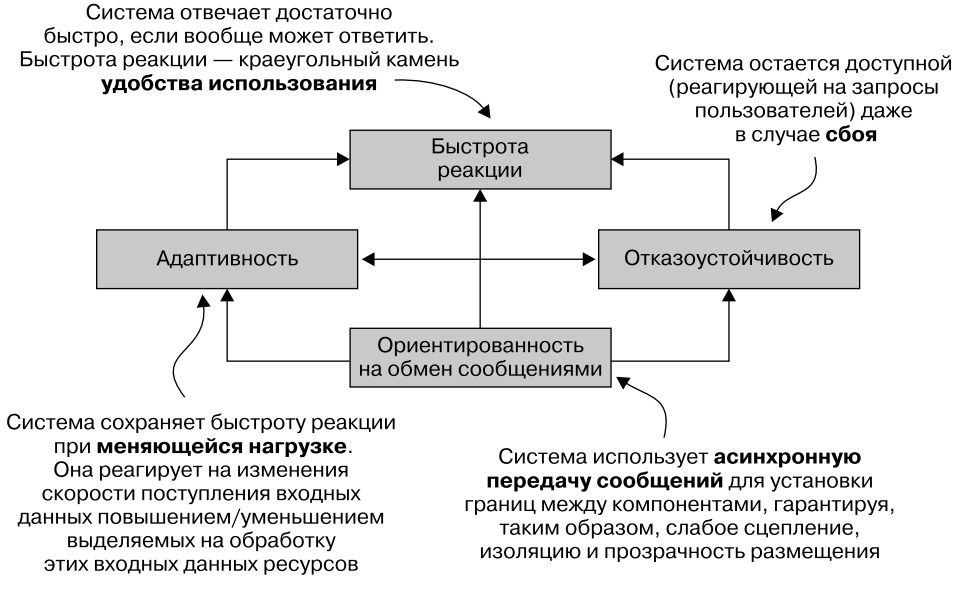


Рис. 17.1. Ключевые особенности реактивных систем

17.1.1. Реактивность на прикладном уровне

Основная особенность реактивного программирования для компонентов прикладного уровня — возможность асинхронного выполнения задач. Как мы будем обсуждать в оставшейся части данной главы, асинхронная неблокирующая обработка потоков событий жизненно важна для максимально полного использования современных многоядерных процессоров и, если быть более точными, конкурирующих за них потоков выполнения. Для этого реактивные фреймворки и библиотеки разделяют потоки выполнения (относительно дорогостоящие и дефицитные ресурсы) между облегченными языковыми конструкциями — фьючерсами, акторами и (чаще всего) циклами ожидания событий, координирующими последовательность обратных вызовов, предназначенных для агрегирования, преобразования и управления обрабатываемыми событиями.

Проверка необходимых базовых знаний

Если вас приводят в замешательство такие термины, как *«событие»*, *«сообщение»*, *«сигнал»* и *«цикл ожидания событий»* (или *«публикация/подписка»*, *«прослушиватель»* и *«противодавление»*, которые используются далее в этой главе), пожалуйста, обратитесь к главе 15. Если нет — читайте дальше.

Эти методики хороши не только меньшим расходом ресурсов, по сравнению с потоками выполнения, но и важнейшим преимуществом с точки зрения разработчика: повышением уровня абстракции реализаций конкурентных и асинхронных приложений, благодаря чему разработчики могут сконцентрировать свои усилия на бизнес-требованиях вместо решения типичных проблем низкоуровневой многопоточной обработки, таких как синхронизация, состояния гонки и взаимные блокировки.

Главное, на что стоит обратить внимание при использовании этих стратегий мультиплексирования потоков выполнения: никогда не выполнять блокирующих операций внутри основного цикла ожидания событий. Желательно включать в число блокирующих операций все связанные с вводом/выводом операции, например обращение к базе данных, файловой системе или удаленному сервису, которое может занять длительное (или заранее не прогнозируемое) время. Проще и интереснее всего будет пояснить на практическом примере, почему следует избегать блокирующих операций.

Представьте себе упрощенный, но вполне типичный сценарий мультиплексирования с пулом из двух потоков выполнения, которые обрабатывают три потока событий. Одновременно можно обрабатывать только два потока данных, и потокам данных приходится как можно эффективнее конкурировать за эти два потока выполнения. Теперь представьте себе, что при обработке события одного из этих потоков данных возникает необходимость в потенциально занимающей длительное время операции ввода/вывода, например записи в файловую систему или извлечении данных из базы данных с помощью блокирующего API. Как демонстрирует рис. 17.2, в такой ситуации поток выполнения 2 оказывается заблокирован в ожи-

дании завершения операции ввода/вывода, так что, хотя поток выполнения 1 может обрабатывать первый поток данных, обработка третьего потока данных невозможна вплоть до завершения блокирующей операции.

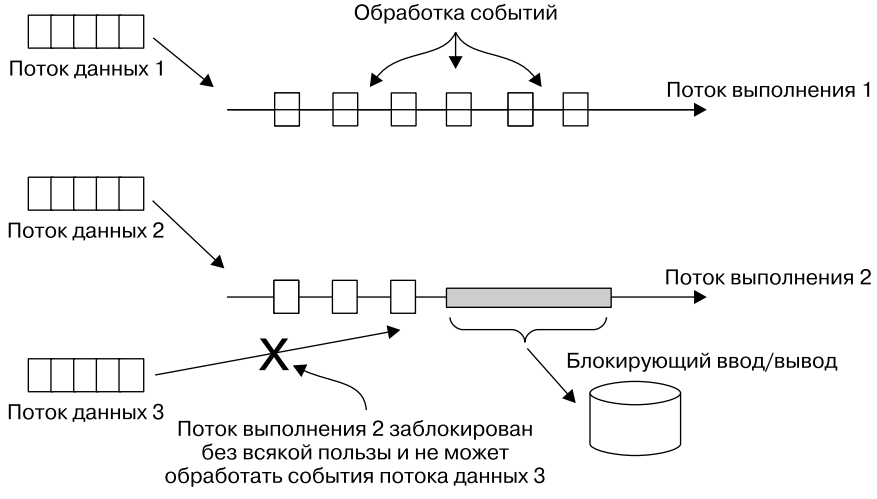


Рис. 17.2. Блокирующая операция удерживает без всякой пользы поток выполнения, в результате чего он не может производить другие вычисления

Для решения этой проблемы наиболее реактивные фреймворки (такие как RxJava и Akka) позволяют выполнять блокирующие операции с помощью отдельного выделенного пула потоков выполнения. Все потоки выполнения основного пула могут работать бесперебойно, за счет чего загрузка всех ядер процессора максимальна. Преимущество отдельных пулов потоков выполнения для операций, ограничиваемых возможностями процессора и ввода/вывода, состоит еще и в большей мелкомодульности и более точном мониторинге производительности этих двух видов задач.

Разработка приложений на основе принципов реактивности — лишь один из аспектов реактивного программирования, причем зачастую не самый трудный. Эффективная работа по отдельности идеально разработанных реактивных приложений ничуть не более важна, чем их взаимодействие в рамках слаженно функционирующей реактивной системы.

17.1.2. Реактивность на системном уровне

Реактивная система (reactive system) — вид архитектуры ПО, при котором несколько приложений образуют единую гармоничную, отказоустойчивую платформу, будучи при этом в достаточной степени расцепленными, так что при сбое одного из них система в целом продолжает работать. Основное различие между реактивным приложением и реактивной системой состоит в том, что первые обычно производят

вычисления на основе кратковременных потоков данных, о них говорят как об ориентированных на обработку событий. Назначение же вторых состоит в объединении приложений и упрощении взаимодействия между ними. О подобных системах часто говорят как об ориентированных на обработку сообщений.

Еще одно важное различие между сообщениями и событиями: сообщения направляются в одно заданное место назначения, а события представляют собой факты, получателями которых выступают все зарегистрированные на их наблюдение компоненты. В реактивных системах важную роль играет асинхронность этих сообщений, благодаря чему операции отправки и получения оказываются расцепленными с отправителем и получателем соответственно. Такое расцепление необходимо для полной изоляции компонентов друг от друга и жизненно важно для сохранения *быстроты реакции* системы как в случае сбоев *(отказоустойчивость)*, так и при высокой нагрузке *(адаптивность)*.

Точнее говоря, отказоустойчивость достигается в реактивных архитектурах за счет изолирования сбоев в пределах компонентов, где они произошли, дабы предотвратить нарушение работоспособности системы вследствие их распространения на соседние компоненты, а оттуда каскадом на всю оставшуюся часть системы. Отказоустойчивость в таком реактивном смысле — не просто нечувствительность к сбоям. Система не прекращает работу, а полностью восстанавливается от последствий сбоя за счет его изоляции и возобновления работоспособного состояния. Это достигается благодаря локализации ошибок и материализации их в виде сообщений, отправляемых другим компонентам, играющим роль наблюдателей. Таким образом, можно обработать проблему из безопасного контекста, внешнего по отношению к сбойному компоненту.

Подобно тому как изоляция и расцепление играют ключевую роль в обеспечении отказоустойчивости, так необходимым условием адаптивности является *прозрачность размещения* (location transparency), благодаря которой любой компонент реактивной системы может взаимодействовать с любым другим сервисом, независимо от местоположения последнего. Прозрачность размещения, в свою очередь, дает возможность репликации системы и (автоматического) масштабирования любого из приложений, в зависимости от текущей нагрузки. Подобное не зависящее от местоположения масштабирование — еще одно отличие между реактивными приложениями (асинхронными, конкурентными и расцепленными по времени) и реактивными системами (допускающими расцепленность по рабочему пространству посредством прозрачности размещения).

Далее в главе мы опробуем часть этих идей на практике, в нескольких примерах реактивного программирования, и, в частности, изучим Flow API Java 9.

17.2. Реактивные потоки данных и Flow API

Реактивное программирование (reactive programming) — программирование с использованием реактивных потоков данных. Реактивные потоки данных — типовая методика (на основе протокола публикации/подписки, о котором мы рассказывали в главе 15) асинхронной обработки потенциально неограниченных потоков дан-

ных последовательно и с принудительным неблокирующим противодавлением. Противодавление — механизм управления движением событий/сообщений, используемый в протоколе публикации/подписки для предотвращения переполнения «медленного» потребителя событий потока данных одним или несколькими более быстрыми генераторами. В подобном случае аварийный сбой проблемного компонента или неконтролируемое игнорирование событий недопустимы. Компоненту необходимо каким-либо образом «попросить» у расположенных ближе по конвейеру генераторов замедлить темпы отправки или сообщить им, сколько событий он может принять и обработать в конкретный момент времени, прежде чем получать остальные данные.

Стоит отметить, что обоснование требования встроенных возможностей противодавления заключается в асинхронной природе потоковой обработки. Фактически при выполнении синхронных вызовов блокирующие API неявно реализуют противодавление. К сожалению, это мешает выполнять другие полезные задачи, пока не закончится блокирующая операция, так что при ожидании впустую расходуется множество ресурсов. И напротив, в случае асинхронных API аппаратное обеспечение используется максимально полно, так что существует риск перегрузить какой-либо медленный компонент, расположенный ниже по конвейеру. В подобной ситуации оказываются полезными противодавление и механизмы управления движением событий/сообщений; они организуют протокол, предотвращающий перегрузку получателей данных без блокировки каких-либо потоков выполнения.

Эти требования и поведение, которое они влекут, собраны воедино в проекте Reactive Streams (http://www.reactive-streams.org), над которым работали инженеры из Netflix, Red Hat, Twitter, Lightbend и других компаний. Он включает определения четырех взаимосвязанных интерфейсов, отражающих минимальный набор возможностей, которые должна предоставлять любая реализация Reactive Streams. Эти интерфейсы уже включены в Java 9 — вложены в класс `java.util.concurrent.Flow` и реализуются множеством сторонних библиотек, в том числе Akka Streams (Lightbend), Reactor (Pivotal), RxJava (Netflix) и Vert.x (Red Hat). В следующем подразделе мы подробно изучим объявленные в этих интерфейсах методы и поясним, как с их помощью выражать реактивные компоненты.

17.2.1. Знакомство с классом Flow

В Java 9 добавился новый класс, предназначенный для реактивного программирования: `java.util.concurrent.Flow`. Этот класс содержит только статические компоненты и не допускает реализации. Класс `Flow` содержит четыре вложенных интерфейса, предназначенных для выражения модели публикации/подписки реактивного программирования, стандартизованной проектом Reactive Streams:

- `Publisher`;
- `Subscriber`;
- `Subscription`;
- `Processor`.

Класс `Flow` дает возможность на основе взаимосвязанных интерфейсов и статических методов формировать управляемые потоком сообщений/событий компоненты, в которых издатели (типа `Publisher`) генерируют элементы, потребляемые одним или несколькими подписчиками (типа `Subscriber`) под управлением подписки (`Subscription`). Издатель генерирует потенциально неограниченное число последовательных событий, сдерживаемое механизмом противодавления для генерации в соответствии с потребностями подписчиков. `Publisher` — функциональный интерфейс Java (объявляет только один абстрактный метод), позволяющий подписчикам (`Subscriber`) регистрироваться в качестве прослушивателей издаваемых им событий; управление потоком сообщений/событий, включая противодавление, между издателем и подписчиками осуществляет подписка (объект `Subscription`). Эти три интерфейса вместе с интерфейсом `Processor` приведены в листингах 17.1–17.4.

Листинг 17.1. Интерфейс издателя: Flow.Publisher

```
@FunctionalInterface
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
```

С другой стороны, у интерфейса `Subscriber` имеется четыре метода обратного вызова, вызываемых издателем при генерации соответствующих событий.

Листинг 17.2. Интерфейс подписчика: Flow.Subscriber

```
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
```

Эти события необходимо генерировать (и вызывать соответствующие методы) в строгом соответствии с задаваемой протоколом последовательностью:

```
onSubscribe onNext* (onError | onComplete)?
```

Эта нотация означает, что в качестве первого события всегда вызывается метод `onSubscribe`, за которым следует произвольное число сигналов `onNext`. Данный поток событий может продолжаться бесконечно, а может быть прерван обратным вызовом `onComplete`, означающим, что больше элементов сгенерировано не будет, или вызовом `onError` в случае сбоя издателя (сравните это с чтением из терминала при таких возможных исходах, как получение строкового значения, или признак конца файла, или ошибка ввода/вывода).

При регистрации подписчика в издателе первое действие издателя — вызов метода `onSubscribe` для передачи обратно подписки (объекта `Subscription`). В интерфейсе `Subscription` объявлено два метода. Первый из них используется подпис-

чиком для уведомления издателя о готовности обработать заданное число событий, второй метод служит для отмены подписки, то есть уведомления издателя о том, что подписчик больше не хочет получать его события.

Листинг 17.3. Интерфейс подписки: Flow.Subscription

```
public interface Subscription {
    void request(long n);
    void cancel();
}
```

Спецификация API Flow Java 9 описывает набор правил взаимодействия реализаций этих интерфейсов. Вот краткая сводка этих правил.

- Издатель обязан отправить подписчику число элементов, не превышающее заданное с помощью метода `request` подписки. Впрочем, издатель может отправить меньше сигналов `onNext`, чем было запрошено, и завершить подписку путем вызова метода `onComplete`, если операция завершилась успешно, или `onError` в противном случае. В подобных ситуациях после достижения конечного состояния (`onComplete` или `onError`) издатель не может больше отправлять каких-либо сигналов подписчикам, а подписка считается отмененной.
- Подписчик обязан оповестить издателя о своей готовности к получению и обработке *n* элементов. Таким образом, подписчик производит противодавление на издателя, предотвращающее переполнение подписчика чрезмерным количеством событий. Более того, при обработке сигналов `onComplete` или `onError` подписчик не может вызывать какие-либо методы издателя или подписки и обязан считать подписку отмененной. Наконец, подписчик должен быть готов к получению этих завершающих сигналов даже без какого-либо предшествующего вызова метода `Subscription.request()` и к получению одного сигнала `onNext` или более даже после того, как вызвал `Subscription.cancel()`.
- Подписка совместно используется ровно одним издателем и подписчиком и отражает уникальную взаимосвязь между ними. Поэтому она должна разрешать синхронный вызов подписчиком своего метода `request`, как из метода `onSubscribe`, так и из метода `onNext`. Стандарт определяет, что реализация метода `Subscription.cancel()` обязана быть идемпотентной (результат нескольких вызовов не отличается от однократного вызова) и потокобезопасной, так что после первого ее вызова никакие последующие вызовы эффекта не дадут. Вызов этого метода требует от издателя в конечном итоге игнорировать любые ссылки на соответствующего подписчика. Повторная подписка с тем же объектом `Subscriber` не рекомендуется, но спецификация не требует генерации в этом случае исключения, поскольку при этом не понадобилось бы хранение всех отмененных подписок в течение неограниченного времени.

На рис. 17.3 приведен типичный жизненный цикл приложения, реализующего определяемые Flow API интерфейсы.

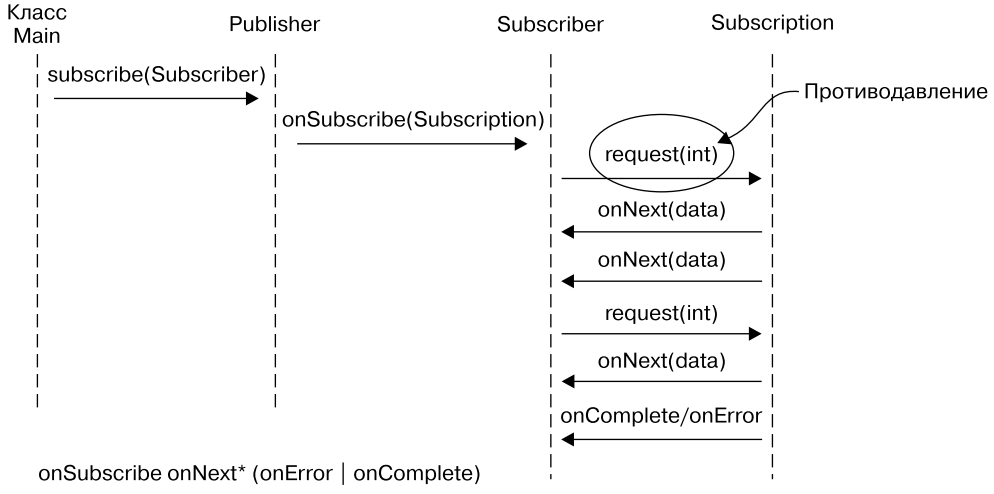


Рис. 17.3. Жизненный цикл реактивного приложения, использующего Flow API

Четвертый, последний член класса `Flow` — интерфейс `Processor`, расширяющий интерфейсы `Publisher` и `Subscriber` и не требующий каких-либо дополнительных методов.

Листинг 17.4. Интерфейс обработчика: Flow.Processor

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

Фактически этот интерфейс представляет этап преобразования событий в реактивном потоке данных. При получении сообщения об ошибке обработчик может либо восстановиться после нее (а затем считать подписку отмененной), либо транслировать сигнал `onError` его подписчикам. Обработчик должен также отменить расположенную ближе по конвейеру подписку при отмене подписки последним из своих подписчиков, чтобы транслировать сигнал отмены (хотя такая отмена необязательна согласно спецификации).

API Flow/API Reactive Streams Java 9 требуют, чтобы все реализации всех методов интерфейса `Subscriber` никогда не блокировали издателя, но не указывают, должны ли эти методы обрабатывать события синхронно или асинхронно. Отметим, впрочем, что все описанные в этих интерфейсах методы возвращают `void`, так что их можно реализовать совершенно асинхронно.

В следующем подразделе мы попытаемся применить на практике все вышеизложенное на простом реальном примере.

17.2.2. Создаем ваше первое реактивное приложение

Описанные в классе `Flow` интерфейсы в большинстве случаев не предназначены для непосредственной реализации. На удивление, библиотека Java 9 также не содержит реализующих их классов! Эти интерфейсы реализованы в уже упоминавшихся

нами реактивных библиотеках (Akka, RxJava и т. д.). Спецификация `java.util.concurrency.Flow` языка Java 9 работает как контракт, которого должны придерживаться все эти библиотеки, так же как лингва-франка, позволяющий реактивным приложениям, разработанным на основе различных реактивных библиотек, работать совместно и взаимодействовать друг с другом. Более того, возможности этих реактивных библиотек обычно намного шире (включая классы и методы для преобразования и слияния реактивных потоков данных, далеко выходящие за пределы минимального подмножества, специфицируемого классом `java.util.concurrency.Flow`).

С учетом этого имеет смысл разработать первое наше реактивное приложение непосредственно на основе API Flow Java 9, чтобы прочувствовать, как обсуждавшиеся ранее четыре интерфейса работают совместно. С этой целью мы напишем на основе реактивных принципов простое приложение для сводок температуры. Эта программа включает два компонента:

- класс `TempInfo`, имитирующий удаленный термометр (который непрерывно выдает отчеты о случайно выбираемых значениях температуры в диапазоне от 0 до 99 градусов по Фаренгейту[1], подходящем для городов США);
- класс `TempSubscriber`, просматривающий эти отчеты и выводящий поток температур, сообщаемых установленным в заданном городе датчиком.

Первый шаг — описание простого класса для передачи сообщаемой в настоящий момент температуры, как показано в листинге 17.5.

Листинг 17.5. Java-компонент для передачи текущего отчета о температуре

```
import java.util.Random;

public class TempInfo {

    public static final Random random = new Random();

    private final String town;
    private final int temp;

    public TempInfo(String town, int temp) {
        this.town = town;
        this.temp = temp;
    }

    public static TempInfo fetch(String town) {
        if (random.nextInt(10) == 0)
            throw new RuntimeException("Error!");
        return new TempInfo(town, random.nextInt(100));
    }

    @Override
    public String toString() {
```

Создаем экземпляр TempInfo для заданного города с помощью статического фабричного метода

В одном случае из десяти (выбираемом случайным образом) в операции получения текущей температуры происходит сбой

Возвращает случайную температуру в диапазоне от 0 до 99 градусов по Фаренгейту

[1] Приблизительно от –17 до 37 градусов по шкале Цельсия. — *Примеч. пер.*

```
        return town + " : " + temp;
    }

    public int getTemp() {
        return temp;
    }

    public String getTown() {
        return town;
    }
}
```

После описания этой простой модели предметной области можно реализовать интерфейс `Subscription` для подписки на отчет о температурах в заданном городе. Он бы отправлял отчет при каждом запросе его подписчиком, как показано в листинге 17.6.

Листинг 17.6. Подписка, отправляющая своему подписчику поток объектов TempInfo

```
import java.util.concurrent.Flow.*;

public class TempSubscription implements Subscription {

    private final Subscriber<? super TempInfo> subscriber;
    private final String town;

    public TempSubscription( Subscriber<? super TempInfo> subscriber,
                             String town ) {
        this.subscriber = subscriber;
        this.town = town;
    }

    @Override
    public void request( long n ) {
        for (long i = 0L; i < n; i++) {
            try {
                subscriber.onNext( TempInfo.fetch( town ) );
            } catch (Exception e) {
                subscriber.onError( e );
                break;
            }
        }
    }

    @Override
    public void cancel() {
        subscriber.onComplete();
    }
}
```

Извлекаем данные в цикле в соответствии с запросом подписчика

Отправляем данные о текущей температуре подписчику

В случае сбоя при извлечении данных о температуре транслируем ошибку подписчику

В случае отмены подписки отправляем подписчику сигнал завершения (onComplete)

Следующий шаг — создание подписчика, который при каждом получении нового элемента выводил бы полученные от подписки температуры и запрашивал новый отчет, как показано в листинге 17.7.

Листинг 17.7. Подписчик, выводящий в консоль полученные данные о температуре

```
import java.util.concurrent.Flow.*;

public class TempSubscriber implements Subscriber<TempInfo> {

    private Subscription subscription;

    @Override
    public void onSubscribe( Subscription subscription ) {
        this.subscription = subscription;
        subscription.request( 1 );
    }

    @Override
    public void onNext( TempInfo tempInfo ) {
        System.out.println( tempInfo );
        subscription.request( 1 );
    }

    @Override
    public void onError( Throwable t ) {
        System.err.println(t.getMessage());
    }

    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
}
```

Сохраняет подписку и отправляет первый запрос

Выводит полученный отчет о температуре в консоль и запрашивает следующий

Выводит сообщение об ошибке в случае ее возникновения

В листинге 17.8 мы заставим наше реактивное приложение работать, описав класс `Main`, в котором создается издатель и выполняется подписка на него объекта `TempSubscriber`.

Листинг 17.8. Класс Main: создание издателя и подписка на него TempSubscriber

```
import java.util.concurrent.Flow.*;

public class Main {
    public static void main( String[] args ) {
        getTemperatures( "New York" ).subscribe( new TempSubscriber() );
    }

    private static Publisher<TempInfo> getTemperatures( String town ) {
        return subscriber -> subscriber.onSubscribe(
                                new TempSubscription( subscriber, town ) );
    }
}
```

Создаем нового издателя для температур в Нью-Йорке и подписываем на него подписчика типа TempSubscriber

Возвращаем издателя, отправляющего подписку типа TempSubscription подписанному на него TempSubscriber

В этом листинге метод `getTemperatures` возвращает лямбда-выражение, принимающее в качестве аргумента подписчика (объект `TempSubscriber`) и вызывающее его метод `onSubscribe`, с передачей последнему нового экземпляра `TempSubscription`.

А поскольку сигнатура этого лямбда-выражения совпадает с единственным абстрактным методом функционального интерфейса `Publisher`, то компилятор Java может автоматически преобразовать это лямбда-выражение в объект `Publisher` (как мы знаем из главы 3). Метод `main` создает издателя для температур в Нью-Йорке, после чего подписывает на него новый экземпляр класса `TempSubscriber`. Результаты запуска метода `main` могут выглядеть, например, вот так:

```
New York : 44
New York : 68
New York : 95
New York : 30
Error!
```

При этом запуске экземпляр `TempSubscription` успешно получил температуру в Нью-Йорке четыре раза, а при пятой попытке чтения произошел сбой. Похоже, мы правильно решили нашу задачу с помощью трех из четырех интерфейсов Flow API. Но как убедиться, что в коде нет никаких ошибок? Поразмышляйте над этим вопросом и выполните следующее контрольное задание.

Контрольное задание 17.1

В разработанном выше примере есть одна малозаметная проблема. Впрочем, она скрыта, поскольку поток температур в какой-то случайный момент времени прерывается сгенерированной внутри фабричного метода `TempInfo` ошибкой. Как вы думаете, что произойдет, если закомментировать оператор случайной генерации ошибки и позволить методу `main` выполняться достаточно долго?

Ответ: проблема с написанным выше кодом в том, что всякий раз, когда подписчик `TempSubscriber` получает новый элемент в методе `onNext`, он отправляет новый запрос подписке `TempSubscription`, после чего метод `request` отправляет подписчику `TempSubscriber` еще один элемент. Эти рекурсивные вызовы помещаются один за другим в стек до тех пор, пока стек не окажется переполнен и не будет сгенерировано исключение `StackOverflowError`, примерно вот так:

```
Exception in thread "main" java.lang.StackOverflowError
    at java.base/java.io.PrintStream.print(PrintStream.java:666)
    at java.base/java.io.PrintStream.println(PrintStream.java:820)
    at flow.TempSubscriber.onNext(TempSubscriber.java:36)
    at flow.TempSubscriber.onNext(TempSubscriber.java:24)
    at flow.TempSubscription.request(TempSubscription.java:60)
    at flow.TempSubscriber.onNext(TempSubscriber.java:37)
    at flow.TempSubscriber.onNext(TempSubscriber.java:24)
    at flow.TempSubscription.request(TempSubscription.java:60)
 ...
```

Как же решить эту проблему и избежать переполнения стека вызовов? Одно из возможных решений: добавить в подписку `TempSubscription` исполнитель и отправлять с его помощью новые элементы подписчику `TempSubscriber` из другого потока

выполнения. Для этого необходимо модифицировать класс `TempSubscription` так, как показано в листинге 17.9 (это лишь часть класса; полное описание включает оставшиеся описания из листинга 17.6).

Листинг 17.9. Добавляем исполнитель в класс TempSubscription

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TempSubscription implements Subscription {

    private static final ExecutorService executor =
                                        Executors.newSingleThreadExecutor();

    @Override
    public void request( long n ) {
        executor.submit( () -> {
            for (long i = 0L; i < n; i++) {
                try {
                    subscriber.onNext( TempInfo.fetch( town ) );
                } catch (Exception e) {
                    subscriber.onError( e );
                    break;
                }
            }
        });
    }
}
```

Мы опустили здесь непоменявшиеся части кода исходного класса TempSubscription

Отправляем следующие элементы подписчику из другого потока выполнения

До сих пор мы использовали только три из четырех объявленных API Flow интерфейсов. А как насчет интерфейса `Processor`? Хороший пример для демонстрации работы этого интерфейса — создание издателя, выдающего отчеты о температуре по шкале Цельсия, а не шкале Фаренгейта (для подписчиков за пределами США).

17.2.3. Преобразование данных с помощью интерфейса Processor

Как описывалось в подразделе 17.2.1, обработчик совмещает в себе черты издателя и подписчика. Он предназначен для подписки на издателя, и повторной публикации полученных от того данных после их преобразования. В качестве примера его применения реализуем обработчик, который подписывается на издателя, выдающего температуру в градусах по Фаренгейту, и повторно публикует их после преобразования в градусы по Цельсию, как показано в листинге 17.10.

Листинг 17.10. Обработчик, преобразующий градусы по Фаренгейту в градусы по Цельсию

```
import java.util.concurrent.Flow.*;

public class TempProcessor implements Processor<TempInfo, TempInfo> {

    private Subscriber<? super TempInfo> subscriber;
```

Обработчик, преобразующий один объект TempInfo в другой

```
    @Override
    public void subscribe( Subscriber<? super TempInfo> subscriber ) {
        this.subscriber = subscriber;
    }

    @Override
    public void onNext( TempInfo temp ) {
        subscriber.onNext( new TempInfo( temp.getTown(),
                                         (temp.getTemp() - 32) * 5 / 9) );
    }

    @Override
    public void onSubscribe( Subscription subscription ) {
        subscriber.onSubscribe( subscription );
    }

    @Override
    public void onError( Throwable throwable ) {
        subscriber.onError( throwable );
    }

    @Override
    public void onComplete() {
        subscriber.onComplete();
    }
}
```

Повторно публикует объект TempInfo после преобразования температуры в градусы по Цельсию

Все остальные сигналы передаются в неизменном виде подписчику, расположенному ближе по конвейеру

Отметим, что какую-либо бизнес-логику содержит только один метод класса `TempProcessor` — `onNext`, который повторно публикует температуру после преобразования ее из градусов по шкале Фаренгейта в градусы по шкале Цельсия. Все остальные реализующие интерфейс `Subscriber` методы просто передают (делегируют) все полученные сигналы в неизменном виде расположенному ближе по конвейеру подписчику, а метод `subscribe` издателя регистрирует в обработчике расположенного ближе по конвейеру подписчика.

В листинге 17.11 мы применим класс `TempProcessor`, воспользовавшись им в нашем классе `Main`.

Листинг 17.11. Класс Main: создание издателя и подписка TempSubscriber на него

```
import java.util.concurrent.Flow.*;

public class Main {
    public static void main( String[] args ) {
        getCelsiusTemperatures( "New York" )
            .subscribe( new TempSubscriber() );
    }

    public static Publisher<TempInfo> getCelsiusTemperatures(String town) {
        return subscriber -> {
            TempProcessor processor = new TempProcessor();
```

Создаем нового издателя для температур в Нью-Йорке в градусах по Цельсию

Подписываем подписчика TempSubscriber на этого издателя

```
            processor.subscribe( subscriber );
            processor.onSubscribe( new TempSubscription(processor, town) );
        };
    }
}
```

Создаем экземпляр TempProcessor и помещаем его между подписчиком и возвращенным издателем

На этот раз запуск метода `main` приводит к следующему выводу в консоль с обычными для шкалы Цельсия температурами:

```
New York : 10
New York : -12
New York : 23
Error!
```

В этом разделе мы непосредственно реализовали описанные в Flow API интерфейсы и при этом познакомились с асинхронной потоковой обработкой на основе протокола публикации/подписки, образующего фундамент Flow API. Но в этом примере было кое-что странное, о чем мы и поговорим в следующем подразделе.

17.2.4. Почему Java не предоставляет реализации Flow API

Flow API в Java 9 довольно странный. Библиотека Java обычно включает реализации для интерфейсов, но в данном случае разработчику приходится реализовывать Flow API самому. Давайте сравним его с List API. Как вы знаете, в Java есть интерфейс `List<T>`, реализованный множеством классов, включая `ArrayList<T>`. Если точнее, класс `ArrayList<T>` расширяет (причем довольно незаметно для пользователя) абстрактный класс `AbstractList<T>`, реализующий интерфейс `List<T>`. В противоположность этому Java 9 объявляет интерфейс `Publisher<T>`, но не дает никакой его реализации[1], отчего (если не считать педагогической ценности) разработчик должен описать свою собственную. Будем откровенны: сам по себе интерфейс может помочь упорядочить мысли программиста, но вовсе не ускоряет процесса написания программ!

В чем здесь дело? Разгадка — в истории Java: существовало несколько библиотек реактивных потоков данных с кодом на Java (таких как Akka и RxJava). Изначально они разрабатывались отдельно, и хотя реализовывали реактивное программирование на основе протокола публикации/подписки, но использовали различные API и спецификации. В процессе стандартизации Java 9 эти библиотеки эволюционировали так, что их классы формально стали реализовывать интерфейсы из `java.util.concurrent.Flow`, а не просто реактивные идеи. Этот стандарт упрощает взаимодействие различных библиотек.

Отметим, что создание реализаций реактивных потоков данных — непростая задача, так что большинство пользователей применяют уже существующие. Подобно большинству реализующих интерфейс классов, обычно они предоставляют больше возможностей, чем необходимо для минимальной реализации.

---

[1] Отметим, что, начиная с Java 9, существует класс SubmissionPublisher, реализующий интерфейс Flow.Publisher (https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/SubmissionPublisher.html). — *Примеч. пер.*

В следующем разделе мы воспользуемся одной из чаще всего применяемых библиотек: разработанной Netflix библиотекой RxJava (Reactive eXtensions to Java, реактивные расширения Java), а именно текущей ее версией RxJava 2.0, реализующей интерфейсы класса `Flow` Java 9.

17.3. Реактивная библиотека RxJava

RxJava входит в число первых библиотек для разработки реактивных приложений на языке Java. Она появилась в результате адаптации компанией Netflix разработанного Microsoft в среде .NET проекта Reactive Extensions (Rx). Версия 2.0 RxJava была модифицирована так, чтобы соответствовать упоминавшемуся ранее в данной главе API реактивных потоков данных, ставшему в Java 9 классом `java.util.concurrent.Flow`.

Использование в Java внешней библиотеки хорошо заметно по инструкциям импорта. Например, для импорта интерфейсов класса `Flow` Java, включая `Publisher`, необходимо написать следующую строку:

```
import java.lang.concurrent.Flow.*;
```

Также необходимо импортировать соответствующие классы реализаций, например:

```
import io.reactivex.Observable;
```

чтобы иметь возможность использовать реализацию `Observable` интерфейса `Publisher`, что мы будем делать далее в этой главе.

Необходимо отметить один архитектурный нюанс: хорошо разработанные системы избегают видимости в масштабах всей системы каких-либо мелкомодульных концепций, используемых только в одной части системы. Соответственно, рекомендуется применять `Observable` только там, где необходимы дополнительные структуры `Observable`, а во всех прочих местах использовать интерфейс `Publisher`. Обратите внимание, что мы, даже не задумываясь, следовали этой рекомендации при работе с интерфейсом `List`. И хотя в метод может передаваться значение типа `ArrayList`, мы объявляли параметр для этого значения типа `List`, избегая, таким образом, раскрытия и ограничения подробностей реализации. Поэтому последующее изменение реализации с типа `ArrayList` на тип `LinkedList` не потребовало глобальных изменений.

В оставшейся части данного раздела мы опишем систему для отчетов о температуре с помощью реализации реактивных потоков данных библиотеки RxJava. Первая проблема, с которой нам предстоит столкнуться, — наличие в RxJava двух различных классов, реализующих интерфейс `Flow.Publisher`.

Прочитав документацию RxJava, вы обнаружите, что один из этих классов — `io.reactivex.Flowable`, включающий реактивные возможности противодавления класса `Flow` Java 9 (с помощью метода `request`), проиллюстрированные листингами 17.7 и 17.9. Противодавление предотвращает переполнение подписчика данными, генерируемыми быстрым издателем. Второй класс — первоначальная версия издателя библиотеки RxJava, `io.reactivex.Observable`, без поддержки противодавления. Этот

класс проще в программировании и лучше подходит для событий пользовательского интерфейса (таких как перемещения мыши); эти события представляют собой потоки данных, для которых невозможна разумная стратегия противодавления (нельзя же попросить пользователя замедлить движения или перестать двигать мышку!). Поэтому RxJava предоставляет два класса, реализующих одну идею потока событий.

RxJava рекомендует использовать класс `Observable`, в котором нет противодавления, если речь идет о потоке данных не более чем из тысячи элементов или о работе с событиями GUI, например с перемещениями мыши или с событиями касания, которые не поддаются противодавлению и в любом случае происходят не очень часто.

Поскольку мы уже анализировали сценарий противодавления, когда обсуждали Flow API в предыдущем разделе, то не будем обсуждать класс `Flowable`; вместо этого продемонстрируем интерфейс `Observable` на деле, в сценарии использования без противодавления. Стоит отметить, что для фактического отключения противодавления любому подписчику достаточно выполнить для подписки вызов `request(Long.MAX_VALUE)`, хотя это и не рекомендуется, разве что вы уверены, что подписчик всегда сможет обработать все полученные события достаточно быстро.

17.3.1. Создание и использование наблюдаемых объектов

Классы `Observable` и `Flowable` содержат множество удобных фабричных методов, позволяющих создавать различные типы реактивных потоков данных (и `Observable` и `Flowable` реализуют интерфейс `Publisher`, так что эти фабричные методы публикуют реактивные потоки данных).

Простейший из наблюдаемых объектов, которые вам может понадобиться создать, состоит из фиксированного числа заранее определенных элементов:

