

O'REILLY®

«Хотите знать, как решать повседневные задачи в последних версиях Java? Тогда можете больше не искать. Кен Коузен продемонстрирует ряд типичных проблем и без лишних слов представит решение, которым все мы сможем воспользоваться.»
— Д-р Венкат Субраманиам, основатель компании Agile Developer, Inc.

«Великолепный способ приобщиться к последней версии Java быстро и эффективно. В этой книге каждый разработчик, стремящийся подняться на новый уровень, найдет что-то полезное для себя.»
— Триша Джу, Java Champion и Java Developer Advocate, компания Jet Brains

Включение средств функционального программирования в Java ознаменовало революционное изменение почтенного объектно-ориентированного языка. Лямбда-выражения, ссылки на методы и потоки принципиально изменили идиомы языка. С тех пор многие разработчики стараются не отстать от жизни. И в этом поможет настоящий сборник рецептов. На примере более 70 подробных рецептов Кен Коузен демонстрирует использование новых возможностей языка для решения широкого круга задач.

Разработчики, хорошо знакомые с предыдущими версиями Java, найдут здесь почти все нововведения, появившиеся в Java SE 8, а также отдельную главу, посвященную новшествам в Java 9. Хотите понять, как функциональные идиомы могут изменить подход к написанию кода? Тогда эта книга — буквально набитая конкретными примерами — для вас.

Краткое содержание книги:

- основы лямбда-выражений и ссылок на методы;
- интерфейсы в пакете java.util.function;
- потоковые операции для преобразования и фильтрации данных;
- компараторы и коллекторы для сортировки и преобразования потоковых данных в коллекции;
- создание экземпляров типа Optional и извлечение хранящихся в них значений;
- новые средства ввода-вывода, поддерживающие функциональные потоки;
- Date-Time API, пришедший на смену унаследованным классам Date и Calendar;
- механизмы для экспериментов с конкурентностью и параллелизмом.

Кен Коузен (Ken Kousen) — независимый консультант и преподаватель, специализирующийся на Android, Spring, Hibernate/JPA, Groovy, Grails и Gradle. Обладатель ряда технических сертификатов, а также ученых степеней в области математики, авиакосмической техники и информатики.

Интернет-магазин:
www.dmkpress.com
Книга — почтой:
orders@aliants-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел.(499)782-38-89
books@aliants-kniga.ru


ИЗДАТЕЛЬСТВО
www.dmk.ru

ISBN 978-5-97060-134-1



9 785970 601341 >

O'REILLY®



Современный Java

Рецепты программирования

Кен Коузен


ИЗДАТЕЛЬСТВО

Современный Java

Кен Коузен

Современный Java: рецепты программирования

Ken Kousen

Modern Java Recipes

**Simple Solutions
to Difficult Problems
in Java 8 and 9**

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Кен Коузен

Современный Java: рецепты программирования

Простые решения
трудных задач на Java 8 и 9



Москва, 2018

УДК 004.438Java
ББК 32.973.2-018.2
К78

Коузен К.

К78 Современный Java: рецепты программирования / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2018. – 274 с.: ил.

ISBN 978-5-97060-134-1

Эта книга представляет собой рецепты программирования на языке Java, описывающие нововведения версий 8 и 9. В книге вы найдете массу примеров кода, демонстрирующих почти все обсуждаемые языковые и библиотечные средства. Эти примеры намеренно сделаны как можно более простыми, чтобы сосредоточиться на основных отличительных моментах. Все они могут послужить вам неплохой отправной точкой для разработки собственного кода.

Издание будет полезно опытным программистам, уже работающим на Java.

УДК 004.438Java
ББК 32.973.2-018.2

Authorized Russian translation of the English edition of Modern Java Recipes ISBN 9781491973172 © 2017 Ken Kousen.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-491-97317-2 (анг.)
ISBN 978-5-97060-134-1 (рус.)

Copyright © 2017 Ken Kousen
© Оформление, издание, перевод, ДМК Пресс, 2018

Привет, Ксандер, это тебе посвящается. Сюрприз!

Содержание

Предисловие	9
Вступление	11
Глава 1. Основные понятия	18
1.1. Лямбда-выражения	19
1.2. Ссылки на методы	22
1.3. Ссылки на конструкторы	26
1.4. Функциональные интерфейсы	30
1.5. Методы по умолчанию в интерфейсах	33
1.6. Статические методы в интерфейсах	36
Глава 2. Пакет java.util.function	39
2.1. Потребители	39
2.2. Поставщики	42
2.3. Предикаты	44
2.4. Функции	48
Глава 3. Потoki	51
3.1. Создание потоков	51
3.2. Оборнутые потоки	55
3.3. Операции редукции	57
3.4. Проверка правильности сортировки с помощью редукции	65
3.5. Отладка потоков с помощью reek	66
3.6. Преобразование строк в потоки и наоборот	69
3.7. Подсчет элементов	72
3.8. Сводные статистики	73
3.9. Нахождение первого элемента в потоке	76
3.10. Методы anyMatch, allMatch и noneMatch	81
3.11. Методы flatMap и map	83
3.12. Конкатенация потоков	85
3.13. Ленивые потоки	89

