Содержание

Об авторе

Ричард Уорбэртон – технолог-эмпирик, увлекающийся решением сложных технических задач, требующих глубокого понимания предмета. Профессионально занимался проблемами статического анализа, верификацией части компилятора и разработкой усовершенствованной автоматизированной технологии обнаружения ошибок. Позже заинтересовался методами анализа данных для высокопроизводительных вычислений. Является руководителем лондонского сообщества пользователей Java и членом комитета JCP, организует процесс подачи запросов на улучшение для Java 8 в части лямбда-выражений и механизмов работы с датой и временем. Ричард также часто выступает на конференциях, в том числе JavaOne, DevoxxUK и JAX London. Получил степень доктора философии по информатике в Варвикском университете, где занимался теоретическими вопросами построения компиляторов.

Предисловие

В течение многих лет функциональное программирование считалось уделом небольшой кучки специалистов, неизменно провозглашавших его превосходство, но не способных убедить массы в мудрости своего подхода. И эту книгу я написал прежде всего для того, чтобы оспорить идею о том, будто функциональному стилю присущи какое-то особое превосходство и убежденность в том, что он доступен лишь немногим избранным.

Последние два года я убеждал разработчиков, входящих в лондонское сообщество пользователей Java, попробовать те или иные аспекты Java 8. Как оказалось, многим членам нашего сообщества очень нравятся предоставленные им новые идиомы и библиотеки. Возможно, их несколько смущают терминология и элитарность технологии, но преимущества, которые несет с собой толика несложного функционального программирования, никого не оставляют равнодушными. Все согласны, что гораздо проще читать код манипуляции объектами и коллекциями, написанный с использованием нового Streams API, – например, для выделения музыкальных альбомов, выпущенных в Великобритании, из списка `List` всех альбомов.

Из опыта проведения таких мероприятий я вынес важный урок – все зависит от примеров. Человек учится, решая простые примеры и осознавая стоящие за ними закономерности. Я также понял, что терминология легко может оттолкнуть учащегося, поэтому всегда стараюсь объяснять трудные идеи простыми словами.

Для многих механизмы функционального программирования, включенные в Java 8, представляются невероятно ограниченными: ни тебе монад[1], ни отложенных вычислений на уровне языка, ни дополнительной поддержки неизменяемости. С точки зрения программиста-прагматика, это прекрасно; нам нужна возможность выражать абстракции на уровне библиотек, чтобы можно быть писать простой и чистый код, решающий конкретную задачу. Даже лучше, если кто-то уже написал за нас эти библиотеки, чтобы мы могли сосредоточиться на своей повседневной работе.

---

[1] Больше это слово в тексте ни разу не встретится.

Зачем мне читать эту книгу?

В этой книге мы рассмотрим следующие вопросы.

- Как писать простой, чистый и понятный читателю код, особенно в части работы с коллекциями.
- Как с помощью параллелизма повысить производительность.
- Как более точно моделировать предметную область и создавать более качественные предметно-ориентированные языки.
- Как писать более простой и безошибочный параллельный код.
- Как тестировать и отлаживать лямбда-выражения.

Повышение продуктивности разработчика – не единственная причина добавления лямбда-выражений в Java; действуют еще и глубинные течения в нашей индустрии.

Кому стоит прочитать эту книгу?

Эта книга предназначена разработчикам, пишущим на Java, знакомым с основами Java SE и желающим идти в ногу со значительными изменениями, появившимися в Java 8.

Если вам интересно узнать о том, что такое лямбда-выражения и как они могут повысить ваш профессионализм, читайте дальше! Не предполагается никаких предварительных знаний о лямбда-выражениях или еще каких-то новшествах в базовых библиотеках; все необходимые сведения будут изложены по ходу дела.

Конечно, мне хотелось бы, чтобы каждый разработчик приобрел эту книгу, но, по совести говоря, она нужна не всем. Если вы вообще не знаете языка Java, то эта книга не для вас. С другой стороны, хотя тема лямбда-выражений в Java рассматривается очень подробно, я ничего не рассказываю о том, как они используются в других языках.

Не ожидайте введения в такие аспекты Java SE, как коллекции, анонимные внутренние классы или механизм обработки событий в Swing. Предполагается, что все это вы уже знаете.