```
Observable<String> strings = Observable.just( "first", "second" );
```

Фабричный метод[1] `just` здесь служит для преобразования одного или нескольких элементов в наблюдаемый объект, который далее эти элементы выдает. Подписчик этого наблюдаемого объекта получит сообщения `onNext("first")`, `onNext("second")` и `onComplete()` — именно в таком порядке.

Еще один весьма распространенный, особенно в случае, когда приложение взаимодействует с пользователем в режиме реального времени, фабричный метод класса `Observable` выдает события с фиксированной частотой:

```
Observable<Long> onePerSec = Observable.interval(1, TimeUnit.SECONDS);
```

Фабричный метод `interval` возвращает наблюдаемый объект с названием `onePerSec`, выдающий бесконечную последовательность возрастающих значений типа `long`, начиная с 0, через фиксированные промежутки времени (1 секунда в этом примере). Теперь этот `onePerSec` можно использовать в качестве основы другого

[1] Соглашение о наименованиях здесь несколько неудачное, ведь, начиная с Java 8, для подобных фабричных методов используется название of(), широко распространившееся благодаря Stream API и Optional API.

наблюдаемого объекта, который будет выдавать отчеты о температуре в заданном городе каждую секунду.

В качестве промежуточного этапа на пути к этой цели можно выводить в консоль температуру каждую секунду. Для этого нам нужно подписаться на объект `onePerSec`, получать от него уведомления об истечении каждой очередной секунды, и извлекать и выводить в консоль температуру в интересующем нас городе. В библиотеке RxJava класс `Observable`[1] (наблюдаемый объект) играет роль интерфейса `Publisher` в Flow API, а интерфейс `Observer` (наблюдатель) соответствует интерфейсу `Subscriber` класса `Flow`. В интерфейсе `Observer` библиотеки RxJava объявлены те же методы, что и в интерфейсе `Subscriber` Java 9, приведенном в листинге 17.2, с тем отличием, что метод `onSubscribe` включает аргумент типа `Disposable`, а не `Subscription`. Как мы уже упоминали ранее, класс `Observable` не поддерживает противодавление, так что у него отсутствует метод `request`, включенный в `Subscription`. Полное описание интерфейса `Observer` выглядит следующим образом:

```
public interface Observer<T> {
    void onSubscribe(Disposable d);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
  }
```

Отметим, впрочем, что API RxJava — более гибкий (насчитывает больше перегруженных вариантов методов), чем нативный Flow API Java 9. Например, подписаться на наблюдаемый объект можно путем передачи лямбда-выражения с сигнатурой метода `onNext`, опустив остальные три метода. Другими словами, для подписки на объект `Observable` достаточно экземпляра `Observer`, реализующего лишь метод `onNext` с потребителем получаемого события, причем для остальных методов используются ничего не делающие реализации по умолчанию для завершения и обработки ошибок. Благодаря этой возможности можно подписаться на наблюдаемый объект `onePerSec` и выводить с его помощью температуру в Нью-Йорке раз в секунду, и все это в одной строке кода:

```
onePerSec.subscribe(i -> System.out.println(TempInfo.fetch( "New York" )));
```

В этом операторе наблюдаемый объект `onePerSec` выдает по одному событию в секунду, при получении которого подписчик извлекает температуру в Нью-Йорке и выводит ее в консоль. Впрочем, если вы поместите этот оператор в метод `Main` и запустите его, то ничего не увидите, поскольку наблюдаемый объект, публикующий по одному событию в секунду, выполняется в потоке выполнения из пула потоков RxJava, состоящего из потоков-демонов[2]. Но выполнение нашей программы `main`

---

[1] Обратите внимание, что интерфейс Observer и класс Observable считаются устаревшими, начиная с Java 9. Более новый код должен использовать Flow API. Посмотрим, как в этой связи будет развиваться RxJava.

[2] Из документации это не вполне ясно, хотя можно найти упоминания о данной особенности на сайте онлайн-сообщества разработчиков stackoverflow.com.

завершается сразу же и при этом прерывает выполнение потока-демона, прежде чем он сможет вывести какие-либо результаты.

С помощью небольшого трюка можно предотвратить немедленное завершение выполнения, приостановив поток выполнения после предыдущего оператора. А еще лучше будет воспользоваться методом `blockingSubscribe`, вызывающим в текущем потоке выполнения (в данном случае в основном потоке выполнения) функции обратного вызова. Для наших демонстрационных целей метод `blockingSubscribe` отлично подходит. При промышленной эксплуатации, впрочем, обычно используется метод `subscribe`, вот так:

```
onePerSec.blockingSubscribe(
    i -> System.out.println(TempInfo.fetch( "New York" ))
);
```

Результаты будут выглядеть примерно следующим образом:

```
New York : 87
New York : 18
New York : 75
java.lang.RuntimeException: Error!
at flow.common.TempInfo.fetch(TempInfo.java:18)
at flow.Main.lambda$main$0(Main.java:12)
at io.reactivex.internal.observers.LambdaObserver
           .onNext(LambdaObserver.java:59)
at io.reactivex.internal.operators.observable
           .ObservableInterval$IntervalObserver.run(ObservableInterval.java:74)
```

К сожалению, по нашему замыслу при извлечении температуры происходит случайный сбой (что и показано выше, после третьей строки). А поскольку наш наблюдатель реализует только оптимистический вариант и в нем нет никакой обработки ошибок вроде реализации метода `onError`, то этот сбой доходит до пользователя в виде неперехваченного исключения.

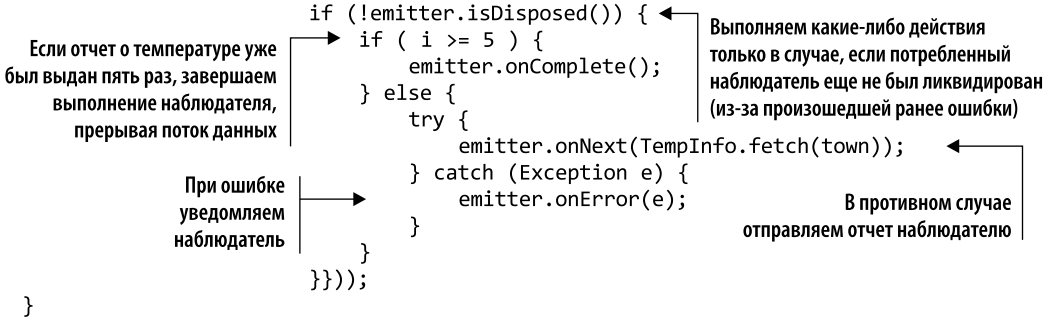
Пора поднять планку и немного усложнить наш пример. Необходимо добавить не только обработку ошибок. Нужно также обобщить уже сделанное. Желательно не выводить температуру сразу же, а предоставить пользователям фабричный метод, который бы возвращал наблюдаемый объект, выдающий эти показания температуры раз в секунду, скажем, не более пяти раз перед завершением выполнения. Это легко сделать с помощью фабричного метода `create`, предназначенного для создания объекта `Observable` из лямбда-выражения, принимающего в качестве аргумента объект `Observer` и возвращающего `void`, как показано в листинге 17.12.

Листинг 17.12. Создание наблюдаемого объекта, выдающего показания температуры раз в секунду

Создаем наблюдаемый объект (Observable) из функции, потребляющей наблюдатель (Observer)

Наблюдаемый объект, выдающий бесконечную последовательность возрастающих значений типа long по одному в секунду

```
public static Observable<TempInfo> getTemperature(String town) {
    return Observable.create(emitter ->
              Observable.interval(1, TimeUnit.SECONDS)
                    .subscribe(i -> {
```



Здесь мы создаем возвращаемый наблюдаемый объект на основе функции, которая потребляет объект типа `ObservableEmitter`, отправляя в него нужные события. Интерфейс `ObservableEmitter` библиотеки RxJava расширяет ее простой интерфейс `Emitter`, который можно рассматривать как `Observer` без метода `onSubscribe`:

```
public interface Emitter<T> {
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
```

и с несколькими дополнительными методами с целью установки для `Emitter` нового `Disposable` и проверки, не была ли последовательность уже ликвидирована далее по конвейеру.

«Под капотом» мы подписываемся на наблюдаемый объект, например, `onePerSec`, публикующий бесконечную последовательность возрастающих значений типа `long` по одному в секунду. Внутри функции, производящей подписку (передается как аргумент в метод `subscribe`), сначала проверяется, не был ли потребленный наблюдатель уже ликвидирован с помощью предназначенного для такой цели метода `isDisposed` интерфейса `ObservableEmitter` (это возможно, если на более ранней итерации произошла ошибка). Если отчеты о температуре уже были выданы пять раз, код завершает наблюдатель, прерывая поток данных; в противном случае он отправляет наиболее свежий отчет о температуре для требуемого города наблюдателю в блоке `try/catch`. Если во время извлечения данных о температуре происходит ошибка, то она транслируется наблюдателю.

Теперь нам не составит труда реализовать полноценный наблюдатель, которым мы затем воспользуемся для подписки на возвращаемый методом `getTemperature` наблюдаемый объект и вывода в консоль публикуемых им данных о температуре, как показано в листинге 17.13.

Листинг 17.13. Наблюдатель, выводящий в консоль полученные данные о температуре

```
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;

public class TempObserver implements Observer<TempInfo> {
```

```
    @Override
    public void onComplete() {
        System.out.println( "Done!" );
    }

    @Override
    public void onError( Throwable throwable ) {
        System.out.println( "Got problem: " + throwable.getMessage() );
    }

    @Override
    public void onSubscribe( Disposable disposable ) {
    }

    @Override
    public void onNext( TempInfo tempInfo ) {
        System.out.println( tempInfo );
    }
}
```

Этот наблюдатель напоминает класс `TempSubscriber` из листинга 17.7 (который был реализацией интерфейса `Flow.Subscriber` Java 9), но немного упрощенный. Поскольку наблюдаемые объекты RxJava не должны поддерживать противодавление, можно не запрашивать последующие элементы после обработки опубликованных.

В листинге 17.14 мы создадим основную программу, в которой подпишем этот наблюдатель на наблюдаемый объект, возвращаемый методом `getTemperature` из листинга 17.12.

Листинг 17.14. Класс Main, выводящий в консоль температуру в Нью-Йорке

```
public class Main {
    public static void main(String[] args) {
        Observable<TempInfo> observable = getTemperature( "New York" );
        observable.blockingSubscribe( new TempObserver() );

    }
}
```

Создаем наблюдаемый объект, выдающий отчеты о температуре в Нью-Йорке раз в секунду

Подписываем на этот наблюдаемый объект простой наблюдатель для вывода температуры в консоль

Если на этот раз при извлечении температуры не произойдет никакой ошибки, метод `main` пять раз с интервалом в секунду выведет в консоль по строке, после чего наблюдаемый объект выдаст сигнал `onComplete`, так что результаты будут выглядеть примерно следующим образом:

```
New York : 69
New York : 26
New York : 85
New York : 94
New York : 29
Done!
```

Пора еще немного усовершенствовать наш пример RxJava, в частности посмотреть, какие операции над одним или несколькими реактивными потоками данных позволяет выполнять эта библиотека.

17.3.2. Преобразование и группировка наблюдаемых объектов

Одно из главных преимуществ RxJava и других реактивных библиотек при работе с реактивными потоками данных, по сравнению с возможностями нативного API Flow Java 9 — расширенный набор функций для группировки, создания и фильтрации любых потоков данных. Как мы уже продемонстрировали в предыдущих подразделах, поток данных может служить источником данных для другого потока. Вы также узнали про интерфейс `Flow.Processor` Java 9, использовавшийся в подразделе 17.2.3 для преобразования температуры из градусов по шкале Фаренгейта в градусы по шкале Цельсия. Но, кроме этого, можно и отфильтровать поток данных, получив из него поток, содержащий только интересующие вас элементы, преобразовать эти элементы с помощью заданной функции отображения (и то и другое можно сделать с помощью `Flow.Processor`) или даже сливать и объединять потоки данных различными способами (чего нельзя сделать с помощью `Flow.Processor`).

Подобные функции преобразования и объединения могут быть довольно сложными, до такой степени, что объяснение их работы на обычном языке будет представлять собой набор неуклюжих, запутанных предложений. В качестве примера взгляните на документацию RxJava для функции `mergeDelayError`:

> *«Схлопывает наблюдаемый объект, выдающий наблюдаемые объекты, в один наблюдаемый объект, таким образом, чтобы наблюдатель мог получать все успешно сгенерированные элементы от всех наблюдаемых объектов-источников без прерываний в виде уведомлений об ошибках от какого-либо из них, ограничивая в то же время число конкурентных подписок на эти наблюдаемые объекты».*

Согласитесь, что далеко не сразу можно понять, что делает функция. Для борьбы с этой проблемой сообщество разработчиков реактивных потоков данных приняло решение документировать поведение подобных функций наглядно, с помощью «мраморных» диаграмм. «Мраморная» диаграмма наподобие той, что показана на рис. 17.4, отражает временно упорядоченную последовательность элементов в реактивном потоке данных в виде геометрических фигур на горизонтальной линии; сигналам ошибки и завершения при этом соответствуют специальные символы. Прямоугольники указывают, каким образом поименованные операторы преобразуют эти элементы или выполняют объединение нескольких потоков данных.

С помощью таких обозначений можно с легкостью наглядно изобразить все возможности функций библиотеки RxJava, как показано на рис. 17.5, иллюстрирующем операции `map` (предназначенную для преобразования публикуемых наблюдаемым объектом элементов) и `merge` (предназначенную для группировки в одно событие выдаваемых двумя или более наблюдаемыми объектами событий).

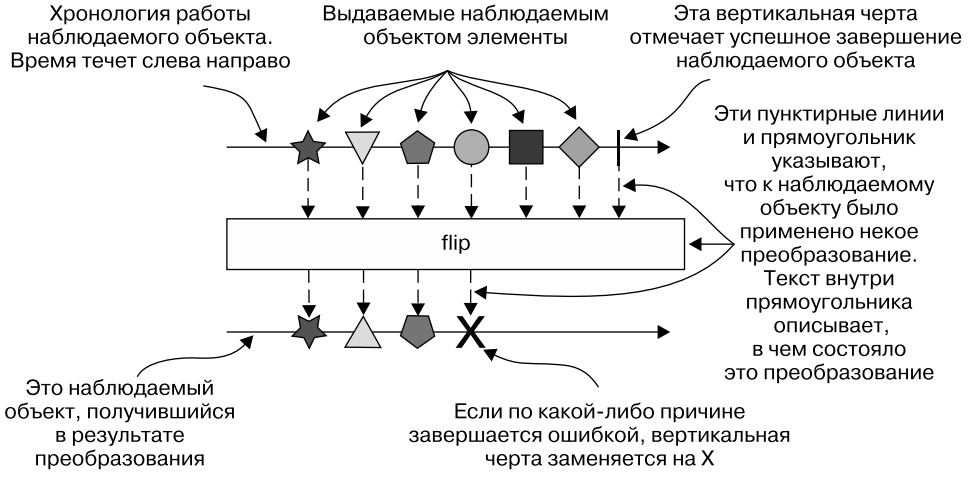


Рис. 17.4. Легенда «мраморной» диаграммы, документирующей оператор из типичной реактивной библиотеки

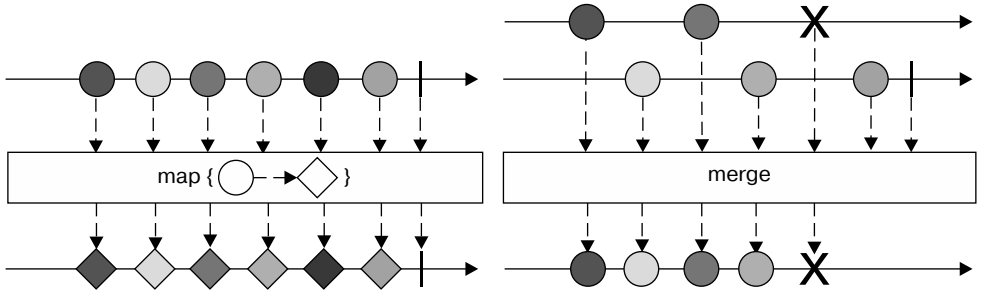


Рис. 17.5. «Мраморные» диаграммы для функций map и merge

Наверное, вам интересно, как с помощью операций `map` и `merge` усовершенствовать разработанный в предыдущем разделе пример RxJava и добавить в него новые возможности. Использование `map` позволяет описать преобразование из шкалы Фаренгейта в шкалу Цельсия более лаконично, по сравнению с использовавшимся выше интерфейсом `Processor` Flow API, как показано в листинге 17.15.

Листинг 17.15. Использование функции map для преобразования градусов по Фаренгейту в градусы по Цельсию

```
public static Observable<TempInfo> getCelsiusTemperature(String town) {
    return getTemperature( town )
                .map( temp -> new TempInfo( temp.getTown(),
                                            (temp.getTemp() - 32) * 5 / 9) );
}
```

Этот несложный метод принимает на входе наблюдаемый объект, возвращаемый методом `getTemperature` из листинга 17.12, и возвращает еще один наблюдаемый объект, который заново выдает опубликованную предыдущим наблюдаемым объектом температуру (с интервалом в секунду) после преобразования градусов по шкале Фаренгейта в градусы по шкале Цельсия.

Чтобы лучше разобраться в возможных операциях над выдаваемыми наблюдаемым объектом элементами, попробуйте воспользоваться еще одной функцией преобразования в следующем контрольном задании.

Контрольное задание 17.2. Отфильтровываем только отрицательную температуру

Метод `filter` класса `Observable` принимает в качестве аргумента предикат и возвращает наблюдаемый объект, выдающий только удовлетворяющие критерию этого предиката элементы. Пускай вам нужно разработать систему оповещений пользователя о риске гололедицы. Как с помощью этого оператора создать наблюдаемый объект, который бы выдавал зарегистрированную в нужном городе температуру по шкале Цельсия лишь в том случае, если она ниже нуля? Очень удобно, что за 0 по шкале Цельсия принимается точка замерзания воды.

Ответ: для решения этой задачи достаточно применить к возвращаемому методом из листинга 17.15 наблюдаемому объекту оператор `filter` с предикатом, пропускающим лишь отрицательную температуру:

```
public static Observable<TempInfo> getNegativeTemperature(String town) {
    return getCelsiusTemperature( town )
                .filter( temp -> temp.getTemp() < 0 );
}
```

Представьте себе теперь, что вас попросили обобщить этот метод и предоставить пользователям наблюдаемый объект, который бы выдавал температуру не для одного, а для целого набора городов. Эту задачу решает листинг 17.16, в котором метод из листинга 17.15 вызывается по одному разу для каждого города и все полученные в результате этих вызовов наблюдаемые объекты объединяются в один с помощью функции `merge`.

Листинг 17.16. Слияние отчетов о температуре для одного или нескольких городов

```
public static Observable<TempInfo> getCelsiusTemperatures(String... towns) {
    return Observable.merge(Arrays.stream(towns)
                                  .map(TempObservable::getCelsiusTemperature)
                                  .collect(toList()));
}
```

Этот метод принимает аргумент переменной длины со списком городов, для которых необходимо получить температуру. Данный аргумент переменной длины преоб-

разуется в поток строковых значений, после чего каждое из этих значений передается методу `getCelsiusTemperature` из листинга 17.11 (который мы усовершенствовали в листинге 17.15). Таким образом, каждый из городов преобразуется в наблюдаемый объект, ежесекундно выдающий температуру для этого города. Наконец, поток наблюдаемых объектов собирается в список, который передается статическому фабричному методу `merge` самого класса `Observable`. Этот метод принимает на входе `Iterable` наблюдаемых объектов и объединяет выдаваемые ими результаты так, что они ведут себя как единый наблюдаемый объект. Другими словами, получившийся в результате наблюдаемый объект выдает все события, публикуемые всеми содержащимися в этом `Iterable` наблюдаемыми объектами, с сохранением их временной упорядоченности.

Попробуем этот метод в деле, воспользовавшись им в окончательном классе `Main`, приведенном в листинге 17.17.

Листинг 17.17. Класс Main, выводящий в консоль температуру в трех городах

```
public class Main {
    public static void main(String[] args) {
        Observable<TempInfo> observable = getCelsiusTemperatures(
                                    "New York", "Chicago", "San Francisco" );
        observable.blockingSubscribe( new TempObserver() );
    }
}
```

Этот класс `Main` идентичен классу из листинга 17.14, за исключением того, что теперь мы подписываемся на наблюдаемый объект, возвращаемый методом `getCelsiusTemperatures` из листинга 17.16, и выводим температуру, зарегистрированную в трех городах. В результате запуска этой функции `Main` выводятся примерно следующие результаты:

```
New York : 21
Chicago : 6
San Francisco : -15
New York : -3
Chicago : 12
San Francisco : 5
Got problem: Error!
```

Каждую секунду `Main` выводит в консоль температуру для каждого из требуемых городов, до тех пор пока одна из операций получения температуры не вернет ошибку, которая будет транслирована в наблюдаемый объект, прерывая поток данных.

Задача этой главы состояла не во всестороннем обзоре RxJava (или какой-либо другой реактивной библиотеки), для чего потребовалась бы целая книга, а лишь в демонстрации возможностей подобного инструментария и знакомства читателя с принципами реактивного программирования. Мы лишь прошлись по верхам этого стиля программирования, однако надеемся, что продемонстрировали хотя бы некоторые из его преимуществ и заинтересовали вас.

Резюме

- Идеям, лежащим в основе реактивного программирования, от 20 до 30 лет, но популярность они обрели лишь недавно, вследствие высоких требований современных приложений в смысле количества обрабатываемых данных и ожиданий пользователей.
- Эти идеи были формализованы в Манифесте реактивности, который гласит, что реактивное программное обеспечение должно отличаться четырьмя взаимосвязанными особенностями: быстротой реакции, отказоустойчивостью, адаптивностью и ориентированностью на обмен сообщениями.
- Принципы реактивного программирования применимы с некоторыми различиями как при разработке отдельного приложения, так и при проектировании реактивной системы, объединяющей несколько приложений.
- В основе реактивных приложений лежит асинхронная обработка одного или нескольких потоков событий, передаваемых реактивными потоками данных. В силу важности роли реактивных потоков данных для разработки реактивных приложений консорциум, включающий компании Netflix, Pivotal, Lightbend и Red Hat, стандартизировал соответствующие концепции ради максимальной совместимости различных реализаций.
- Вследствие асинхронности обработки реактивных потоков данных в их архитектуру включен механизм противодавления — для предотвращения переполнения медленного потребителя более быстрыми генераторами.
- Результат этого процесса стандартизации и проектирования был включен в Java. Flow API Java 9 описывает четыре базовых интерфейса: `Publisher`, `Subscriber`, `Subscription` и `Processor`.
- Эти интерфейсы в большинстве случаев не предназначены для непосредственной реализации разработчиками, а играют роль лингва-франка для различных библиотек, реализующих парадигму реактивности.
- В число наиболее часто используемых подобных наборов инструментов входит библиотека RxJava, предоставляющая (помимо описываемых Flow API Java 9 базовых функций) множество полезных, обладающих широкими возможностями операторов. В их числе: операторы для удобного преобразования и фильтрации элементов, публикуемых отдельным реактивным потоком данных, а также операторы для объединения и агрегирования нескольких потоков данных.

Часть VI

Функциональное программирование и эволюция языка Java

В шестой, заключительной части этой книги мы немного отступим от темы, чтобы предоставить читателю руководство по написанию эффективных программ в стиле функционального программирования на языке Java, а также сравнить возможности Java 8 с возможностями языка Scala.

Глава 18 предлагает полное руководство по функциональному программированию, знакомит с его терминологией и объясняет, как писать программы в функциональном стиле на языке Java.

Глава 19 охватывает более продвинутые методики функционального программирования, включая функции высшего порядка, каррирование, персистентные структуры данных, отложенные списки и сопоставление с шаблоном. Эту главу можно рассматривать как смесь практических методик, которые можно использовать в своем коде, с чисто теоретической информацией, направленной на расширение ваших познаний как программиста.

Глава 20 продолжает тему сравнения возможностей Java 8 с возможностями языка Scala — языка, который, как и Java, основан на использовании JVM и быстро развивается, угрожая занять часть ниши Java в экосистеме языков программирования.

Наконец, в главе 21 мы охватим взглядом пройденный во имя изучения Java 8 путь и нашу пропаганду функционального программирования. В дополнение мы поговорим о возможных будущих усовершенствованиях и замечательных новых возможностях Java в следующих за Java 8 и Java 9 версиях.

18 Мыслим функционально

В этой главе

- Почему именно функциональное программирование.
- Отличительные свойства функционального программирования.
- Декларативное программирование и функциональная прозрачность.
- Рекомендации по написанию функционального кода на языке Java.
- Итерация и рекурсия.

Мы часто встречали термин *«функциональный»* на протяжении этой книги. У вас уже, возможно, сложилось некоторое представление о том, что подразумевает эпитет «функциональный». Но идет ли речь о лямбда-выражениях и полноправных функциях или о запрете на изменение объектов? Какие преимущества несет функциональный стиль программирования?

В этой главе мы прольем свет на данные вопросы. Мы расскажем, что такое функциональное программирование, и познакомим вас с определенной его терминологией. Во-первых, мы изучим понятия, лежащие в его основе, — такие как побочные эффекты, неизменяемость, декларативное программирование и функциональная прозрачность, — после чего соотнесем их с кодом на Java 8. В главе 19 мы подробнее изучим такие методики функционального программирования, как функции высшего порядка, каррирование, персистентные структуры данных, отложенные списки, сопоставление с шаблоном и комбинаторы.

18.1. Создание и сопровождение систем

Для начала представьте себе, что вам нужно произвести модернизацию большой программной системы, которую вы в глаза не видели. Соглашаться ли вам на сопровождение подобной системы? Лишь отчасти шуточное правило, в соответствии с которым опытные Java-подрядчики принимают подобные решения: «Начните с поиска слова `synchronized`, если найдете — сразу отвечайте "нет", учитывая сложности исправления ошибок, связанных с конкурентностью. В противном случае изучите структуру системы подробнее». Мы поговорим об этом детальнее в следующих разделах. Сначала, впрочем, мы отметим, что, как вы уже видели в предыдущих главах, появление в Java 8 потоков данных дает возможность пользоваться параллелизмом, не задумываясь о блокировках, при условии, что вас устраивает отсутствие сохранения состояния. То есть функции в конвейере потоковой обработки не взаимодействуют, никакие функции не читают/не записывают данные в переменные, уже записанные другими функциями.

Что еще требуется от программы, чтобы с ней легко было работать? Она должна быть хорошо структурированной, обладать понятной иерархией классов, отражающей структуру системы. Существуют возможности оценки подобной структуры с помощью таких метрик инженерии ПО, как *сцепление* (степень взаимозависимости частей системы) и *взаимосвязь* (степень связи различных частей системы друг с другом).

Но основная ежедневная забота многих программистов — отладка в процессе сопровождения системы, например, при фатальном сбое какого-то кода из-за встречи с неожиданным значением. Но как узнать, какие части программы участвовали в создании и модификации этого значения? Задумайтесь, сколько ваших личных проблем в процессе сопровождения относится к этой категории?[1] Оказывается, что здесь могут помочь принципы *отсутствия побочных эффектов* и *неизменяемости*, поощряемые функциональным программированием. Мы изучим их подробнее в следующих разделах.

18.1.1. Разделяемые изменяемые данные

Основная причина проблем с неожиданными значениями переменных, обсуждавшаяся выше, — чтение и модификация разделяемых изменяемых структур данных более чем одним методом. Представьте себе, что ссылка на список хранится в нескольких классах. Вам, как отвечающему за сопровождение, необходимо найти ответы на следующие вопросы.

- Кто является владельцем этого списка?
- Что произойдет, если один из классов модифицирует список?
- Ожидают ли остальные классы подобного изменения?
- Как эти остальные классы узнают про данное изменение?
- Нужно ли оповещать классы про изменение для удовлетворения всех допущений списка или они должны создавать для себя защитные копии?

[1] Мы рекомендуем вам обратиться за дополнительной информацией по этому вопросу к книге Working Effectively with Legacy Code Майкла Физерса (Michael Feathers) (Prentice Hall, 2004).

Другими словами, разделяемые изменяемые структуры данных затрудняют отслеживание изменений в различных частях программы. Эта идея проиллюстрирована на рис. 18.1.

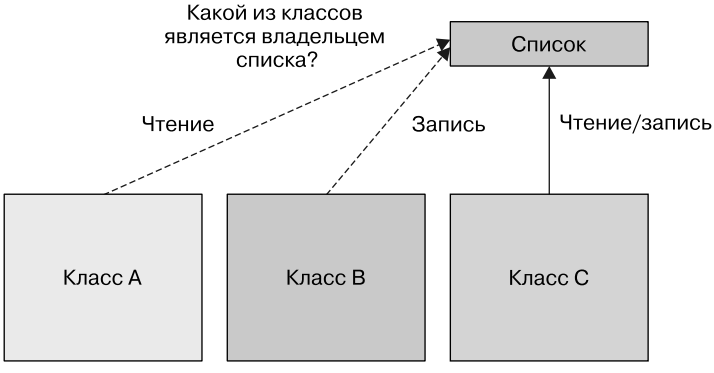


Рис. 18.1. Изменяемые данные, совместно используемые несколькими классами. Понять, кто является владельцем списка, непросто

Представьте себе систему, в которой нет никаких изменяемых структур данных. Поддержка такой системы — мечта разработчика: никаких неприятных сюрпризов в виде неожиданной модификации какой-либо структуры данных каким-нибудь объектом. Метод, не меняющий ни состояние охватывающего класса, ни состояние каких-либо других объектов и возвращающий все результаты с помощью оператора `return`, называют *чистым* (pure) или *лишенным побочных эффектов* (side-effect-free).

Что представляет собой побочный эффект? По существу, *побочный эффект* — это действие, выходящее за пределы самой функции. Вот несколько примеров:

- модификация структуры данных без создания дополнительных структур, включая присвоение значения любому ее полю, за исключением инициализации внутри конструктора (методы-сеттеры);
- генерация исключения;
- выполнение операций ввода/вывода, например записи в файл.

Другая сторона идеи отсутствия побочных эффектов — неизменяемые объекты. *Неизменяемым* (immutable) называется объект, который не меняет своего состояния после начальной инициализации, так что никакие действия функции на него не влияют. Единожды инициализированный, неизменяемый объект уже не попадет ни в какое неожиданное состояние. Такие объекты можно совместно использовать без необходимости копирования даже в многопоточной среде, поскольку их нельзя модифицировать.

Идея отсутствия побочных эффектов может показаться жестким ограничением, и, наверное, вы сомневаетесь, можно ли разрабатывать в таком стиле реальные системы. Мы надеемся к концу данной главы убедить вас, что это вполне возможно. Хорошая новость заключается в том, что придерживающиеся этого принципа компоненты системы могут использовать многоядерный параллелизм без блокировок,

поскольку методы не будут мешать друг другу. Кроме того, эта концепция позволяет сразу же определить, какие части программы независимы друг от друга.

Источник этих идей — функциональное программирование, к которому мы перейдем в следующем разделе.

18.1.2. Декларативное программирование

Для начала мы обсудим идею декларативного программирования, лежащего в основе программирования функционального.

Можно рассматривать программную реализацию системы с двух точек зрения. Один вариант: сосредоточить свое внимание на способе выполнения (сначала сделать то, потом обновить это и т. д.). Например, если необходимо вычислить наиболее ресурсозатратную транзакцию из списка, выполняется последовательность команд (извлечь транзакцию из списка и сравнить ее с предыдущей наиболее ресурсозатратной транзакцией; если данная транзакция является наиболее ресурсозатратной, запоминаем ее в качестве текущей наиболее ресурсозатратной; повторяем эти действия со следующей транзакцией в списке и т. д.).

Подобный стиль, отвечающий на вопрос «как?», отлично подходит для классического объектно-ориентированного программирования (иногда называемого *императивным программированием*), поскольку он включает инструкции, имитирующие низкоуровневый лексикон компьютера (например, присваивания, условное ветвление и циклы), как показано в следующем коде:

```
Transaction mostExpensive = transactions.get(0);
if(mostExpensive == null)
    throw new IllegalArgumentException("Empty list of transactions");
for(Transaction t: transactions.subList(1, transactions.size())){
    if(t.getValue() > mostExpensive.getValue()){
        mostExpensive = t;
    }
}
```

Другой способ концентрируется на том, что нужно сделать. Вы видели в главах 4 и 5, что с помощью Stream API можно описать следующий запрос:

```
Optional<Transaction> mostExpensive =
    transactions.stream()
                .max(comparing(Transaction::getValue));
```

Нюансы реализации этого запроса скрыты в библиотеке. Такая идея носит название *внутренней итерации*. Важное ее преимущество состоит в том, что запрос в точности отражает формулировку задачи и сразу становится понятен, в отличие от последовательности команд, для которой нужно выяснять, что она делает.

Подобный стиль, отвечающий на вопрос «что?», часто называют *декларативным программированием*. Вы лишь задаете правила, описывающие, что вам нужно, а система должна сама определить, как достичь этой цели. Достоинство подобного стиля программирования — в том, что формулировка задачи сразу очевидна из кода.

18.1.3. Почему функциональное программирование?

Функциональное программирование служит примером идей декларативного программирования (например, использование не взаимодействующих между собой выражений, реализацию для которых выбирает система) и вычислений без побочных эффектов, о которых мы говорили выше в главе. Эти две идеи упрощают реализацию и сопровождение систем.

Отметим, что для чтения и написания в естественном декларативном стиле необходимы определенные языковые возможности, например операции композиции и передачи поведения (с которыми мы познакомились в главе 3 при изучении лямбда-выражений). С помощью потоков данных можно выражать сложные запросы, связывая цепочкой несколько операций. Такие возможности — отличительная особенность функциональных языков программирования. Мы внимательнее изучим эти возможности в виде комбинаторов в главе 19.

Чтобы наша дискуссия была более предметной и связанной с новыми возможностями Java 8, в следующем разделе мы приведем четкое определение понятия функционального программирования и его представления в Java. Хотелось бы подчеркнуть, что с помощью функционального программирования можно писать серьезные программы без использования в них побочных эффектов.

18.2. Что такое функциональное программирование

Чрезвычайно упрощенный ответ на вопрос «Что такое функциональное программирование?» звучит так: «Программирование с помощью функций». Но что такое функция?

Можно представить себе метод, принимающий в качестве аргументов значения типов `int` и `double` и возвращающий `double` с побочным эффектом подсчета числа вызовов путем обновления изменяемой переменной, как показано на рис. 18.2.

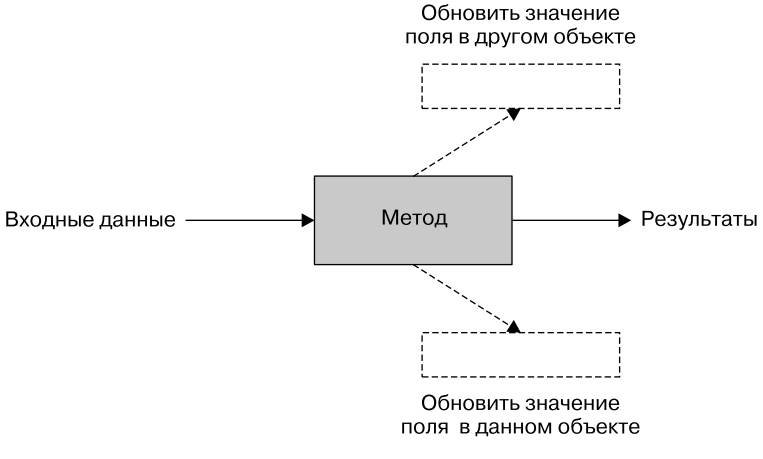


Рис. 18.2. Функция с побочными эффектами

В контексте функционального программирования, однако, термин *функция* понимается в математическом смысле: она принимает ноль или более аргументов, возвращает один или более результатов и не имеет побочных эффектов. Функцию можно рассматривать как «черный ящик», получающий какие-либо входные данные и выдающий какие-то результаты, как показано на рис. 18.3.

Рис. 18.3. Функция без побочных эффектов

Подобные функции и методы таких языков программирования, как Java, отличаются разительно (нельзя себе даже представить, что у таких математических функций, как `log` или `sin`, могут быть побочные эффекты). В частности, математические функции всегда возвращают один и тот же результат, будучи вызванными с одними аргументами. Данная особенность исключает из рассмотрения такие функции, как `Random.nextInt`, а далее, в подразделе 18.2.2, мы поговорим подробнее об этой концепции функциональной прозрачности.

Говоря *«функциональный»*, мы подразумеваем «подобный математическому», без побочных эффектов. Но теперь всплывают некоторые тонкости, связанные с программированием. Имеется ли в виду, что все функции должны быть основаны на функциональных и математических идеях, как `if-then-else`? Или внутри функции могут выполняться нефункциональные действия, при условии, что они не оказывают побочных эффектов на остальную систему? Другими словами, если программа производит невидимый для вызывающей стороны побочный эффект, то можно ли считать, что этого побочного эффекта нет? Вызывающей стороне о нем знать необязательно, поскольку он никак на нее не влияет.

Чтобы подчеркнуть это различие, мы будем называть первый вариант *чисто функциональным программированием*, а второй — просто *функциональным программированием* или *программированием в функциональном стиле*.

18.2.1. Функциональное программирование на языке Java

На практике невозможно программировать в чисто функциональном стиле на языке Java. Модель ввода/вывода Java, например, состоит из методов с побочными эффектами (побочный эффект вызова `Scanner.nextLine` — потребление строки из файла, так что результат при повторном вызове будет другим). Тем не менее вполне можно написать основные компоненты системы так, как будто они чисто функциональные. На языке Java мы будем писать программы в функциональном стиле.

Во-первых, существует нюанс относительно невидимости побочных эффектов и, следовательно, смысла слова *«функциональный»*. Пусть у метода или функции нет побочных эффектов, за исключением увеличения на единицу значения определенного поля при входе в функцию и уменьшения после выхода. С точки зрения однопоточной программы у этой функции нет видимых побочных эффектов, ее можно считать функ-

циональной. С другой стороны, если к этому полю может обращаться другой поток выполнения — или метод может вызываться конкурентно, — то считать его функциональным нельзя. Эту проблему можно скрыть с помощью адаптера с блокировкой для тела этого метода, что позволит утверждать, что он является функциональным. Но при этом мы потеряем возможность параллельного выполнения двух вызовов метода двумя ядрами многоядерного процессора. Побочный эффект может быть незаметен программе, но он проявляется для программиста в виде снижения быстродействия.

Мы рекомендуем считать функциональными только функции/методы, которые изменяют лишь значения локальных переменных. Кроме того, должны быть неизменяемыми и объекты, на которые они ссылаются, — то есть все поля должны быть `final`, а все поля ссылочного типа должны транзитивно ссылаться на другие неизменяемые объекты. Далее можно разрешить модификацию полей объектов, только что созданных в данном методе, поскольку они ниоткуда не видимы и не сохраняются, так что не могут повлиять на последующие вызовы этого метода.

Впрочем, приведенные рекомендации неполны. Чтобы быть функциональными, функция/метод должны удовлетворять дополнительному требованию: *не генерировать никаких исключений*. Обосновывается это требование тем, что генерация исключения означает сигнал о результате выполнения функции, отличный от возврата значения; см. модель «черного ящика» на рис. 18.3. Впрочем, это достаточно дискуссионный вопрос, некоторые авторы утверждают, что неперехваченные исключения, отражающие фатальные ошибки, допустимы, а нефункциональным способом управления потоком в данном случае является перехват исключения. Подобное применение исключений все же нарушает простую метафору «передать аргумент, вернуть результат», показанную в модели «черного ящика», приводя к третьей стрелке, соответствующей исключению, как показано на рис. 18.4.

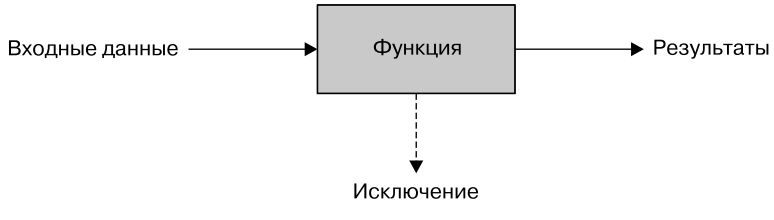


Рис. 18.4. Генерирующая исключение функция

Функции и частичные функции

В математике функция должна возвращать ровно один результат для каждого возможного значения аргумента. Но множество распределенных математических операций правильнее было бы назвать частичными функциями. То есть для некоторых или большинства входных значений они возвращают ровно один результат, но для части входных значений они не определены и вообще не возвращают никакого результата. Типичный пример: операция деления с нулевым вторым операндом или извлечение квадратного корня для отрицательного аргумента. Подобные ситуации обычно моделируются в Java путем генерации исключения.

Как выразить функции, подобные делению, без генерации исключений? С помощью типов вроде `Optional<T>`. Вместо сигнатуры «`double sqrt(double)`, но может генерировать исключение» у функции `sqrt` может быть сигнатура `Optional<Double> sqrt(double)`. Она или возвращает значение, означающее успешное выполнение, или указывает в возвращаемом значении, что выполнение требуемой операции невозможно. И да, это означает, что вызывающая сторона должна проверять, не пустой ли опционал возвращает каждый вызов такого метода. Выглядит изрядной «головной болью», но если подойти к делу прагматично и следовать нашим рекомендациям по программированию в функциональном стиле вместо чисто функционального программирования, то можно использовать исключения локально, не делая их видимыми в широкомасштабных интерфейсах и, таким образом, получая все преимущества функционального программирования без излишнего раздувания кода.

Чтобы считаться функциональной, функция (метод) должна вызывать только те библиотечные функции с побочными эффектами, которые позволяют скрыть нефункциональное поведение (то есть гарантировать, что любые изменения структур данных скрыты от вызывающей стороны, например, с помощью предварительного их копирования и путем перехвата всех исключений). В подразделе 18.2.4 мы будем скрывать использование в методе `insertAll` библиотечной функции `List.add`, имеющей побочный эффект, путем копирования списка.

Подобные указания зачастую отмечают с помощью комментариев или объявления метода с добавлением аннотации-маркера — с теми же ограничениями, что налагались на функции, передаваемые в операции потоковой обработки (например, `Stream.map`) в главах 4–7.

Наконец, исходя из чисто прагматических соображений, для функционального кода может пригодиться возможность вывода отладочной информации в какой-либо файл журнала. Такой код нельзя считать строго функциональным, но на практике при этом не теряется практически никаких преимуществ функционального программирования.

18.2.2. Функциональная прозрачность

Ограничения, связанные с отсутствием видимых побочных эффектов (никакие изменяемые структуры не видны вызывающей стороне, никакого ввода/вывода, никаких исключений), отражают концепцию функциональной прозрачности. Функция считается функционально прозрачной, если она всегда возвращает один результат при вызове с одним и тем же значением аргумента. Метод `String.replace`, например, функционально прозрачен, поскольку `"raoul".replace('r', 'R')` всегда возвращает один и тот же результат (`replace` возвращает новый объект `String`, все `r` в нижнем регистре из которого заменены на `R` в верхнем регистре) вместо обновления объекта `this`, так что его можно считать функцией.

Иными словами, функция неизменно возвращает один результат при одних входных данных, независимо от того, где и когда она вызвана. Отсюда понятно также, почему метод `Random.nextInt` нельзя считать функциональным. В Java использование объекта `Scanner` для получения вводимых с клавиатуры данных нарушает функциональную прозрачность, поскольку метод `nextLine` может при каждом вызове возвра-

щать другой результат. Но сложение значений двух переменных, объявленных как `final int`, всегда возвращает один и тот же результат, поскольку их содержимое не может измениться.

Функциональная прозрачность — свойство программы, которое делает ее гораздо понятнее. Она также включает оптимизацию типа «сохранение вместо повторного вычисления» для дорогостоящих или длительных операций — процесс, носящий название *мемоизации* (memoization) или *кэширования* (caching). Хотя данный вопрос и важный, для этой главы он носит лишь побочный характер, так что мы рассмотрим его отдельно в главе 19.

В языке Java есть одна небольшая проблема, связанная с функциональной прозрачностью. Представьте себе, что выполняете два вызова метода, возвращающего список. Эти вызовы могут вернуть ссылки на различные, хотя и содержащие одинаковые элементы, списки в оперативной памяти. Если рассматривать их как изменяемые объектно-ориентированные значения (а значит, неодинаковые), то метод не будет функционально прозрачным. Если же вы собираетесь использовать эти списки в качестве чистых (неизменяемых) значений, то их имеет смысл считать одинаковыми, так что функция будет функционально прозрачной. Вообще говоря, в функциональном коде подобные функции считаются функционально прозрачными.

В следующем подразделе мы поговорим о вопросе изменения значений в более широком контексте.

18.2.3. Объектно-ориентированное и функциональное программирование

Мы начнем с противопоставления программирования в функциональном стиле (экстремальному) классическому объектно-ориентированному программированию, прежде заметим, что Java 8 рассматривает эти стили просто как две противоположности в объектно-ориентированном спектре. Как Java-программист, даже не задумываясь об этом, вы наверняка используете определенные свойства функционального программирования и некоторые свойства того, что мы будем называть экстремальным объектно-ориентированным программированием. Как мы отметили в главе 1, изменения в аппаратном обеспечении (такие как многоядерность) и ожиданиях программистов (например, SQL-подобных запросов для операций над данными) сдвигают стиль разработки программного обеспечения на языке Java в сторону функционального конца этого спектра, и одна из целей данной книги как раз и состоит в том, чтобы помочь вам адаптироваться к такой «смене климата».

На одном конце этого спектра располагается экстремальное объектно-ориентированное представление: все рассматривается в качестве объектов и в основе работы программ лежит обновление полей и вызов методов, обновляющих связанные с ними объекты. На другом конце этого спектра располагается функционально прозрачное функциональное программирование без (видимых) изменений значений. На практике Java-программисты всегда смешивали эти стили. Можно выполнить обход структуры данных с помощью итератора с изменяемым внутренним состоянием, но использовать его для вычисления, допустим, суммы значений структуры данных в функциональном стиле (в языке Java, как обсуждалось ранее, этот процесс может включать изменение локальных переменных). Одна из целей данной главы заключается в обсуждении

методик программирования и знакомстве с возможностями функционального программирования, с помощью которых можно писать программы с лучшей модульной организацией, более подходящие для многоядерных процессоров. Можете считать эти идеи дополнительными инструментами в вашем арсенале программиста.

18.2.4. Функциональное программирование в действии

Для начала решим упражнение по программированию, предлагавшееся студентам-первокурсникам. Оно хорошо иллюстрирует функциональный стиль: по заданному значению `List<Integer>`, например, {1, 4, 9} сформировать значение `List<List<Integer>>`, элементы которого представляют собой все подмножества {1, 4, 9} в произвольном порядке. Подмножествами {1, 4, 9} являются: {1, 4, 9}, {1, 4}, {1, 9}, {4, 9}, {1}, {4}, {9} и {}.

Общее число подмножеств — восемь, включая пустое подмножество, записанное в виде {}. Каждое из подмножеств представлено объектом типа `List<Integer>`, а значит, ответ должен иметь тип `List<List<Integer>>`.

Студенты зачастую не знают, с чего начать, им приходится подсказывать[1]: «Подмножества {1, 4, 9} или включают 1, или не включают». Те, которые не включают, являются подмножествами {4, 9}, а те, что включают, можно получить путем вставки 1 в каждое из подмножеств {4, 9}. Есть, правда, один нюанс: не забывайте, что у пустого множества есть ровно одно подмножество — оно само. Понимание этого позволяет написать на языке Java естественный, нисходящий, функциональный код для решения задачи[2]:

```
static List<List<Integer>> subsets(List<Integer> list) {
    if (list.isEmpty()) {
        List<List<Integer>> ans = new ArrayList<>();
        ans.add(Collections.emptyList());
        return ans;
    }

    Integer fst = list.get(0);
    List<Integer> rest = list.subList(1,list.size());
    List<List<Integer>> subAns = subsets(rest);
    List<List<Integer>> subAns2 = insertAll(fst, subAns);
    return concat(subAns, subAns2);
}
```

Если входной список пуст, у него есть только одно подмножество: сам пустой список

В противном случае убираем из списка один элемент, fst, находим все подмножества оставшейся его части и присваиваем переменной subAns, которая составляет половину итогового ответа

Вторая половина итогового ответа, subAns2, состоит из всех списков из subAns с добавлением в их начало fst

Производим конкатенацию двух половинок ответа

---

[1] Трудные (одаренные!) студенты иногда указывают на изящный трюк с использованием двоичного представления чисел. (Код решения на Java соответствует 000,001,010,011,100,-101,110,111.) Мы рекомендуем таким студентам вычислить вместо этого список всех перестановок списка; для нашего примера {1, 4, 9} их будет шесть.

[2] Для большей конкретности в приведенном коде применяется List<Integer>, но можно заменить его в описании метода на обобщенный List<T>; после чего можно будет использовать модифицированный метод subsets как для List<Integer>, так и для List<String>.

При получении в качестве входных данных {1, 4, 9} эта программа генерирует {{}, {9}, {4}, {4, 9}, {1}, {1, 9}, {1, 4}, {1, 4, 9}}. Попробуйте запустить ее, когда мы опишем два недостающих метода.

Напомним: мы считали, что недостающие методы `insertAll` и `concat` — сами функциональные, из чего заключили, что наша функция `subsets` также будет функциональной, поскольку никакие операции в ней не изменяют каких-либо существующих структур данных (если вы знакомы с математикой, то узнали, наверное, вывод по индукции).

Теперь взглянем на описание `insertAll`. Здесь нас поджидает первая опасность. Предположим, что мы описали `insertAll` так, что она изменяет свои аргументы, например включая `fst` во все элементы `subAns`. В таком случае `subAns` будет ошибочно модифицирована программой аналогично `subAns2` и мы получим результат, который почему-то содержит восемь копий {1, 4, 9}. Поэтому мы опишем `insertAll` иначе, функционально:

```
static List<List<Integer>> insertAll(Integer fst,
                                      List<List<Integer>> lists) {
    List<List<Integer>> result = new ArrayList<>();
    for (List<Integer> list : lists) {
        List<Integer> copyList = new ArrayList<>();
        copyList.add(fst);
        copyList.addAll(list);
        result.add(copyList);
    }
    return result;
}
```

Копируем список, чтобы можно было добавлять в него значения. Мы не будем копировать структуры более низкого уровня, даже если они изменяемые (значения типа Integer не являются изменяемыми)

Обратите внимание, что мы создали новый список, содержащий все элементы `subAns`. Мы воспользовались тем фактом, что объекты типа `Integer` неизменяемые; в противном случае пришлось бы их все тоже клонировать. Представление о методе `insertAll` как функциональном указывает нам естественное место для размещения всего этого аккуратно скопированного кода: не на вызывающей стороне, а внутри `insertAll`.

Наконец, нам нужно описать метод `concat`. В данном случае решение просто, но мы умоляем вас его не использовать; оно приведено здесь лишь для того, чтобы у вас была возможность сравнить различные стили:

```
static List<List<Integer>> concat(List<List<Integer>> a,
                                   List<List<Integer>> b) {
    a.addAll(b);
    return a;
}
```

Вместо этого кода мы рекомендуем написать следующий:

```
static List<List<Integer>> concat(List<List<Integer>> a,
                                   List<List<Integer>> b) {
    List<List<Integer>> r = new ArrayList<>(a);
    r.addAll(b);
    return r;
}
```

Почему? Вторая версия `concat` представляет собой чистую функцию. Внутри нее может происходить изменение значений (добавление элементов в список `r`), но она возвращает результат, полученный на основе аргументов, не модифицируя ни одного из них. Напротив, первая ее версия полагается на тот факт, что после вызова `concat(subAns, subAns2)` никто не обращается снова к значению `subAns`. Именно так и происходит, согласно нашему определению метода `subsets`, так что, конечно, расходующая меньше ресурсов версия `concat` лучше. Вопрос в том, насколько вы цените свое время. Сравните время, которое вы потратите позднее в поисках запутанных ошибок, с дополнительными затратами ресурсов на создание копии.

Неважно, насколько хорошо вы опишете в комментариях к «неочищенной» версии `concat`, что она «должны применяться лишь тогда, когда допустима перезапись первого аргумента, она предназначена для использования лишь в методе `subsets`, а любые изменения кода `subsets` должны учитывать этот комментарий», кто-то когда-нибудь обязательно решит задействовать ее в каком-то фрагменте кода, где она вроде бы работает. В этот момент начнется история вашего будущего кошмара при отладке. Мы вернемся к данному вопросу в главе 19.

Вывод: если рассматривать задачи программирования с точки зрения функциональных методов, характеризуемых только их входными аргументами и выходными результатами *(что делать)*, зачастую можно добиться большего, чем обдумывание на слишком раннем этапе цикла проектирования того, *как сделать* и какие значения изменять.

В следующем разделе мы подробнее обсудим рекурсию.

18.3. Рекурсия и итерация

Рекурсия (recursion) — поощряемая в функциональном программировании методика, направляющая мысли разработчиков в русло ответа на вопрос «что делать?». Чисто функциональные языки программирования обычно не содержат таких итеративных конструкций, как циклы `while` и `for`. Подобные конструкции часто служат неявными приглашениями к изменениям значений. Например, необходимо обновлять условие в цикле `while`; в противном случае цикл выполнится ноль или бесконечное число раз. Во многих случаях, впрочем, циклы вполне допустимы. Мы говорили, что и в функциональном стиле программирования изменения значений могут иметь место, если они никому не видны, так что изменение значений локальных переменных допустимо. Цикл `for-each` в Java, например, `for(Apple apple : apples) { }`, транслируется в следующий итератор:

```
Iterator<Apple> it = apples.iterator();
while (it.hasNext()) {
   Apple apple = it.next();
   // ...
}
```

Трансляция в такой код, впрочем, не вызывает проблем, поскольку изменение (состояния итератора с помощью метода `next` и присвоение значения переменной `apple` внутри тела `while`) не видимы стороне, вызывающей метод. Но при приме-

нении цикла `for-each`, например в поисковом алгоритме, происходящее вызывает проблемы, поскольку тело цикла обновляет структуру данных, используемую совместно с вызывающей стороной:

```
public void searchForGold(List<String> l, Stats stats){
    for(String s: l){
        if("gold".equals(s)){
            stats.incrementFor("gold");
        }
    }
}
```

Так что у тела цикла есть побочный эффект, который не позволяет считать его функциональным: он изменяет состояние объекта `stats`, используемого совместно с остальными частями программы.

Поэтому в чисто функциональных языках программирования вроде Haskell стараются избегать подобных операций с побочными эффектами. Как же вам писать программы? Теоретически каждую программу можно переписать, используя вместо итерации рекурсию, которая не требует изменяемости. Благодаря рекурсии можно избавиться от обновляемых на каждом шаге итерационных переменных. Классическая школьная задача: итеративное и рекурсивное вычисление факториала (для положительных аргументов), как показано в листингах 18.1, 18.2.

Листинг 18.1. Итеративное вычисление факториала

```
static long factorialIterative(long n) {
    long r = 1;
    for (int i = 1; i <= n; i++) {
        r *= i;
    }
    return r;
}
```

Листинг 18.2. Рекурсивное вычисление факториала

```
static long factorialRecursive(long n) {
    return n == 1 ? 1 : n * factorialRecursive(n-1);
}
```

Первый листинг демонстрирует стандартный вариант на основе цикла: переменные `r` и `i` обновляются на каждом шаге итерации. Во втором листинге показано рекурсивное определение (функция вызывает саму себя), более близкое к математической форме. Производительность рекурсивных вариантов в языке Java обычно ниже, как мы обсудим в следующем же примере (листинг 18.3).

Листинг 18.3. Вычисление факториала с помощью потока данных

```
static long factorialStreams(long n){
    return LongStream.rangeClosed(1, n)
                     .reduce(1, (long a, long b) -> a * b);
}
```

Займемся теперь вопросом производительности. Пользователям Java следует с осторожностью относиться к словам фанатичных приверженцев функционального программирования, призывающих всегда использовать рекурсию вместо итерации. В общем случае рекурсивный вызов функции всегда требует больше ресурсов, чем необходимая для итерации простая команда перехода машинного уровня. При каждом вызове функции `factorialRecursive` создается новый фрейм стека в стеке вызовов для хранения состояния каждого из вызовов функции (операции умножения, которую она должна произвести) вплоть до завершения выполнения рекурсии.

Такое рекурсивное определение факториала требует объема оперативной памяти, пропорционального входному значению. Так что, если запустить функцию `factorialRecursive` с большим входным значением, можно запросто получить исключение `StackOverflowError`:

```
Exception in thread "main" java.lang.StackOverflowError
```

Так что, рекурсия бесполезна? Конечно же, нет! Функциональные языки программирования решают эту проблему с помощью оптимизации путем *хвостового вызова* (tail-call). Основная ее идея: написание рекурсивного определения факториала, при котором рекурсивный вызов выполняется в функции в последнюю очередь (то есть этот вызов находится в хвосте). Подобный вид рекурсии допускает существенную оптимизацию быстродействия. Такое определение факториала приведено в листинге 18.4.

Листинг 18.4. Хвостовое рекурсивное вычисление факториала

```
static long factorialTailRecursive(long n) {
    return factorialHelper(1, n);
}

static long factorialHelper(long acc, long n) {
    return n == 1 ? acc : factorialHelper(acc * n, n-1);
}
```

Функция `factorialHelper` реализует хвостовую рекурсию, поскольку рекурсивный вызов выполняется в функции в последнюю очередь. Напротив, в предыдущем определении `factorialRecursive` последней операцией в функции было умножение `n` на результат рекурсивного вызова.

Подобная форма рекурсии полезна тем, что вместо хранения всех промежуточных результатов рекурсии в отдельных фреймах стека компилятор может переиспользовать один фрейм стека. И действительно, в определении `factorialHelper` промежуточные результаты (частичные результаты вычисления факториала) передаются напрямую функции в качестве аргументов. Нет нужды сохранять промежуточный результат каждого из рекурсивных вызовов в отдельном фрейме стека; он доступен непосредственно в виде первого аргумента `factorialHelper`. Рисунки 18.5 и 18.6 иллюстрируют различия между рекурсивным и хвостовым рекурсивным определениями факториала.

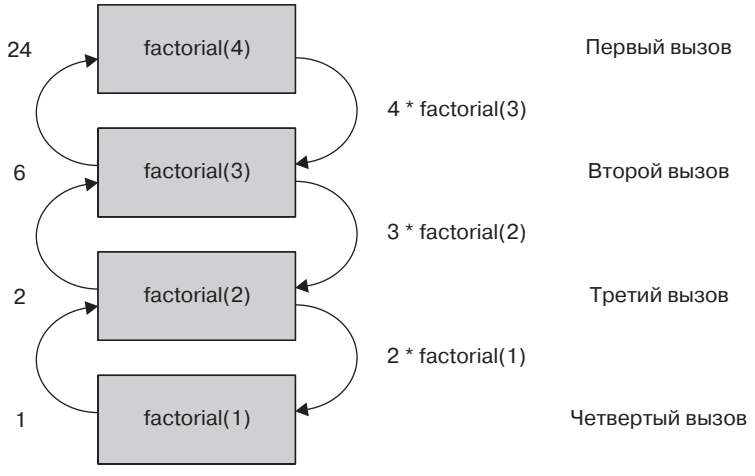


Рис. 18.5. Рекурсивное определение факториала, требующее нескольких фреймов стека

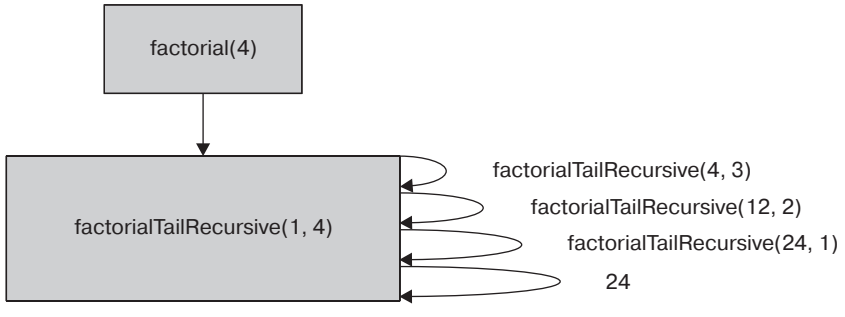


Рис. 18.6. Хвостовое рекурсивное определение факториала, которое может переиспользовать один фрейм стека

Плохая новость: Java не поддерживает подобную оптимизацию. Но использование хвостовой рекурсии все равно предпочтительнее, чем обычной, поскольку в будущем в компиляторе может появиться ее поддержка. Многие современные языки программирования, использующие JVM, например Scala, Groovy и Kotlin, умеют оптимизировать подобную рекурсию, эквивалентную итерации (и выполняющуюся с аналогичным быстродействием). В результате сторонники чисто функциональных языков получают желаемую чистоту плюс быстродействие в придачу.

При написании кода на Java 8 следует учитывать, что итерацию зачастую можно заменить потоками данных во избежание изменения значений. Кроме того, итерацию можно заменить рекурсией в случаях, когда она позволяет реализовать алгоритм более лаконично, без побочных эффектов. Безусловно, рекурсия облегчает чтение, написание и понимание примеров (как в примере с подмножествами, приведенном выше в данной главе), а эффективность работы программиста зачастую важнее, чем небольшие различия в быстродействии.

В этом разделе мы обсудили программирование в функциональном стиле и идею функционального метода; все вышесказанное применимо к первой версии Java. В главе 19 мы рассмотрим потрясающие новые возможности, которые приносит появление в Java 8 полноправных функций.

Резюме

- Сокращение совместного использования изменяемых структур данных упрощает в долгосрочной перспективе сопровождение и отладку программ.
- Программирование в функциональном стиле поощряет применение методов без побочных эффектов и декларативного программирования.
- Функциональные методы характеризуются только входными аргументами и выходным результатом.
- Функция является функционально прозрачной, если всегда возвращает один и тот же результат при вызове с одним аргументом. Итеративные конструкции вроде циклов `while` можно заменить рекурсией.
- Хвостовая рекурсия, возможно, лучше классической, поскольку открывает путь потенциальной оптимизации компилятором.

19 Методики функционального программирования

В этой главе

- Полноправные функции, функции высшего порядка, каррирование и частичное применение.
- Персистентные структуры данных.
- Отложенное вычисление и отложенные списки как обобщение потоков данных Java.
- Сопоставление с шаблоном и имитация его в Java.
- Функциональная прозрачность и кэширование.

В главе 18 вы научились функциональному мышлению; мышление на языке методов без побочных эффектов позволяет писать удобный для сопровождения код. В этой главе мы познакомим вас с более продвинутыми методиками функционального программирования. В ней вы найдете смесь практических методик для применения в своем коде с теоретической информацией, которая сделает вас более грамотным программистом. Мы обсудим функции высшего порядка, каррирование, персистентные структуры данных, отложенные списки, сопоставление с шаблоном, кэширование с сохранением функциональной прозрачности и комбинаторы.

19.1. Повсюду функции

В главе 18 с помощью фразы «программирование в функциональном стиле» (функциональное программирование) мы указывали, что поведение функций и методов должно быть аналогичным математическим функциям, без каких-либо побочных

эффектов. Программисты, пишущие на функциональных языках программирования, часто применяют эту фразу в более общем смысле для обозначения того, что функции могут использоваться аналогично прочим значениям: передаваться в качестве аргументов, возвращаться в качестве результатов и храниться в структурах данных. Функции, которые можно использовать подобно прочим значениям, называют *полноправными функциями* (first-class functions). Полноправных функций в ранних версиях Java не было, они были добавлены в Java 8: любой метод теперь можно использовать в качестве функционального значения путем создания ссылки на метод с помощью оператора `::` или непосредственного выражения функциональных значений с помощью лямбда-выражений (таких как `(int x) -> x + 1`). В Java 8 вполне допустимо[1] сохранять метод `Integer.parseInt` в переменной с помощью ссылки на метод:

```
Function<String, Integer> strToInt = Integer::parseInt;
```

19.1.1. Функции высшего порядка

До сих пор мы в основном использовали полноправность функций только для передачи их операциям потоковой обработки Java 8 (как в главах 4–7) и для достижения эффекта параметризации поведения, подобного передаче `Apple::isGreen-Apple` в качестве функционального значения в метод `filterApples` в главах 1 и 2. Еще один встречавшийся нам интересный пример заключался в использовании статического метода `Comparator.comparing`, принимающего в качестве параметра функцию и возвращающего другую функцию (компаратор), как показано в следующем коде и на рис. 19.1.

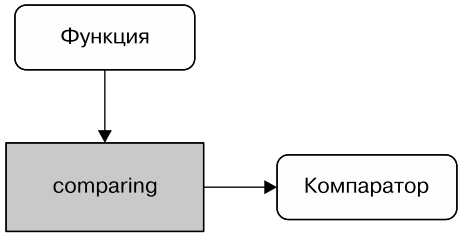


Рис. 19.1. Метод comparing принимает функцию в качестве параметра и возвращает другую функцию

```
Comparator<Apple> c = comparing(Apple::getWeight);
```

Мы уже делали нечто похожее, когда создавали конвейер операций в главе 3 путем формирования цепочки функций:

---

[1] Если вы собираетесь хранить в переменной strToInt только метод Integer::parseInt, то желательно объявить ее с типом ToIntFunction<String>, во избежание упаковки. Мы не делаем этого тут, поскольку это может сделать код менее понятным, даже если и повысит эффективность работы с простыми типами (данных).

```
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::checkSpelling)
             .andThen(Letter::addFooter);
```

Функции (вроде `Comparator.comparing`), которые могут выполнять что-то из нижеприведенного списка, называются в среде функциональных программистов *функциями высшего порядка* (higher-order function). Они могут:

- принимать одну или более функций в качестве параметров;
- возвращать функцию в качестве результата.

Этим характеристикам полностью соответствуют функции Java 8, поскольку их можно не только передавать в качестве аргументов, но и возвращать в качестве результатов, присваивать локальным переменным или даже вставлять в структуры данных. Программа-калькулятор, например, может включать ассоциативный массив `Map<String, Function<Double, Double>>`, содержащий отображение строкового значения `"sin"` на объект типа `Function<Double, Double>`, в котором хранится ссылка на метод `Math::sin`. Нечто подобное мы делали при изучении паттерна проектирования «Фабрика» в главе 8.

Читатели, которым понравился математический пример в конце главы 3, могут считать типом операции дифференцирования следующее:

```
Function<Function<Double,Double>, Function<Double,Double>>
```

поскольку она принимает функцию в виде аргумента (например, `(Double x) -> x * x`) и возвращает функцию в качестве результата (в данном случае `(Double x) -> 2 * x`). Мы записали этот код в виде функционального типа (самая левая функция), чтобы явным образом указать на возможность передачи этой функции дифференцирования в другую функцию. Впрочем, напомним, что вышеприведенный тип и сигнатура:

```
Function<Double,Double> differentiate(Function<Double,Double> func)
```

означают одно и то же.

Побочные эффекты и функции высшего порядка

Мы отмечали в главе 7, что у функций, передаваемых потоковым операциям, обычно отсутствуют побочные эффекты, и указывали на возникающие в противном случае проблемы (например, некорректные и даже непредсказуемые результаты вследствие состояний гонки, которые трудно предугадать заранее). Этот принцип в целом применим и к функциям высшего порядка. При написании функции или метода высшего порядка заранее не известно, какие аргументы будут в него передаваться, и если у аргументов будут побочные эффекты, то какие именно. Анализ выполняемых кодом действий сильно затрудняется в случае непредсказуемых изменений состояния программы пользовательскими функциями-аргументами; подобные функции могут даже вмешиваться в работу кода так, что это будет трудно обнаружить при отладке. Мы рекомендуем указывать в документации, какие побочные эффекты функций-аргументов допустимы. Лучше всего — никакие!

В следующем разделе мы займемся каррированием: методикой, упрощающей модульную организацию функций и переиспользование кода.

19.1.2. Каррирование

Прежде чем дать формальное определение каррирования, приведем пример. Почти все приложения используются в различных странах, так что задача преобразования из одной системы единиц в другую возникает очень часто.

Для преобразования единиц измерения всегда требуется коэффициент преобразования и, иногда, смещение точки отсчета. Например, формула преобразования градусов по Цельсию в градусы по Фаренгейту выглядит следующим образом: `CtoF(x) = x*9/5 + 32`. Стандартная схема всех преобразований единиц измерения такова.

1. Умножить на коэффициент преобразования.
2. Сдвинуть точку отсчета, если нужно.

Эту схему можно выразить с помощью следующего общего метода:

```
static double converter(double x, double f, double b) {
    return x * f + b;
}
```

Здесь `x` представляет собой преобразуемую величину, `f` — коэффициент преобразования, а `b` — смещение. Но этот метод слишком общий. Обычно приходится производить множество преобразований между одной и той же парой единиц измерения, например преобразовывать километры в мили. Конечно, можно все время вызывать метод `converter` с тремя аргументами, но указывать каждый раз коэффициент преобразования и смещение довольно утомительно и существует риск случайной опечатки.

Можно писать для каждого приложения новый метод, но при этом упускается возможность переиспользования логики.

Вот простой способ переиспользования существующей логики, с сохранением адаптации конвертера к конкретным приложениям. Он состоит в описании фабрики, генерирующей функции преобразования с одним аргументом, в качестве иллюстрации идеи каррирования:

```
static DoubleUnaryOperator curriedConverter(double f, double b){
    return (double x) -> x * f + b;
}
```

Остается только передать методу `curriedConverter` коэффициент преобразования и смещение (`f` и `b`), и он любезно вернет функцию (от аргумента `x`), которая будет производить требуемые действия. Теперь можно воспользоваться этой фабрикой для создания любого нужного конвертера:

```
DoubleUnaryOperator convertCtoF = curriedConverter(9.0/5, 32);
DoubleUnaryOperator convertUSDtoGBP = curriedConverter(0.6, 0);
DoubleUnaryOperator convertKmtoMi = curriedConverter(0.6214, 0);
```

А поскольку интерфейс `DoubleUnaryOperator` содержит метод `applyAsDouble`, то полученные конвертеры можно использовать следующим образом:

```
double gbp = convertUSDtoGBP.applyAsDouble(1000);
```

В результате код становится гибче и переиспользует уже существующую логику преобразования!

Задумайтесь над тем, что мы сделали. Вместо передачи методу `converter` одновременно всех аргументов, `x`, `f` и `b`, мы берем только аргументы `f` и `b` и возвращаем другую функцию, которая по заданному аргументу `x` возвращает `x * f + b`. Такой двухэтапный процесс дает возможность переиспользовать логику преобразования и создавать различные функции с различными коэффициентами преобразования.

Формальное определение каррирования

Каррирование — методика, при которой функция `f` двух аргументов (например, `x` и `y`) рассматривается как функция `g` одного аргумента, возвращающая функцию также одного аргумента. Возвращаемое этой последней функцией значение соответствует значению, возвращаемому исходной функцией, то есть `f(x,y) = (g(x))(y)`.

Конечно, это определение можно обобщить. Можно каррировать функцию шести аргументов в функцию, принимающую аргументы 2, 4 и 6 и возвращающую функцию, принимающую аргумент 5, которая возвращает функцию, принимающую оставшиеся аргументы 1 и 3.

Если передается часть аргументов (но не все), о функции говорят, что она *применена частично* (partially applied).

В следующем разделе мы обратимся к другому аспекту функционального программирования — к структурам данных. Можно ли использовать в программах структуры данных, если их запрещено модифицировать?

19.2. Персистентные структуры данных

Существует множество названий для структур данных, используемых в функциональных программах, например функциональные структуры данных и неизменяемые структуры данных, но чаще всего используется название *персистентные структуры данных* (к сожалению, эта терминология идет вразрез с понятием персистентности в базах данных, означающим «хранимый дольше одного выполнения программы»).

Прежде всего следует отметить, что функциональным методам не позволено обновлять никакие глобальные структуры данных или структуры, передаваемые в качестве параметров. Почему? Потому что при повторном вызове, вероятно, результаты окажутся другими, что нарушает функциональную прозрачность и не дает рассматривать метод как простое отображение аргументов в результаты.

19.2.1. Деструктивные и функциональные обновления

Начнем с возможных проблем. Пускай для представления поездок по железной дороге из пункта А в пункт Б используется изменяемый класс `TrainJourney` (простая реализация односвязного списка), в котором поле типа `int` моделирует какую-то информацию о поездке, например цену билета для текущего отрезка пути. Поездки с пересадками состоят из нескольких связанных посредством поля `onward` объектов `TrainJourney`; для прямого поезда или последнего отрезка пути поле `onward` содержит `null`:

```
class TrainJourney {
    public int price;
    public TrainJourney onward;
    public TrainJourney(int p, TrainJourney t) {
        price = p;
        onward = t;
    }
}
```

Теперь пусть у нас есть несколько отдельных объектов `TrainJourney`, представляющих поездки из X в Y и из Y в Z. Нам хотелось бы создать одну поездку, которая бы связывала два объекта `TrainJourney` (то есть поездку из X в Z через Y).

Вот простой традиционный императивный метод, связывающий эти две поездки на поезде:

```
static TrainJourney link(TrainJourney a, TrainJourney b){
    if (a==null) return b;
    TrainJourney t = a;
    while(t.onward != null){
        t = t.onward;
    }
    t.onward = b;
    return a;
}
```

В основе его работы лежит поиск последнего отрезка в объекте `TrainJourney a` и замена значения `null`, отмечающего конец списка `a`, на список `b` (учитываем также частный случай, когда `a` не содержит элементов).

Проблема в следующем: допустим, что переменная `firstJourney` содержит маршрут из X в Y, а переменная `secondJourney` — маршрут из Y в Z. При вызове `link(firstJourney, secondJourney)` вышеприведенный код деструктивно обновляет значение `firstJourney`, включая в него `secondJourney`, так что, хотя один пользователь, которому нужна поездка из X в Z, видит желаемый объединенный маршрут, вдобавок был деструктивно обновлен маршрут из X в Y. И действительно, переменная `firstJourney` содержит теперь не маршрут из X в Y, а маршрут из X в Z, что нарушает работу использующего ее немодифицированного кода! Представьте себе, что `firstJourney` представляет утренний поезд из Лондона в Брюссель и все последующие пользователи, желающие попасть в Брюссель, с удивлением обнаружат

дополнительный отрезок пути, скажем, до Кельна. Всем нам порой приходилось бороться с подобными ошибками, касающимися выбора степени видимости изменений в структурах данных.

Функциональный подход к этой проблеме состоит в запрете на использование таких методов с побочными эффектами. Если нужна структура данных для результата вычислений, следует создать новую, а не изменять уже существующую, как мы делали ранее. Чаще всего это будет рекомендуемой практикой и в обычном объектно-ориентированном программировании. Разработчики часто возражают, что функциональный подход приводит к излишнему копированию, и говорят, что они запомнят или укажут в документации побочные эффекты. Но подобный оптимизм приводит только к проблемам для тех, кому придется сопровождать код в дальнейшем. Функциональное решение выглядит следующим образом:

```
static TrainJourney append(TrainJourney a, TrainJourney b){
    return a==null ? b : new TrainJourney(a.price, append(a.onward, b));
}
```

Этот код явно функциональный (не производит изменений, даже локальных) и не модифицирует никаких существующих структур данных. Отметим, впрочем, что в нем не создается нового объекта `TrainJourney`. Если `a` представляет собой последовательность из n элементов, а `b` — из m элементов, код вернет последовательность из $n + m$ элементов, в которой первые n элементов — новые узлы, а последние m элементов разделяются с объектом `TrainJourney b`. Обратите внимание, что пользователям нельзя изменять результат `append`, поскольку при этом они могут нарушить информацию о поездах, передаваемую в виде последовательности `b`. Рисунки 19.2 и 19.3 иллюстрируют различия между деструктивным `append` и `append` в функциональном стиле.

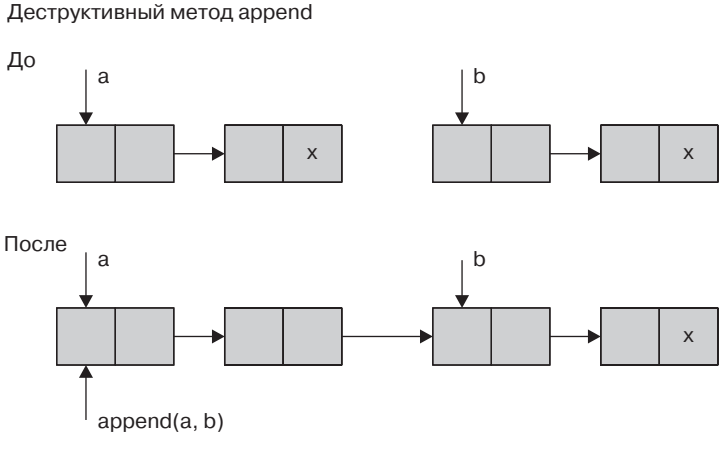


Рис. 19.2. Структура данных обновляется деструктивно

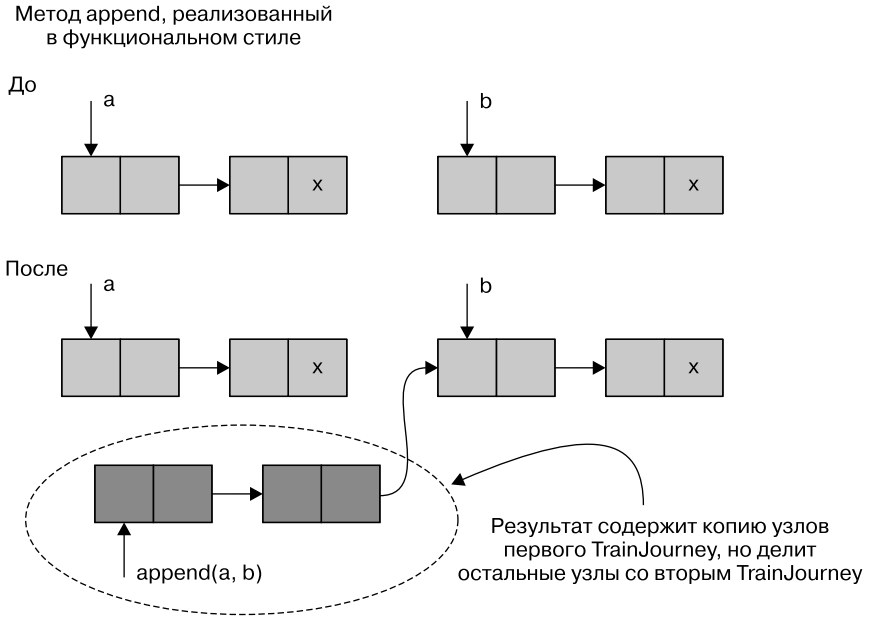


Рис. 19.3. Функциональный стиль без модификаций структуры данных

19.2.2. Другой пример, с деревьями

Прежде чем закончить обсуждение этой темы, рассмотрим еще одну структуру данных: двоичное дерево поиска, подходящее для реализации интерфейса, аналогичного `HashMap`. Идея в том, что дерево содержит строковое значение, соответствующее ключу, и `int`, соответствующий значению, например, имени и возрасту:

```
class Tree {
   private String key;
   private int val;
   private Tree left, right;
   public Tree(String k, int v, Tree l, Tree r) {
     key = k; val = v; left = l; right = r;
   }
}
class TreeProcessor {
    public static int lookup(String k, int defaultval, Tree t) {
        if (t == null) return defaultval;
        if (k.equals(t.key)) return t.val;
        return lookup(k, defaultval,
                      k.compareTo(t.key) < 0 ? t.left : t.right);
    }
    // Прочие методы для работы с деревом
}
```

Мы хотим использовать двоичное дерево поиска для поиска строковых значений с целью получения соответствующих им `int`. Взглянем, как обновить соответству-

ющее заданному ключу значение (предположим для простоты, что ключ уже присутствует в дереве):

```
public static void update(String k, int newval, Tree t) {
    if (t == null) { /* нужно добавить новый узел*/ }
    else if (k.equals(t.key)) t.val = newval;
    else update(k, newval, k.compareTo(t.key) < 0 ? t.left : t.right);
}
```

Добавление нового узла — более хитрая задача. Простейший способ: возвратить из метода `update` обходимый объект `Tree` (неизменный, если не требуется добавить узел). Код стал намного более неуклюжим, поскольку пользователь должен помнить, что метод `update` обновляет дерево без создания дополнительных структур, возвращая то же дерево, что было передано. А если исходное дерево было пустым, в качестве результата возвращается новый узел:

```
public static Tree update(String k, int newval, Tree t) {
    if (t == null)
       t = new Tree(k, newval, null, null);
    else if (k.equals(t.key))
       t.val = newval;
    else if (k.compareTo(t.key) < 0)
       t.left = update(k, newval, t.left);
    else
       t.right = update(k, newval, t.right);
    return t;
}
```

Отметим, что обе эти версии метода `update` изменяют существующее дерево, а значит, все пользователи хранящегося в этом дереве ассоциативного массива видят изменение.

19.2.3. Функциональный подход

Как запрограммировать подобные обновления деревьев функционально? Необходимо создать новый узел для новой пары «ключ/значение». Нужно также создать новые узлы на пути от корня дерева до этого нового узла:

```
public static Tree fupdate(String k, int newval, Tree t) {
     return (t == null) ?
         new Tree(k, newval, null, null) :
          k.equals(t.key) ?
            new Tree(k, newval, t.left, t.right) :
       k.compareTo(t.key) < 0 ?
         new Tree(t.key, t.val, fupdate(k,newval, t.left), t.right) :
         new Tree(t.key, t.val, t.left, fupdate(k,newval, t.right));
}
```

Вообще говоря, этот код не требует больших затрат ресурсов. Если глубина дерева d и оно относительно хорошо сбалансировано, количество записей будет около $2d$, так что мы создаем лишь небольшую его часть.

Мы написали этот код в виде простого условного выражения вместо оператора условного ветвления `if-then-else`, чтобы подчеркнуть: его тело представляет собой единое выражение без побочных эффектов. Но можно и написать эквивалентную цепочку `if-then-else`, каждая ветвь которой содержит оператор `return`.

Какова разница между `update` и `fupdate`? Мы уже отмечали: метод `update` предполагает, что каждый из пользователей готов использовать структуру данных совместно с другими пользователями и согласен видеть обновления, вызываемые работой других частей программы. Следовательно, в нефункциональном коде жизненно важно (хотя часто и упускается из виду) при добавлении какого-либо структурированного значения в дерево копировать его, поскольку рано или поздно кто-нибудь сочтет допустимым его обновление. Напротив, метод `fupdate` — чисто функциональный; он создает в качестве результата новый объект `Tree`, но делит как можно большую его часть с объектом-аргументом. Эта идея проиллюстрирована на рис. 19.4. Дерево состоит из узлов, в которых хранятся имена и возраст людей. Вызов `fupdate` не модифицирует существующее дерево, а создает новые узлы «сбоку» дерева, не нарушая существующей структуры данных.

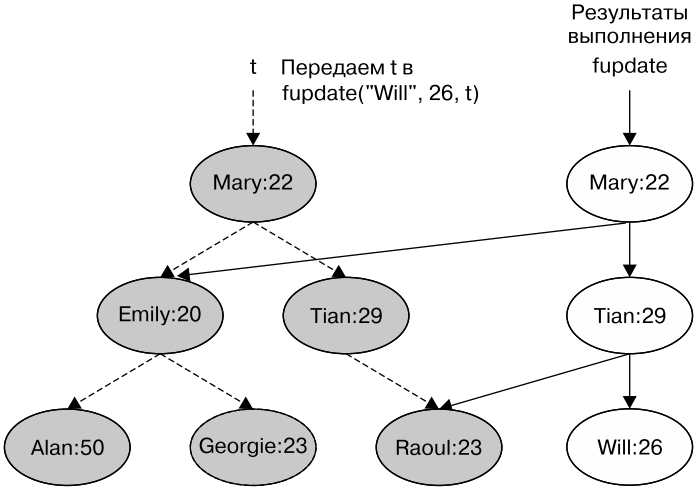


Рис. 19.4. При выполнении этого обновления дерева ни одна существующая структура данных не пострадала

Подобные функциональные структуры данных часто называют *персистентными* (persistent) — их значения сохраняются надолго и изолированы от происходящих в прочих местах изменений — так что программист может быть уверен, что метод `fupdate` не изменит передаваемой в него в качестве аргумента структуры данных. Одна оговорка: противоположная сторона данного соглашения обязывает всех пользователей персистентных структур данных следовать требованиям отсутствия изменений. В противном случае программисты, игнорирующие это условие, могут изменить результаты выполнения `fupdate` (путем изменения значения `20` для `Emily`, например).

В этом смысле метод `fupdate` может работать еще эффективнее. Правило «никаких изменений существующих структур» позволяет структурам, отличающимся лишь незначительно (например, как дерево, которое видит пользователь А, и модифицированная его версия, которую видит пользователь Б), совместно использовать хранилище для общих частей их структуры. Компилятор может обеспечить выполнение этого правила, для чего необходимо объявить поля `key`, `val`, `left` и `right` класса `Tree` как `final`. Но не забывайте, что этот модификатор `final` защищает только поле, а не объект, на который это поле ссылается, — у него тоже нужно объявить поля как `final` и т. д.

Вы можете заявить: «Мне *хотелось бы*, чтобы обновления дерева были видны части пользователей (а части пользователей — нет)». Есть два варианта решения этой задачи. Один как классическое решение Java: при обновлении чего-либо соблюдать осторожность и проверять, не нужно ли сначала скопировать эти данные. Второй вариант: решение в функциональном стиле — создавать новую структуру данных при каждом обновлении (так что никакие структуры никогда не изменяются) и передавать пользователям при необходимости нужную версию структуры данных. Обеспечить соблюдение этой идеи можно посредством API. Клиентам структуры данных, которые должны видеть обновления, следует обращаться через API, возвращающий наиболее свежую версию. А клиенты, которые не хотят видеть обновления (например, в случае долго работающих операций статистического анализа) могут использовать извлеченную ими копию в уверенности, что она не окажется измененной.

Эта методика напоминает обновление файла на CD-R, допускающего строго однократную запись файла путем прожига диска лазером. На компакт-диске сохраняется несколько версий файла (интеллектуальное ПО для записи дисков даже может совмещать общие части этих нескольких версий), а пользователь передает соответствующий адрес блока начала файла (или название, в котором указывается версия) для выбора нужной версии. В Java дела обстоят несколько лучше, чем с CD, в том смысле, что больше не используемые старые версии структуры данных удаляются при сборке мусора.

19.3. Отложенное вычисление с помощью потоков данных

В предыдущих главах вы видели, что потоки данных — отличный способ обработки коллекций данных. Но по различным причинам, включая эффективность реализации, создатели Java 8 добавили потоки данных в Java несколько своеобразным образом. Одно из ограничений — невозможность описать поток данных рекурсивно в силу однократности потребления потока. В следующих разделах мы продемонстрируем проблемы, возникающие вследствие этого.

19.3.1. Самоопределяющиеся потоки данных

Чтобы разобраться с идеей рекурсивного потока данных, вернемся к примеру генерации простых чисел из главы 6. В той главе было показано, что (например, в виде части класса `MyMathUtils`) вычислить поток простых чисел можно вот так:

```
public static Stream<Integer> primes(int n) {
    return Stream.iterate(2, i -> i + 1)
                 .filter(MyMathUtils::isPrime)
```

```
                .limit(n);
}
public static boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return IntStream.rangeClosed(2, candidateRoot)
                    .noneMatch(i -> candidate % i == 0);
}
```

Но это решение — довольно неуклюжее. Оно включает проход в цикле по всем числам для проверки, не делится ли на него потенциальное простое число (на практике, достаточно проверить только числа, которые уже классифицированы как простые).

В идеале поток данных должен отфильтровывать числа, которые делятся на простые числа, генерируемые потоком данных. Этот процесс выглядит примерно так.

1. Необходим поток целых чисел, из которого будут выбираться простые.
2. Берем из этого потока данных первое число (ведущий элемент потока данных), которое будет простым (на первом шаге это число 2).
3. Отфильтровываем все числа, делящиеся на это число, из конца потока данных.
4. Полученный хвост потока данных становится нашим новым потоком чисел. Фактически мы возвращаемся на шаг 1, так что алгоритм носит рекурсивный характер.

Отметим, что это плохой алгоритм, по нескольким причинам[1], но его удобно анализировать в смысле работы с потоками данных. В следующих разделах мы попробуем реализовать его с помощью Stream API.

Шаг 1: получаем поток чисел

Получить бесконечный поток чисел, начинающийся с 2, можно с помощью метода `IntStream.iterate` (описанного в главе 5):

```
static Intstream numbers(){
    return IntStream.iterate(2, n -> n + 1);
}
```

Шаг 2: выбираем из него ведущий элемент

В интерфейсе `IntStream` есть метод `findFirst`, с помощью которого можно получить первый элемент:

```
static int head(IntStream numbers){
    return numbers.findFirst().getAsInt();
}
```

Шаг 3: фильтрация хвоста

Описываем метод для получения хвоста потока данных:

```
static IntStream tail(IntStream numbers){
    return numbers.skip(1);
}
```

[1] Больше информации о том, почему этот алгоритм плох, можно найти в статье http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf.

Произвести фильтрацию чисел при заданном ведущем элементе потока можно следующим образом:

```
IntStream numbers = numbers();
int head = head(numbers);
IntStream filtered = tail(numbers).filter(n -> n % head != 0);
```

Шаг 4: рекурсивное создание потока простых чисел

Здесь кроется небольшой подвох. Наверное, вам кажется, что можно просто вернуть полученный отфильтрованный поток данных, чтобы можно было взять его ведущий элемент и продолжить фильтрацию чисел, вот так:

```
static IntStream primes(IntStream numbers) {
    int head = head(numbers);
    return IntStream.concat(
              IntStream.of(head),
              primes(tail(numbers).filter(n -> n % head != 0))
            );
}
```

Плохие новости

К сожалению, при запуске кода из шага 4 возвращается ошибка: `java.lang.IllegalStateException: stream has already been operated upon or closed`. И действительно, мы использовали две завершающие операции для разбиения потока данных на ведущий элемент и хвост: `findFirst` и `skip`. Как вы помните из главы 4, с вызовом для потока данных завершающей операции его потребление заканчивается навсегда!

Отложенное вычисление

Есть и еще одна, более существенная проблема: статический метод `IntStream.concat` ожидает на входе два экземпляра потока данных, но его второй аргумент представляет собой прямой рекурсивный вызов `primes`, что приводит к бесконечной рекурсии! Для многих задач Java ограничения работы с потоками данных, например запрет рекурсивных определений, не вызывает никаких проблем и лишь повышает выразительность и возможности распараллеливания запросов в стиле баз данных. Таким образом, разработчики Java 8 выбрали оптимальный вариант. Тем не менее удобными дополнениями вашего арсенала программиста могли бы стать более общие возможности и модели потоков данных из функциональных языков программирования, таких как Scala и Haskell. Необходим просто способ отложенного вычисления вызовов метода `primes` из второго аргумента `concat` (на более формальном языке эту концепцию называют *отложенным вычислением* (lazy evaluation), *нестрогим вычислением* (nonstrict evaluation) или даже *вызовом по имени* (call by name)). Поток данных следует вычислять лишь при необходимости выполнения действий над простыми числами (например, с помощью метода `limit`). Язык Scala поддерживает такие возможности. На языке Scala можно описать вышеприведенный алгоритм следующим образом:

```
def numbers(n: Int): Stream[Int] = n #:: numbers(n+1)
def primes(numbers: Stream[Int]): Stream[Int] = {
    numbers.head #:: primes(numbers.tail filter (n => n % numbers.head != 0))
}
```

где оператор `#::` производит отложенную конкатенацию (аргументы вычисляются лишь тогда, когда возникает необходимость потребления потока данных).

Не беспокойтесь, если что-то в этом коде непонятно. Его единственная задача — продемонстрировать различия между Java и другими функциональными языками программирования. Не помешает задуматься на минутку о способе вычисления аргументов в различных языках. При вызове метода в языке Java аргументы полностью вычисляются немедленно. Но при использовании оператора `#::` в языке Scala возврат из операции конкатенации производится немедленно, а элементы вычисляются лишь по мере необходимости.

В следующем подразделе мы займемся реализацией идеи отложенных списков на языке Java.

19.3.2. Наш собственный отложенный список

Потоки данных Java 8 часто описывают как отложенные. Отложенными они являются в том смысле, что поток данных ведет себя как «черный ящик», генерирующий значения по запросу. При применении к потоку данных последовательности операций они просто сохраняются на будущее. Что-либо вычисляется только при применении к потоку данных завершающей операции. Эта задержка очень удобна при применении к потоку данных нескольких операций (например, `filter` и `map`, за которыми следует завершающая операция `reduce`): обход потока совершается лишь один раз, а не для каждой операции.

В этом разделе мы обсудим понятие отложенных списков — одной из разновидностей более общего понятия потоков данных (отложенные списки — концепция, схожая с потоками данных). Кроме того, на примере отложенных списков очень удобно говорить про функции высшего порядка. Функциональное значение помещается в структуру данных, где оно может не использоваться большую часть времени, но при обращении (по требованию) способно увеличить количество элементов этой структуры. Эту идею иллюстрирует рис. 19.5.

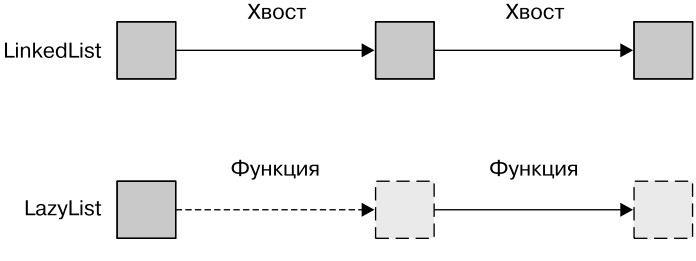


Рис. 19.5. Элементы LinkedList уже существуют в оперативной памяти (разнесены по ней). А элементы LazyList создаются функцией по требованию; их можно рассматривать как разнесенные во времени

Далее мы посмотрим на эту концепцию в действии. Мы собираемся сгенерировать бесконечный список простых чисел с помощью вышеописанного алгоритма.

Создание простого связного списка

Напомним, что в Java можно описать простой класс `MyLinkedList` для связного списка следующим образом (с минимальным интерфейсом `MyList`):

```
interface MyList<T> {
    T head();
    MyList<T> tail();
    default boolean isEmpty() {
        return true;
    }
}
class MyLinkedList<T> implements MyList<T> {
    private final T head;
    private final MyList<T> tail;
    public MyLinkedList(T head, MyList<T> tail) {
        this.head = head;
        this.tail = tail;
    }
    public T head() {
        return head;
    }
    public MyList<T> tail() {
        return tail;
    }
    public boolean isEmpty() {
        return false;
    }
}
class Empty<T> implements MyList<T> {
    public T head() {
        throw new UnsupportedOperationException();
    }
    public MyList<T> tail() {
        throw new UnsupportedOperationException();
    }
}
```

Теперь можно сформировать значение типа `MyLinkedList` вот так:

```
MyList<Integer> l =
    new MyLinkedList<>(5, new MyLinkedList<>(10, new Empty<>()));
```

Создание простейшего списка с отложенным вычислением

Простейший способ воплотить в этом классе концепцию отложенных списков — сделать так, чтобы хвост списка не присутствовал в оперативной памяти целиком, а генерировался поэлементно с помощью поставщика (объекта `Supplier<T>`), который мы обсуждали в главе 3 (его можно также рассматривать как фабрику

с функциональным дескриптором `void -> T`). При такой архитектуре код будет выглядеть следующим образом:

```
import java.util.function.Supplier;
class LazyList<T> implements MyList<T>{
    final T head;
    final Supplier<MyList<T>> tail;
    public LazyList(T head, Supplier<MyList<T>> tail) {
        this.head = head;
        this.tail = tail;
    }
    public T head() {
        return head;
    }
    public MyList<T> tail() {
        return tail.get();
    }
    public boolean isEmpty() {
        return false;
    }
}
```

Обратите внимание, что использование поставщика для хвоста списка создает эффект отложенности, по сравнению с ведущей частью списка

В результате вызова метода `get` из поставщика создается узел списка `LazyList` (подобно тому как фабрика создает новый объект).

Теперь для создания бесконечного отложенного списка чисел, начинающегося с *n*, достаточно передать в качестве аргумента `tail` конструктора `LazyList` поставщик, который создает следующий элемент ряда чисел:

```
public static LazyList<Integer> from(int n) {
    return new LazyList<Integer>(n, () -> from(n+1));
}
```

При выполнении следующего кода в консоль выводится `2 3 4`. Конечно, числа генерируются по требованию. Для проверки вставим соответствующий оператор `System.out.println` и просто отметим, что `from(2)` выполнялся бы бесконечно, если бы пытался вычислить все числа, начиная с 2:

```
LazyList<Integer> numbers = from(2);
int two = numbers.head();
int three = numbers.tail().head();
int four = numbers.tail().tail().head();
System.out.println(two + " " + three + " " + four);
```

Возвращаемся к генерации простых чисел

Давайте взглянем, как воспользоваться вышеизложенным для генерации самоопределяющегося отложенного списка простых чисел (того, что нам не удавалось сделать с помощью Stream API). Если переделать вышеприведенный код, который использовал Stream API, на основе нового класса `LazyList`, получится примерно следующее:

```
public static MyList<Integer> primes(MyList<Integer> numbers) {
    return new LazyList<>(
```

```
                numbers.head(),
                () -> primes(
                        numbers.tail()
                               .filter(n -> n % numbers.head() != 0)
                            )
    );
}
```

Реализация отложенного фильтра

К сожалению, в классе `LazyList` (а точнее, в интерфейсе `List`) не описан метод `filter`, так что предыдущий код не скомпилируется! Для решения этой проблемы объявим метод `filter`:

```
public MyList<T> filter(Predicate<T> p) {
    return isEmpty() ?
           this :
           p.test(head()) ?
               new LazyList<>(head(), () -> tail().filter(p)) :
               tail().filter(p);
}
```

Можно вернуть new Empty<>(), но this подходит не хуже и тоже пуст

Код компилируется и готов к использованию! Первые три простых числа можно вычислить, связав цепочкой вызовы `tail` и `head`, вот так:

```
LazyList<Integer> numbers = from(2);
int two = primes(numbers).head();
int three = primes(numbers).tail().head();
int five = primes(numbers).tail().tail().head();
System.out.println(two + " " + three + " " + five);
```

Этот код выводит в консоль `2 3 5` — первые три простых числа. Теперь можно с ним поэкспериментировать. Например, можно вывести в консоль все простые числа (программа будет работать бесконечно, выполняя метод `printAll`, который итеративно выводит ведущий и хвостовой элементы списка):

```
static <T> void printAll(MyList<T> list){
    while (!list.isEmpty()){
        System.out.println(list.head());
        list = list.tail();
    }
}
printAll(primes(from(2)));
```

Поскольку эта глава посвящена функциональному программированию, нельзя не упомянуть, что этот код можно переписать в изящном рекурсивном виде:

```
static <T> void printAll(MyList<T> list){
    if (list.isEmpty())
        return;
    System.out.println(list.head());
    printAll(list.tail());
}
```

Однако эта программа не будет работать бесконечно. К сожалению, в конце концов она завершится сбоем из-за переполнения стека, поскольку Java не поддерживает оптимизацию хвостовой рекурсии, как обсуждалось в главе 18.

Критический обзор

Мы создали немало кода с помощью отложенных списков и функций, где они использовались лишь для описания структуры данных, содержащей все простые числа. Но где они применяются на практике? Мы видели, как поместить функции внутрь структур данных (поскольку Java 8 предоставляет разработчику такую возможность), и эти функции можно использовать для создания частей структуры данных по требованию, а не при создании самой структуры. Эта возможность может быть очень полезна при написании компьютерных игр, например шахмат, для создания структуры данных, условно хранящей все дерево возможных ходов (заведомо слишком большое для немедленного вычисления, но вполне доступное для вычисления отложенного). Этой структурой данных будет отложенное дерево, а не отложенный список. Мы сосредоточили в данной главе наше внимание на отложенных списках, поскольку это позволило нам обсудить достоинства и недостатки потоков данных по сравнению с отложенными списками.

Остается вопрос производительности. Несложно догадаться, что отложенное выполнение лучше немедленного. И действительно, лучше вычислить только необходимые программе в данный момент значения и структуры данных, чем создавать все эти значения (а может, и еще больше), как при обычном выполнении. К сожалению, на практике не все так просто. Накладные расходы отложенного выполнения (например, дополнительные поставщики между элементами нашего `LazyList`) перевешивают относительное преимущество, если анализируется, скажем, более 10 % структуры данных. Наконец, есть нюанс, делающий наши значения `LazyList` не вполне отложенными. При обходе значения `LazyList`, например `from(2)`, до десятого элемента все узлы создаются по два раза, то есть 20 вместо 10. Подобный результат вряд ли можно назвать отложенным выполнением. Проблема в том, что поставщик в `tail` вызывается снова и снова при каждом просмотре по требованию списка `LazyList`. Исправить эту ситуацию можно, если сделать так, чтобы поставщик в `tail` вызывался лишь при первом просмотре по требованию, а полученное значение кэшировалось, фактически фиксируя список по состоянию на этот момент. Для этого добавьте в определение класса `LazyList` поле `private Optional<LazyList<T>> alreadyComputed` и сделайте так, чтобы метод `tail` обращался к нему и обновлял его соответствующим образом.

Чисто функциональный язык программирования Haskell устроен так, что все его структуры данных являются отложенными в этом смысле. Если вам интересно узнать больше — прочитайте какую-либо из множества статей по языку Haskell.

Запомните, что отложенные структуры данных могут оказаться полезными инструментами и не помешают в вашем арсенале разработчика. Их имеет смысл использовать, когда они упрощают написание приложения; если они снижают производительность — лучше переписать их в более традиционном стиле.

В следующем разделе мы поговорим о возможности, имеющейся практически во всех функциональных языках программирования, за исключением Java: сопоставлении с шаблоном.

19.4. Сопоставление с шаблоном

У того, что обычно считают функциональным программированием, есть и еще одна важная отличительная черта — (структурное) *сопоставление с шаблоном* (pattern matching) (не путать с сопоставлением с шаблоном применительно к регулярным выражениям). Глава 1 заканчивалась наблюдением относительно возможности в математике писать определения вида:

```
f(0) = 1
f(n) = n*f(n-1) в противном случае
```

в то время как в Java приходится писать оператор `if-then-else` или `switch`. По мере усложнения типов данных количество необходимого для их обработки кода (и создаваемый беспорядок) растет. А благодаря использованию сопоставления с шаблоном можно этот беспорядок уменьшить.

Для иллюстрации разберем древовидную структуру данных, которую нужно обойти. Рассмотрим простой арифметический язык, состоящий из чисел и двоичных операций:

```
class Expr { ... }
class Number extends Expr { int val; ... }
class BinOp extends Expr { String opname; Expr left, right; ... }
```

Предположим, что нам нужно написать метод для упрощения определенных выражений. Например, 5 + 0 можно упростить до 5. С помощью нашего класса `Expr` можно упростить `new BinOp("+", new Number(5), new Number(0))` до `Number(5)`. Произвести обход структуры `Expr` можно следующим образом:

```
Expr simplifyExpression(Expr expr) {
    if (expr instanceof BinOp
          && ((BinOp)expr).opname.equals("+"))
          && ((BinOp)expr).right instanceof Number
          && ... // код становится очень неуклюжим
          && ... ) {
       return (Binop)expr.left;
     }
     ...
}
```

Как видите, код очень быстро становится безобразным!

19.4.1. Паттерн проектирования «Посетитель»

Другой способ распаковки данных подобного типа — применение паттерна проектирования «Посетитель». При этом, по существу, создается отдельный класс, который инкапсулирует алгоритм «посещения» конкретного типа данных.

Класс-посетитель принимает на входе конкретный экземпляр типа данных, после чего может обращаться ко всем его членам. Вот конкретный пример. Во-первых, добавьте в класс `BinOp` метод `accept`, принимающий в качестве аргумента объект типа `SimplifyExprVisitor` и передающий в него себя (нужно еще добавить аналогичный метод в класс `Number`):

```
class BinOp extends Expr{
    ...
    public Expr accept(SimplifyExprVisitor v){
        return v.visit(this);
    }
}
```

Теперь объект `SimplifyExprVisitor` сможет получить доступ к объекту `BinOp` и распаковать его:

```
public class SimplifyExprVisitor {
    ...
    public Expr visit(BinOp e){
        if("+".equals(e.opname) && e.right instanceof Number && …){
            return e.left;
        }
        return e;
    }
}
```

19.4.2. На помощь приходит сопоставление с шаблоном

Более простое решение состоит в применении возможности, именуемой *сопоставлением с шаблоном*. Эта возможность недоступна в Java, так что для ее иллюстрации мы приведем несколько маленьких примеров кода на языке Scala. Эти примеры дадут вам представление о возможностях, которые могли бы быть в языке Java, если бы он поддерживал сопоставление с шаблоном.

Код для разбора арифметического выражения типа `Expr` на языке Scala (который мы выбрали за сходство синтаксиса с языком Java) выглядит следующим образом:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e   // Прибавляем ноль
    case BinOp("*", e, Number(1)) => e   // Умножаем на единицу
    case BinOp("/", e, Number(1)) => e   // Делим на единицу
    case _ => expr                       // Невозможно упростить выражение
}
```

С помощью подобного сопоставления с шаблоном можно очень лаконично и выразительно производить операции над различными древовидными структурами данных. Обычно такая методика применяется для создания компиляторов и механизмов обработки бизнес-правил. Отметим, что синтаксис Scala:

```
Expression match { case Pattern => Expression ... }
```

аналогичен синтаксису Java:

```
switch (Expression) { case Constant : Statement ... }
```

где последнее выражение `case _` играет роль `default:` в Java благодаря джокерному шаблону (`_`). Наиболее заметное синтаксическое различие состоит в том, что язык Scala ориентирован на обработку выражений, а Java — скорее операторов. Но для программиста главное различие — в смысле выразительности: шаблоны Java в метках `case` ограничены несколькими простыми типами данных, перечислимыми типами, парочкой специальных классов-адаптеров для определенных простых типов данных и строковыми значениями. На практике одно из важнейших преимуществ языков программирования, поддерживающих сопоставление с шаблоном, — отсутствие длинных цепочек операторов `switch` или `if-then-else`, переплетенных с операциями выбора полей.

Совершенно очевидно, что по выразительности сопоставление с шаблоном языка Scala намного опережает язык Java, и можно лишь надеяться, что в будущих версиях Java выразительность операторов `switch` повысится (наши конкретные предложения по этому поводу вы найдете в главе 21).

Тем временем мы покажем вам, как реализовать подобный сопоставлению с шаблоном код на языке Java с помощью лямбда-выражений. Цель нашего описания этой методики состоит исключительно в демонстрации еще одного интересного способа применения лямбда-выражений.

Имитация сопоставления с шаблоном на языке Java

Во-первых, взгляните, насколько велики возможности выражения `match` сопоставления с шаблоном в языке Scala. Выражение:

```
def simplifyExpression(expr: Expr): Expr = expr match {
    case BinOp("+", e, Number(0)) => e
    ...
```

означает «проверить, относится ли `expr` к классу `BinOp`, извлечь три его компонента (`opname`, `left`, `right`), после чего сопоставить эти компоненты с шаблонами — первый со строкой `+`, второй — с переменной `e` (всегда соответствует) и третий — с шаблоном `Number(0)`». Другими словами, сопоставление с шаблоном в Scala (и многих других функциональных языках программирования) является многоуровневым. Сопоставление с шаблоном, моделируемое с помощью лямбда-выражений Java 8, является лишь одноуровневым. В предыдущем примере мы смогли бы выразить только `BinOp(op, l, r)` или `Number(n)`, но не `BinOp("+", e, Number(0))`.

Во-первых, обнаружился удивительный факт: при наличии лямбда-выражений можно в принципе не использовать `if-then-else`. Код вроде `condition ? e1 : e2` можно заменить вызовом метода:

```
myIf(condition, () ->  e1, () ->  e2);
```

А где-то, например в библиотеке, будет находиться определение (обобщенное по типу `T`):

```
static <T> T myIf(boolean b, Supplier<T> truecase, Supplier<T> falsecase) {
    return b ? truecase.get() : falsecase.get();
}
```

Тип `T` играет роль типа результата для условного выражения. В принципе, подобное можно сделать и для других конструкций управления потоком выполнения программы, например `switch` и `while`.

В обычном коде подобное только затуманит происходящее, поскольку `if-then-else` прекрасно передает соответствующую идиому. Но, как мы отмечали, `switch` и `if-then-else` не передают идиомы сопоставления с шаблоном, и оказывается, что лямбда-выражения могут выражать (одноуровневое) сопоставление с шаблоном несколько более изящно, чем цепочки `if-then-else`.