Глава 4. Компараторы и коллекторы	91
4.1. Сортировка с помощью компаратора.....	91
4.2. Преобразование потока в коллекцию	95
4.3. Добавление линейной коллекции в отображение	97
4.4. Сортировка отображений.....	99
4.5. Разбиение и группировка.....	102
4.6. Подчиненные коллекторы.....	104
4.7. Нахождение минимального и максимального значений.....	106
4.8. Создание неизменяемых коллекций.....	109
4.9. Реализация интерфейса Collector	111
Глава 5. Применение потоков, лямбда-выражений и ссылок на методы	115
5.1. Класс java.util.Objects	115
5.2. Лямбда-выражения и эффективная финальность.....	117
5.3. Потоки случайных чисел	120
5.4. Методы по умолчанию интерфейса Map.....	122
5.5. Конфликт между методами по умолчанию.....	126
5.6. Обход коллекций и отображений	128
5.7. Протоколирование с помощью Supplier	130
5.8. Композиция замыканий.....	132
5.9. Применение вынесенного метода для обработки исключений	135
5.10. Контролируемые исключения и лямбда-выражения	138
5.11. Использование универсальной обертки исключений.....	141
Глава 6. Тип Optional	143
6.1. Создание Optional	143
6.2. Извлечение значения из Optional.....	146
6.3. Optional в методах чтения и установки.....	149
6.4. Методы flatMap и map класса Optional	150
6.5. Отображение объектов Optional	154
Глава 7. Файловый ввод-вывод	157
7.1. Обработка файлов	158
7.2. Получение файлов в виде потока.....	160
7.3. Обход файловой системы	161
7.4. Поиск в файловой системе	163

Глава 8. Пакет java.time	165
8.1. Основные классы для работы с датами и временем	166
8.2. Создание даты и времени на основе существующих экземпляров	169
8.3. Корректоры и запросы	173
8.4. Преобразование java.util.Date в java.time.LocalDate	178
8.5. Разбор и форматирование	182
8.6. Нахождение часовых поясов с необычным смещением	185
8.7. Нахождение названий регионов по смещению	187
8.8. Время между событиями	189
Глава 9. Параллелизм и конкурентность	192
9.1. Преобразование последовательного потока в параллельный	193
9.2. Когда распараллеливание помогает	196
9.3. Изменение размера пула	201
9.4. Интерфейс Future	203
9.5. Завершение CompletableFuture	206
9.6. Координация нескольких CompletableFuture, часть 1	210
9.7. Координация нескольких CompletableFuture, часть 2	215
Глава 10. Нововведения в Java 9	222
10.1. Модули в проекте Jigsaw	223
10.2. Закрытые методы в интерфейсах	227
10.3. Создание неизменяемых коллекций	229
10.4. Интерфейс Stream: ofNullable, iterate, takeWhile и dropWhile	233
10.5. Подчиненные коллекторы: filtering и flatMapping	236
10.6. Класс Optional: методы stream, or, ifPresentOrElse	240
10.7. Диапазоны дат	243
Приложение А. Универсальные типы и Java 8	246
Общие сведения	246
Что знает каждый	246
Чего некоторые разработчики не осознают	249
Метатипы и PECS	250
Примеры из Java 8 API	255
Резюме	262
Предметный указатель	263
Об авторе	272
Об иллюстрации на обложке	273

Предисловие

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

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

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

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

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

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

Вступление

СОВРЕМЕННЫЙ JAVA

Иногда трудно поверить, что язык, имеющий 20-летнюю историю обратной совместимости, мог так радикально измениться. До выхода версии Java SE 8 в марте 2014-го¹ Java, несмотря на все успехи в качестве языка программирования серверов, пользовался репутацией «COBOL'a XXI века». Стабильный, вездесущий, ориентированный на производительность. Изменения происходили медленно, если вообще происходили. Компании не особенно торопились переходить на новые версии, когда они наконец появлялись.

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

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

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

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

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

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

Кому стоит прочитать эту книгу?

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

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

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

О СТРУКТУРЕ КНИГИ

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

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

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

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

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

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

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

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

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

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

В главе 10 «Нововведения в Java 9» рассматриваются многие изменения, включенные в версию Java 9, выход которой был намечен на 21 сентября 2017 г. Детали проекта `Jigsaw` заслуживают отдельной книги¹, но основные элементы понятны и описаны в этой главе. Здесь же рассмотрены закрытые методы в интерфейсах, новые методы потоков, коллекторов и типа `Optional`, а также вопрос о создании потока дат².

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

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

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

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

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ И СОГЛАШЕНИЯ, ПРИНЯТЫЕ В КНИГЕ

В книге используются следующие типографские соглашения:

Курсив

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

Моноширинный шрифт

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

Моноширинный полужирный шрифт

Используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно без изменений.

Моноширинный курсив

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



Такая пиктограмма обозначает совет или рекомендацию.



Такая пиктограмма обозначает указание или примечание общего характера.



Эта пиктограмма обозначает предупреждение или предостережение.

ОТЗЫВЫ И ПОЖЕЛАНИЯ

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

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

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

СКАЧИВАНИЕ ИСХОДНОГО КОДА ПРИМЕРОВ

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

СПИСОК ОПЕЧАТОК

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

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

НАРУШЕНИЕ АВТОРСКИХ ПРАВ

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

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

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

БЛАГОДАРНОСТИ

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

С тех пор я вот уже два года провожу презентации по новым функциональным возможностям Java. Осенью 2016-го я закончил свою последнюю книгу¹ и, поскольку собирался писать еще один сборник рецептов для того же самого издательства, сгоряча решил, что проект будет легким.

