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`:

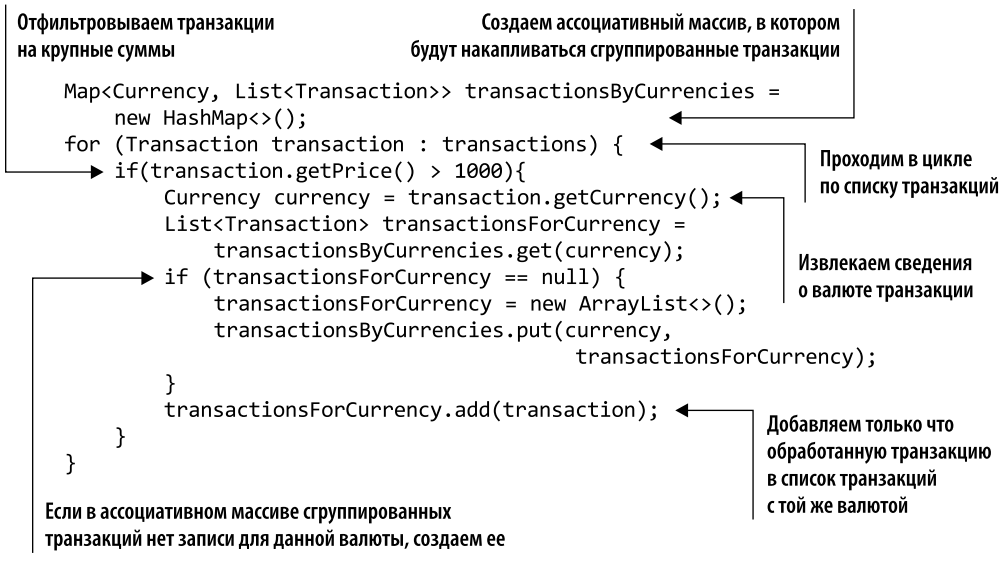
```
filter(inventory, (Apple a) -> a.getWeight() > 150 );
```

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

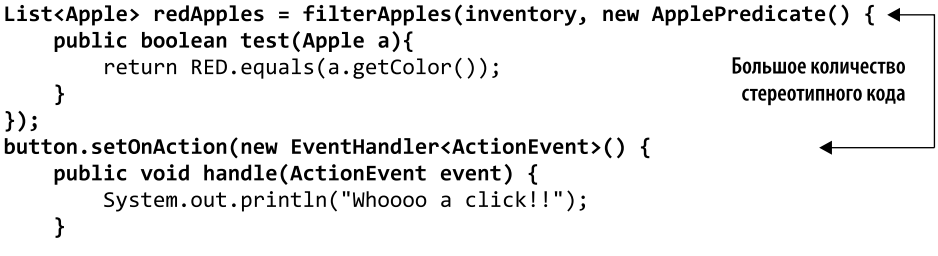
Практически любое приложение Java *создает* и *обрабатывает* коллекции. Однако работать с коллекциями не всегда удобно. Например, представьте себе,

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:



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:



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!

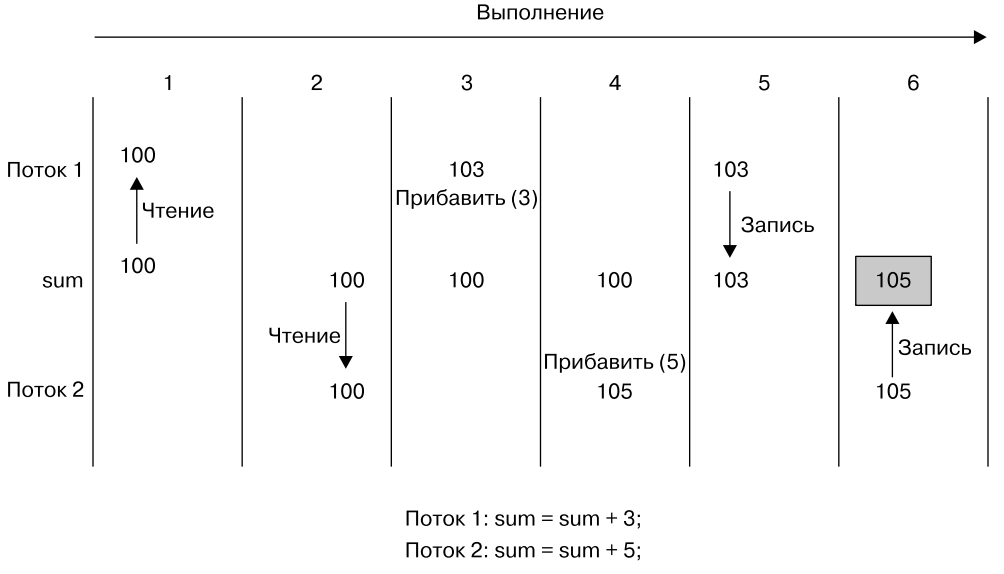


Рис. 1.5. Возможная проблема с двумя потоками выполнения, пытающимися прибавить число к разделяемой переменной sum. Результат равен 105 вместо ожидаемых 108

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.

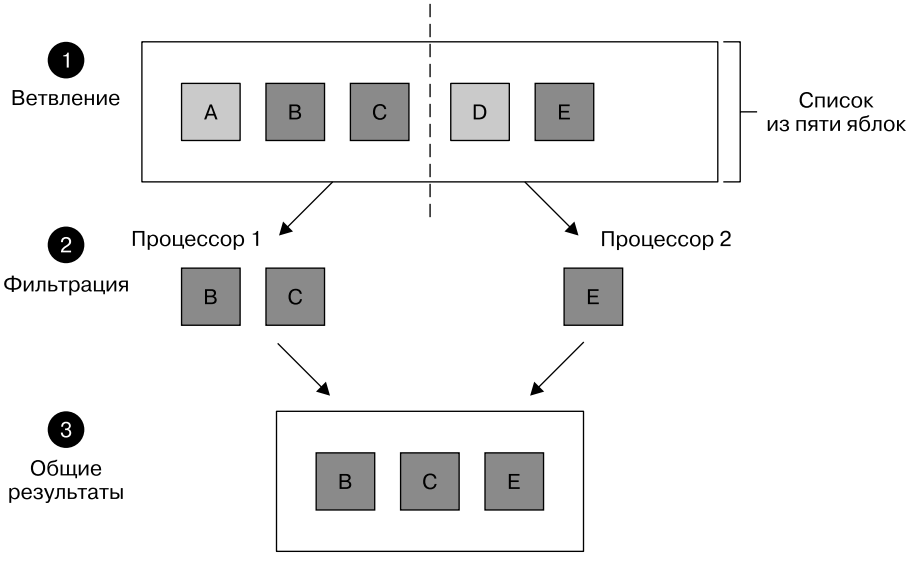


Рис. 1.6. Ветвление filter на два процессора и объединение результатов

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, что и то и другое — полезные инструменты, которые не будут лишними в вашем арсенале. К сожалению, Java 8 не полностью поддерживает сопоставление с шаблоном,

---

[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)).

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`:

```
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?

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.