Возвращаемся к сопоставлению с шаблоном для значений класса `Expr` (у которого есть два подкласса — `BinOp` и `Number`). Можно описать метод `patternMatchExpr` (опять же обобщенный по типу `T` — типу результата сопоставления с шаблоном):

```
interface TriFunction<S, T, U, R>{
    R apply(S s, T t, U u);
}
static <T> T patternMatchExpr(
                        Expr e,
                        TriFunction<String, Expr, Expr, T> binopcase,
                        Function<Integer, T> numcase,
                        Supplier<T> defaultcase) {
    return
     (e instanceof BinOp) ?
        binopcase.apply(((BinOp)e).opname, ((BinOp)e).left,
                                           ((BinOp)e).right) :
     (e instanceof Number) ?
        numcase.apply(((Number)e).val) :
        defaultcase.get();
}
```

В результате следующий вызов:

```
patternMatchExpr(e, (op, l, r) -> {return  binopcode ;},
                    (n) -> {return  numcode ;},
                    () -> {return  defaultcode ;});
```

определяет, относится ли `e` к типу `BinOp` (и если да, выполняет метод `binopcase`, у которого есть доступ к полям `BinOp` через идентификаторы `op, l, r`) или `Number` (и если да, выполняет метод `numcode`, у которого есть доступ к значению `n`). В этом методе даже предусмотрен `defaultcode`, который будет выполняться, если кто-то позднее создаст узел дерева, не являющийся ни `BinOp`, ни `Number`.

В листинге 19.1 показано, как упрощать выражения для сложения и умножения с помощью метода `patternMatchExpr`.

Листинг 19.1. Реализация сопоставления с шаблоном для упрощения выражения

```
public static Expr simplify(Expr e) {
    TriFunction<String, Expr, Expr, Expr> binopcase =
        (opname, left, right) -> {
            if ("+".equals(opname)) {
```

Обработка выражения BinOp

Обработка случая сложения

```
                if (left instanceof Number && ((Number) left).val == 0) {
                    return right;
                }
                if (right instanceof Number && ((Number) right).val == 0) {
                    return left;
                }
            }                                          Обработка случая умножения
            if ("*".equals(opname)) {
                if (left instanceof Number && ((Number) left).val == 1) {
                    return right;
                }
                if (right instanceof Number && ((Number) right).val == 1) {
                    return left;
                }
            }
            return new BinOp(opname, left, right);
        };                                                    Обработка Number
Function<Integer, Expr> numcase = val -> new Number(val);
Supplier<Expr> defaultcase = () -> new Number(0);
return patternMatchExpr(e, binopcase, numcase, defaultcase);
```

Обработка по умолчанию, на случай, если выражение пользователя не получается распознать

Применение сопоставления с шаблоном

Теперь можно вызвать метод `simplify` вот так:

```
Expr e = new BinOp("+", new Number(5), new Number(0));
Expr match = simplify(e);
System.out.println(match);    ←——— Выводит 5
```

Вы уже встретили много нового в этой главе: функции высшего порядка, каррирование, персистентные структуры данных, отложенные списки и сопоставление с шаблоном. В следующем разделе мы поговорим о некоторых нюансах, которые отложили на после, чтобы излишне не усложнять текст.

19.5. Разное

В этом разделе мы изучим два нюанса функциональности и функциональной прозрачности, один из которых касается эффективности, а второй — возврата того же самого результата. Это интересные вопросы, но мы поместили их обсуждение в данный раздел, поскольку они касаются побочных эффектов и не являются принципиально важными. Мы также поговорим об идее *комбинаторов* — методов или функций, принимающих на входе две и более функции и возвращающих другую функцию. Эта идея вдохновила множество дополнений к API Java 8, а позднее и Flow API Java 9.

19.5.1. Кэширование или мемоизация

Пускай у нас есть метод без побочных эффектов `computeNumberOfNodes(Range)`, подсчитывающий число узлов в заданном диапазоне сети с древовидной топологией. Допустим, что сеть не меняется (то есть структура неизменяема), а для

вызова метода `computeNumberOfNodes` требуется достаточно много ресурсов в силу необходимости рекурсивного обхода структуры. Результаты его работы могут понадобиться не один раз. В случае функциональной прозрачности можно легко избежать этих дополнительных накладных расходов. Одно из стандартных решений — *мемоизация* (memoization) — добавление к методу адаптера для кэширования (например, в структуре `HashMap`). Во-первых, этот адаптер будет обращаться к кэшу и проверять, не содержится ли там уже пара «аргумент, результат». Если да, он сразу возвращает сохраненный результат. В противном случае вызывается метод `computeNumberOfNodes`, но перед возвратом из адаптера новая пара «аргумент, результат» сохраняется в кэше. Строго говоря, это решение не является чисто функциональным, поскольку оно изменяет структуру данных, совместно используемую несколькими вызывающими сторонами, но эту обернутую версию кода можно считать функционально прозрачной.

На практике такой код работает следующим образом:

```
final Map<Range,Integer> numberOfNodes = new HashMap<>();
Integer computeNumberOfNodesUsingCache(Range range) {
    Integer result = numberOfNodes.get(range);
    if (result != null){
        return result;
    }
    result = computeNumberOfNodes(range);
    numberOfNodes.put(range, result);
    return result;
}
```

ПРИМЕЧАНИЕ

Для подобных случаев в Java 8 интерфейс Map был дополнен (см. приложение Б) методом computeIfAbsent. С его помощью можно написать более понятный код:

```
Integer computeNumberOfNodesUsingCache(Range range) {
    return numberOfNodes.computeIfAbsent(range,
                                         this::computeNumberOfNodes);
}
```

Метод `computeNumberOfNodesUsingCache` явно функционально прозрачен (при условии, что метод `computeNumberOfNodes` также функционально прозрачен). Но изменяемое разделяемое состояние `NumberOfNodes` и то, что `HashMap` — не `synchronized`[1], означает, что данный код не является потокобезопасным. Даже использование защищаемого блокировками класса `Hashtable` или класса `ConcurrentHashMap` (конкурентного без блокировок) вместо `HashMap` не гарантирует ожидаемой производительности при параллельных обращениях к `NumberOfNodes` из различных ядер

[1] Именно здесь главный рассадник ошибок. При использовании класса HashMap легко можно забыть (или просто не обратить внимания в силу однопоточности текущей программы), что документация Java отмечает его не многопоточное исполнение.

процессора. Между обнаружением факта отсутствия `range` в ассоциативном массиве и вставкой в него пары «аргумент, результат» возникает состояние гонки, а значит, вычисление одного и того же значения для добавления к ассоциативному массиву могут производить несколько процессов.

Наверное, главный вывод, который можно сделать из всего этого, — то, что смешивать изменяемое состояние с конкурентностью небезопасно. При функциональном программировании такой практики избегают, за исключением низкоуровневых трюков вроде кэширования. Второй вывод: при программировании в функциональном стиле никогда не нужно заботиться о том, синхронизирован ли другой функциональный метод, если только не считать трюков вроде кэширования, поскольку *точно известно*, что у него нет разделяемого изменяемого состояния.

19.5.2. Что значит «вернуть тот же самый объект»?

Рассмотрим опять пример с двоичным деревом из подраздела 19.2.3. На рис. 19.4 переменная `t` указывает на уже существующий объект `Tree`, и рисунок демонстрирует эффект от вызова `fupdate("Will", 26, t)` для создания нового объекта `Tree`, который, вероятно, присваивается переменной `t2`. Из рисунка ясно, что переменная `t` и все доступные из нее структуры данных не изменяются. Предположим, что мы выполняем точно такой же вызов в дополнительном операторе присваивания:

```
t3 = fupdate("Will", 26, t);
```

Теперь переменная `t3` указывает еще на три только что созданных узла, содержащих те же данные, что и узлы из `t2`. Вопрос в том, является ли функция `fupdate` функционально прозрачной. Функциональная прозрачность означает «одинаковые аргументы (как раз наш случай) приводят к одинаковым результатам». Проблема в том, что ссылки у `t2` и `t3` различны, так что `(t2 == t3)` равно `false`, и, похоже, нужно признать функцию функционально непрозрачной. Хотя при использовании немодифицируемых персистентных структур данных никаких логических различий между переменными `t2` и `t3` нет.

На эту тему можно долго спорить, но в функциональном программировании для сравнения структурированных значений обычно используется `equals`, а не `==` (равенство по ссылкам), поскольку данные не изменяются, а при такой модели функция `fupdate` функционально прозрачна.

19.5.3. Комбинаторы

В функциональном программировании часто можно встретить функцию высшего порядка (скажем, в виде метода), принимающую, допустим, две функции и возвращающую третью функцию, каким-либо образом сочетающую две первых. Для описания такой концепции обычно используют термин *«комбинатор»* (combinator). Эта идея лежит в основе значительной части API Java 8, например метода `thenCombine` из класса `CompletableFuture`, в который можно передать два объекта, `CompletableFuture` и `BiFunction`, для получения еще одного объекта, `CompletableFuture`.

Хотя подробное обсуждение комбинаторов в функциональном программировании выходит за рамки данной книги, стоит взглянуть на несколько частных случаев, чтобы осознать, что принимающие и возвращающие функции операции — широко распространенная и естественная конструкция функционального программирования. Следующий метод воплощает идею композиции функций:

```
static <A,B,C> Function<A,C> compose(Function<B,C> g, Function<A,B> f) {
    return x -> g.apply(f.apply(x));
}
```

Этот метод принимает в качестве аргументов функции `f` и `g` и возвращает функцию, которая выполняет сначала `f`, а затем `g`. Далее можно описать в виде комбинатора операцию, воплощающую идею внутренней итерации. Допустим, что вам нужно применить функцию `f` к полученным данным `n` раз, как в цикле. Наша операция (назовем ее `repeat`) принимает на входе функцию `f`, описывающую происходящее в одной итерации цикла, и возвращает функцию, описывающую происходящее при `n` итерациях. Вызов:

```
repeat(3, (Integer x) -> 2*x);
```

должен возвращать `x ->(2*(2*(2*x))`, что эквивалентно `x -> 8*x`.

Протестировать этот код можно с помощью следующего оператора:

```
System.out.println(repeat(3, (Integer x) -> 2*x).apply(10));
```

который выводит в консоль `80`.

Код для метода `repeat` выглядит таким образом (обратите внимание на частный случай цикла из 0 итераций):

```
static <A> Function<A,A> repeat(int n, Function<A,A> f) {
    return n==0 ? x -> x
                : compose(f, repeat(n-1, f));
}
```

Возвращает ничего не делающую тождественную функцию, если n равно 0

В противном случае выполняем функцию f n – 1 раз, а потом еще один раз

Используя эту идею, можно моделировать более продвинутые варианты итерации, включая функциональную модель передачи между итерациями изменяемого состояния. Но время двигаться дальше. Задача этой главы заключалась лишь в обзоре функционального программирования в достаточном для изучения Java 8 объеме. Существует множество замечательных книг, в которых функциональное программирование изучается гораздо глубже.

Резюме

- Полноправными называются функции, которые можно передавать в качестве аргументов, возвращать как результаты и хранить в структурах данных.
- Функции высшего порядка принимают одну или несколько функций на входе и возвращают другую функцию. В числе типичных функций высшего порядка в Java — `comparing`, `andThen` и `compose`.
- Каррирование — методика, упрощающая модульную организацию и переиспользование кода.
- Персистентная структура данных сохраняет свою предыдущую версию при модификации. В результате предотвращается излишнее защитное копирование.
- Потоки данных в языке Java не могут быть самоопределяющимися.
- Отложенные списки — более выразительные версии потоков данных Java. Благодаря отложенным спискам можно генерировать элементы списка по требованию, с помощью поставщика, который может создавать дополнительные элементы структуры данных.
- Сопоставление с шаблоном — функциональная возможность, предназначенная для распаковки типов данных. Сопоставление данных можно рассматривать как обобщение оператора `switch` языка Java.
- Функциональная прозрачность дает возможность кэширования результатов вычислений.
- Комбинаторы — функциональные идеи, служащие для объединения двух или более функций или других структур данных.

20 Смесь ООП и ФП: сравнение Java и Scala

В этой главе

- Введение в Scala.
- Взаимосвязь Java и Scala.
- Функции в Java по сравнению с функциями Scala.
- Классы и типажи.

Scala — язык программирования, в котором смешиваются объектно-ориентированное и функциональное программирование. Его часто рассматривают в качестве альтернативного варианта разработчики, желающие использовать функциональные возможности в статически типизированном языке программирования, работающем на JVM, с сохранением духа Java. Возможности Scala намного шире, чем Java: более совершенная система типов, вывод типов, сопоставление с шаблоном (как вы уже видели в главе 19), языковые конструкции, упрощающие описание предметно-ориентированных языков и т. д. Кроме того, из кода Scala можно обращаться ко всем библиотекам Java.

Наверное, вы недоумеваете, зачем нужна посвященная Scala глава в книге о Java. Основное внимание в данной книге было посвящено принятию на вооружение разработчика функционального стиля программирования в Java. Scala, как и Java, поддерживает функциональную обработку коллекций (то есть потокоподобные операции), полноправные функции и методы с реализацией по умолчанию. Но Scala идет дальше и предоставляет намного больше возможностей по сравнению с Java. Полагаем, вам будет интересно сравнить подходы Scala и Java и увидеть ограничения Java. В этой главе мы хотели бы пролить свет на данный вопрос и утолить ваше любопытство. Мы не настаиваем на использовании Scala вместо Java. Заслуживают внимания и другие

новые языки программирования на JVM, например Kotlin. Цель главы состоит лишь в расширении вашего кругозора в плане имеющихся, помимо Java, возможностей. Мы убеждены, что всесторонне образованный разработчик должен быть в курсе всей экосистемы языков программирования, а не только используемого им языка.

Имейте также в виду, что задача этой главы вовсе не в том, чтобы научить вас писать идиоматический код на Scala или рассказать вам о Scala все. Scala поддерживает множество недоступных в Java возможностей (например, сопоставление с шаблоном, for-включения и неявные параметры, преобразования, классы и т. д.), о которых мы не будем говорить. Наша цель — сравнить возможности Java и Scala и представить вам полную картину. Например, показать, что на Scala можно писать более лаконичный и удобочитаемый код, по сравнению с Java.

Эта глава начинается с введения в Scala: написания простых программ и работы с коллекциями. Далее мы обсудим функции в Scala: полноправные функции, замыкания и каррирование. И наконец, рассмотрим классы в Scala и типажи — возможность, бросающую вызов интерфейсам и методам с реализацией по умолчанию.

20.1. Введение в Scala

В этом разделе мы вкратце познакомим вас с основными возможностями Scala, необходимыми для написания простых программ на этом языке. Начнем со слегка видоизмененного примера Hello world, написанного в императивном и функциональном стилях. Затем мы рассмотрим некоторые из поддерживаемых Scala структур данных — `List`, `Set`, `Map`, `Stream`, `Tuple` и `Option` — и сравним их с аналогами в Java. Наконец, мы продемонстрируем *типажи*, которые в Scala заменяют интерфейсы Java и также поддерживают наследование методов во время создания объектов.

20.1.1. Приложение Hello beer

Отойдем немного от классического Hello world и поменяем мир (world) на пиво (beer). Мы хотим получить в консоли следующее:

```
Hello 2 bottles of beer
Hello 3 bottles of beer
Hello 4 bottles of beer
Hello 5 bottles of beer
Hello 6 bottles of beer
```

Императивный код на Scala

```
object Beer {
  def main(args: Array[String]){
    var n : Int = 2
    while( n <= 6){
      println(s"Hello ${n} bottles of beer")
      n += 1
    }
  }
}
```

Строковая интерполяция

Инструкции по запуску этого кода вы можете найти на официальном сайте Scala (https://docs.scala-lang.org/getting-started.html). Внешний вид этой программы напоминает программы на Java, да и структура похожа: состоит из одного метода с названием `main`, принимающего в качестве аргумента массив строковых значений (синтаксис аннотаций типов имеет вид `s : String` вместо `String s`, как в Java). Метод `main` не возвращает значения, и описывать возвращаемый тип в Scala не обязательно, в то время как в Java необходимо использовать ключевое слово `void`.

ПРИМЕЧАНИЕ

Вообще говоря, в нерекурсивных объявлениях методов в Scala явный тип возвращаемого значения указывать не обязательно, поскольку Scala может сам вывести этот тип.

Прежде чем заняться телом метода `main`, необходимо обсудить объявление `object`. В конце концов, в Java внутри класса требуется объявлять метод `main`. Объявление `object` представляет собой объект-одиночку: объявление класса `Beer` с одновременным созданием его экземпляра. Создается только один экземпляр. Это первый пример классического паттерна проектирования (паттерна проектирования «Одиночка»), реализованного в виде встроенной возможности языка. Кроме того, можно считать объявленные внутри `object` методы статическими, именно поэтому сигнатура метода `main` и не содержит явным образом модификатора `static`.

Теперь взглянем на тело метода `main`. Он также напоминает методы Java, но точка с запятой в конце операторов не обязательна. Тело состоит из цикла `while` с наращиванием значения изменяемой переменной `n`. Для каждого нового значения `n` строковое значение выводится на экран с помощью встроенного метода `println`. Строка с `println` демонстрирует еще одну возможность Scala — *строковую интерполяцию* (string interpolation), позволяющую использовать переменные и выражения прямо в строковых литералах. В предыдущем коде мы использовали переменную `n` прямо внутри строкового литерала `s"Hello ${n} bottles of beer"`. Это возможно благодаря указанию перед строковым значением интерполятора `s`. В Java обычно для этого приходится выполнять явную конкатенацию, вот так: `"Hello " + n + " bottles of beer"`.

Функциональный код на Scala

После наших разговоров про функциональное программирование на протяжении всей книги хотелось бы знать, что может предложить в этом плане Scala. Предыдущий код можно переписать в более функциональном виде на Java, вот так:

```
public class Foo {
    public static void main(String[] args) {
        IntStream.rangeClosed(2, 6)
                 .forEach(n -> System.out.println("Hello " + n +
                                                  " bottles of beer"));
    }
}
```

А вот как этот код выглядит на Scala:

```
object Beer {
  def main(args: Array[String]){
    2 to 6 foreach { n => println(s"Hello ${n} bottles of beer") }
  }
}
```

Код на Scala похож на Java-код, но более лаконичен. Во-первых, можно создавать диапазоны с помощью выражений вида `2 to 6`. Потрясающий факт: `2` представляет собой объект типа `Int`. В Scala абсолютно *все* является объектами; понятия простых типов данных, как в Java, здесь нет, так что Scala — полностью объектно-ориентированный язык программирования. Класс `Int` в Scala поддерживает метод `to`, который принимает в качестве аргумента другой объект `Int` и возвращает диапазон. Вместо вышеприведенного кода можно было написать `2.to(6)`, но принимающие один аргумент методы можно записывать в инфиксной форме. Далее, `foreach` (с буквой `e` в нижнем регистре) аналогичен оператору `forEach` в Java (с буквой `E` в верхнем регистре). Этот метод применим к диапазонам (тоже используется инфиксная форма записи), он принимает в качестве аргумента лямбда-выражение, которое применяется к каждому из элементов. Синтаксис лямбда-выражений напоминает их синтаксис в Java, но стрелка имеет вид `=>` вместо `->`[1]. Предыдущий код является функциональным, поскольку не происходит изменения значения переменной, как в предыдущем примере с циклом `while`.

20.1.2. Основные структуры данных Scala: List, Set, Map, Stream, Tuple и Option

Как вы себя чувствуете, утолив жажду парой бутылок пива? Большинству настоящих программ приходится хранить данные и производить над ними различные операции, так что в этом разделе мы займемся операциями над коллекциями Scala и сравним этот процесс с Java.

Создание коллекций

Создавать коллекции в Scala очень просто, благодаря присущей Scala лаконичности. Например, вот как можно создать ассоциативный массив:

```
val authorsToAge = Map("Raoul" -> 23, "Mario" -> 40, "Alan" -> 53)
```

Несколько вещей новы для нас в этой строке кода. Во-первых, впечатляет возможность создания ассоциативного массива и связывания ключей со значениями непосредственно, с помощью синтаксиса `->`. Нет необходимости добавлять элементы вручную, как в Java:

```
Map<String, Integer> authorsToAge = new HashMap<>();
authorsToAge.put("Raoul", 23);
authorsToAge.put("Mario", 40);
authorsToAge.put("Alan", 53);
```

[1] Обратите внимание, что в языке Scala для аналога лямбда-выражений Java используются попеременно термины «анонимные функции» (anonymous functions) и «замыкания» (closures).

Впрочем, из главы 8 вы узнали, что в Java 9 под влиянием Scala появилось несколько фабричных методов, упрощающих подобный код:

```
Map<String, Integer> authorsToAge
    = Map.ofEntries(entry("Raoul", 23),
                    entry("Mario", 40),
                    entry("Alan", 53));
```

Во-вторых, можно не указывать тип переменной `authorsToAge`. Мы могли явным образом написать `val authorsToAge : Map[String, Int]`, но Scala может вывести информацию о типе этой переменной (отметим, что код все равно проверяется статически. Тип всех переменных известен на этапе компиляции). Мы вернемся к этой возможности в главе 21. В-третьих, вместо ключевого слова `var` мы используем `val`. В чем различие между ними? Ключевое слово `val` означает, что переменная предназначена только для чтения и повторное присвоение ей значения невозможно (подобно модификатору `final` в Java). Ключевое слово `var` означает, что переменная доступна как для чтения, так и для записи.

А как насчет других типов коллекций? Можно легко создать `List` (односвязные списки) или `Set` (множество без дубликатов):

```
val authors = List("Raoul", "Mario", "Alan")
val numbers = Set(1, 1, 2, 3, 5, 8)
```

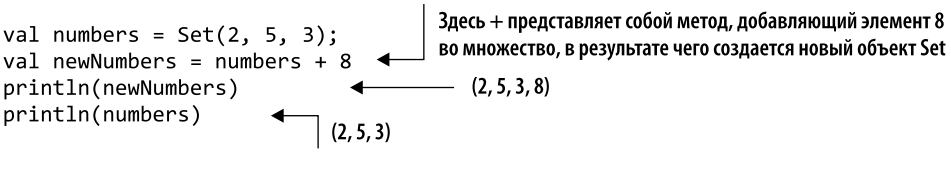
Переменная `authors` включает три элемента, а переменная `numbers` — пять.

Неизменяемые и изменяемые коллекции

Важно не забывать, что созданные выше коллекции неизменяемы по умолчанию, то есть их нельзя менять после создания. Благодаря неизменяемости вы можете быть уверены, что при обращении к коллекции в любой момент времени вы получите коллекцию из одних и тех же элементов.

Как же обновлять неизменяемые коллекции в Scala? В рамках терминологии главы 19, подобные коллекции в Scala можно назвать *персистентными*. Обновление коллекции приводит к созданию новой коллекции, разделяющей как можно большую часть с предыдущей версией, которая сохраняется, никак не затронутая изменениями (как мы продемонстрировали на рис. 19.3 и 19.4). Вследствие этого свойства в коде оказывается меньше неявных зависимостей данных: меньше путаница относительно того, из какого места кода и в какой момент времени происходит обновление коллекции (или какой-либо другой совместно используемой структуры данных).

Эту идею иллюстрирует следующий пример. Добавим элемент в `Set`:



В примере не модифицируется числовое множество, а создается новый объект `Set` с дополнительным элементом.

Заметим, что Scala не заставляет использовать неизменяемые коллекции, а всего лишь упрощает воплощение идеи неизменяемости в коде. Кроме того, изменяемые версии можно найти в пакете `scala.collection.mutable`.

Немодифицируемые и неизменяемые коллекции

В Java есть несколько способов создания немодифицирумых коллекций. В следующем коде переменная `newNumbers` представляет собой предназначенное только для чтения представление множества `numbers`:

```
Set<Integer> numbers = new HashSet<>();
Set<Integer> newNumbers = Collections.unmodifiableSet(numbers);
```

Этот код значит, что добавлять новые элементы через переменную `newNumbers` вы не сможете. Но немодифицируемая коллекция представляет собой лишь адаптер для модифицируемой коллекции, так что вы все равно сможете добавлять элементы посредством обращения к переменной `numbers`.

Напротив, неизменяемые коллекции гарантируют невозможность изменения коллекции, независимо от того, сколько переменных на нее указывает.

Мы объясняли в главе 19, как создать персистентную структуру данных — неизменяемую структуру данных, сохраняющую свою предыдущую версию при модификации. При любых ее модификациях создается новая структура данных.

Операции над коллекциями

Теперь, когда мы разобрались, как создавать коллекции, давайте выясним, что с ними можно делать. Коллекции в Scala поддерживают операции, аналогичные Stream API Java. Наверное, вы узнаете операции `filter` и `map` в следующем примере и на рис. 20.1:

```
val fileLines = Source.fromFile("data.txt").getLines.toList()
val linesLongUpper
  = fileLines.filter(l => l.length() > 10)
             .map(l => l.toUpperCase())
```

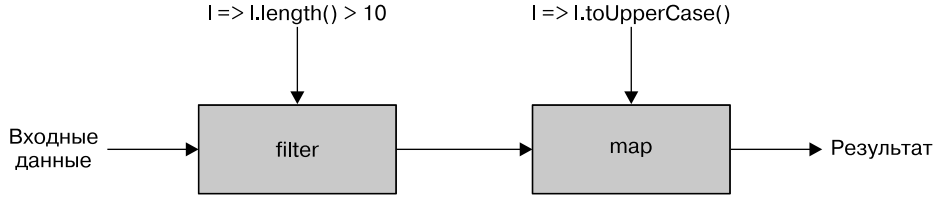


Рис. 20.1. Потокоподобные операции над списками в языке Scala

Первая строка преобразует файл в список строковых значений, состоящих из строк файла (подобно методу `Files.readAllLines` в Java). Вторая строка создает конвейер из двух операций:

- `filter`, выбирающей только строки с длиной более 10;
- `map`, преобразующей эти длинные строки в верхний регистр.

Этот код можно также написать следующим образом:

```
val linesLongUpper = fileLines filter (_.length() > 10) map(_.toUpperCase())
```

Мы воспользовались инфиксной нотацией, а также джокерным символом подчеркивания (`_`), который соответствует произвольному аргументу на соответствующей позиции. В данном случае `_.length()` можно рассматривать как `l => l.length()`. В передаваемых в операции `filter` и `map` функциях символ подчеркивания привязан к параметру, соответствующему обрабатываемой строке.

В Collection API Scala есть еще множество удобных операций. Мы рекомендуем вам заглянуть в документацию Scala, чтобы составить о них представление (https://docs.scala-lang.org/overviews/collections/introduction.html). Заметим, что этот API обладает бóльшими возможностями, чем Stream API Java (включая поддержку операции `zip`, предназначенной для объединения элементов двух списков), так что вы наверняка вынесете из этого прочтения несколько полезных идиом программирования. Эти идиомы вполне могут появиться в Stream API в следующих версиях Java.

Наконец, вспомним, что в Java доступно выполнение конвейера в параллельном режиме путем вызова для потока данных метода `parallel`. В Scala тоже есть такая возможность. Для этого достаточно воспользоваться методом `par`:

```
val linesLongUpper = fileLines.par filter (_.length() > 10) map(_.toUpperCase())
```

Кортежи

В этом пункте мы рассмотрим еще одну возможность, которая зачастую мучительно нелаконично выражается в языке Java, — кортежи. С помощью кортежей можно, например, сгруппировать людей по имени и номеру телефона (в данном случае это просто пары) без объявления специального нового класса и создания объекта: `("Raoul", "+44 7700 700042")`, `("Alan", "+44 7700 700314")` и т. д.

К сожалению, Java не поддерживает кортежи, так что приходится создавать свои собственные структуры данных. Вот простой класс `Pair`:

```
public class Pair<X, Y> {
    public final X x;
    public final Y y;
    public Pair(X x, Y y){
        this.x = x;
        this.y = y;
    }
}
```

Кроме того, конечно, приходится создавать экземпляры пар явным образом:

```
Pair<String, String> raoul = new Pair<>("Raoul", "+44 7700 700042");
Pair<String, String> alan = new Pair<>("Alan", "+44 7700 700314");
```

Ладно, допустим, но как насчет троек и кортежей произвольного размера? Описывать отдельный класс для каждого из размеров кортежей утомительно и в конечном итоге влияет на удобочитаемость и удобство сопровождения программ.

Scala предоставляет для этой цели кортежные литералы, позволяющие создавать кортежи с помощью простого «синтаксического сахара» в обычной математической нотации:

```
val raoul = ("Raoul", "+44 7700 700042")
val alan = ("Alan", "+44 7700 700314")
```

Scala поддерживает и кортежи произвольного размера[1], так что возможно и следующее:

```
val book = (2018 "Modern Java in Action", "Manning")
val numbers = (42, 1337, 0, 3, 14)
```

Кортеж с элементами типов (Int, String, String)

Кортеж с элементами типов (Int, Int, Int, Int, Int)

Получить доступ к элементам кортежей можно по их позиции в кортеже с помощью методов доступа `_1`, `_2` (нумерация начинается с 1), как показано в следующем примере:

```
println(book._1)
println(numbers._4)
```

Выводит в консоль 2018

Выводит в консоль 3

Разве не изящнее этот пример, чем код, который нам пришлось написать на Java? Хорошая новость: сейчас идет активное обсуждение внедрения кортежных литералов в будущие версии Java (см. более подробную информацию о планируемых новых возможностях Java в главе 21).

Потоки данных

Все обсуждавшиеся до сих пор коллекции — `List`, `Set`, `Map` и `Tuple` — вычисляются немедленно. Мы уже знаем, что потоки данных в Java вычисляются по требованию (отложенным образом). Как мы видели в главе 5, благодаря этому свойству потоки данных можно использовать для представления бесконечной последовательности, не опасаясь переполнения памяти.

В Scala также есть соответствующая отложенная структура данных — `Stream`. Но потоки данных в Scala обладают намного более широкими возможностями, чем потоки данных в Java. Потоки данных в Scala запоминают вычисленные значения, так что можно обращаться к предыдущим элементам. Кроме того, потоки данных

[1] С ограничением 22 элемента.

в Scala индексируются, так что к элементам можно обращаться по индексу, как в списке. Обратите внимание, что за эти дополнительные возможности приходится платить бóльшим расходом памяти, по сравнению с потоками данных Java, поскольку для обращения к предыдущим элементам необходимо их запоминать (кэшировать).

Класс Option

Еще одна знакомая нам структура данных — класс `Option`, представляющий собой Scala-версию класса `Optional` языка Java, который мы обсуждали в главе 11. Мы говорили, что для улучшения API следует применять опционалы везде, где только возможно, чтобы пользователь мог по сигнатуре метода сразу сказать, можно ли ожидать возврата значения-опционала. Эту структуру данных следует использовать вместо `null` везде, где только возможно, чтобы избежать генерации исключений, связанных с нулевым указателем.

Как вы видели в главе 11, опционал можно использовать для возврата названия застраховавшей человека компании в случае, если возраст этого человека превышает определенное пороговое значение:

```
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

Класс `Option` в Scala используется аналогично классу `Optional` в Java:

```
def getCarInsuranceName(person: Option[Person], minAge: Int) =
  person.filter(_.age >= minAge)
        .flatMap(_.car)
        .flatMap(_.insurance)
        .map(_.name)
        .getOrElse("Unknown")
```

Структура и названия методов в этом коде совпадают, за исключением `getOrElse` — аналога метода `orElse` из Java. Как видите, новые понятия, которые вы узнаете из этой книги, непосредственно применимы к другим языкам программирования! К сожалению, в Scala также существует `null`, из соображений совместимости с Java, но использовать его крайне не рекомендуется.

20.2. Функции

Функции Scala можно рассматривать как последовательности инструкций, сгруппированных для выполнения единой задачи. Они удобны для абстрагирования поведения и составляют краеугольный камень функционального программирования.

Вы хорошо знакомы с *методами* Java — функциями класса. Вы также встречали лямбда-выражения, которые можно считать анонимными функциями. Scala поддерживает более широкий набор возможностей для функций, чем Java, о которых мы и поговорим в этом разделе. Scala предоставляет следующие возможности.

- *Функциональные типы* — «синтаксический сахар», соответствующий дескрипторам функций в Java (то есть нотации для сигнатуры объявленного в функциональном интерфейсе абстрактного метода), которые мы описывали в главе 3.
- Анонимные функции, у которых нет ограничений на запись нелокальных переменных, в отличие от лямбда-выражений Java.
- Поддержка *каррирования* — разбиения принимающей несколько аргументов функции на последовательность функций, каждая из которых принимает часть этих аргументов.

20.2.1. Полноправные функции

Функции в языке Scala являются *полноправными значениями* (first-class values), то есть их можно передавать в качестве параметров, возвращать в качестве результата и сохранять в переменных, подобно значениям типов `Integer` и `String`. Как мы уже показывали в предыдущих главах, ссылки на методы и лямбда-выражения в Java тоже можно рассматривать как полноправные функции.

Приведем пример работы полноправных функций в Scala. Пускай у нас есть список строковых значений, соответствующих полученным твитам. Необходимо отфильтровать этот список в соответствии с различными критериями, например выбрать твиты с упоминанием слова *Java* или твиты определенной длины. Эти два критерия можно представить в виде предикатов (возвращающих `Boolean` функций):

```
def isJavaMentioned(tweet: String) : Boolean = tweet.contains("Java")
def isShortTweet(tweet: String) : Boolean = tweet.length() < 20
```

В Scala можно передать эти методы непосредственно во встроенный метод `filter` (аналогично передаче их с помощью ссылок на методы в Java):

```
val tweets = List(
    "I love the new features in Java",
    "How's it going?",
    "An SQL query walks into a bar, sees two tables and says 'Can I join
you?'"
)
tweets.filter(isJavaMentioned).foreach(println)
tweets.filter(isShortTweet).foreach(println)
```

Теперь посмотрим на сигнатуру встроенного метода `filter`:

```
def filter[T](p: (T) => Boolean): List[T]
```

Наверное, вам интересно, что означает параметр `p` (в данном случае с дескриптором `(T) => Boolean`), поскольку в Java следовало бы ожидать в этом месте функциональный

интерфейс. Подобный синтаксис Scala в Java пока еще недоступен, но он описывает *функциональный тип* (function type). Этот тип здесь соответствует функции, принимающей на входе объект типа `T` и возвращающей `Boolean`. В Java этот тип можно описать в виде `Predicate<T>` или `Function<T, Boolean>` с той же сигнатурой, что и методы `isJavaMentioned` и `isShortTweet`, так что их можно передать в качестве аргументов в метод `filter`. Разработчики языка Java решили не использовать подобный синтаксис для функциональных типов, чтобы сохранить совместимость с предыдущими версиями языка (появление слишком большого количества нового синтаксиса в очередной версии языка — лишняя головная боль для разработчиков, которым придется к нему привыкать).

20.2.2. Анонимные функции и замыкания

Scala также поддерживает анонимные функции, синтаксис которых схож с синтаксисом лямбда-выражений. В следующем примере переменной `isLongTweet` присваивается анонимная функция, проверяющая длину конкретного твита:

```
val isLongTweet : String => Boolean
    = (tweet : String) => tweet.length() > 60
```

Переменная функционального типа String => Boolean

Анонимная функция

В Java с помощью лямбда-выражения можно создать экземпляр функционального интерфейса. Есть подобный механизм и в Scala. Вышеприведенный код представляет собой «синтаксический сахар» для объявления анонимного класса типа `scala.Function1` (функция с одним параметром), реализующего метод `apply`:

```
val isLongTweet : String => Boolean
    = new Function1[String, Boolean] {
        def apply(tweet: String): Boolean = tweet.length() > 60
      }
```

А поскольку в переменной `isLongTweet` содержится объект типа `Function1`, допустимо вызвать метод `apply`, который можно рассматривать как вызов этой функции:

```
isLongTweet.apply("A very short tweet")
```

Возвращает false

В Java можно написать вот так:

```
Function<String, Boolean> isLongTweet = (String s) -> s.length() > 60;
boolean long = islongTweet.apply("A very short tweet");
```

Java предоставляет несколько встроенных функциональных интерфейсов (например, `Predicate`, `Function` и `Consumer`) для лямбда-выражений. Scala для этой цели предоставляет типажи (можете пока что считать их интерфейсами): начиная с `Function0` (функция без параметров и возвращаемого результата) и заканчивая `Function22` (функция с 22 параметрами), во всех из которых есть метод `apply`.

Еще один ловкий трюк Scala позволяет неявно вызывать метод `apply` с помощью «синтаксического сахара», похожего на вызов функции:

```
isLongTweet("A very short tweet")  ◄—— Возвращает false
```

Компилятор автоматически преобразует вызов `f(a)` в `f.apply(a)` и в общем случае вызов `f(a1, ..., an)` в вызов `f.apply(a1, ..., an)`, если `f` — объект типа, поддерживающего метод `apply` (обратите внимание, что у `apply` может быть произвольное число аргументов).

Замыкания

В главе 3 мы обсуждали, являются ли лямбда-выражения замыканиями. *Замыкание* (closure) — это экземпляр функции, который может без каких-либо ограничений ссылаться на нелокальные переменные этой функции. Но у лямбда-выражений в Java есть ограничение: они не могут модифицировать содержимое локальных переменных метода, в котором они определены. Эти переменные оказываются неявным образом терминальными. Удобно считать, что лямбда-выражения замыкают *значения*, а не *переменные*.

Напротив, анонимные функции в Scala могут захватывать сами переменные, а не *значения*, на которые эти переменные в данный момент ссылаются. В Scala возможен, например, такой код:

```
def main(args: Array[String]) {
    var count = 0
    val inc = () => count+=1   ◄—— Замыкание, захватывающее и увеличивающее на 1 переменную count
    inc()
    println(count)             ◄—— Выводит в консоль 1
    inc()
    println(count)             ◄—— Выводит в консоль 2
}
```

Но в Java нижеприведенный код приведет к ошибке компиляции, поскольку переменная `count` является неявным образом терминальной[1]:

```
public static void main(String[] args) {
    int count = 0;
    Runnable inc = () -> count+=1;   ◄—— Ошибка: переменная count должна быть объявлена final или фактически являться таковой
    inc.run();
    System.out.println(count);
    inc.run();
}
```

В главах 7, 18 и 19 мы приводили аргументы в пользу того, что изменяемости следует избегать по мере возможности, чтобы упростить распараллеливание и сопровождение программ, так что старайтесь использовать эту возможность только в случае крайней необходимости.

[1] Хотя и не объявлена с модификатором final. — *Примеч. пер.*

20.2.3. Каррирование

В главе 19 мы описали методику под названием *«каррирование»*, при которой функция `f` двух аргументов (допустим, `x` и `y`) рассматривается как функция `g` одного аргумента, которая возвращает другую функцию также одного аргумента. Это определение можно обобщить на случай функций нескольких аргументов, возвращающих несколько функций одного аргумента. Другими словами, принимающую несколько аргументов функцию можно разбить на последовательность функций, каждая из которых принимает на входе некое подмножество этих аргументов. Scala предоставляет языковую конструкцию, сильно упрощающую каррирование существующей функции.

Чтобы разобраться, что предлагает Scala, взглянем еще раз на пример на Java. Опишем простой метод для умножения двух целых чисел:

```
static int multiply(int x, int y) {
  return x * y;
}
int r = multiply(2, 10);
```

Это определение требует передачи всех аргументов. Метод `multiply` можно вручную разбить так, чтобы он возвращал другую функцию:

```
static Function<Integer, Integer> multiplyCurry(int x) {
    return (Integer y) -> x * y;
}
```

Возвращаемая методом `multiply` функция захватывает значение `x`, умножает его на аргумент `y` и возвращает объект типа `Integer`. В результате функцию `multiplyCurry` можно использовать в операции `map` для умножения всех элементов на 2:

```
Stream.of(1, 3, 5, 7)
      .map(multiplyCurry(2))
      .forEach(System.out::println);
```

В результате выполнения этого кода выводится `2, 6, 10, 14`. Он работает, поскольку операция `map` ожидает в качестве аргумента функцию, и `multiplyCurry` возвращает функцию.

Создавать каррированные формы функций на Java, разбивая их вручную, немного утомительно, особенно в случае функции с несколькими аргументами. В Scala существует специальный синтаксис для автоматизации этого процесса. Обычный метод `multiply` можно описать следующим образом:

```
def multiply(x : Int, y: Int) = x * y
val r = multiply(2, 10)
```

А вот каррированная форма:

```
def multiplyCurry(x :Int)(y : Int) = x * y
val r = multiplyCurry(2)(10)
```

Описание каррированной функции

Вызов каррированной функции

При использовании синтаксиса `(x: Int)(y: Int)` метод `multiplyCurry` принимает *два* списка аргументов, состоящих каждый из одного параметра типа `Int`. И напротив, метод `multiply` принимает *один* список аргументов, состоящий из двух параметров типа `Int`. Что происходит при вызове метода `multiplyCurry`? Первый вызов `multiplyCurry` с одним объектом `Int` (параметр `x`), `multiplyCurry(2)`, возвращает другую функцию, которая принимает параметр `y` и умножает его на захваченное значение `x` (в данном случае равное 2). В подобном случае говорят, что функция *применена частично*, поскольку указаны не все аргументы. Второй вызов приводит к перемножению значений `x` и `y`. Первое обращение к `multiplyCurry` можно сохранить в переменной и переиспользовать его позднее:

```
val multiplyByTwo : Int => Int = multiplyCurry(2)
val r = multiplyByTwo(10)                          ←——— 20
```

По сравнению с Java в Scala не требуется вручную указывать каррированную форму функции, как в предыдущем примере. В Scala имеется удобный синтаксис описания функций для указания на наличие у функции нескольких каррированных списков аргументов.

20.3. Классы и типажи

В этом разделе мы сравним классы и интерфейсы Java с их аналогами в Scala. Эти две языковые конструкции жизненно важны для проектирования приложений. Вам предстоит убедиться, что классы и интерфейсы Scala обеспечивают бóльшую гибкость, чем их аналоги в Java.

20.3.1. Код с классами Scala становится лаконичнее

Поскольку Scala — полноценный объектно-ориентированный язык программирования, в нем можно создавать классы и их экземпляры. Простейшая форма синтаксиса объявления класса и создания его экземпляра аналогична Java. Например, объявить класс `Hello` можно следующим образом:

```
class Hello {
  def sayThankYou(){
    println("Thanks for reading our book")
  }
}
val h = new Hello()
h.sayThankYou()
```

Геттеры и сеттеры

Интересное в Scala начинается, когда у класса есть поля. Случалось ли вам сталкиваться с классом Java, в котором просто описан список полей и нужно объявить длинный список геттеров, сеттеров и соответствующий конструктор? Настоящая головная боль! Кроме того, нередко можно встретить тесты для реализации каждого

из методов. Подобные классы обычно занимают значительный объем кода в корпоративных приложениях Java. Рассмотрим простой класс `Student`:

```
public class Student {
    private String name;
    private int id;
    public Student(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

Нам пришлось вручную описать конструктор для инициализации всех полей, два геттера и два сеттера. Такой простой класс теперь занимает почти 20 строк кода. В генерации его могут помочь несколько IDE (интегрированных сред разработки) и утилит, но база кода все равно будет включать большое количество лишнего кода, не слишком полезного по сравнению с настоящей бизнес-логикой.

В Scala конструкторы, геттеры и сеттеры можно сгенерировать неявным образом, благодаря чему код становится более лаконичным:

```
class Student(var name: String, var id: Int)
val s = new Student("Raoul", 1)
println(s.name)
s.id = 1337
println(s.id)
```

Инициализация объекта Student

Получаем имя и выводим в консоль Raoul

Задаем значение поля id

Выводим в консоль 1337

В Java можно добиться подобного поведения за счет описания общедоступных полей, но конструктор все равно придется описать явным образом. Классы Scala позволяют сэкономить на стереотипном коде.

20.3.2. Типажи Scala и интерфейсы Java

В Scala имеется еще одна возможность, полезная для абстрагирования поведения, — *типажи* (traits), которые заменяют в Scala интерфейсы Java. В типажах могут объявляться как абстрактные методы, так и методы с реализацией по умолчанию. Типажи поддерживают множественное наследование, как и интерфейсы в Java, так что их можно рассматривать как аналог интерфейсов Java, поддерживающих методы с реа-

лизацией по умолчанию. Типажи также могут включать такие поля, как абстрактные классы — возможность, не поддерживаемая интерфейсами Java. Напоминают ли типажи абстрактные классы? Нет, поскольку, в отличие от абстрактных классов, типажи поддерживают множественное наследование. В Java всегда существовало множественное наследование типов в виде наследования классом нескольких интерфейсов. В Java 8 в виде методов с реализацией по умолчанию появилось множественное наследование поведений, но множественное наследование состояния по-прежнему не поддерживается, в отличие от типажей Scala.

Посмотрим, как выглядят типажи в Scala. Опишем типаж `Sized`, включающий одно изменяемое поле `size` и один метод `isEmpty` с реализацией по умолчанию:

```
trait Sized {
  var size : Int = 0
  def isEmpty() = size == 0
}
```

Поле size

Метод isEmpty с реализацией по умолчанию

На основе этого кода можно на этапе объявления создать, например, класс `Empty`, значение поля `size` которого всегда равно `0`:

```
class Empty extends Sized
println(new Empty().isEmpty())
```

Унаследованный от типажа Sized класс

Выводит в консоль true

Что интересно, в отличие от интерфейсов Java композиция типажей возможна во время создания экземпляров (но это все равно остается операцией времени компиляции). Например, можно создать класс `Box` и попытаться сделать так, чтобы один конкретный экземпляр поддерживал описанные в типаже `Sized` операции:

```
class Box
val b1 = new Box() with Sized
println(b1.isEmpty())
val b2 = new Box()
b2.isEmpty()
```

Композиция типажа во время создания объекта

Выводит в консоль true

Ошибка компиляции: объявление класса Box не наследует типаж Sized

А что происходит, если наследуются несколько типажей, в которых объявлены методы с одинаковой сигнатурой или поля с одинаковыми названиями? В Scala на такой случай предусмотрены правила разрешения, аналогичные правилам для методов с реализацией по умолчанию (см. главу 13).

Резюме

- Java и Scala сочетают объектно-ориентированные и функциональные возможности в рамках одного языка программирования; оба работают на JVM и во многом способны взаимодействовать.
- Scala поддерживает абстракции коллекций, аналогичные таковым в Java, — `List`, `Set`, `Map`, `Stream`, `Option`, — а также отсутствующие в Java кортежи.
- Функциональные возможности Scala значительно шире, чем у Java. В их числе функциональные типы, замыкания без ограничений на доступ к локальным переменным, а также встроенные каррированные формы функций.
- Классы в Scala могут включать неявные конструкторы, геттеры и сеттеры.
- Scala поддерживает типажи — интерфейсы, которые могут включать поля и методы с реализацией по умолчанию.

21 Заключение и дальнейшие перспективы Java

В этой главе

- Новые возможности Java 8 и их эволюционное влияние на стиль программирования.
- Новая система модулей Java 9.
- Новый шестимесячный жизненный цикл инкрементных обновлений Java.
- Первый инкрементный выпуск Java 10.
- Реализацию каких возможностей ожидать в следующих версиях Java.

Мы охватили в этой книге очень много материала и надеемся, что вы уже готовы приступить к использованию новых возможностей Java 8 и 9 в своем коде, возможно отталкиваясь от наших примеров и контрольных заданий. В этой главе мы проанализируем пройденный путь изучения Java 8 и нашу легкую пропаганду функционального программирования, а также преимущества новых возможностей модульной организации и другие мелкие усовершенствования, появившиеся в Java 9. Мы также расскажем, какие возможности были включены в Java 10. В дополнение мы обсудим предполагаемые будущие усовершенствования и замечательные новые возможности Java, выходящие за пределы Java 9, 10, 11 и 12.

21.1. Обзор возможностей Java 8

Чтобы было понятнее, насколько язык Java 8 практичный и удобный, мы по очереди напомним вам его возможности. Вместо простого их перечисления мы покажем их во взаимосвязанном виде, чтобы представить их не просто как набор возможностей, а как обзор «с высоты птичьего полета» составных частей логически

последовательной архитектуры языка Java 8. Другая цель данной обзорной главы — подчеркнуть, насколько большинство этих новых возможностей Java 8 облегчает функциональное программирование в Java. Помните, поддержка функционального программирования не прихоть создателей языка, а осознанная проектная стратегия, основанная на двух тенденциях, которые мы рассматривали в модели из главы 1 в качестве изменений климата программирования.

- Рост потребности в полном использовании возможностей многоядерных процессоров, поскольку в настоящий момент исходя из особенностей технологии производства микросхем ежегодный рост числа транзисторов в соответствии с законом Мура более не приводит к повышению тактовой частоты отдельных ядер процессоров. Проще говоря, для повышения быстродействия кода требуется его распараллеливание.
- Растущая тенденция к лаконичным манипуляциям с коллекциями в декларативном стиле, например извлечение из какого-либо источника всех соответствующих заданному критерию данных и применение к результату какой-либо операции (вычисление сводных показателей или создание из результатов коллекции для дальнейшей обработки). Такой стиль программирования связан с использованием неизменяемых объектов и коллекций, в результате обработки которых генерируются последующие неизменяемые значения.

Ни один из этих факторов не учитывается в должной мере в традиционном объектно-ориентированном императивном подходе, в основе которого лежит изменение полей и применение итераторов. Изменение данных в одном ядре процессора и чтение их в другом на удивление дорогостоящая операция, не говоря уже о том, что при этом приходится использовать чреватые ошибками блокировки. Аналогично, если вы привыкли к циклам и изменению существующих объектов, потоковая идиома может казаться чуждой. Но функциональное программирование поддерживает две вышеупомянутые тенденции, именно поэтому в Java 8 центр тяжести несколько сместился с традиционного для Java.

В этой главе мы охватим в единой укрупненной картине изученное в книге и продемонстрируем, как все согласуется с этим новым климатом.

21.1.1. Параметризация поведения (лямбда-выражения и ссылки на методы)

Для написания метода, подходящего для переиспользования (такого как `filter`), необходимо указать в качестве его аргумента описание критерия фильтрации. Хотя опытные разработчики могли сделать это и в предыдущих версиях Java, передав экземпляр класса-адаптера для критерия фильтрации, такое решение было слишком неудобным для написания и сопровождения, а поэтому для всеобщего употребления не подходило.

Как вы обнаружили в главах 2 и 3, в Java 8 появился заимствованный из функционального программирования способ передачи элементов кода в методы. Для этого в Java есть два удобных варианта.

- Передача лямбда-выражения (одноразового фрагмента кода), например, `apple -> apple.getWeight() > 150`.
- Передача в существующий метод *ссылки на метод*, например, `Apple::isHeavy`.

Типы этих значений имеют вид `Function<T, R>`, `Predicate<T>` и `BiFunction<T, U, R>`, получатель может выполнять их с помощью методов `apply`, `test` и т. д. Как вы знаете из главы 3, такие типы называются функциональными интерфейсами и содержат один абстрактный метод. Сами по себе лямбда-выражения — скорее узкоспециализированная концепция, но благодаря широкому применению в новом Stream API они начинают занимать в Java довольно-таки центральное положение.

21.1.2. Потоки данных

Классы коллекций в Java, наряду с итераторами и конструкцией `for-each` верой и правдой служили программистам на протяжении многих лет. Создателям Java 8 было бы проще добавить такие методы, как `filter` и `map`, в коллекции и использовать лямбда-выражения для выражения запросов в стиле баз данных. Но они решили добавить вместо этого новый Stream API, которому посвящены главы 4–7, и имеет смысл обсудить почему.

Какие недостатки коллекций вынудили создателей языка заменить/дополнить их довольно схожей концепцией потоков данных? Попробуем вкратце сформулировать ответ вот так: применение к большой коллекции трех операций (например, отображение объектов из коллекции в сумму двух их полей, фильтрация сумм по некоему критерию и сортировка результата) требует трех отдельных обходов этой коллекции. Stream API позволяет организовать эти операции в конвейер отложенным образом и выполнить все операции за один обход конвейера. Для больших наборов данных эффективность подобного процесса намного выше, и по таким причинам, как наличие кэшей памяти, чем больше набор данных, тем важнее минимизировать число обходов.

Еще одна, не менее важная причина связана с параллельной обработкой элементов, без которой невозможна эффективная эксплуатация многоядерных процессоров. Поток данных можно, в частности, благодаря методу `parallel` пометить как подходящий для параллельной обработки. Как вы помните, параллелизм и изменяемое состояние очень плохо сочетаются, так что для использования потоков данных в параллельном режиме (при операциях `map`, `filter` и тому подобных) оказываются жизненно важны основные концепции функционального программирования (операции без побочных эффектов и методы, параметризуемые лямбда-выражениями и ссылками на методы, что позволяет использовать внутреннюю итерацию вместо внешней, как обсуждалось в главе 4).

В следующем разделе мы проведем прямую аналогию между этими идеями, которые обозначили на языке потоков данных, и архитектурой завершаемых фьючерсов.

21.1.3. Класс CompletableFuture

Интерфейс `Future` появился еще в Java 5. Фьючерсы сильно облегчают эксплуатацию многоядерных процессоров благодаря порождению дополнительной задачи,

выполняемой в другом потоке выполнения или на другом ядре процессора параллельно с выполнением исходной задачи в своем потоке/ядре. Для получения результата исходная задача может воспользоваться методом `get`, который будет ожидать завершения фьючерса (возврата значения).

Глава 16 посвящена реализации интерфейса `Future` в Java 8 в виде класса `CompletableFuture`, опять же использующем лямбда-выражения. Его удобное, хотя и чуть неточное, описание: «Завершаемый фьючерс по отношению к фьючерсу — то же, что и поток данных по отношению к коллекции». Для сравнения:

- потоки данных дают возможность конвейерной организации операций и обеспечивают параметризацию поведения с помощью операций `map`, `filter` и им подобных, позволяя выбросить стереотипный код, необходимый при использовании итераторов;
- класс `CompletableFuture` включает такие операции, как `thenCompose`, `thenCombine` и `allOf`, кодирующих в лаконичном стиле функционального программирования связанные с фьючерсами распространенные паттерны проектирования и опять же позволяющих избежать подобного императивного стереотипного кода.

Подобный стиль операций, хотя и в более простом сценарии, применим также к операциям Java 8 над опционалами, которые мы обсудим в следующем разделе.

21.1.4. Класс Optional

Библиотека Java 8 включает класс `Optional<T>`, который позволяет указывать, идет ли речь об обычном значении типа `T` или возвращенном статическим методом `Optional.empty` отсутствующем значении. Эта возможность делает программу более понятной и упрощает написание документации. Вместо того чтобы, как раньше, использовать чреватый ошибками указатель `null` для указания на отсутствующие значения (при котором программист никогда не может быть уверен, идет ли речь о запланированном отсутствующем значении или случайном `null`, возникшем в результате ошибки в предыдущей операции вычисления), применяется тип данных с явно указываемым отсутствующим значением.

Как обсуждалось в главе 11, при систематическом применении опционалов генерации исключения `NullPointerException` в программе происходить не должно. Опять же вы можете счесть эту ситуацию единичной, не связанной с остальным языком Java 8 и задать вопрос: «Как переход с одной формы отсутствующих значений на другую поможет в написании программ?» Если взглянуть поближе, окажется, что в классе `Optional<T>` есть методы `map`, `filter` и `ifPresent`. Они ведут себя аналогично соответствующим методам потоков данных и могут применяться для конвейерной организации вычислений, опять же в функциональном стиле, причем проверка на отсутствующие значения производится библиотекой, а не пользовательским кодом. Выбор внутренних, а не внешних проверок в классе `Optional<T>` совершенно аналогичен выполнению библиотекой потоков данных внутренней итерации в пользовательском коде вместо внешней. В Java 9 были добавлены различные новые методы в Optional API, включая `stream()`, `or()` и `ifPresentOrElse()`.

21.1.5. Flow API

Язык Java 9 стандартизировал реактивные потоки данных и протокол реактивного противодавления, механизм, предназначенный для предотвращения переполнения «медленного» потребителя событий потока данных одним или несколькими более быстрыми генераторами. Flow API содержит четыре базовых интерфейса, реализация которых библиотеками позволяет добиться большей совместимости: `Publisher`, `Subscriber`, `Subscription` и `Processor`.

Последняя тема, которой мы коснемся в этом разделе, связана не с функциональным программированием, а с поддержкой совместимых снизу вверх расширений библиотек, понадобившихся для разработки ПО.

21.1.6. Методы с реализацией по умолчанию

В Java 8 появились и другие дополнения, не слишком повлиявшие на выразительность каких-либо программ. Но одно дополнение оказалось очень удобным для разработчиков библиотек: возможность добавления в интерфейсы методов с реализацией по умолчанию. До Java 8 в интерфейсах описывались только сигнатуры методов; теперь там можно описывать реализации по умолчанию для методов, которые, по мнению разработчика библиотеки, не все клиенты захотят задавать явным образом.

Это замечательный инструмент для разработчиков библиотек, поскольку позволяет им дополнять интерфейсы новыми операциями, не требуя от клиентов (реализующих этот интерфейс классов) добавления нового кода для описания соответствующих методов. Следовательно, методы с реализацией по умолчанию важны также и для пользователей библиотек, поскольку защищают последних от будущих изменений интерфейсов (см. главу 13).

21.2. Система модулей Java 9

В Java 8 были внесены колоссальные изменения как в смысле новых возможностей (лямбда-выражения и методы с реализацией по умолчанию в интерфейсах, например), так и новых полезных классов в нативном API, например `Stream` и `CompletableFuture`. В Java 9 не появилось никаких новых языковых возможностей, а просто доведено до ума начатое в Java 8, появившиеся там классы дополнены некоторыми полезными методами (например, `takeWhile` и `dropWhile` — класс `Stream` и `completeOnTimeout` — класс `CompletableFuture`). На самом деле главным новшеством Java 9 было появление новой системы модулей. Эта новая система не влияет на язык, за исключением нового файла `module-info.java`, но тем не менее улучшает проектирование и написание приложений с архитектурной точки зрения, более четко разграничивая части системы и определяя их взаимодействие.

Java 9, к сожалению, нанес больший ущерб обратной совместимости Java, чем любой другой выпуск (попробуйте, например, скомпилировать большую базу кода Java 8 с помощью компилятора Java 9). Но преимущества должной модульной организации оправдывают эту цену. Во-первых, она гарантирует лучшую и более строгую инкапсуляцию на уровне пакетов. На самом деле модификаторы видимости Java

предназначены для задания инкапсуляции методов и классов, но на уровне пакетов возможен только один тип видимости: `public`. Этот недостаток затрудняет должную модульную организацию системы, в частности, из-за него сложно указать, какие части модуля предназначены для общего использования, а какие представляют собой нюансы реализации и должны быть скрыты от прочих модулей и приложений.

Во-вторых, из слабости инкапсуляции на уровне пакетов непосредственно следует вывод, что без должной системы модулей неизбежно становятся видимыми элементы функциональности, относящиеся к обеспечению безопасности всего работающего в данной среде кода. Вредоносный код может получить доступ к критически важным частям модуля в обход всех мер безопасности.

Наконец, новая система модулей Java позволяет разбить среду выполнения Java на более мелкие части, так что можно использовать только те части, которые необходимы для приложения. Например, было бы удивительно, если бы вашему новому проекту Java понадобилась CORBA, но вполне вероятно, что она будет включена во все ваши приложения. Хотя для обычных вычислительных устройств это вряд ли имеет значение, но существенно для встраиваемых устройств и для все чаще встречающейся ситуации, когда Java-приложения работают в контейнеризованной среде. Другими словами, система модулей Java — механизм, благодаря которому среда выполнения Java может использоваться в приложениях Интернета вещей (IoT) и в облачных приложениях.

Как обсуждалось в главе 14, система модулей Java решает эти задачи на уровне языка посредством механизма модульной организации больших систем и самой среды выполнения Java. В числе преимуществ системы модулей Java следующие.

- *Надежность конфигурации* — явное объявление зависимостей модулей позволяет обнаруживать ошибки во время сборки, а не выполнения в случае отсутствующих, конфликтующих или циклических зависимостей.
- *Сильная инкапсуляция* — система модулей Java позволяет модулям экспортировать только отдельные пакеты и отделять общедоступные границы модулей от внутренней реализации.
- *Повышенная безопасность* — запрет на вызов пользователем определенных частей модуля значительно затрудняет для злоумышленников обход встроенных в модуль мер безопасности.
- *Повышение производительности* — эффективность многих методик оптимизации возрастает, когда у класса есть возможность обращаться только к нескольким компонентам, а не ко всем остальным классам, загруженным средой выполнения.
- *Масштабируемость* — система модулей Java позволяет разбить платформу Java SE на меньшие части, содержащие только необходимые для текущего приложения возможности.

Вообще говоря, модульная организация — непростая тема и она вряд ли будет играть основную роль во внедрении Java 9, подобно той роли, которую лямбда-выражения играли для Java 8. Впрочем, мы полагаем, что в долгосрочной перспективе усилия, затраченные на модуляризацию приложения, всегда окупаются в виде упрощения сопровождения.

Пока что мы подвели итоги охваченных в данной книге концепций Java 8 и 9. В следующем разделе мы переключимся на изучение более щекотливого вопроса будущих усовершенствований и замечательных возможностей в грядущих версиях Java.

21.3. Вывод типов локальных переменных в Java 10

Первоначально в Java при создании переменной или метода одновременно указывался и их тип. Пример:

```
double convertUSDToGBP(double money) { ExchangeRate e = ...; }
```

содержит три типа, соответствующих типу возвращаемого методом результата `convertUSDToGBP`, типу его аргумента `money` и типу его локальной переменной `e`. Со временем это требование было смягчено в двух направлениях. Во-первых, можно не указывать параметры обобщенных типов в выражениях, когда они понятны из контекста. Этот пример:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

можно сократить до следующего, начиная с Java 7:

```
Map<String, List<String>> myMap = new HashMap<>();
```

Во-вторых, благодаря той же идее транслирования определяемого контекстом типа в выражение можно сократить лямбда-выражение:

```
Function<Integer, Boolean> p = (Integer x) -> booleanExpression;
```

до вида:

```
Function<Integer, Boolean> p = x -> booleanExpression;
```

опустив типы. В обоих случаях компилятор выводит пропущенные типы.

Преимуществ у вывода типов не очень много, если тип состоит из одного идентификатора; основное из них состоит в сокращении объема редактирования при замене одного типа другим. Но по мере роста размеров описаний типов и параметризации обобщенных типов другими обобщенными типами вывод типов может существенно повысить удобочитаемость[1]. Языки Scala и C# допускают замену типа в объявлении инициализируемой локальной переменной на ключевое слово (с ограниченной областью видимости) `var`; компилятор берет соответствующий тип с правой стороны объявления. Показанное ранее объявление `myMap` на Java можно преобразовать в следующее:

```
var myMap = new HashMap<String, List<String>>();
```

[1] Важно подходить к выводу типов обдуманно. Лучше всего вывод типов работает, когда существует только один способ восстановления пропущенного пользователем типа. Проблемы начинаются, когда система выводит не тот тип данных, что задумывался пользователем. Так что при продуманном выводе типов должна возвращаться ошибка, если может быть выведено два несовместимых типа; эвристический алгоритм может создавать видимость выбора случайным образом, не того, который нужен.

Эта идея носит название *вывода типов локальных переменных* (local variable type inference) и была включена в Java 10.

Впрочем, здесь есть небольшой повод для беспокойства. Какой тип в следующем объявлении:

```
var x = new Car();
```

неявно объявляется для переменной `x` — `Car` или `Vehicle` (или даже `Object`) в случае, если класс `Car` наследует тип `Vehicle`? В данном случае прекрасно работает простое объяснение, что опущенный тип должен соответствовать типу инициализатора (здесь это тип `Car`). Java 10 формализует этот вывод, при этом четко заявляя, что в отсутствие инициализатора `var` использовать нельзя.

21.4. Что ждет Java в будущем?

Некоторые из рассматриваемых в этом разделе вопросов подробнее обсуждаются на веб-сайте предложений по расширению JDK (http://openjdk.java.net/jeps/0). Но здесь мы объясним, почему кажущиеся разумными идеи вызывают неочевидные проблемы или так взаимодействуют с уже существующими возможностями, что прямое внедрение их в Java невозможно.

21.4.1. Вариантность по месту объявления

Язык Java поддерживает джокерные символы — гибкий механизм, с помощью которого становится возможным расширение обобщенных типов (обычно называемое *вариантностью по месту использования* (use-site variance)). Благодаря этой поддержке следующее присвоение вполне допустимо:

```
List<? extends Number> numbers = new ArrayList<Integer>();
```

А вот такое присвоение, без `"? extends"`, приводит к ошибке во время компиляции:

```
List<Number> numbers = new ArrayList<Integer>();   ◄——— Несовместимые типы
```

Многие языки программирования, такие как C# и Scala, поддерживают другой механизм вариантности: *вариантность по месту объявления* (declaration-site variance). В этих языках программисты могут указывать вариантность при описании обобщенного класса. Подобная возможность полезна для классов, которые по своей природе являются вариантными. Интерфейс `Iterator`, например, является по своей природе ковариантным, а `Comparator` — контрвариантным, так что не следует думать о них на языке `? extends` и `? super`. Добавление вариантности по месту объявления в Java могло бы оказаться полезным, позволив перенести подобные характеристики в объявления классов. В результате несколько снизилась бы когнитивная нагрузка на программистов. Отметим, что на момент написания данной книги (2018 год) включение вариантности по месту объявления в будущие версии Java входит в число предложений по расширению JDK (http://openjdk.java.net/jeps/300).

21.4.2. Сопоставление с шаблоном

Как уже обсуждалось в главе 19, в функциональных языках программирования обычно существует какая-либо из форм сопоставления с шаблоном — расширенной формы `switch`, — при которой можно спросить: «Относится ли это значение к заданному классу?» — и (не обязательно) рекурсивно запросить информацию о конкретных значениях его полей. Простой модульный тест в Java выглядит примерно так:

```
if (op instanceof BinOp){
    Expr e = ((BinOp) op).getLeft();
}
```

Заметьте, что приходится повторять `BinOp` внутри выражения приведения типа, хотя очевидно, что `op` ссылается на объект этого типа.

Конечно, обрабатываемая иерархия выражений может быть достаточно запутанной и связывание цепочкой нескольких условий `if` только приведет к «раздуванию» кода. Не лишним будет напомнить, что обычная объектно-ориентированная архитектура не одобряет использования `switch` и поощряет использование таких паттернов, как посетитель, в которых контроль потока команд (на основе типов данных) производится путем назначения методов вместо `switch`. На другом конце спектра языков программирования, в функциональном программировании, самым удобным способом проектирования программы обычно оказывается сопоставление значений типов данных с шаблоном.

Добавление полноценного сопоставления с шаблоном в стиле Scala в Java выглядит непростой задачей, но вслед за недавним обобщением оператора `switch`, в котором теперь допустимы строковые значения, можно легко представить более умеренное расширение синтаксиса: возможность работы `switch` с объектами посредством синтаксиса `instanceof`. На самом деле существует предложение по расширению JDK, в котором сопоставление с шаблоном рассматривается как кандидат на включение в Java (http://openjdk.java.net/jeps/305). В следующем примере мы вернемся к примеру из главы 19, предположив, что у класса `Expr` есть подклассы `BinOp` и `Number`:

```
switch (someExpr) {
      case (op instanceof BinOp):
         doSomething(op.getOpName(), op.getLeft(), op.getRight());
      case (n instanceof Number):
         dealWithLeafNode(n.getValue());
      default:
         defaultAction(someExpr);
}
```

Отметим пару нюансов. Во-первых, этот код заимствует у сопоставления с шаблоном идею, что `op` в условии `case (op instanceof BinOp):` — новая локальная переменная (типа `BinOp`), которая привязывается к значению `someExpr`. Аналогично в `case` для `Number n` становится переменной типа `Number`. А в `case` по умолчанию привязываемая переменная отсутствует. Это предложение позволяет значительно уменьшить объем стереотипного кода, по сравнению с использованием цепочек `if-then-else` и приведением к подтипу. Создатель программ в традиционном

объектно-ориентированном стиле, вероятно, возразит, что выражать подобный код для назначения типов данных лучше с помощью переопределяемых в подтипах методов-посетителей, но с точки зрения функционального программирования подобное решение лишь приводит к разбрасыванию связанного кода по описаниям нескольких классов. Подобное классическое противопоставление архитектур называется в литературе проблемой выражения[1].

21.4.3. Более полнофункциональные формы обобщенных типов

В этом разделе мы обсудим два ограничения обобщенных типов Java и рассмотрим возможные пути их смягчения.

Материализованные обобщенные типы

Когда обобщенные типы только появились в Java 5, от них требовалась обратная совместимость с существующей JVM. Поэтому представления времени выполнения у `ArrayList<String>` и `ArrayList<Integer>` идентичны. Эта модель носит название *«модель стирания полиморфизма обобщенных типов»* (erasure model of generic polymorphism). Подобный выбор требует определенных небольших затрат ресурсов во время выполнения, но главное для программистов — то, что параметрами обобщенных типов могут быть только объекты, а не простые типы данных. Предположим на минуту, что в Java разрешили использовать, допустим, `ArrayList<int>`. В результате этого можно выделить место в куче для объекта `ArrayList`, содержащего значение простого типа, например, `int 42`, но в контейнере `ArrayList` при этом не будет никакого индикатора того, содержит ли он значение-объект, например, типа `String` или значение простого типа `int`, например, `42`.

До некоторой степени эта ситуации выглядит вполне безобидной. Какая разница, при получении значения простого типа вроде `42` и объекта `String` вроде `"abc"`, что контейнеры `ArrayList` неразличимы? К сожалению, дело в сборке мусора, поскольку из-за отсутствия информации о типе во время выполнения относительно содержимого `ArrayList` JVM не сможет определить, был ли элемент 13 из `ArrayList` ссылкой на строковое значение (которое нужно отслеживать и пометить для сборки мусора) или простым значением типа `int` (отслеживать которое, безусловно, не нужно).

В языке C# представления времени выполнения `ArrayList<String>`, `ArrayList<Integer>` и `ArrayList<int>` различаются коренным образом. Но даже если бы они были одинаковы, во время выполнения хранится информация о типах, достаточная для того, чтобы механизм сборки мусора мог определить, является ли поле ссылкой или относится к простому типу данных. Эта модель называется *материализованной моделью полиморфизма обобщенных типов* (reified model of generic polymorphism) или попросту *материализованными обобщенными типами* (reified generics). Слово «материализация» означает «делать нечто неявное явным».

Материализованные обобщенные типы явно предпочтительнее, поскольку позволяют добиться большего единообразия простых типов данных и соответствующих объектных типов — довольно проблемный вопрос, как мы увидим в следующих раз-

[1] Более подробные пояснения можно найти в статье http://en.wikipedia.org/wiki/Expression_problem.

делах. Основная проблема для Java — обратная совместимость на уровне как JVM, так и уже существующих программ, которые используют рефлексию и ожидают стирания обобщенных типов.

Увеличение синтаксической гибкости обобщенных типов для функциональных типов

После добавления в Java 5 обобщенные типы проявили себя как замечательная возможность. Они также прекрасно подходят для выражения типов множества лямбда-выражений и ссылок на методы Java 8. Функцию с одним аргументом можно описать, например, следующим образом:

```
Function<Integer, Integer> square = x -> x * x;
```

В случае функции с двумя аргументами можно воспользоваться типом `BiFunction<T, U, R>`, где `T` — первый параметр, `U` — второй, а `R` — результат. Но типа `TriFunction` не существует, его можно разве что объявить самостоятельно.

Аналогично нельзя использовать `Function<T, R>` для методов, принимающих ноль аргументов и возвращающих результат типа `R`; вместо этого следует использовать `Supplier<R>`.

По существу, лямбда-выражения Java 8 обогатили возможности написания кода, но система типов Java за гибкостью кода не поспевает. В многих функциональных языках можно, например, написать тип `(Integer, Double) => String`, соответствующий `BiFunction<Integer, Double, String>` в Java 8, а также `Integer => String`, соответствующий `Function<Integer, String>`, и даже `() => String` для `Supplier<String>`. Оператор `=>` можно считать инфиксной версией `Function`, `BiFunction`, `Supplier` и им подобных. Простое расширение синтаксиса Java для инфиксного оператора `=>` сделало бы типы более удобочитаемыми, аналогично типам Scala, как обсуждалось в главе 20.

Версии для простых типов данных и обобщенные типы

В Java у всех простых типов данных (например, `int`) есть соответствующие объектные типы (в данном случае `java.lang.Integer`). Зачастую программисты называют эти типы неупакованными и упакованными соответственно. Хотя это различие проводится с похвальной целью максимизации производительности во время выполнения, существует опасность путаницы. Почему, например, нужно писать в Java 8 `Predicate<Apple>` вместо `Function<Apple, Boolean>`? Дело в том, что при вызове методом `test` объект типа `Predicate<Apple>` возвращает данные простого типа `boolean`.

И напротив, как и все обобщенные типы, `Function` может параметризоваться только объектными типами. В случае `Function<Apple, Boolean>` эту роль играет объектный тип `Boolean`, а не простой тип данных `boolean`. Быстродействие `Predicate<Apple>` выше, поскольку он не требует упаковки `boolean` для получения `Boolean`. Эта проблема привела к созданию множества аналогичных интерфейсов, например `LongToIntFunction` и `BooleanSupplier`, еще больше повышающих понятийную нагрузку на программиста.

Еще один пример связан с различиями между ключевым словом `void` (у которого отсутствуют значения и которое годится только для задания возвращаемого типа метода) и объектным типом `Void`, содержащим одно значение `null` (вопрос, регулярно всплывающий на форумах). Частные случаи `Function`, такие как `Supplier<T>`, который можно записать в виде `() => T` в предложенной в предыдущем разделе новой нотации, еще больше подтверждают последствия различий между простыми и объектными типами данных. Мы уже обсуждали ранее, как материализованные обобщенные типы могут помочь в решении многих из этих проблем.

21.4.4. Расширение поддержки неизменяемости

Самые опытные из читателей этой книги могли несколько расстроиться, когда мы сказали, что в Java 8 есть три вида значений:

- значения простых типов данных;
- (ссылки на) объекты;
- (ссылки на) функции.

С одной стороны, мы будем держаться своих слов и скажем: «Но именно такие значения метод может сейчас принимать в качестве аргументов и возвращать в качестве результатов». Но нужно признать, что это объяснение немного противоречиво. До какой степени можно считать, что возвращается (математическое) значение в случае возврата ссылки на изменяемый массив? Объект `String` или неизменяемый массив явно значение, но ситуация не столь ясна в случае изменяемого объекта или массива. Метод может вернуть массив, элементы которого упорядочены в порядке возрастания, а какой-то другой код — позднее поменять один из его элементов.