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

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

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

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

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

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

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

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

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

Глава 1

Основные понятия

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

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

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

Все дело в том, что мир разработки программного обеспечения меняется, и если язык хочет остаться успешным, то должен адаптироваться к новому. В середине 1990-х годов, когда Java еще сверкала новизной, действовал закон Мура¹. Надо было подождать каких-то два года, чтобы быстродействие вашего компьютера возросло в два раза.

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

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

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

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

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

1.1. ЛЯМБДА-ВЫРАЖЕНИЯ

Проблема

Вы хотите использовать в своем коде лямбда-выражения.

Решение

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

Обсуждение

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

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

Пример 1.1 ❖ Анонимный внутренний класс, реализующий интерфейс `Runnable`

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

```

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

```

❶ Анонимный внутренний класс

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

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

Пример 1.2 ❖ Использование лямбда-выражения в конструкторе `Thread`

```

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

```

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

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

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

Пример 1.3 ❖ Присваивание лямбда-выражения переменной

```

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

```

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

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

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

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

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

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

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

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

Пример 1.4 ❖ Реализация `FilenameFilter` с помощью анонимного внутреннего класса

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

❶ Анонимный внутренний класс

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

В примере 1.5 приведена версия с лямбда-выражением.

Пример 1.5 ❖ Лямбда-выражение, реализующее интерфейс `FilenameFilter`

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

❶ Лямбда-выражение

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

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

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

Пример 1.6 ❖ Лямбда-выражение с явно заданными типами данных

```
File directory = new File("./src/main/java");
String[] names = directory.list((File dir, String name) -> {
    name.endsWith(".java");
});
```

❶ Явные типы данных

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

Пример 1.7 ❖ Блочное лямбда-выражение

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

❶ Блочный синтаксис

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

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

1.2. Ссылки НА МЕТОДЫ

Проблема

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

Решение

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

Обсуждение

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

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

Пример 1.8 ❖ Использование ссылки на метод для доступа к `println`

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

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

Нотация с двойным двоеточием дает ссылку на метод `println` объекта `System.out`, т. е. экземпляра типа `PrintStream`. В конце ссылки на метод скобки не ставятся. В примере выше все элементы потока выводятся в стандартный вывод¹.

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

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

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

Пример 1.9 ❖ Ссылка на статический метод

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

- ❶ Статический метод
- ❷ Метод экземпляра

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

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

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

Синтаксис

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

`object::instanceMethod`

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

`Class::staticMethod`

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

`Class::instanceMethod`

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

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

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

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

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

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

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

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

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

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

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

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

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

- ❶ Ссылка на метод и эквивалентное лямбда-выражение

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

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

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

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

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

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

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

Пример 1.12 ❖ Лямбда-выражения, эквивалентные ссылкам на методы

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

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

См. также

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

1.3. ССЫЛКИ НА КОНСТРУКТОРЫ

Проблема

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

Решение

Использовать в ссылке на метод ключевое слово `new`.

Обсуждение

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

Пример 1.13 ❖ Преобразование списка людей в список имен

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

// или

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

- ❶ Лямбда-выражение
- ❷ Ссылка на метод

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

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

Пример 1.14 ❖ Класс `Person`

```
public class Person {
    private String name;

    public Person() {}

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

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

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

Пример 1.15 ❖ Преобразование строк в экземпляры класса `Person`

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

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

// или

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

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

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

Копирующий конструктор

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

Пример 1.16 ❖ Копирующий конструктор класса `Person`

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

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

Пример 1.17 ❖ Преобразование списка в поток и обратно

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

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

assertTrue(before == after);           ❶

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

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

Копирующий конструктор позволяет разорвать эту связь.

Пример 1.18 ❖ Применение копирующего конструктора

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

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

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

- ❶ Используется копирующий конструктор
- ❷ Объекты разные
- ❸ Но эквивалентные

Теперь при вызове метода `map` контекстом является поток объектов `Person`. Поэтому `Person::new` вызывает конструктор, который принимает `Person` и возвращает новый, но эквивалентный экземпляр. Тем самым связь между ссылками *before* и *after* разрывается¹.

Конструктор с переменным числом аргументов

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

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

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

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

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

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

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

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

Пример 1.20. ❖ Использование конструктора с переменным числом аргументов

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

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

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

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

то получится такой результат:

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

Массивы

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

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

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

Пример 1.21 ❖ Создание массива объектов `Person`

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

- ❶ Ссылка на конструктор `Person`
- ❷ Ссылка на конструктор массива `Person`

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

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

См. также

Ссылки на методы обсуждаются в рецепте 1.2.

1.4. ФУНКЦИОНАЛЬНЫЕ ИНТЕРФЕЙСЫ

Проблема

Требуется использовать уже имеющийся функциональный интерфейс или написать свой собственный.

Решение

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

Обсуждение

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

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

Пример 1.22 ❖ Интерфейс `PalindromeChecker`

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

Все методы интерфейса являются открытыми¹, поэтому модификатор доступа можно опускать, точно так же как ключевое слово `abstract`.

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

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

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

Functional Interface:

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

Функциональный интерфейс:

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

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

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

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

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

    static void myStaticMethod() {
```

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


```

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

```

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

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

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

Пример 1.24 ❖ После расширения функциональный интерфейс перестает быть функциональным

```

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

```

- ❶ Дополнительный абстрактный метод

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

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

Method Summary				
All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method and Description			
int	<code>compare(T o1, T o2)</code> Compares its two arguments for order.			
boolean	<code>equals(Object obj)</code> Indicates whether some other object is "equal to" this comparator.			