Как читать эту книгу

Эта книга построена на примерах: вслед за знакомством с новой концепцией сразу идет код. Иногда в коде может встретиться что-то такое, с чем вы не совсем знакомы. Не пугайтесь – объяснение последует очень скоро, чаще всего в следующем же абзаце.

У такого подхода есть еще и то достоинство, что он позволяет по ходу дела экспериментировать с новыми идеями. Более того, в конце многих глав имеются дополнительные примеры для самостоятельной работы. Я настоятельно рекомендую выполнять эти упражнения – ката. Навык мастера ставит, и – как известно любому программисту-прагматику – очень легко впасть в заблуждение, думая, что понимаешь какой-то код, тогда как на самом деле упустил из виду важную деталь.

Поскольку смысл лямбда-выражений заключается в том, чтобы абстрагировать сложность, убрав ее в библиотеки, то я остановлюсь на нескольких приятных нововведениях в общих библиотеках. В главах 2–6 рассматриваются изменения в самом языке и усовершенствованные библиотеки, входящие в состав JDK 8.

Последние три главы касаются практических применений функционального программирования. В главе 7 я расскажу о нескольких приемах, упрощающих тестирование и отладку кода. В главе 8 объясняется, как применить к лямбда-выражениям общепринятые принципы правильного проектирования программного обеспечения. Затем, в главе 9, мы поговорим о параллелизме и о том, как с помощью лямбда-выражений писать понятный параллельный код, пригодный для сопровождения. Там, где это уместно, я буду знакомить вас со сторонними библиотеками.

Первые четыре главы, наверное, стоит рассматривать как вводный материал – вещи, которые должен знать всякий, кто хочет правильно использовать Java 8. Последующие главы сложнее, зато они научат вас полноценно и уверенно применять лямбда-выражения в собственных проектах. По всей книге рассыпаны упражнения, решения к ним имеются на сайте в GitHub. Если вы не будете пренебрегать упражнениями, то очень скоро овладеете лямбда-выражениями в совершенстве.

Графические выделения

В книге применяются следующие графические выделения:

Курсив
: Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`
: Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный полужирный
: Команды и иной текст, который пользователь должен вводить точно в указанном виде.

Моноширинный курсив
: Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

 Так обозначаются совет или рекомендация.

 Так обозначаются примечания общего характера.

 Так обозначаются предупреждения или предостережения.

О примерах кода

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать с сайта *https://github.com/RichardWarburton/java-8-lambdas-exercises*.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Java 8 Lambdas by Richard Warburton (O'Reilly). Copyright 2014 Richard Warburton, 978-1-449-37077-0».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу *permissions@oreilly.com*.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: *http://oreil.ly/java_8_lambdas*.

Замечания и вопросы технического характера следует отправлять по адресу *bookquestions@oreilly.com*.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу *http://www.oreilly.com*.

Читайте нас на Facebook: *http://facebook.com/oreilly*.

Следите за нашей лентой в Twitter: *http://twitter.com/oreillymedia*.

Смотрите нас на YouTube: *http://www.youtube.com/oreillymedia*.

Благодарности

Хотя на обложке книги стоит мое имя, ее выходу в свет немало поспособствовали и многие другие.

Прежде всего, хочу сказать спасибо редактору Меган и всему коллективу издательства O'Reilly, благодаря которым процесс оказался очень приятным, а сроки – достаточно гибкими. И если бы Мартин и Бен не представили меня Меган, то эта книга никогда и не появилась бы.

Рецензирование сильно помогло улучшить книгу, поэтому я сердечно благодарен всем, кто принимал участие в официальном и неофициальном рецензировании: Мартину Вербургу (Martijn Verburg), Джиму Гафу (Jim Gough), Джону Оливеру (John Oliver), Эдварду Вонгу (Edward Wong), Брайану Гетцу (Brian Goetz), Даниэлю Брайанту (Daniel Bryant), Фреду Розенбергу (Fred Rosenberger), Джайкиран Пай (Jaikiran Pai) и Мани Саркар (Mani Sarkar). А особенно Мартину, который таки убедил меня написать техническую книгу.

Нельзя также не упомянуть группу разработчиков проекта Project Lambda в Oracle. Модернизировать уже сложившийся язык – задача не из легких, и они отлично справились с этой работой в Java 8, заодно предоставив мне материал, о котором можно писать. Благодарности заслуживает также лондонское сообщество пользователей Java, кото-

рое так активно участвовало в тестировании ранних выпусков Java, демонстрируя, какие ошибки допускают разработчики и как их можно исправить.