При желании программировать в функциональном стиле на Java необходима языковая поддержка понятия «неизменяемое значение». Как обсуждалось в главе 18, ключевого слова `final` для этого недостаточно; оно только предотвращает обновление квалифицируемого поля. Рассмотрим пример:

```
final int[] arr = {1, 2, 3};
final List<T> list = new ArrayList<>();
```

Первая строка запрещает дальнейшие присваивания вида `arr = ...`, но не запрещает присваивания `arr[1] = 2`; вторая строка запрещает присваивания `list`, но не запрещает изменения другими методами числа элементов в `list`. Ключевое слово `final` отлично работает для значений простых типов данных, но для ссылок на объекты зачастую лишь дает ложное ощущение безопасности.

Вот к чему мы ведем: поскольку функциональное программирование подчеркивает важность неизменяемости существующих структур, вполне разумно будет ввести в язык что-то вроде ключевого слова `transitively_final` для квалификации полей ссылочного типа с гарантией невозможности модификации поля или какого-либо прямо или косвенно доступного через это поле объекта.

Подобные типы иллюстрируют факт, который подсказывает нам интуиция: значения неизменяемы и только переменную, содержащую значение, можно изменить так, чтобы она содержала другое неизменяемое значение. Как мы отмечали в начале

данного раздела, разработчики Java (включая нас самих) иногда непоследовательно говорят насчет Java-значения, которое является изменяемым массивом. В следующем разделе мы вернемся к правильному интуитивно понятному представлению и обсудим идею типов-значений, которые могут содержать только неизменяемые значения, хотя переменные типов-значений по-прежнему можно обновлять, разве что они объявлены с модификатором `final`.

21.4.5. Типы-значения

В этом разделе мы обсудим различия между простыми и объектными типами данных, а далее поговорим о необходимости типов-значений, упрощающих написание программ в функциональном стиле, подобно тому как объектные типы необходимы для объектно-ориентированного программирования. Многие из обсуждаемых здесь вопросов взаимосвязаны, так что рассказывать о них поодиночке не получится. Вместо этого мы раскроем общую проблему через ее частные аспекты.

Разве компилятор не может обрабатывать Integer и int одинаково?

С учетом всей неявной распаковки и упаковки, понемногу накапливавшейся в Java, начиная с версии 1.1, возникает закономерный вопрос: не пора ли Java начать обрабатывать `Integer` и `int` одинаково в надежде, что компилятор Java сам выберет оптимальную для JVM форму.

В принципе, это замечательная идея, но давайте взглянем на проблемы, возникающие при добавлении в Java типа для комплексных чисел, чтобы понять, в чем трудности с упаковкой. Тип `Complex`, моделирующий так называемые комплексные числа, включающие вещественную и мнимую части, можно описать следующим образом:

```
class Complex {
    public final double re;
    public final double im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    public static Complex add(Complex a, Complex b) {
        return new Complex(a.re+b.re, a.im+b.im);
    }
}
```

Но значения типа `Complex` относятся к ссылочным типам, так что любая операция над объектом `Complex` требует выделения памяти под объект, затмевая, например, затраты на сложение двух чисел в операции `add`. Программистам нужен аналог `Complex` в виде простого типа данных, например, с названием `complex`.

Проблема в том, что программистам требуются неупакованные объекты, которые фактически не поддерживают ни Java, ни JVM. Можно, конечно, снова начать охать: «Наверняка же компилятор может это как-то оптимизировать». К сожалению, это намного более сложная задача, чем кажется; хотя компилятор иногда может на основе так называемого escape-анализа определить, что распаковка допустима, но его применимость ограничивается допущениями Java относительно

объектов, существующими еще с версии Java 1.1. Рассмотрим следующую задачу с подвохом:

```
double d1 = 3.14;
double d2 = d1;
Double o1 = d1;
Double o2 = d2;
Double ox = o1;
System.out.println(d1 == d2 ? "yes" : "no");
System.out.println(o1 == o2 ? "yes" : "no");
System.out.println(o1 == ox ? "yes" : "no");
```

Результат выполнения: `"yes"`, `"no"`, `"yes"`. Опытный Java-программист, вероятно, скажет: «Какой глупый код. Все знают, что в последних двух строках следует использовать `equals` вместо `==`». И хотя все эти простые типы и объекты содержат неизменяемое значение `3.14` и должны быть неразличимы, определения `o1` и `o2` создают новые объекты и для оператора `==` (сравнения идентичности) они различаются. Обратите внимание, что для простых типов данных сравнение идентичности производится путем побитового сравнения, а для объектов выполняется сравнение ссылок. Разработчики зачастую случайно создают новый отдельный объект `Double`, что компилятору приходится учитывать, поскольку этого требует семантика класса `Object` — одного из родительских для `Double`. Мы уже обсуждали это, как при разговоре о типах-значениях в этой главе, так и в главе 19, где мы обсуждали функциональную прозрачность методов, функционально обновляющих существующие структуры данных.

Типы-значения: не все в Java является простым типом данных или объектом

Мы считаем, что решение этой проблемы лежит в плоскости изменения предположений Java, гласящих, что: 1) все, что не относится к простым типам данных, — объект, а значит, наследует класс `Object` и 2) все ссылки являются ссылками на объекты.

В этом направлении уже ведутся разработки. Значения принимают две формы:

- объектные типы, поля которых являются изменяемыми, если иное не указано явным образом с помощью модификатора `final`, и обладают идентичностью, которую можно проверить с помощью оператора `==`;
- типы-значения, неизменяемые и не обладающие ссылочной идентичностью. Простые типы данных также относятся к этой категории.

Можно разрешить пользовательские типы-значения (например, начинающиеся с буквы в нижнем регистре, чтобы подчеркнуть их сходство с простыми типами данных вроде `int` и `boolean`). Типы-значения оператор `==` по умолчанию будет сравнивать поэлементно, аналогично аппаратному побитовому сравнению для `int`. Особое внимание следует уделить членам классов с плавающей точкой, поскольку их сравнение — операция несколько более сложная. Тип `Complex` — прекрасный пример не простого типа-значения, подобные типы напоминают тип `struct` языка C#.

Помимо этого, типы-значения менее требовательны к памяти, поскольку у них нет ссылочной идентичности. Рисунок 21.1 иллюстрирует массив размером 3, элементы которого — 0, 1 и 2 — выделены светло-серым, серым и темно-серым цветом соответственно. Схема слева демонстрирует типичные потребности в памяти в случае, когда `Pair` и `Complex` являются объектами, а справа — более экономный

вариант, когда `Pair` и `Complex` — типы-значения. Обратите внимание, что на схеме они называются `pair` и `complex` (в нижнем регистре), чтобы подчеркнуть их сходство с простыми типами данных. Отметим также, что быстродействие типов-значений, вероятно, тоже будет выше не только в смысле доступа к данным (вместо нескольких уровней косвенной адресации указателей — одна инструкция адресации по индексу), но и использования аппаратного кэша (вследствие связности данных).

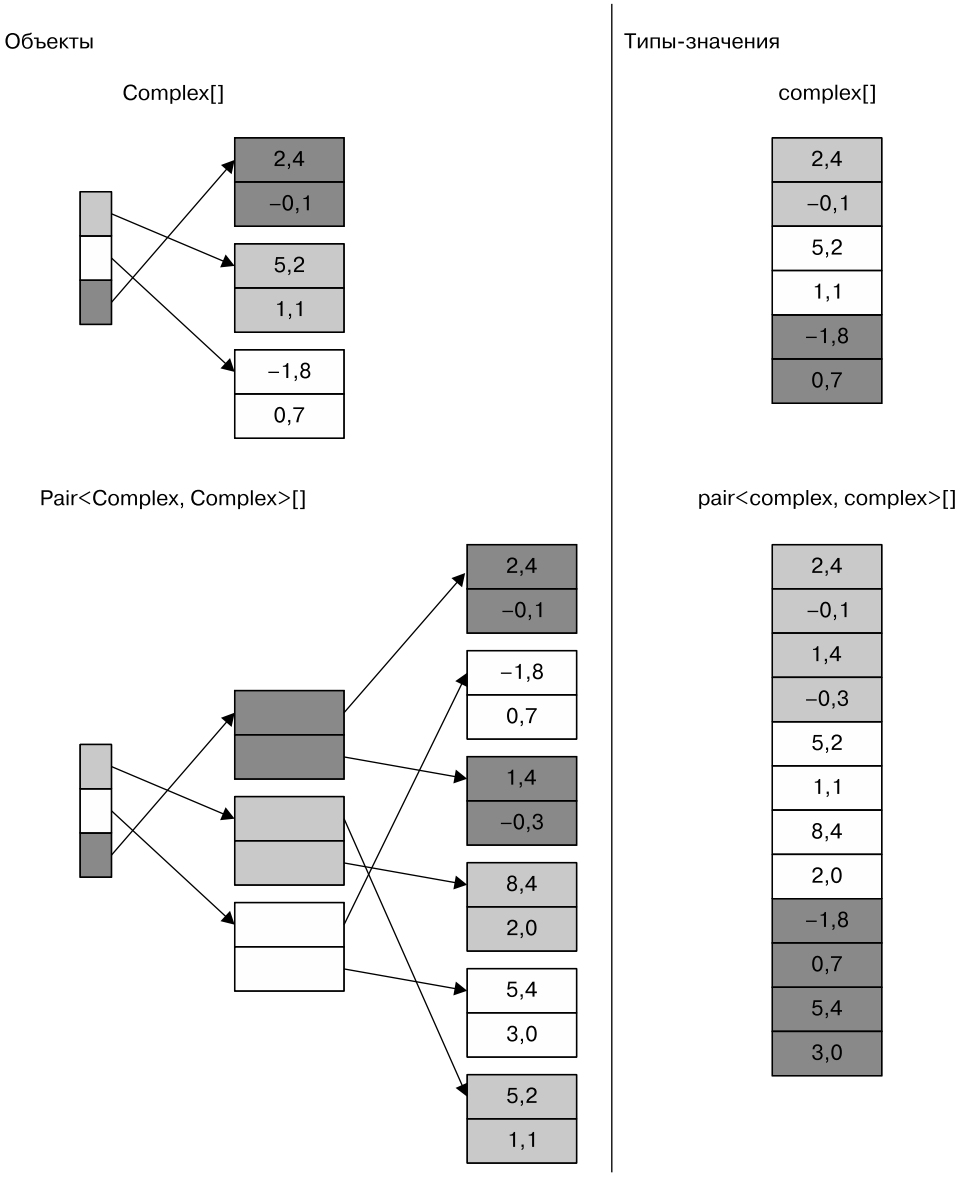


Рис. 21.1. Объекты и типы-значения

Обратите внимание, что в силу отсутствия у типов-значений ссылочной идентичности компилятор может распаковывать/упаковывать их по своему выбору. При передаче `complex` в качестве аргумента из одной функции в другую компилятор может вполне логично передать их в виде двух отдельных `double` (возвращать их без упаковки в JVM — более сложная задача, конечно, поскольку в JVM существуют только инструкции возврата из методов для передачи представимых в 64-битных аппаратных регистрах значений). Но если передать в качестве аргумента тип-значение большего размера (например, большой неизменяемый массив), то компилятор после упаковки может передать его незаметно для пользователя в виде ссылки. Аналогичная технология уже существует в C#. В документации по C# написано (https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types):

> «Переменная типа-значения фактически содержит значение соответствующего типа. Например, переменная типа int может содержать значение 42. Это отличается от переменных ссылочных типов, содержащих ссылку на экземпляр соответствующего типа, называемый также объектом. При присвоении нового значения переменной типа-значения, оно копируется. При присвоении нового значения переменной ссылочного типа, копируется только ссылку на объект, а не сам объект».

На момент написания данной книги (2018 год) ожидает решения предложение по расширению JDK относительно включения в Java типов-значений (http://openjdk.java.net/jeps/169).

Упаковка, обобщенные типы и типы-значения: проблема взаимозависимости

Типы-значения желательны в Java, поскольку программы в функциональном стиле работают с неизменяемыми значениями, у которых нет идентичности. Было бы удобно считать простые типы данных частным случаем типов-значений, но принятая в настоящий момент в Java модель стирания полиморфизма обобщенных типов означает, что типы-значения нельзя использовать с обобщенными типами без упаковки. Объектные (упакованные) версии (например, `Integer`) простых типов данных (таких как `int`) продолжают играть важнейшую роль для коллекций и обобщенных типов Java вследствие их модели стирания, но в настоящее время то, что они наследуют `Object` (а значит, сравнение ссылок), рассматривается как недостаток. Решение любой из этих проблем требует решения их всех.

21.5. Ускорение развития Java

За 22 года вышло десять старших версий Java — в среднем между выпусками проходило более двух лет. В некоторых случаях ждать новой версии приходилось пять лет. Архитекторы Java осознали, что такая ситуация неприемлема, поскольку язык при этом развивается недостаточно быстро и новые языки на JVM (например, Scala и Kotlin) сильно отрываются от Java в смысле возможностей. Столь длительный цикл выпуска версий в какой-то мере имел смысл для колоссальных революционных изменений вроде лямбда-выражений и системы модулей Java, но

означал также, что для внедрения мелких усовершенствований в язык придется ждать без какой-либо веской причины завершения реализации одного из этих больших изменений. Фабричные методы коллекций, обсуждавшиеся в главе 8, например, были готовы задолго до завершения разработки системы модулей Java. Поэтому было решено, что теперь у Java будет шестимесячный цикл разработки. Другими словами, новая старшая версия Java и JVM будет выходить каждые шесть месяцев, начиная с вышедшего в марте 2018 выпуска Java 10 и вышедшего в сентябре 2018-го выпуска[1] Java 11. Архитекторы Java также поняли, что, хотя такой ускоренный цикл разработки полезен для самого языка, а также быстро адаптирующихся компаний и разработчиков, любящих экспериментировать с новыми технологиями, он может создать проблемы для более консервативных организаций, которые привыкли обновлять ПО реже. Поэтому архитекторы Java также приняли решение о том, что каждые три года будет выпускаться версия с долгосрочной (на протяжении следующих трех лет) поддержкой (long-term support, LTS). Java 9 не LTS-выпуск, так что теперь, когда вышел Java 10, его жизненный цикл считается завершенным. То же самое ожидает Java 10. Вышедший же в сентябре 2018 года Java 11, напротив, LTS-версия, поддержка которой будет продолжаться до сентября 2021 года. На рис. 21.2 приведен жизненный цикл версий Java, планируемых к выпуску в ближайшие несколько лет.

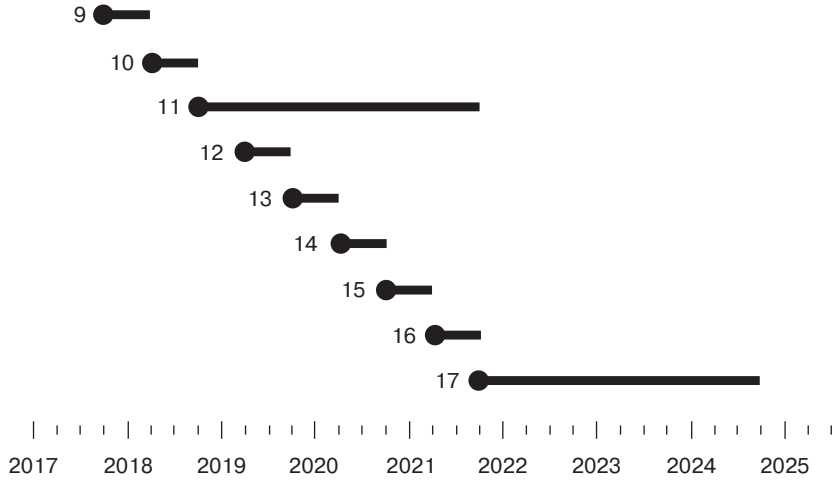


Рис. 21.2. Жизненный цикл будущих выпусков Java

Мы горячо поддерживаем решение сократить цикл разработки Java, особенно в нынешней обстановке, когда всем программным системам и языкам программирования приходится развиваться как можно быстрее. Более короткий цикл разработки позволит Java эволюционировать с нужной скоростью и сохранить актуальность в будущем.

[1] 19 марта 2019 года был опубликован выпуск Java 12. — *Примеч. пер.*

21.6. Заключительное слово

В этой книге мы изучили основные новые возможности, появившиеся в Java 8 и 9. Java 8 представляет собой, вероятно, крупнейший эволюционный шаг в истории Java. Единственным сравнимым по масштабу шагом, вероятно, было появление десятью годами ранее (в 2004 году) обобщенных типов в Java 5. Важнейшая отличительная черта Java 9 — появление долгожданной системы модулей, более интересной скорее для архитекторов ПО, чем для разработчиков. Java 9 также стандартизировал протокол для использования реактивных потоков данных в Flow API. В Java 10 появился вывод типов локальных переменных — популярная возможность в других языках программирования, помогающая повысить эффективность работы программистов. Java 11 позволяет использовать синтаксис `var` вывода типов локальных переменных в списках параметров неявно типизируемых лямбда-выражений. Но что еще более важно, Java 11 взял на вооружение идеи конкурентности и реактивного программирования, обсуждавшиеся в данной книге, и включает новую асинхронную клиентскую HTTP-библиотеку, в полной мере использующую завершаемые фьючерсы. Наконец, на момент написания данной книги было объявлено, что Java 12 будет поддерживать расширенную конструкцию `switch`, допускающую использование в качестве выражения, а не просто оператора, — ключевая возможность языков функционального программирования. Фактически выражения `switch` прокладывают дорогу для сопоставления с шаблоном в Java, которое мы обсуждали в подразделе 21.4.2. Все эти новшества в языке демонстрируют, что идеи и влияние функционального программирования продолжат победоносное шествие по Java в будущем!

В этой главе мы рассмотрели основные побудительные причины эволюции Java. В заключение хотели бы сформулировать следующее утверждение: *Java 8, 9, 10 и 11 — замечательные площадки, чтобы задержаться, но отнюдь не остановиться!*

Мы надеемся, что вам понравилось познавательное путешествие с нами по миру языка Java и нам удалось зажечь в вас искру интереса к дальнейшей его эволюции.

Приложения

А Прочие изменения языка

В этом приложении мы обсудим еще три усовершенствования языка в Java 8: повторяющиеся аннотации (repeating annotations), аннотации типов (type annotations) и вывод обобщенных целевых типов (generalized target-type inference). В приложении Б обсуждаются обновления библиотек в Java 8. Мы не будем обсуждать обновления JDK 8, например движок Nashorn и сжатые профили (compact profiles), поскольку они представляют собой новые возможности JVM, а эта книга посвящена обновлениям *библиотек* и *языка* Java. Если вас интересуют Nashorn и сжатые профили, рекомендуем прочитать информацию, расположенную по следующим ссылкам: http://openjdk.java.net/projects/nashorn/ и http://openjdk.java.net/jeps/161.

А.1. Аннотации

Механизм аннотаций в Java 8 подвергся усовершенствованиям в двух направлениях:

- можно повторять аннотации;
- можно аннотировать любые применения типов.

Прежде чем мы расскажем про эти обновления, стоит вкратце напомнить, что можно было делать с помощью аннотаций до Java 8.

Аннотации (annotations) в Java представляют собой механизм, с помощью которого можно снабжать элементы программы дополнительной информацией (отметим, что до Java 8 аннотировать можно было только объявления). Другими словами, это разновидность *синтаксических метаданных*. Например, аннотации часто используют с фреймворком JUnit. В следующем фрагменте кода метод `setUp` снабжен аннотацией `@Before`, а метод `testAlgorithm` — аннотацией `@Test`:

```
@Before
public void setUp(){
    this.list = new ArrayList<>();
}
```

```
@Test
public void testAlgorithm(){
     ...
    assertEquals(5, list.size());
}
```

Аннотации подходят для нескольких сценариев использования.

- В контексте JUnit с помощью аннотаций можно отделять методы, выполняемые в качестве модульных тестов, от вспомогательных методов, предназначенных для настройки тестов.
- Аннотации можно использовать для документирования. Например, аннотация `@Deprecated` применяется для указания, что метод более не следует использовать.
- Компилятор Java может также обрабатывать аннотации для обнаружения ошибок, подавления предупреждений и генерации кода.
- Аннотации часто используются на платформе Java EE, для задания настроек корпоративных приложений.

A.1.1. Повторяющиеся аннотации

В предыдущих версиях Java не разрешалось снабжать объявление более чем одной аннотацией одинаковой разновидности. Поэтому следующий код был некорректен:

```
@interface Author { String name(); }
@Author(name="Raoul") @Author(name="Mario")  @Author(name="Alan")
class Book{ }
```

Ошибка: дублирование аннотации

Программисты Java EE часто использовали такой способ, чтобы обойти это ограничение: объявлялась новая аннотация, содержащая массив экземпляров аннотации, которую нужно было повторить. Выглядело это примерно следующим образом:

```
@interface Author { String name(); }
@interface Authors {
    Author[] value();
}

@Authors(
  { @Author(name="Raoul"), @Author(name="Mario") , @Author(name="Alan")}
)
class Book{}
```

Вложенная аннотация для класса `Book` выглядит довольно уродливо. Поэтому-то в Java 8 это ограничение было убрано, что несколько исправило положение дел. Теперь можно снабжать одно объявление несколькими аннотациями одной разновидности при условии, что аннотация явным образом помечена как повторяемая. Это не считается вариантом поведения по умолчанию, необходимо явным образом указать, что аннотация повторяемая.

Делаем аннотацию повторяемой

Если аннотация изначально создавалась как повторяемая, то никаких дополнительных действий не требуется. Если же речь идет об аннотации, созданной вами для

ваших пользователей, то необходимо указать, что она может повторяться. Для этого требуется два шага.

1. Отметить аннотацию как `@Repeatable`.
2. Указать аннотацию-контейнер.

Сделать, например, аннотацию `@Author` повторяемой можно следующим образом:

```
@Repeatable(Authors.class)
@interface Author { String name(); }
@interface Authors {
    Author[] value();
}
```

В результате для класса `Book` можно указать несколько аннотаций `@Author`:

```
@Author(name="Raoul") @Author(name="Mario")  @Author(name="Alan")
class Book{ }
```

Во время компиляции считается, что класс `Book` аннотирован `@Authors({@Author(name="Raoul"), @Author(name="Mario"), @Author(name="Alan")})`, так что этот новый механизм можно рассматривать как «синтаксический сахар» — обертку для вышеприведенной идиомы, использовавшейся Java-программистами ранее. Аннотации по-прежнему обертываются в контейнер для обеспечения функциональной совместимости с методами рефлексии предыдущих версий. Метод `getAnnotation(Class<T> annotationClass)` из API Java возвращает для аннотированного элемента аннотацию типа `T`. Но какую аннотацию должен вернуть этот метод, если элемент снабжен несколькими аннотациями типа `T`?

Если не углубляться в подробности, то класс `Class` поддерживает новый метод `getAnnotationsByType`, упрощающий работу с повторяющимися аннотациями. Например, для вывода всех аннотаций `Author` класса `Book` можно воспользоваться им следующим образом:

```
public static void main(String[] args) {
    Author[] authors = Book.class.getAnnotationsByType(Author.class);
    Arrays.asList(authors).forEach(a -> { System.out.println(a.name()); });
}
```

Извлекает массив, состоящий из повторяющихся аннотаций Author

Чтобы этот код работал, как у повторяющейся аннотации, так и у ее контейнера должна быть стратегия сохранения RUNTIME. Дополнительную информацию о совместимости с методами рефлексии предыдущих версий можно найти здесь: http://cr.openjdk.java.net/~abuckley/8misc.pdf.

A.1.2. Аннотации типов

В Java 8 аннотации могут указываться для любых применений типов, включая оператор `new`, приведение типов, проверку `instanceof`, аргументы обобщенных типов, а также выражения `implements` и `throws`. В следующем коде мы с помощью аннотации `@NonNull` указываем, что имя переменной типа `String` не может быть `null`:

```
@NonNull String name = person.getName();
```

Аналогично можно аннотировать тип элементов списка:

```
List<@NonNull Car> cars = new ArrayList<>();
```

Значение этой возможности заключается в потенциальной пользе аннотаций типов для анализа программ. В приведенных двух примерах гарантируется, что `getName` не возвращает `null` и что элементы списка машин не пусты. Это помогает снизить число неожиданных ошибок в коде.

Java 8 не предоставляет готовых аннотаций или инструмента для их использования, а только возможность использования аннотаций типов. К счастью, существует такая утилита, как фреймворк Checker, в котором описывается несколько *ти*повых аннотаций и предоставляется возможность усовершенствования с их помощью проверки типов. Если она вас заинтересовала, рекомендуем взглянуть на обучающее руководство: http://www.checkerframework.org/. Дополнительную информацию о применении аннотаций можно найти здесь: http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.7.4.

A.2. Обобщенный вывод целевых типов

Java 8 расширяет возможности вывода обобщенных аргументов. Вы уже знакомы с выводом типов в предыдущих версиях Java с помощью контекстной информации. Например, метод `emptyList` в Java описан следующим образом:

```
static <T> List<T> emptyList();
```

Этот метод параметризован параметром типа `T`. Чтобы задать конкретный тип для параметра типа, его можно вызвать вот так:

```
List<Car> cars = Collections.<Car>emptyList();
```

Но Java умеет выводить обобщенный аргумент. Следующий код эквивалентен вышеприведенному:

```
List<Car> cars = Collections.emptyList();
```

До появления Java 8 возможности этого механизма вывода, основанного на контексте (то есть ожидаемом в данном контексте типе), были ограничены. Например, было невозможно следующее:

```
static void cleanCars(List<Car> cars) {

}
cleanCars(Collections.emptyList());
```

При попытке компиляции этого кода вы бы получили следующую ошибку:

```
cleanCars (java.util.List<Car>)cannot be applied to
     (java.util.List<java.lang.Object>)
```

Для ее исправления необходимо указать явный аргумент типа, такой как был показан выше.

В Java 8 целевой тип учитывает и аргументы метода, так что указывать явным образом обобщенный аргумент не нужно:

```
List<Car> cleanCars = dirtyCars.stream()
                               .filter(Car::isClean)
                               .collect(Collectors.toList());
```

Именно это усовершенствование позволяет писать в данном коде `Collectors.toList()` вместо `Collectors.<Car>toList()`.

Б Прочие изменения в библиотеках

Здесь обсуждаются основные изменения, внесенные в библиотеки Java 8.

Б.1. Коллекции

Главная новинка Collection API — появление потоков данных, которые мы обсуждали в главах 4–6. В него были внесены и другие изменения, обсуждавшиеся в главе 9, а также различные мелкие дополнения, которые мы вкратце подытожим в этом приложении.

Б.1.1. Добавленные методы

Создатели API Java выжали максимум из уже существующих методов и добавили в интерфейсы и классы коллекций несколько новых. Эти новые методы перечислены в табл. Б.1.

Таблица Б.1. Новые методы, добавленные в классы и интерфейсы коллекций

Класс/интерфейс	Новые методы
Map	getOrDefault, forEach, compute, computeIfAbsent, computeIfPresent, merge, putIfAbsent, remove(key, value), replace, replaceAll, of, ofEntries
Iterable	forEach, spliterator
Iterator	forEachRemaining
Collection	removeIf, stream, parallelStream
List	replaceAll, sort, of
BitSet	stream
Set	of

Map

В интерфейс `Map` было внесено больше всего обновлений в виде нескольких удобных новых методов. Например, методом `getOrDefault` можно заменить существующую идиому для проверки того, содержит ли объект `Map` соответствие для заданного

ключа (если нет, то теперь можно указать значение по умолчанию, которое будет возвращаться взамен). Ранее приходилось писать что-то вроде:

```
Map<String, Integer> carInventory = new HashMap<>();
Integer count = 0;
if(map.containsKey("Aston Martin")){
  count = map.get("Aston Martin");
}
```

Теперь же можно просто сделать следующее:

```
Integer count = map.getOrDefault("Aston Martin", 0);
```

Отметим, что это работает только при отсутствии соответствия в объекте `Map`. Например, если ключ явным образом отображается на значение `null`, то никакого значения по умолчанию возвращено не будет.

Еще один очень удобный метод — `computeIfAbsent`, который мы мельком упоминали в главе 19, когда обсуждали мемоизацию. Он удобен для воплощения паттерна кэширования. Допустим, что нужно извлекать и обрабатывать данные с различных сайтов. При таком сценарии полезно кэшировать данные, чтобы не выполнять дорогостоящую операцию их извлечения несколько раз:

```
public String getData(String url){
    String data = cache.get(url);
    if(data == null){
        data = getData(url);
        cache.put(url, data);
    }
    return data;
}
```

Проверяем, не кэшированы ли уже эти данные

Если нет, извлекаем данные и затем кэшируем их в ассоциативном массиве для использования в будущем

Теперь можно написать этот код в более лаконичном виде с помощью метода `computeIfAbsent`:

```
public String getData(String url){
    return cache.computeIfAbsent(url, this::getData);
}
```

Описание всех остальных методов вы можете найти в официальной документации API Java (http://docs.oracle.com/javase/8/docs/api/java/util/Map.html). Обратите внимание, что в классе `ConcurrentHashMap` также появились дополнительные методы. Мы обсудим их в разделе Б.2.

Интерфейс Collection

Для удаления всех элементов коллекции, соответствующих заданному условию, можно воспользоваться методом `removeIf`. Обратите внимание, что он отличается от метода `filter` из Stream API. Метод `filter` из Stream API генерирует новый поток, а не меняет текущий поток или источник данных.

Интерфейс List

Метод `replaceAll` заменяет все элементы объекта `List` на получающиеся в результате применения к ним заданного оператора. Он напоминает метод `map` потока данных, но меняет элементы объекта `List`, в то время как метод `map` создает новые элементы.

Например, следующий код выводит [2, 4, 6, 8, 10], поскольку модифицируются элементы исходного списка:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.replaceAll(x -> x * 2);
System.out.println(numbers);
```

Выводит [2, 4, 6, 8, 10]

Б.1.2. Класс Collections

Класс `Collections`, предназначенный для работы с коллекциями или их возврата, существует уже долгое время. Теперь он включает и дополнительные методы для возврата неизменяемых, синхронизированных, проверяемых и пустых объектов `NavigableMap` и `NavigableSet`. Кроме того, он включает метод `checkedQueue`, возвращающий представление заданной очереди, дополненное динамической проверкой типов.

Б.1.3. Интерфейс Comparator

Интерфейс `Comparator` теперь включает статические методы и методы с реализацией по умолчанию. Мы использовали статический метод `Comparator.comparing` в главе 3 для возврата объекта `Comparator` на основе переданной ему в виде аргумента функции, извлекающей ключ сортировки.

Среди новых методов экземпляра:

- `reversed` — возвращает `Comparator` с обратной упорядоченностью по сравнению с текущим `Comparator`;
- `thenComparing` — возвращает объект `Comparator`, использующий в случае равенства двух объектов другой заданный `Comparator`;
- `thenComparingInt`, `thenComparingDouble`, `thenComparingLong` — работают аналогично методу `thenComparing`, но принимают на входе функцию, работающую с простыми типами данных (соответственно `ToIntFunction`, `ToDoubleFunction` и `ToLongFunction`).

В числе новых статических методов:

- `comparingInt`, `comparingDouble`, `comparingLong` — работают аналогично методу `thenComparing`, но принимают на входе функцию, работающую с простыми типами данных (соответственно `ToIntFunction`, `ToDoubleFunction` и `ToLongFunction`);
- `naturalOrder` — возвращает объект `Comparator`, сравнивающий объекты `Comparable` в естественном порядке;
- `nullsFirst`, `nullsLast` — возвращает объект `Comparator`, который считает, что `null` располагается перед или после не-`null` объектов;
- `reverseOrder` — эквивалентен вызову `naturalOrder().reverse()`.

Б.2. Конкурентность

В Java 8 появилось и несколько усовершенствований, связанных с конкурентностью. Прежде всего, конечно, появление параллельных потоков данных, которые мы обсуждали в главе 7. А кроме того, появление класса `CompletableFuture`, о котором вы можете узнать из главы 16.

Есть и другие заметные изменения. Например, класс `Arrays` теперь поддерживает параллельные операции. Мы обсудим их в разделе Б.3.

В текущем же разделе мы рассмотрим обновления пакета `java.util.concurrent.atomic`, предназначенного для работы с атомарными переменными. Мы также обсудим изменения, внесенные в класс `ConcurrentHashMap`, поддерживающий теперь несколько новых методов.

Б.2.1. Пакет Atomic

В пакете `java.util.concurrent.atomic` имеется несколько классов, предназначенных для представления числовых значений, в том числе `AtomicInteger` и `AtomicLong`, поддерживающие атомарные операции над отдельными переменными. Они были обновлены в Java 8 и поддерживают теперь следующие новые методы.

- `getAndUpdate` — атомарно заменяет текущее значение результатом применения к нему заданной функции, возвращая при этом предыдущее значение.
- `updateAndGet` — атомарно заменяет текущее значение результатом применения к нему заданной функции, возвращая при этом новое значение.
- `getAndAccumulate` — атомарно заменяет текущее значение результатом применения заданной функции к нему и другим заданным значениям, возвращая при этом предыдущее значение.
- `accumulateAndGet` — атомарно заменяет текущее значение результатом применения заданной функции к нему и другим заданным значениям, возвращая при этом новое значение.

Вот пример атомарного присвоения переменной минимума из указанного значения `10` и ее текущего целочисленного значения:

```
int min = atomicInteger.accumulateAndGet(10, Integer::min);
```

Сумматоры и накопители

API Java рекомендует использовать новые классы `LongAdder`, `LongAccumulator`, `DoubleAdder` и `DoubleAccumulator` вместо эквивалентных классов пакета `Atomic` в тех случаях, когда в нескольких потоках выполнения данные часто обновляются, а читаются реже (например, в контексте статистических вычислений). Эти классы спроектированы так, чтобы накапливать данные динамически, снижая тем самым издержки на переключение потоков.

Классы `LongAdder` и `DoubleAdder` поддерживают только операции сложения, в то время как `LongAccumulator` и `DoubleAccumulator` позволяют указывать функцию (точнее, лямбда-выражение типа `LongBinaryOperator`. — *Примеч. пер.*) для объединения входящих значений. Например, для вычисления суммы нескольких значений можно воспользоваться классом `LongAdder` следующим образом (листинг Б.1).

Листинг Б.1. Применение класса LongAdder для вычисления суммы значений

```
LongAdder adder = new LongAdder();
adder.add(10);
// ...
long sum = adder.sum();
```

В какой-то момент получаем сумму

Складываем значения в нескольких разных потоках выполнения

Конструктор по умолчанию устанавливает начальное значение равным 0

Или можно воспользоваться классом `LongAccumulator` (листинг Б.2).

Листинг Б.2. Применение класса LongAccumulator для вычисления суммы значений

```
LongAccumulator acc = new LongAccumulator(Long::sum, 0);
acc.accumulate(10);
// ...
long result = acc.get();
```

Накапливаем значения в нескольких разных потоках выполнения

В какой-то момент получаем результат

Б.2.2. ConcurrentHashMap

Класс `ConcurrentHashMap` появился в качестве более современной и поддерживающей конкурентные операции версии класса `HashMap`. Класс `ConcurrentHashMap` поддерживает конкурентное добавление и обновление данных с блокировкой лишь определенных частей внутренней структуры данных. Таким образом, удалось повысить производительность операций чтения и записи по сравнению с альтернативным вариантом синхронизированного `Hashtable`.

Производительность

Внутренняя структура `ConcurrentHashMap` была обновлена ради повышения производительности. Элементы ассоциативного массива обычно хранятся в корзинах, доступ к которым осуществляется по хеш-коду ключа. Однако большое число ключей с одинаковым хеш-кодом может привести к снижению производительности, поскольку корзины реализованы в виде списков со сложностью извлечения порядка $O(n)$. Когда размер корзин в Java 8 становится слишком велик, они динамически заменяются на упорядоченные деревья, сложность извлечения из которых имеет порядок $O(\log(n))$. Отметим, что это возможно только в случае ключей с типом, реализующим интерфейс `Comparable` (например, классов `String` или `Number`).

Потокоподобные операции

Класс `ConcurrentHashMap` поддерживает три новых вида операций, напоминающих операции с потоками данных.

- `forEach` — выполняет заданное действие для каждой пары «ключ, значение».
- `reduce` — объединяет все пары «ключ, значение» в единый результат с помощью заданной функции свертки.
- `search` — применяет заданную функцию к каждой паре «ключ, значение» вплоть до получения непустого результата[1].

Каждый из этих видов операций существует в четырех формах, принимающих на входе функции со следующими аргументами: ключи, значения, объекты `Map.Entry`, и пары «ключ, значение»:

- работают с ключами и значениями (`forEach`, `reduce`, `search`);
- работают с ключами (`forEachKey`, `reduceKey`, `searchKey`);

[1] И возвращает этот непустой результат или null, если такового не нашлось. — *Примеч. пер.*

- работают со значениями (`forEachValue`, `reduceValues`, `searchValues`);
- работают с объектами `Map.Entry` (`forEachEntry`, `reduceEntries`, `searchEntries`).

Обратите внимание, что эти операции не блокируют состояние `ConcurrentHashMap`. Они работают с элементами по мере обхода данных. Указываемые в этих операциях функции не должны зависеть от какого-либо порядка обхода, от других объектов или значений, которые могут поменяться во время проведения вычислений.

Кроме того, для всех этих операций необходимо указывать аргумент `parallelismThreshold`. Если, по оценке, текущий размер `map` меньше этого заданного порогового значения, то операции выполняются последовательно. Значение `1` этого аргумента означает максимально возможный параллелизм за счет максимально полного использования общего пула потоков выполнения. При значении же `Long.MAX_VALUE` операция будет ограничена одним потоком выполнения.

В следующем примере мы воспользуемся методом `reduceValues` для поиска максимального значения в объекте `Map`:

```
ConcurrentHashMap<String, Integer> Map = new ConcurrentHashMap<>();
Optional<Integer> maxValue =
    Optional.of(map.reduceValues(1, Integer::max));
```

Отметим, что для каждой из операций `reduce` есть соответствующие функции свертки к простым типам данных — `int`, `long` и `double` (например, `reduceValuesToInt`, `reduceKeysToLong` и т. д.).

Подсчет

Класс `ConcurrentHashMap` предоставляет новый метод `mappingCount`, который служит для подсчета количества элементов в ассоциативном массиве в виде значения типа `long`. Рекомендуется использовать его вместо метода `size`, возвращающего значение типа `int`, поскольку количество элементов может оказаться слишком велико и превысить предел типа `int`.

Представление в виде Set

Класс `ConcurrentHashMap` предоставляет новый метод `keySet`, возвращающий представление объекта `ConcurrentHashMap` в виде объекта типа `Set` (изменения в объекте `Map` отражаются в объекте `Set`, и наоборот). Можно также создать объект `Set`, основанный на данных из `ConcurrentHashMap`, с помощью нового статического метода `newKeySet`.

Б.3. Массивы

Класс `Arrays` предоставляет различные статические методы для работы с массивами. Теперь в нем появилось еще четыре новых метода (с перегруженными вариантами для работы с различными простыми типами данных).

Б.3.1. Использование метода parallelSort

Метод `parallelSort` сортирует заданный массив параллельно в естественном порядке или с помощью заданного объекта `Comparator` для массива объектов.

Б.3.2. Использование методов setAll и parallelSetAll

Методы `setAll` и `parallelSetAll` предназначены для присвоения значений всем объектам заданного массива соответственно последовательно или параллельным образом с использованием передаваемой им функции для вычисления значения каждого элемента. Поскольку метод `parallelSetAll` выполняется параллельным образом, то упомянутая функция должна быть без побочных эффектов, как объяснялось в главах 7 и 18.

В качестве примера воспользуемся методом `setAll` для генерации массива значений 0, 2, 4, 6...