Рис. 1.1 ❖ Абстрактные методы интерфейса `Comparator`

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

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

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

См. также

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

1.5. МЕТОДЫ ПО УМОЛЧАНИЮ В ИНТЕРФЕЙСАХ

Проблема

Требуется предоставить реализацию метода в самом интерфейсе.

Решение

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

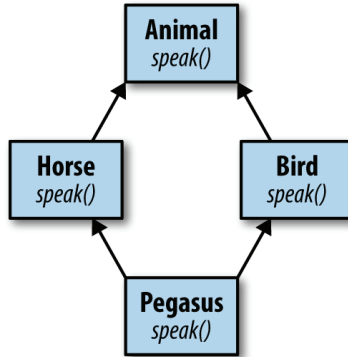
Обсуждение

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

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

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

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

Рис. 1.2 ❖ Иерархия наследования класса `Animal`

В разных языках к этой проблеме подходят по-разному. Так, в C++ множественное наследование разрешено, но если класс наследует конфликтующие реализации, то программа не откомпилируется¹. В Eiffel² компилятор позволяет выбрать нужную реализацию.

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

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

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

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

Пример 1.25 ❖ Реализация `isEmpty` в терминах `size`

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

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

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

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

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

Пример 1.26 ❖ Интерфейс `Employee` с методом по умолчанию

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

    String getLast();

    void convertCaffeineToCodeForMoney();

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

❶ Метод по умолчанию, имеющий реализацию

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

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

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

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

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

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

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

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

Пример 1.27 ❖ Использование методов по умолчанию

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

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

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

См. также

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

1.6. СТАТИЧЕСКИЕ МЕТОДЫ В ИНТЕРФЕЙСАХ

Проблема

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

Решение

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

Обсуждение

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

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

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

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

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

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

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

Пример 1.28 ❖ Сортировка строк

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

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

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

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

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

sorted = bonds.stream()
```

```

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

```

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

В этом примере продемонстрировано применение нескольких статических методов интерфейса `Comparator` для сортировки списка актеров, в разные годы игравших роль Джеймса Бонда¹. Мы еще будем обсуждать компараторы в рецепте 4.1.

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

Запомните следующие положения:

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

См. также

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

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

Глава 2

Пакет `java.util.function`

В предыдущей главе мы обсуждали базовый синтаксис лямбда-выражений и ссылок на методы. Очень важно, что в обоих случаях обязательно существует контекст. Лямбда-выражения и ссылки на методы всегда присваиваются переменным типа функционального интерфейса, так что у компилятора есть информация о том, какой единственный абстрактный метод реализуется.

Хотя в стандартной библиотеке Java имеется много интерфейсов, содержащих единственный абстрактный метод и потому являющихся функциональными, в версии Java 8 появился новый пакет, в котором находятся только функциональные интерфейсы, используемые в разных частях библиотеки. Этот пакет называется `java.util.function`.

Интерфейсы, включенные в пакет `java.util.function`, можно разбить на четыре категории: (1) потребители, (2) поставщики, (3) предикаты и (4) функции. Потребитель принимает аргумент универсального типа и ничего не возвращает. Поставщик не принимает аргументов и возвращает значение. Предикат принимает аргумент и возвращает значение типа `boolean`. Функция принимает один аргумент и возвращает значение.

С каждым из основных интерфейсов связано несколько вариантов. Например, у интерфейса `Consumer` имеются варианты для примитивных типов (`IntConsumer`, `LongConsumer` и `DoubleConsumer`), а также вариант (`BiConsumer`), который принимает два аргумента и возвращает `void`.

Хотя все интерфейсы, рассматриваемые в этой главе, по определению, содержат единственный абстрактный метод, в большинстве из них есть дополнительные методы, статические или по умолчанию. Знакомство с этими методами облегчает работу программиста.

2.1. ПОТРЕБИТЕЛИ

Проблема

Требуется написать лямбда-выражение, реализующее интерфейс `java.util.function.Consumer`.

Решение

Реализовать метод `void accept(T t)`, применяя лямбда-выражение или ссылку на метод.

Обсуждение

Интерфейс `java.util.function.Consumer` содержит единственный абстрактный метод `void accept(T t)`. См. пример 2.1.

Пример 2.1 ❖ Методы интерфейса `java.util.function.Consumer`

```
void accept(T t) ❶
default Consumer<T> andThen(Consumer<? super T> after) ❷
```

- ❶ Единственный абстрактный метод
- ❷ Метод по умолчанию для композиции

Метод `accept` принимает аргумент универсального типа и возвращает `void`. Один из самых распространенных примеров метода, принимающего `Consumer` в качестве аргумента, – метод по умолчанию `forEach` интерфейса `java.util.Iterable`.

Пример 2.2 ❖ Метод `forEach` интерфейса `Iterable`

```
default void forEach(Consumer<? super T> action) ❶
```

- ❶ Передает каждый элемент итерируемой коллекции потребителю, заданному аргументом `action`

Этот интерфейс реализуют все линейные коллекции, что дает им возможность применять указанное действие к каждому элементу, как в примере 2.3.

Пример 2.3 ❖ Печать элементов коллекции