Помощь и поддержку на протяжении всей работы над книгой мне оказывало множество людей. Особенно хочется выделить родителей, которые приходили на выручку по первому зову. И невыразимо приятно было получать ободрение и позитивные замечания от друзей, в том числе от старых членов компьютерного общества, а особенно от Садика Джаффера (Sadiq Jaffer) и Бойса Бригейда (Boys Brigade).

Глава 1

Введение

Прежде чем вплотную заняться вопросом о том, что такое лямбда-выражения и как ими пользоваться, надо бы понять, для чего вообще они существуют. В этой главе я расскажу об этом, а заодно опишу структуру книги и причины ее появления.

Зачем понадобилось снова изменять Java?

Версия Java 1.0 была выпущена в январе 1996 года, и с тех пор мир программирования претерпел кое-какие изменения. Бизнес требует все более сложных приложений, а программы по большей части исполняются на машинах с мощными многоядерными процессорами. Появление целого ряда виртуальных машин Java (JVM) с эффективными JIT-компиляторами означает, что теперь программисты могут сосредоточиться на создании чистого, удобного для сопровождения кода, а не выжимать из оборудования все до последнего такта процессора, трясясь над каждым байтом памяти.

Все знают о нашествии многоядерных процессоров, но мало кто задумывается об этом. Алгоритмы с применением блокировок чреваты ошибками и требуют много времени на разработку. В пакете `java.util.concurrent` и многочисленных внешних библиотеках предлагаются различные абстракции параллелизма, помогающие писать код, эффективно работающий на многоядерных процессорах. К сожалению, до сих пор мы продвинулись не слишком далеко. Но времена меняются.

В настоящее время есть пределы уровню абстрагирования, доступному авторам библиотек на Java. Хороший пример – отсутствие эффективных параллельных операций с большими коллекциями данных. Java 8 позволяет записывать сложные алгоритмы обработ-

ки коллекций, а путем простого изменения всего лишь одного вызова метода этот код будет эффективно исполняться на многоядерных процессорах. Но чтобы такие библиотеки распараллеливания массовых операций над данными были возможны, в Java пришлось внести дополнение на уровне языка: лямбда-выражения.

Разумеется, за все нужно платить. В данном случае придется научиться читать и писать код с лямбда-выражениями, но это неплохая сделка. Программисту проще изучить новый синтаксис и несколько новых идиом, чем писать вручную горы сложного потокобезопасного кода. Хорошие библиотеки и каркасы существенно сократили временные и финансовые затраты на разработку корпоративных приложений, а теперь следует устранить все барьеры на пути создания простых в использовании и эффективных библиотек.

Идея абстракции знакома всем, кто занимается объектно-ориентированным программированием. Различие же состоит в том, что в объектно-ориентированном программировании абстрагируются главным образом данные, а в функциональном – поведение. В реальном мире, как и в наших программах, все перемешано, поэтому мы можем и должны изучать обе тенденции.

У нового аспекта абстракции есть и другие достоинства – существенные для тех из нас, кому не приходится постоянно писать код, который должен выполняться максимально эффективно. Теперь вы можете писать код, который проще читать, уделяя главное внимание способам выражения своих намерений, а не механизмам их достижения. А код, который легко читать, легко также сопровождать, он более надежен и в меньшей степени подвержен ошибкам.

При создании обратных вызовов и обработчиков событий вы больше не связаны многословностью и неудобочитаемостью анонимных вложенных классов. При таком подходе программисту проще работать с системами обработки событий. Возможность передавать функции из одного места в другое позволяет без особого труда писать отложенный код, в котором значения инициализируются в тот момент, когда это необходимо.

Ко всему прочему, появление в языке методов коллекций по умолчанию (`default`) позволяет программисту лучше сопровождать собственные библиотеки.

Сегодня язык Java не тот, на котором писал ваш дедушка, – и это хорошо.

Что такое функциональное программирование?

Разные люди понимают под словами *функциональное программирование* разные вещи. В его основе лежит осмысление предметной области в терминах неизменяемых значений и функций, которые их преобразуют.