```
int[] evenNumbers = new int[10];
Arrays.setAll(evenNumbers, i -> i * 2);
```

Б.3.3. Использование метода parallelPrefix

Метод `parallelPrefix` параллельно накапливает результат в элементах данного массива на основе указанного бинарного оператора. В листинге Б.3 мы сгенерируем значения 1, 2, 3, 4, 5, 6, 7...

Листинг Б.3. Накопление результата в элементах массива с помощью метода parallelPrefix

```
int[] ones = new int[10];
Arrays.fill(ones, 1);
Arrays.parallelPrefix(ones, (a, b) -> a + b);
```

Массив ones теперь содержит значения [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Б.4. Классы Number и Math

В API Java 8 классы `Number` и `Math` тоже дополнены новыми методами.

Б.4.1. Класс Number

Вот новые методы класса `Number`.

- В классы `Short`, `Integer`, `Long`, `Float` и `Double` включены статические методы `sum`, `min` и `max`. Вы уже видели эти методы в сочетании с операцией `reduce` в главе 5.
- Классы `Integer` и `Long` включают методы `compareUnsigned`, `divideUnsigned`, `remainderUnsigned` и `toUnsignedString` для работы с беззнаковыми значениями.
- Классы `Integer` и `Long` также включают соответственно статические методы `parseUnsignedInt` и `parseUnsignedLong` для синтаксического разбора строк в беззнаковые значения типа `int` или `long`.
- Классы `Byte` и `Short` включают статические методы `toUnsignedInt` и `toUnsignedLong`, преобразующие аргумент в `int` или `long` путем беззнакового преобразования. Аналогично класс `Integer` теперь включает статический метод `toUnsignedLong`.

- Классы `Double` и `Float` включают статический метод `isFinite` для проверки того, представляет ли аргумент собой конечное значение с плавающей точкой.
- Класс `Boolean` теперь включает статические методы `logicalAnd`, `logicalOr` и `logicalXor` для выполнения логических операций `and`, `or` и `xor` между двумя булевыми значениями.
- Класс `BigInteger` включает методы `byteValueExact`, `shortValueExact`, `intValueExact` и `longValueExact`, служащие для преобразования этого объекта `BigInteger` в соответствующий простой тип данных. Но в случае потери информации при таком преобразовании эти методы генерируют арифметическое исключение.

Б.4.2. Класс Math

Класс `Math` включает новые методы, генерирующие арифметическое исключение в случае переполнения при выполнении операции. В числе этих методов `addExact`, `subtractExact`, `multiplyExact`, `incrementExact`, `decrementExact` и `negateExact` с аргументами типов `int` или `long`. Кроме того, появился статический метод `toIntExact` для преобразования значения `long` в `int`. Среди других дополнений — статические методы `floorMod`, `floorDiv` и `nextDown`.

Б.5. Класс Files

В числе наиболее заметных новшеств класса `Files` — возможность генерации потоков данных на основе файлов. Мы упоминали в главе 5 новый статический метод `Files.lines`, который предоставляет возможность отложенного чтения данных из файла как потока. Среди других удобных статических методов, возвращающих поток данных:

- `Files.list` — генерирует объект `Stream<Path>`, состоящий из элементов заданного каталога. Этот перечень не рекурсивен. Данный метод удобен при обработке очень больших каталогов, поскольку поток данных потребляется отложенным образом;
- `Files.walk` — аналогично `Files.list` этот метод генерирует объект `Stream<Path>`, состоящий из элементов заданного каталога. Отличие состоит в том, что этот перечень рекурсивен, причем уровень вложенности можно настраивать. Отметим, что обход выполняется «вглубь»;
- `Files.find` — генерирует объект `Stream<Path>` на основе рекурсивного обхода каталога для поиска элементов, соответствующих заданному условию.

Б.6. Рефлексия

Мы обсудили несколько изменений в механизме аннотаций в Java 8 в приложении А. Для поддержки этих изменений был обновлен и Reflection API.

Еще одно изменение в этом API — возможность доступа к информации о параметрах методов (например, названиях и модификаторах) с помощью нового

класса `java.lang.reflect.Parameter`. На него ссылается новый класс `java.lang.reflect.Executable` — суперкласс для общей функциональности классов `Method` и `Constructor`.

Б.7. Класс String

Класс `String` теперь включает удобный статический метод `join`, предназначенный — как вы легко можете догадаться — для объединения строк с разделителем! Использовать его можно следующим образом:

```
String authors = String.join(", ", "Raoul", "Mario", "Alan");
System.out.println(authors);
```

Получаем на выходе Raoul, Mario, Alan

В *Параллельное выполнение нескольких операций над потоком данных*

Одно из главных ограничений потоков данных Java 8 — то, что их можно обрабатывать лишь однократно и получать лишь один результат. И действительно, если попытаться прочитать поток данных второй раз, можно получить разве что вот такое исключение:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

Несмотря на это, существуют ситуации, когда необходимо получить несколько результатов обработки одного потока данных. Например, произвести синтаксический разбор файла журнала, как в подразделе 5.7.3, но получить за один раз несколько сводных показателей. Или, в случае модели данных меню (на примере которой мы поясняли возможности потоков данных в главах 4, 5 и 6), извлекать различную информацию при обходе потока блюд.

Другими словами, было бы удобно пропускать поток данных через несколько различных лямбда-выражений за один проход, для чего нужен некий метод ветвления с применением различных функций к каждому из полученных в результате него потоков. А еще лучше было бы производить эти операции параллельно, вычисляя различные требуемые результаты в различных потоках выполнения.

К сожалению, таких возможностей текущая реализация потоков данных Java 8 не содержит, но в данном приложении мы продемонстрируем реализацию этой полезной возможности (и удобного API для нее) с помощью интерфейса `Spliterator` и, в частности, его возможностей позднего связывания вместе с интерфейсами `BlockingQueue` и `Future`[1].

[1] В основе представленной в данном приложении реализации лежит решение, опубликованное Полом Сандозом (Paul Sandoz) в рассылке lambda-dev: http://mail.openjdk.java.net/pipermail/lambda-dev/2013-November/011516.html.

B.1. Ветвление потока данных

Прежде всего для параллельного выполнения нескольких операций над потоком данных нам нужно создать адаптер для исходного потока — класс `StreamForker`, в котором мы опишем различные операции, которые хотели бы выполнять. Взгляните на листинг B.1.

Листинг B.1. Описание класса StreamForker для выполнения нескольких операций над потоком данных

```
public class StreamForker<T> {

    private final Stream<T> stream;
    private final Map<Object, Function<Stream<T>, ?>> forks =
                                                        new HashMap<>();
    public StreamForker(Stream<T> stream) {
        this.stream = stream;
    }

    public StreamForker<T> fork(Object key, Function<Stream<T>, ?> f) {
        forks.put(key, f);
        return this;
    }

    public Results getResults() {
        // Необходимо реализовать
    }
}
```

Задает ключ, индексирующий функцию, которая будет обрабатывать поток данных

Возвращает this для обеспечения возможности текучих многократных вызовов метода fork

В данном случае метод `fork` принимает два аргумента.

- Объект типа `Function`, предназначенный для преобразования потока данных в результат произвольного типа, соответствующего одной из операций.
- Ключ, с помощью которого можно будет извлекать результат операции, и накапливать пары «ключ/функция» во внутреннем ассоциативном массиве.

Метод `fork` возвращает сам объект `StreamForker`; следовательно, с помощью ветвления на несколько операций можно построить конвейер. Рисунок B.1 иллюстрирует основные идеи, заложенные в классе `StreamForker`.

В данном случае над потоком данных должны быть выполнены три операции, которым соответствуют три ключа. `StreamForker` обходит исходный поток, производя ветвление его на три других потока. После этого можно параллельно применить три нужные операции к трем полученным в результате ветвления потокам и поместить результаты с соответствующими ключами в итоговый ассоциативный массив.

Выполнение всех добавленных через метод `fork` операций запускается с помощью вызова метода `getResults`, возвращающего реализацию интерфейса `Results`, описание которого выглядит следующим образом:

```
public static interface Results {
    public <R> R get(Object key);
}
```

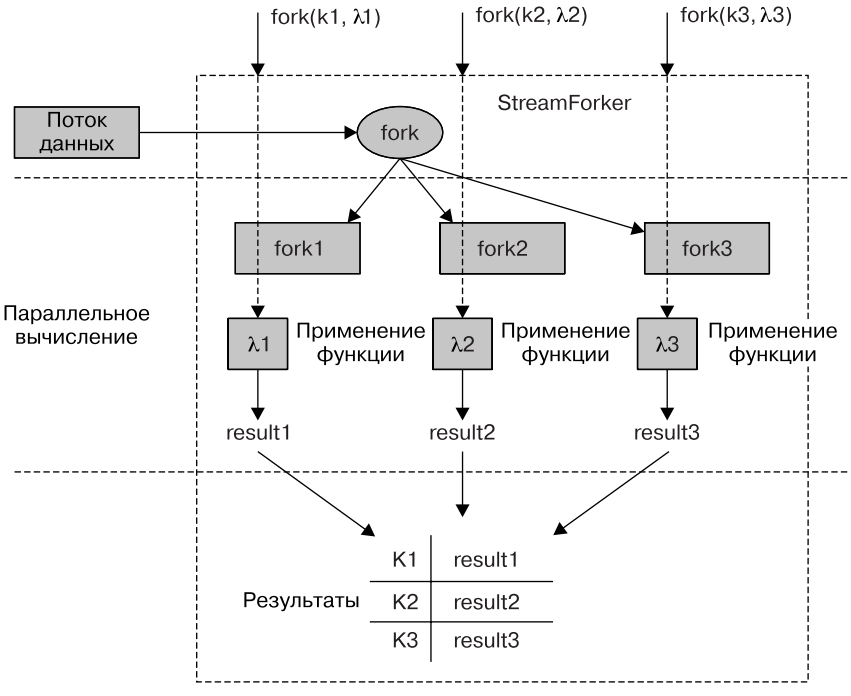


Рис. B.1. Класс StreamForker в действии

Этот интерфейс включает всего один метод, в который передается один из используемых в каком-либо из вызовов метода `fork` ключей типа `Object`, и возвращается результат соответствующей этому ключу операции.

B.1.1. Реализация интерфейса Results на основе класса ForkingStreamConsumer

Метод `getResults` можно реализовать следующим образом:

```
public Results getResults() {
    ForkingStreamConsumer<T> consumer = build();
    try {
        stream.sequential().forEach(consumer);
    } finally {
        consumer.finish();
    }
    return consumer;
}
```

Класс `ForkingStreamConsumer` реализует одновременно описанный выше интерфейс `Results` и интерфейс `Consumer`. Как вы увидите далее, когда мы будем анализировать данную реализацию подробно, его основная задача состоит в потреблении всех элементов потока данных и размножении их по блокирующим очередям (объектам `BlockingQueue`), число которых соответствует числу отправленных на

выполнение через метод `fork` операций. Отметим, что в коде обеспечивается последовательность потока данных, поскольку выполнение метода `forEach` для параллельного потока данных может привести к попаданию его элементов в очереди в неправильном порядке. Метод `finish` добавляет в эти очереди специальные завершающие элементы, указывающие, что больше обрабатывать нечего. В методе `build` создается объект `ForkingStreamConsumer`, как показано в листинге В.2.

Листинг В.2. Метод build, используемый для создания объекта ForkingStreamConsumer

```
private ForkingStreamConsumer<T> build() {
    List<BlockingQueue<T>> queues = new ArrayList<>();

    Map<Object, Future<?>> actions =
            forks.entrySet().stream().reduce(
                    new HashMap<Object, Future<?>>(),
                    (map, e) -> {
                        map.put(e.getKey(),
                                getOperationResult(queues, e.getValue()));
                        return map;
                    },
                    (m1, m2) -> {
                        m1.putAll(m2);
                        return m1;
                    });

    return new ForkingStreamConsumer<>(queues, actions);
}
```

Создаем список очередей, по одной для каждой операции

Создаем ассоциативный массив с фьючерсами, которые будут содержать результаты операций, и ключами, служащими для идентификации этих операций

В листинге В.2 мы сначала создаем список упомянутых выше блокирующих очередей. Далее мы создаем ассоциативный массив с теми же ключами, что используются для идентификации выполняемых над потоком данных операций, и фьючерсами в качестве значений, которые будут содержать соответствующие результаты этих операций. Далее список блокирующих очередей и ассоциативный массив фьючерсов передаются конструктору класса `ForkingStreamConsumer`. Фьючерсы создаются с помощью метода `getOperationResult`, как показано в листинге В.3.

Листинг В.3. Создание фьючерсов с помощью метода getOperationResult

```
private Future<?> getOperationResult(List<BlockingQueue<T>> queues,
                                     Function<Stream<T>, ?> f) {
    BlockingQueue<T> queue = new LinkedBlockingQueue<>();
    queues.add(queue);

    Spliterator<T> spliterator = new BlockingQueueSpliterator<>(queue);
    Stream<T> source = StreamSupport.stream(spliterator, false);
    return CompletableFuture.supplyAsync( () -> f.apply(source) );
}
```

Создаем очередь и добавляем ее в список очередей

Создаем сплитератор для обхода элементов данной очереди

Создаем поток данных с вышеупомянутым сплитератором в качестве источника данных

Создаем фьючерс для асинхронного вычисления заданной функции на основе данных этого потока

Метод `getOperationResult` создает новую блокирующую очередь и добавляет ее в список очередей. Эта очередь затем передается конструктору для создания объекта `BlockingQueueSpliterator` (сплитератор с поздним связыванием), задачей которого является чтение обрабатываемых элементов из очереди; чуть позже мы обсудим, как это происходит.

Далее мы создаем последовательный поток данных с вышеупомянутым объектом `Spliterator` в качестве источника и наконец создаем фьючерс для асинхронного применения к данному потоку соответствующей ему функции. Создается этот фьючерс с помощью статического фабричного метода класса `CompletableFuture`, реализующего интерфейс `Future`. Это еще один новый класс, появившийся в Java 8, мы обсуждали его подробно в главе 16.

B.1.2. Разработка классов ForkingStreamConsumer и BlockingQueueSpliterator

Последние две части, которые нам нужно разработать, — это упоминавшиеся выше классы `ForkingStreamConsumer` и `BlockingQueueSpliterator`. Первый из них можно реализовать следующим образом (листинг B.4).

Листинг B.4. Класс ForkingStreamConsumer, предназначенный для добавления элементов потока данных в несколько очередей

```
static class ForkingStreamConsumer<T> implements Consumer<T>, Results {
    static final Object END_OF_STREAM = new Object();

    private final List<BlockingQueue<T>> queues;
    private final Map<Object, Future<?>> actions;

    ForkingStreamConsumer(List<BlockingQueue<T>> queues,
                          Map<Object, Future<?>> actions) {
        this.queues = queues;
        this.actions = actions;
    }

    @Override
    public void accept(T t) {
        queues.forEach(q -> q.add(t));
    }

    void finish() {
        accept((T) END_OF_STREAM);
    }

    @Override
    public <R> R get(Object key) {
        try {
            return ((Future<R>) actions.get(key)).get();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Трансляция текущего обрабатываемого элемента потока данных во все очереди

Добавляем в очередь специальный последний элемент, который указывает, что чтение потока завершено

Возвращаем результат операции, индексированной заданным ключом и ждем завершения вычисляющего ее фьючерса

Этот класс реализует оба интерфейса, `Results` и `Consumer`, и содержит ссылку на список блокирующих очередей, а также ассоциативный массив фьючерсов, выполняющих различные операции над потоком.

Интерфейс `Consumer` требует реализации метода `accept`. В данном случае всякий раз, когда `ForkingStreamConsumer` получает (`accept`) элемент потока данных, он добавляет его во все блокирующие очереди. Кроме того, по завершении добавления всех элементов исходного потока данных во все очереди метод `finish` производит добавление в каждую их них по одному последнему элементу. Когда объект `BlockingQueueSpliterator` встречает такой элемент, он понимает, что больше обрабатывать нечего.

Интерфейс `Results` требует реализации метода `get`. В данном случае он извлекает фьючерс, которому в ассоциативном массиве соответствует переданный в него аргумент `key` и распаковывает результат (или ждет, пока результат станет доступен).

Наконец, для каждой выполняемой над потоком данных операции должен существовать объект `BlockingQueueSpliterator`. В каждом из объектов `BlockingQueueSpliterator` будет ссылка на один из объектов `BlockingQueue`, заполняемых `ForkingStreamConsumer`. Реализовать его можно, например, так, как показано в листинге В.5.

Листинг В.5. Класс, реализующий интерфейс Spliterator и читающий элементы из BlockingQueue

```
class BlockingQueueSpliterator<T> implements Spliterator<T> {
    private final BlockingQueue<T> q;
    BlockingQueueSpliterator(BlockingQueue<T> q) {
        this.q = q;
    }

    @Override
    public boolean tryAdvance(Consumer<? super T> action) {
        T t;
        while (true) {
            try {
                t = q.take();
                break;
            } catch (InterruptedException e) { }
        }
        if (t != ForkingStreamConsumer.END_OF_STREAM) {
            action.accept(t);
            return true;
        }
        return false;
    }

    @Override
    public Spliterator<T> trySplit() {
        return null;
    }

    @Override
    public long estimateSize() {
```

```
        return 0;
    }

    @Override
    public int characteristics() {
        return 0;
    }
}
```

Цель реализации интерфейса `Spliterator` в этом листинге не задание стратегии разбиения потока данных, а лишь использование его возможностей динамического связывания. Поэтому метод `trySplit` мы здесь не реализовывали.

Кроме того, невозможно вернуть какое-либо осмысленное значение из метода `estimatedSize`, поскольку невозможно предугадать, сколько еще элементов будет получено из очереди. Кроме того, выполнение такой оценки не имеет смысла, поскольку мы не пытаемся производить разбиение потока данных.

У этой реализации отсутствуют перечисленные в табл. 7.2 характеристики сплитераторов, так что метод `characteristics` возвращает `0`.

Единственный реализованный здесь метод — `tryAdvance`. Он ждет получения из блокирующей очереди элементов исходного потока (добавленных в нее `ForkingStreamConsumer`). Эти элементы он отправляет в `Consumer`, служащий (в зависимости от того, как был создан сплитератор в методе `getOperationResult`) источником будущего потока (к которому будет применяться соответствующая функция, передаваемая при одном из вызовов метода `fork`). Метод `tryAdvance` возвращает `true`, тем самым сообщая вызывающему его методу, что элементы для потребления еще есть, до тех пор, пока не встретит добавленный `ForkingStreamConsumer` специальный объект, указывающий, что больше в очереди элементов нет. На рис. B.2 приведена общая схема `StreamForker` и его составных частей.

На этом рисунке `StreamForker` в верхнем левом углу содержит ассоциативный массив, в котором каждой операции, которая должна выполняться над потоком данных, соответствует ключ. `ForkStreamConsumer` справа от него включает очередь для каждой из этих операций и потребляет все элементы исходного потока данных, размножая их для помещения в эти очереди.

Внизу рисунка у каждой очереди есть свой `BlockingQueueSpliterator`, предназначенный для извлечения ее элементов и служащий источником для другого потока. Наконец, каждый из этих потоков данных, полученных путем ветвления исходного потока данных, передается в качестве аргумента в одну из функций, что приводит к выполнению одной из запланированных операций. Все компоненты нашего `StreamForker` готовы, можно его использовать.

B.1.3. Применяем StreamForker на практике

Давайте воспользуемся `StreamForker` для модели данных меню, описанной в главе 4. Произведем ветвление исходного потока блюд на четыре для параллельного выполнения над ним четырех различных операций, как показано в листинге B.6. В частности, мы сгенерируем разделенный запятыми список названий всех имеющихся

блюд, вычислим суммарную калорийность меню, найдем самое калорийное блюдо и сгруппируем все блюда по типу.

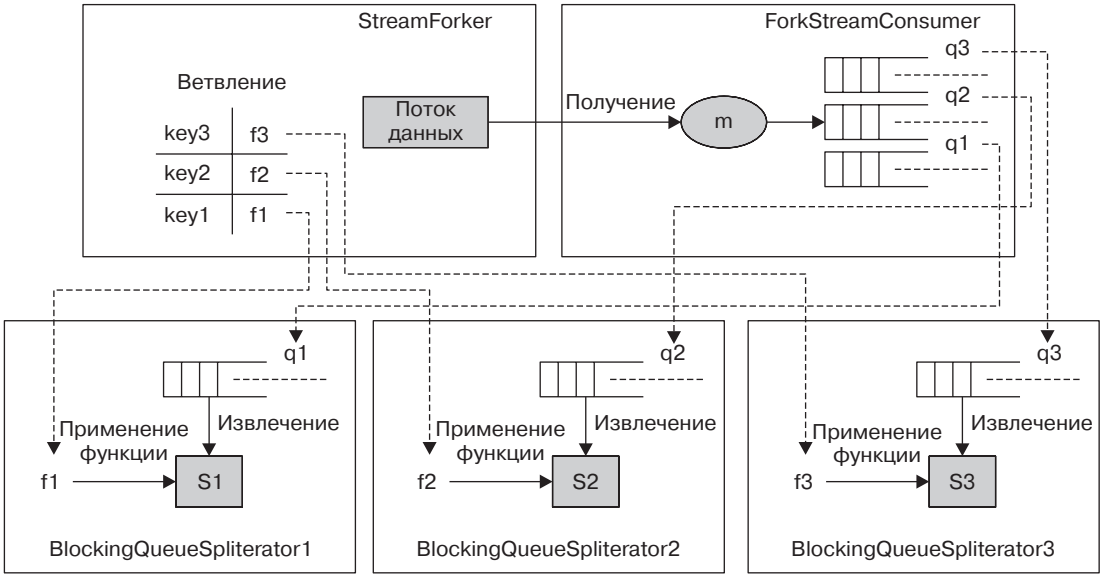


Рис. В.2. Составные части класса StreamForker

Листинг В.6. Используем StreamForker на практике

```
Stream<Dish> menuStream = menu.stream();

StreamForker.Results results = new StreamForker<Dish>(menuStream)
        .fork("shortMenu", s -> s.map(Dish::getName)
                                 .collect(joining(", ")))
        .fork("totalCalories", s -> s.mapToInt(Dish::getCalories).sum())
        .fork("mostCaloricDish", s -> s.collect(reducing(
                (d1, d2) -> d1.getCalories() > d2.getCalories() ? d1 : d2))
                .get())
        .fork("dishesByType", s -> s.collect(groupingBy(Dish::getType)))
        .getResults();

String shortMenu = results.get("shortMenu");
int totalCalories = results.get("totalCalories");
Dish mostCaloricDish = results.get("mostCaloricDish");
Map<Dish.Type, List<Dish>> dishesByType = results.get("dishesByType");
System.out.println("Short menu: " + shortMenu);
System.out.println("Total calories: " + totalCalories);
System.out.println("Most caloric dish: " + mostCaloricDish);
System.out.println("Dishes by type: " + dishesByType);
```

Класс `StreamForker` предоставляет удобный API для ветвления потока данных и выполнения различных операций над каждым из полученных потоков. Эти операции выражаются в терминах, применяемых к потокам функций, для их идентификации могут служить любые объекты; в данном случае мы выбрали для этой цели строковые значения. Когда все нужные вызовы `fork` уже добавлены, можно вызвать для объекта `StreamForker` метод `getResults` для запуска выполнения всех назначенных операций и получения `StreamForker.Results`. А поскольку эти операции на самом деле выполняются асинхронно, то метод `getResults` возвращает результат сразу же, а не ждет, пока все результаты будут доступны.

Получить результат конкретной операции можно путем передачи соответствующего ей ключа интерфейсу `StreamForker.Results`. Если к тому времени выполнение этой операции уже закончилось, метод `get` вернет соответствующий результат; в противном случае он будет удерживать блокировку, пока этот результат не станет доступным.

Как и можно было ожидать, вышеприведенный код выводит в консоль следующие результаты:

```
Short menu: pork, beef, chicken, french fries, rice, season fruit, pizza,
    prawns, salmon
Total calories: 4300
Most caloric dish: pork
Dishes by type: {OTHER=[french fries, rice, season fruit, pizza], MEAT=[pork,
    beef, chicken], FISH=[prawns, salmon]}
```

B.2. Вопросы производительности

Если говорить о производительности, то не стоит считать, что такой подход более эффективен, чем обход потока несколько раз. Накладные расходы на использование блокирующих очередей легко могут перевесить преимущества параллельного выполнения различных операций, если поток состоит из данных, полностью расположенных в оперативной памяти.

И напротив, однократное обращение к потоку данных может оказаться выигрышным ходом, если включает какие-либо дорогостоящие операции ввода/вывода, например, если источник потока представляет собой огромный файл; так что (как и всегда) единственное разумное правило оптимизации производительности приложения звучит как «Просто проверьте!».

Этот пример демонстрирует возможности выполнения над одним потоком нескольких операций за один проход. А главное, мы убеждены, что это доказывает: даже когда нативный API Java не предоставляет какой-либо конкретной возможности, благодаря гибкости лямбда-выражений вы легко можете ее реализовать сами при небольшой толике творческой фантазии в переиспользовании и сочетании уже имеющихся возможностей.

Г Лямбда-выражения и байт-код JVM

Наверное, вам интересно, как компилятор Java реализует лямбда-выражения и как их обрабатывает виртуальная машина Java (JVM). Если вы полагаете, что лямбда-выражения можно просто транслировать в анонимные классы — лучше почитайте дальше. В этом приложении вкратце обсуждается вопрос компиляции лямбда-выражений путем исследования сгенерированных `class`-файлов.

Г.1. Анонимные классы

Мы показывали в главе 2, как использовать анонимные классы для одновременного объявления и создания экземпляров классов. В результате, как и лямбда-выражения, они позволяют создавать реализации для функциональных интерфейсов.

А поскольку лямбда-выражение обеспечивает реализацию абстрактного метода функционального интерфейса, было бы, казалось, логично, чтобы компилятор Java транслировал лямбда-выражения в анонимные классы во время процесса компиляции. Но анонимные классы отличаются определенными нежелательными свойствами, отрицательно влияющими на быстродействие приложений.

- *Для каждого анонимного класса компилятор генерирует новый class-файл.* Названия этих файлов обычно выглядят наподобие `ClassName$1`, где `ClassName` — название класса, в котором встретился данный анонимный класс, за которым следует символ доллара и число. Генерация большого количества class-файлов нежелательна, поскольку каждый из них необходимо загрузить и проверить перед использованием, что влияет на время загрузки приложения. Если бы лямбда-выражения транслировались в анонимные классы, то у каждого лямбда-выражения был бы свой отдельный `class`-файл.
- *Каждый новый анонимный класс означает новый подтип класса или интерфейса.* Сотня различных лямбда-выражений для компараторов в вашей программе будет

означать сотню различных подтипов интерфейса `Comparator`. В определенных ситуациях это может серьезно затруднить оптимизацию виртуальной машиной Java производительности программы во время выполнения.

Г.2. Генерация байт-кода

Компилятор Java компилирует файлы исходного кода Java в байт-код Java. В дальнейшем JVM может выполнить сгенерированный байт-код и запустить приложение. Инструкции для анонимных классов и лямбда-выражений отличаются. Вы можете заглянуть в байт-код и набор констант любого `class`-файла с помощью команды:

```
javap -c -v ClassName
```

Попробуем реализовать экземпляр интерфейса `Function` с помощью старого синтаксиса Java 7, в виде анонимного внутреннего класса, как показано в листинге Г.1.

Листинг Г.1. Объект Function, реализованный в виде анонимного внутреннего класса

```
import java.util.function.Function;
public class InnerClass {
    Function<Object, String> f = new Function<Object, String>() {
        @Override
        public String apply(Object obj) {
            return obj.toString();
        }
    };
}
```

Соответствующий сгенерированный байт-код для созданного в виде анонимного внутреннего класса объекта `Function` будет выглядеть примерно так:

```
 0: aload_0
 1: invokespecial #1        // Метод java/lang/Object."<init>":()V
 4: aload_0
 5: new           #2        // Класс InnerClass$1
 8: dup
 9: aload_0
10: invokespecial #3        // Метод InnerClass$1."<init>":(LInnerClass;)V
13: putfield      #4        // Поле f:Ljava/util/function/Function;
16: return
```

Из этого байт-кода видно следующее.

- Экземпляр типа `InnerClass$1` создается с помощью операции `new` байт-кода. Одновременно ссылка на этот только что созданный объект помещается в стек.
- Операция `dup` копирует эту ссылку в стеке.
- Это значение потребляется инструкцией `invokespecial`, инициализирующей объект.
- Верхушка стека теперь содержит ссылку на объект, сохраненный в поле `f1` класса `LambdaBytecode` с помощью инструкции `putfield`.

Компилятор сгенерировал для этого анонимного класса название `InnerClass$1`. Для перестраховки можете заглянуть также в `class`-файл `InnerClass$1`, где вы найдете код реализации интерфейса `Function`:

```
class InnerClass$1 implements
    java.util.function.Function<java.lang.Object, java.lang.String> {
  final InnerClass this$0;
  public java.lang.String apply(java.lang.Object);
    Code:
       0: aload_1
       1: invokevirtual #3 // Метод java/lang/Object.toString:()Ljava/lang/String;
       4: areturn
}
```

Г.3. Invokedynamic спешит на помощь

Теперь попробуем то же самое с новым синтаксисом Java 8 для лямбда-выражений. Взглянем на `class`-файл, сгенерированный для кода из листинга Г.2.

Листинг Г.2. Объект Function, реализованный с помощью лямбда-выражения

```
import java.util.function.Function;
public class Lambda {
    Function<Object, String> f = obj -> obj.toString();
}
```

Вы обнаружите следующие инструкции байт-кода:

```
 0: aload_0
 1: invokespecial #1     // Метод java/lang/Object."<init>":()V
 4: aload_0
 5: invokedynamic #2,  0 // InvokeDynamic
                            #0:apply:()Ljava/util/function/Function;
10: putfield      #3     // Поле f:Ljava/util/function/Function;
13: return
```

Мы рассказывали выше о недостатках трансляции лямбда-выражения в анонимный внутренний класс, и действительно, как вы видите, результаты сильно различаются. Вместо создания дополнительного класса здесь используется инструкция `invokedynamic`.

Инструкция invokedynamic

Инструкция байт-кода `invokedynamic` появилась в JDK7 для поддержки в JVM языков программирования с динамической типизацией. Инструкция `invokedynamic` добавляет еще один уровень косвенной адресации при вызове метода, чтобы часть логики могла зависеть от определения цели вызова конкретным динамическим языком программирования. Типичный сценарий использования для этой инструкции выглядит следующим образом:

```
def add(a, b) { a + b }
```

Типы `a` и `b` здесь на момент компиляции неизвестны и могут меняться время от времени. Поэтому при первом выполнении инструкции `invokedynamic` виртуальной машиной Java она обращается к методу-загрузчику, реализующему зависящую от языка программирования логику, который определяет, какой именно метод нужно вызвать. Метод-загрузчик возвращает связанную точку вызова. Достаточно вероятно, что если метод `add` был вызван с двумя значениями типа `int` в качестве аргументов, то последующие вызовы также будут с двумя значениями типа `int`. Сама точка вызова может содержать логику, определяющую, при каких условиях ее следует привязать заново.

В листинге Г.2 возможности инструкции `invokedynamic` использовались немного не для той цели, для которой они исходно предназначались. Фактически здесь она использовалась для того, чтобы отложить выбор стратегии трансляции лямбда-выражений в байт-код до момента выполнения. Другими словами, подобное применение инструкции `invokedynamic` позволяет отложить генерацию кода, реализующего лямбда-выражение, до времени выполнения. Такое архитектурное решение имеет положительные следствия.

- Стратегия, используемая для трансляции тела лямбда-выражений в байт-код, становится чистым нюансом реализации. Ее можно будет менять динамически или оптимизировать и модифицировать в будущих реализациях JVM при сохранении обратной совместимости байт-кода.
- Полностью отсутствуют дополнительные накладные расходы вроде дополнительных полей или статического инициализатора.
- В случае лямбда-выражений без сохранения состояния можно создать один экземпляр объекта лямбда-выражения, кэшировать его и всегда возвращать в неизменном виде. Это распространенный сценарий использования, до Java 8 разработчики делали это явным образом, например объявляли конкретный экземпляр `Comparator` в статической терминальной переменной.
- Никакого отрицательного влияния на производительность, поскольку трансляция производится с последующей привязкой результата только при первом обращении к лямбда-выражению. Все последующие вызовы пропускают этот медленный путь и вызывают ранее привязанную реализацию.

Г.4. Стратегии генерации кода

Лямбда-выражения транслируются в байт-код путем помещения их тела в один из создаваемых во время выполнения статических методов. Простейший вариант лямбда-выражения для трансляции — лямбда-выражение без сохранения состояния, не захватывающее никакого состояния из охватывающей области видимости, вроде того, которое мы описали в листинге Г.2. В данном случае компилятор может сгенерировать метод с той же сигнатурой лямбда-выражения, так что результат должен оказаться следующим:

```
public class Lambda {
    Function<Object, String> f = [динамический вызов lambda$1]

    static String lambda$1(Object obj) {
        return obj.toString();
    }
}
```

Случай лямбда-выражения, захватывающего терминальные (или по сути являющиеся терминальными) локальные переменные или поля, как в следующем примере, несколько сложнее:

```
public class Lambda {
    String header = "This is a ";
    Function<Object, String> f = obj -> header + obj.toString();
}
```

В этом случае сигнатура сгенерированного метода не будет такой же, как у лямбда-выражения, поскольку необходимо добавить в нее дополнительные аргументы для передачи дополнительного состояния из внешнего контекста. Простейший способ добиться этого — присоединить спереди к аргументам лямбда-выражения еще по одному аргументу для каждой из захваченных переменных, так что сгенерированный для реализации вышеприведенного лямбда-выражения метод будет выглядеть примерно так:

```
public class Lambda {
    String header = "This is a ";
    Function<Object, String> f = [динамический вызов lambda$1]

    static String lambda$1(String header, Object obj) {
        return obj -> header + obj.toString();
    }
}
```

Больше информации о процессе трансляции лямбда-выражений можно найти по адресу http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html.