```
List<String> strings = Arrays.asList("this", "is", "a", "list", "of", "strings");
```

```
strings.forEach(new Consumer<String>() { ❶
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
});
```

```
strings.forEach(s -> System.out.println(s)); ❷
strings.forEach(System.out::println); ❸
```

- ❶ Реализация с анонимным внутренним классом
- ❷ Лямбда-выражение
- ❸ Ссылка на метод

Лямбда-выражение соответствует сигнатуре метода `accept`, поскольку оно принимает один аргумент и ничего не возвращает. Метод `println` класса `PrintStream`, к которому мы обращаемся через `System.out`, совместим с `Consumer`. Следовательно, то и другое можно использовать там, где ожидается аргумент типа `Consumer`.

В пакете `java.util.function` имеются также варианты `Consumer<T>` для примитивных типов и вариант с двумя аргументами. Детали приведены в табл. 2.1.

Таблица 2.1. Дополнительные интерфейсы семейства `Consumer`

Интерфейс	Единственный абстрактный метод
<code>IntConsumer</code>	<code>void accept(int x)</code>
<code>DoubleConsumer</code>	<code>void accept(double x)</code>
<code>LongConsumer</code>	<code>void accept(long x)</code>
<code>BiConsumer</code>	<code>void accept(T t, U u)</code>

☑ Ожидается, что потребители выполняют то, что от них требуется, с помощью побочных эффектов, как показано в рецепте 2.3.

В интерфейсе `BiConsumer` определен метод `accept`, который принимает два аргумента универсальных типов, обычно разных. В пакете есть три варианта интерфейса `BiConsumer` с различными примитивными типами второго аргумента. В интерфейсе `ObjIntConsumer` метод `accept` принимает два аргумента: универсального типа и типа `int`. Интерфейсы `ObjLongConsumer` и `ObjDoubleConsumer` аналогичны.

Перечислим другие примеры использования интерфейса `Consumer` в стандартной библиотеке:

```
Optional.ifPresent(Consumer<? super T> consumer)
```

Если значение присутствует, вызвать указанного потребителя. В противном случае не делать ничего.

```
Stream.forEach(Consumer<? super T> action)
```

Выполняет действие для каждого элемента потока¹. Метод `Stream.forEachOrdered` похож, но обрабатывает элементы в порядке следования².

```
Stream.peek(Consumer<? super T> action)
```

Возвращает поток, содержащий те же элементы, что существующий, но сначала выполняет указанное действие. Это очень полезно при отладке (см. пример в рецепте 3.5).

См. также

Метод `andThen` интерфейса `Consumer` применяется для композиции. Композиция функций рассматривается в рецепте 5.8. Метод `peek` интерфейса `Stream` рассматривается в рецепте 3.5.

¹ Это настолько частая операция, что метод `forEach` был добавлен также непосредственно в интерфейс `Iterable`. Вариант в `Stream` полезен, когда источник не является коллекцией или когда нужно сделать поток параллельным.

² Это отличие становится существенным в случае параллельного потока. – *Прим. перев.*

2.2. ПОСТАВЩИКИ

Проблема

Требуется реализовать интерфейс `java.util.function.Supplier`.

Решение

Реализовать метод `T get()` интерфейса `java.util.function.Supplier` с помощью лямбда-выражения или ссылки на метод.

Обсуждение

Интерфейс `java.util.function.Supplier` очень прост. В нем нет ни статических методов, ни методов по умолчанию, а есть только один абстрактный метод `T get()`.

Чтобы реализовать `Supplier`, нужно предоставить метод, который не принимает аргументов и возвращает значение универсального типа. В документации по Java оговорено, что необязательно при каждом вызове `get()` возвращать новое или отличающееся от предыдущего значение.

Простой пример реализации `Supplier` дает метод `Math.random`, который не принимает аргументов и возвращает `double`. Поэтому его можно присвоить ссылке типа `Supplier` и вызывать в любое время, как показано в примере 2.4.

Пример 2.4 ❖ Использование `Math.random()` в качестве `Supplier`

```
Logger logger = Logger.getLogger("...");

DoubleSupplier randomSupplier = new DoubleSupplier() { ❶
    @Override
    public double getAsDouble() {
        return Math.random();
    }
};