Сообщества, сформировавшиеся вокруг какого-то языка программирования, полагают, что набор средств, включенных в их любимый язык, – единственно правильный. На данном этапе еще слишком рано говорить о том, как определят функциональное программирование программисты, пишущие на Java. Но в определенном смысле это и не важно; существенно то, как писать *хороший* – а не функциональный – код.

В этой книге предметом моего внимания будет прагматичное функциональное программирование, в том числе приемы, которые легко сможет понять и использовать большинство программистов, чтобы создавать программы, более понятные и удобные для сопровождения.

Пример предметной области

Все примеры в этой книге относятся к общей предметной области: музыке. Точнее, мы будем иметь дело с данными, присутствующими в музыкальных альбомах. Ниже приведена краткая сводка терминов.

Исполнитель

Один человек или группа, исполняющая музыкальные произведения:

- *name*: имя или название исполнителя (например, «The Beatles»);
- *members*: другие исполнители, входящие в состав группы (например, «Джон Леннон»), это поле может быть пустым;
- *origin*: место, где возникла группа (например, «Ливерпуль»).

Произведение

Одно музыкальное произведение:

- *name:* название произведения (например, «Yellow Submarine»).

Альбом

Собрание нескольких музыкальных произведений в одном издании:

- *name:* название альбома (например, «Revolver»);
- *tracks:* список произведений;
- *musicians:* список исполнителей, принимавших участие в работе над альбомом.

На примере этой предметной области мы продемонстрируем применение функционального программирования в типичном бизнес-приложении на Java. Возможно, на ваш взгляд, этот пример не идеален, но он простой, а многие приведенные в этой книге фрагменты кода аналогичны встречающимся в реальных задачах.

Глава 2

Лямбда-выражения

Самое серьезное изменение на уровне языка, появившееся в Java 8, – лямбда-выражения – компактный способ передать поведение из одного места программы в другое. Поскольку это тот элемент, который лежит в основе всей книги, поговорим о том, что же он собой представляет.

Наше первое лямбда-выражение

Swing – это платформенно-независимая Java-библиотека для создания графического интерфейса пользователя (ГИП). В ней повсеместно встречается идиома, в соответствии с которой для выяснения того, что сделал пользователь, необходимо зарегистрировать *прослушиватель событий*. Затем прослушиватель сможет выполнить что-то полезное в ответ на действие пользователя (см. пример 2.1).

Пример 2.1 ❖ Использование анонимного внутреннего класса для связывания поведения с нажатием кнопки

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

Здесь мы создаем новый объект, предоставляющий реализацию интерфейса `ActionListener`. В этом интерфейсе определен единственный метод `actionPerformed`, который вызывается объектом `button`, когда пользователь нажимает кнопку на экране. Анонимный внутренний класс как раз и предоставляет реализацию этого метода. В примере 2.1 он просто печатает сообщение о том, что была нажата кнопка.


На самом деле это пример использования *кода как данных* – мы передаем кнопке объект, который представляет действие.

Анонимные внутренние классы были придуманы, чтобы упростить передачу кода как данных в Java. К сожалению, упрощение оказалось

недостаточным. Мы по-прежнему видим четыре строки стереотипного кода, обрамляющих единственно важную строку, содержащую логику.

Но наличие стереотипного кода – не единственная проблема: этот код довольно трудно читать, потому что он затемняет намерение программиста. Мы не хотим передавать никакой объект, в действительности требуется передать некое поведение. В Java 8 этот код можно переписать в виде лямбда-выражения, как показано в примере 2.2.

Пример 2.2 ❖ Использование лямбда-выражения для связывания поведения с нажатием кнопки

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Вместо объекта, реализующего интерфейс, мы передаем блок кода – функцию без имени. Здесь `event` – имя параметра, такое же, как в примере с анонимным внутренним классом, – а `->` отделяет параметр от тела лямбда-выражения, содержащего код, исполняемый при нажатии кнопки.

Еще одно отличие этого примера от анонимного внутреннего класса – способ объявления переменной `event`. Раньше нужно было явно указать тип – `ActionEvent event`. Теперь тип не указывается вовсе, и тем не менее код компилируется. Подспудно компилятор `javac` выводит тип переменной `event` из контекста – в данном случае из сигнатуры метода `addActionListener`. Это означает, что нет нужды явно выписывать тип в случае, когда он очевиден. Ниже мы рассмотрим механизм выведения типа более подробно, но сначала познакомимся с различными способами записи лямбда-выражений.