randomSupplier = () -> Math.random(); ❷
randomSupplier = Math::random; ❸
logger.info(randomSupplier);
```

- ❶ Реализация с помощью анонимного внутреннего класса
- ❷ Лямбда-выражение
- ❸ Ссылка на метод

Единственный абстрактный метод в интерфейсе `DoubleSupplier` – это `getAsDouble`, который возвращает значение типа `double`. Другие родственные `Supplier` интерфейсы в пакете `java.util.function` перечислены в табл. 2.2.

Таблица 2.2. Дополнительные интерфейсы семейства `Supplier`

Интерфейс	Единственный абстрактный метод
<code>IntSupplier</code>	<code>int getAsInt()</code>
<code>DoubleSupplier</code>	<code>double getAsDouble()</code>
<code>LongSupplier</code>	<code>getAsLong()</code>
<code>BooleanSupplier</code>	<code>boolean getAsBoolean()</code>

Одно из основных применений интерфейсов семейства `Supplier` – поддержка *отложенного выполнения*. Метод `info` класса `java.util.logging.Logger` принимает поставщика `Supplier`, метод `get` которого вызывается, только если уровень протоколирования таков, что сообщение должно быть выведено (детали см. в рецепте 5.7). Идею отложенного выполнения вы можете применить и в своем коде, чтобы запрашивать у поставщика значение только тогда, когда это необходимо.

Другой пример из стандартной библиотеки – метод `orElseGet` класса `Optional`, который тоже принимает `Supplier`. Класс `Optional` обсуждается в главе 6, а пока достаточно знать, что объект типа `Optional` либо пуст, либо обертывает некоторое значение. Обычно такие объекты возвращаются, когда есть основания ожидать, что результата не будет, например при поиске значения в пустой коллекции.

В примере 2.5 показано, как это работает при поиске имени в коллекции.

Пример 2.5 ❖ Поиск имени в коллекции

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "Inara",
    "Zoë", "Jayne", "Simon", "River", "Shepherd Book");

Optional<String> first = names.stream()
    .filter(name -> name.startsWith("C"))
    .findFirst();

System.out.println(first);           ❶
System.out.println(first.orElse("None")); ❷

System.out.println(first.orElse(String.format("Ничего не найдено в %s",
    names.stream().collect(Collectors.joining(", "))))); ❸

System.out.println(first.orElseGet(() ->
    String.format("Ничего не найдено в %s",
    names.stream().collect(Collectors.joining(", "))))); ❹
```

- ❶ Печатается `Optional.empty`
- ❷ Печатается строка "None"
- ❸ Коллекция с запятой-разделителем строится, даже если имя найдено
- ❹ Коллекция с запятой-разделителем строится, только когда `Optional` пусто

Метод `findFirst` интерфейса `Stream` возвращает первый элемент, встретившийся в упорядоченном потоке¹. Поскольку к потоку может быть применен фильтр, не оставляющий в нем никаких элементов, то метод возвращает объект типа `Optional`, который может либо содержать искомый элемент, либо быть пустым. В данном случае ни одно имя не проходит фильтра, поэтому в результате получается пустой `Optional`.

¹ Поток может иметь порядок следования или не иметь никакого порядка – точно так же, как списки упорядочены по индексу, а множества – нет. Этот порядок может отличаться от порядка, в котором элементы обрабатываются. Дополнительные сведения см. в рецепте 3.9.

Метод `orElse` объекта `Optional` возвращает либо обернутый им элемент, либо заданное значение по умолчанию. Это хорошо, если значением по умолчанию является пустая строка, но расточительно, если алгоритм обязательно возвращает какое-то значение.

В данном случае возвращенное значение содержит полный список имен через запятую. Метод `orElse` создает строку вне зависимости от того, содержит `Optional` значение или нет.

С другой стороны, метод `orElseGet` принимает `Supplier` в качестве аргумента. Преимущество заключается в том, что метод `get` интерфейса `Supplier` вызывается, только если `Optional` пуст, поэтому строка не строится, если в этом нет необходимости.

В стандартной библиотеке есть и другие места, где используется интерфейс `Supplier`:

- метод `orElseThrow` класса `Optional`, который принимает аргумент типа `Supplier<X extends Exception>`. `Supplier` вызывается, только если имеет место исключение;
- метод `Objects.requireNonNull(String obj, Supplier<String> messageSupplier)` форматирует ответ, только если первый аргумент не равен `null`;
- метод `CompletableFuture.supplyAsync(Supplier<U> supplier)` возвращает объект типа `CompletableFuture`, который исполняемая задача асинхронно завершает значением, полученным от заданного поставщика `Supplier`;
- в классе `Logger` все методы имеют перегруженные варианты, принимающие не просто строку, а `Supplier<String>` (см. пример в рецепте 5.7).

См. также

Перегруженные методы, принимающие `Supplier`, рассматриваются в рецепте 5.7. Нахождение первого элемента коллекции обсуждается в рецепте 3.9. Завершаемые будущие объекты рассматриваются в нескольких рецептах из главы 9, а класс `Optional` – тема рецептов из главы 6.

2.3. ПРЕДИКАТЫ

Проблема

Требуется отфильтровать данные с помощью интерфейса `java.util.function.Predicate`.

Решение

Реализовать метод `boolean test(T t)` интерфейса `Predicate` с помощью лямбда-выражения или ссылки на метод.

Обсуждение

Предикаты используются главным образом для фильтрации потоков. Метод `filter` интерфейса `java.util.stream.Stream` принимает аргумент типа `Predicate`

и возвращает новый поток, который включает только те элементы входного потока, которые удовлетворяют предикату.

Единственный абстрактный метод интерфейса `Predicate`, `boolean test(T t)`, принимает один аргумент универсального типа и возвращает `true` или `false`. В примере 2.6 перечислены все методы `Predicate`, включая статические и по умолчанию.

Пример 2.6 ❖ Методы интерфейса `java.util.function.Predicate`

```
default Predicate<T> and(Predicate<? super T> other)
static <T> Predicate<T> isEqual(Object targetRef)
default Predicate<T> negate()
default Predicate<T> or(Predicate<? super T> other)
boolean test(T t) ❶
```

❶ Единственный абстрактный метод

Пусть имеется коллекция имен и требуется найти в ней все имена определенной длины. В примере 2.7 показано, как это сделать средствами обработки потоков.

Пример 2.7 ❖ Нахождение строк заданной длины

```
public String getNamesOfLength(int length, String... names) {
    return Arrays.stream(names)
        .filter(s -> s.length() == length) ❶
        .collect(Collectors.joining(", "));
}
```

❶ Предикат для поиска строк заданной длины

А в примере 2.8 показано, как найти имена, начинающиеся заданной строкой.

Пример 2.8 ❖ Нахождение строк с заданным префиксом

```
public String getNamesStartingWith(String s, String... names) {
    return Arrays.stream(names)
        .filter(s -> s.startsWith(s)) ❶
        .collect(Collectors.joining(", "));
}
```

❶ Предикат для поиска строк с заданным префиксом

Этот пример можно обобщить, разрешив клиенту задавать условие.

Пример 2.9 ❖ Нахождение строк, удовлетворяющих произвольному предикату

```
public class ImplementPredicate {
    public String getNamesSatisfyingCondition(
        Predicate<String> condition, String... names) {
        return Arrays.stream(names)
            .filter(condition) ❶
            .collect(Collectors.joining(", "));
    }
}
```

```

}
// ... прочие методы ...
}

```

❶ Фильтр по заданному предикату

Решение весьма гибкое, но вряд ли стоит ожидать, что клиенты будут задавать каждый предикат самостоятельно. Можно вместо этого включить в класс константы, представляющие типичные случаи, как показано в примере 2.10.

Пример 2.10 ❖ Константы для типичных случаев

```

public class ImplementPredicate {
    public static final Predicate<String> LENGTH_FIVE = s -> s.length() == 5;
    public static final Predicate<String> STARTS_WITH_S =
        s -> s.startsWith("S");
// ... остальное, как и раньше ...
}

```

Еще одно преимущество передачи предиката в качестве аргумента состоит в том, что можно использовать методы по умолчанию `and`, `or` и `negate` для построения предиката из составных частей.

Все эти приемы продемонстрированы в примере 2.11.

Пример 2.11 ❖ Тесты JUnit для методов предиката

```

import static functionpackage.ImplementPredicate.*; ❶
import static org.junit.Assert.assertEquals;

// ... другие предложения импорта ...

public class ImplementPredicateTest {
    private ImplementPredicate demo = new ImplementPredicate();
    private String[] names;

    @Before
    public void setUp() {
        names = Stream.of("Mal", "Wash", "Kaylee", "Inara", "Zoë",
            "Jayne", "Simon", "River", "Shepherd Book")
            .sorted()
            .toArray(String[]::new);
    }

    @Test
    public void getNamesOfLength5() throws Exception {
        assertEquals("Inara, Jayne, River, Simon",
            demo.getNamesOfLength(5, names));
    }

    @Test
    public void getNamesStartingWithS() throws Exception {
        assertEquals("Shepherd Book, Simon",
            demo.getNamesStartingWith("S", names));
    }
}

```

```

@Test
public void getNamesSatisfyingCondition() throws Exception {
    assertEquals("Inara, Jayne, River, Simon",
        demo.getNamesSatisfyingCondition(s -> s.length() == 5, names));
    assertEquals("Shepherd Book, Simon",
        demo.getNamesSatisfyingCondition(s -> s.startsWith("S"),
            names));
    assertEquals("Inara, Jayne, River, Simon",
        demo.getNamesSatisfyingCondition(LENGTH_FIVE, names));
    assertEquals("Shepherd Book, Simon",
        demo.getNamesSatisfyingCondition(STARTS_WITH_S, names));
}

@Test
public void composedPredicate() throws Exception {
    assertEquals("Simon",
        demo.getNamesSatisfyingCondition(
            LENGTH_FIVE.and(STARTS_WITH_S), names)); ❶
    assertEquals("Inara, Jayne, River, Shepherd Book, Simon",
        demo.getNamesSatisfyingCondition(
            LENGTH_FIVE.or(STARTS_WITH_S), names)); ❷
    assertEquals("Kaylee, Mal, Shepherd Book, Wash, Zoë",
        demo.getNamesSatisfyingCondition(LENGTH_FIVE.negate(), names)); ❸
}
}

```

- ❶ Статический импорт, чтобы было проще использовать константы
- ❷ Композиция
- ❸ Отрицание

Вот еще несколько методов из стандартной библиотеки, в которых используются предикаты.

`Optional.filter(Predicate<? super T> predicate)`

Если значение имеется и удовлетворяет заданному предикату, то возвращается объект `Optional`, обертывающий это значение, иначе пустой `Optional`.

`Collection.removeIf(Predicate<? super E> filter)`

Удаляет из коллекции все элементы, удовлетворяющие предикату.

`Stream.allMatch(Predicate<? super T> predicate)`

Возвращает `true`, если все элементы потока удовлетворяют заданному предикату. Методы `anyMatch` и `noneMatch` аналогичны.

`Collectors.partitioningBy(Predicate<? super T> predicate)`

Возвращает объект `Collector`, который разбивает поток на два: элементы, удовлетворяющие предикату, и все остальные.

Предикаты полезны во всех случаях, когда из потока следует выбрать только некоторые элементы. Надеюсь, этот рецепт поможет вам понять, когда и как их стоит использовать.

См. также

В рецепте 5.8 обсуждается композиция замыканий. Методы `allMatch`, `anyMatch` и `noneMatch` рассматриваются в рецепте 3.10. Операции разбиения и группировки обсуждаются в рецепте 4.5.