Хотя для записи параметров лямбда-метода требуется меньше стереотипного кода, чем раньше, они все равно статически типизированы. Для большей удобочитаемости объявления типов можно включать явно, а иногда компилятор просто не может вывести их автоматически!

Как опознать лямбда-выражение

В примере 2.3 показано несколько вариантов основного формата записи лямбда-выражений.

Пример 2.3 ❖ Различные способы записи лямбда-выражений

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
```

```
    System.out.print("Hello");
    System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹


BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

❶ показывает, как можно записать лямбда-выражение вообще без аргументов. Отсутствие аргументов обозначается парой пустых скобок `()`. Это лямбда-выражение реализует интерфейс `Runnable`, единственный метод которого `run` не принимает аргументов и «возвращает» значение типа `void`.

В случае ❷ имеется лямбда-выражение с одним аргументом, и в этой ситуации окружающие его скобки можно опустить. Это именно тот формат, который использовался в примере 2.2.

Лямбда-выражение может включать не только единственное выражение, но и целый блок кода, заключенный в фигурные скобки (`{}`), как в случае ❸. К таким блокам применяются те же правила, что и для обычных методов. В частности, они могут возвращать значения и возбуждать исключения. В фигурные скобки можно заключать и однострочное лямбда-выражение, например чтобы яснее показать, где оно начинается и заканчивается.

С помощью лямбда-выражений можно также представлять методы, принимающие несколько аргументов, как в случае ❹. Сейчас стоит поговорить о том, как *читать* такое лямбда-выражение. В этой строке мы не складываем два числа, а создаем функцию, складывающую два числа. Переменная с именем `add`, имеющая тип `BinaryOperator<Long>`, – это не результат сложения чисел, а код, который их складывает.

До сих пор типы параметров лямбда-выражений выводил за нас компилятор. Это замечательно, но иногда удобно иметь возможность задать тип явно. Поступая так, мы должны заключить аргументы лямбда-выражения в круглые скобки. Скобки необходимы и тогда, когда аргументов несколько. Это показано в случае ❺.

Целевым типом лямбда-выражения называется тип контекста, в котором это выражение встречается, – например, тип локальной переменной, которой оно присваивается, или тип параметра метода, вместо которого оно передается.

Во всех этих примерах неявно подразумевается, что тип лямбда-выражения зависит от контекста. Он выводится компилятором. Выведение целевого типа не является новшеством. Как видно из примера 2.4, типы инициализаторов массива всегда выводились в Java из

контекста. Другой знакомый пример – null. Какой тип имеет null, мы понимаем сразу, как видим его присваивание чему-либо.

Пример 2.4 ❖ Тип правой части не указан, он выводится из контекста

```
final String[] array = { "hello", "world" };
```

Использование значений

В прошлом, используя анонимные внутренние классы, вы, наверное, встречались с ситуацией, когда требуется использовать переменную, определенную в объемлющем методе. Для этого переменную нужно было объявлять с ключевым словом final, как показано в примере 2.5. Переменной, объявленной как final, нельзя присвоить другое значение. Это означает, что, используя final-переменную, мы точно знаем, что она сохранит именно то значение, которое ей было когда-то присвоено.

Пример 2.5 ❖ Локальная final-переменная, захваченная анонимным внутренним классом

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hi " + name);
    }
});
```

В Java 8 это ограничение немного ослаблено. Теперь разрешено ссылаться на переменные, в объявлении которых нет слова final; однако они должны быть *эффективно* финальными. Хотя включать final в объявление переменной не требуется, использовать переменные, на которые ссылается лямбда-выражение, как нефинальные, запрещено. Попытка сделать это приведет к ошибке компиляции.

Таким образом, эффективно финальной переменной значение можно присвоить только один раз. По-другому осознать это ограничение можно, поняв, что лямбда-выражение захватывает не переменные, а *значения*. В примере 2.6 name – эффективно финальная переменная.

Пример 2.6 ❖ Эффективно final-переменная, захваченная лямбда-выражением

```
String name = getUserName();
button.addActionListener(event -> System.out.println("hi " + name));
```

Лично мне подобный код читать легче, когда слово final опущено, потому что оно воспринимается как лишний шум. Разумеется, быва-