2.4. ФУНКЦИИ

Проблема

Требуется реализовать интерфейс `java.util.function.Function` для преобразования входного параметра в выходное значение.

Решение

Написать лямбда-выражение, реализующее метод `R apply(T t)`.

Обсуждение

Функциональный интерфейс `java.util.function.Function` содержит единственный абстрактный метод `apply`, который вызывается, чтобы преобразовать входной параметр универсального типа `T` в выходное значение универсального типа `R`. В примере 2.12 перечислены методы интерфейса `Function`.

Пример 2.12 ❖ Методы интерфейса `java.util.function.Function`

```
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
    R apply(T t)
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
static <T> Function<T,T> identity()
```

Чаще всего `Function` используется в качестве аргумента метода `Stream.map`. Например, один из способов сопоставить строке целое число состоит в том, чтобы вызвать метод `length` объекта `String`.

Пример 2.13 ❖ Отображение строки на ее длину

```
List<String> names = Arrays.asList("Mal", "Wash", "Kaylee", "Inara",
    "Zoë", "Jayne", "Simon", "River", "Shepherd Book");
```

```
List<Integer> nameLengths = names.stream()
    .map(new Function<String, Integer>() { ❶
        @Override
        public Integer apply(String s) {
            return s.length();
        }
    })
    .collect(Collectors.toList());
```

```
nameLengths = names.stream()
    .map(s -> s.length()) ❷
    .collect(Collectors.toList());
```

```

nameLengths = names.stream()
    .map(String::length) ❸
    .collect(Collectors.toList());

System.out.printf("nameLengths = %s%n", nameLengths);
// nameLengths == [3, 4, 6, 5, 3, 5, 5, 5, 13]

```

- ❶ Анонимный внутренний класс
- ❷ Лямбда-выражение
- ❸ Ссылка на метод

В табл. 2.3 приведен полный перечень частных случаев этого интерфейса для примитивных типов.

Таблица 2.3. Дополнительные интерфейсы семейства *Function*

Интерфейс	Единственный абстрактный метод
IntFunction	R apply(int value)
DoubleFunction	R apply(double value)
LongFunction	R apply(long value)
ToIntFunction	int applyAsInt(T value)
ToDoubleFunction	double applyAsDouble(T value)
ToLongFunction	long applyAsLong(T value)
DoubleToIntFunction	int applyAsInt(double value)
DoubleToLongFunction	long applyAsLong(double value)
IntToDoubleFunction	double applyAsDouble(int value)
IntToLongFunction	long applyAsLong(int value)
LongToDoubleFunction	double applyAsDouble(long value)
LongToIntFunction	int applyAsInt(long value)
BiFunction	R apply(T t, U u)

Аргументом метода `map` в примере 2.13 могло бы быть значение типа `ToIntFunction`, потому что метод возвращает значение примитивного типа `int`. Метод `Stream.mapToInt` принимает в качестве аргумента `ToIntFunction`, методы `mapToDouble` и `mapToLong` аналогичны. Методы `mapToInt`, `mapToDouble` и `mapToLong` возвращают значения типа `IntStream`, `DoubleStream` и `LongStream` соответственно.

Что, если типы аргумента и возвращаемого значения одинаковы? На такой случай в пакете `java.util.function` определен интерфейс `UnaryOperator`. Как и следовало ожидать, существуют также интерфейсы `IntUnaryOperator`, `DoubleUnaryOperator` и `LongUnaryOperator`, для которых входные и выходные аргументы имеют соответственно типы `int`, `double` и `long`. В качестве примера применения `UnaryOperator` приведем метод `reverse` класса `StringBuilder`, где входное и выходное значения – строки.

В определении интерфейса `BiFunction` предполагается, что два входных аргумента и возвращаемое значение имеют универсальные типы, в общем случае различные. На случай, когда все три типа одинаковы, в пакет включен интерфейс `BinaryOperator`. Примером бинарного оператора является метод `Math.max`,

поскольку оба его входа и выход могут иметь типы `int`, `double`, `float` или `long`. Разумеется, определены также интерфейсы `IntBinaryOperator`, `DoubleBinaryOperator` и `LongBinaryOperator`¹.

И в завершение упомянем варианты `BiFunction` для примитивных типов, все они перечислены в табл. 2.4.

Таблица 2.4. Дополнительные интерфейсы семейства *Function*

Интерфейс	Единственный абстрактный метод
<code>ToIntBiFunction</code>	<code>int applyAsInt(T t, U u)</code>
<code>ToDoubleBiFunction</code>	<code>double applyAsDouble(T t, U u)</code>
<code>ToLongBiFunction</code>	<code>long applyAsLong(T t, U u)</code>

Основной сферой применения интерфейса `Function` являются методы `Stream.map`, но встречается он и в других контекстах, например:

```
Map.computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
```

Если с заданным ключом не ассоциировано значение, то оно вычисляется с помощью заданной функции `Function` и помещается в отображение `Map`.

```
Comparator.comparing(Function<? super T,? extends U> keyExtractor)
```

Этот метод, обсуждаемый в рецепте 4.1, генерирует компаратор, который сортирует коллекцию по ключу, вычисляемому заданной функцией.

```
Comparator.thenComparing(Function<? super T,? extends U> keyExtractor)
```

Метод экземпляра, также используемый при сортировке. Он вызывается, если после первой сортировки в коллекции оказались одинаковые ключи.

Функции также активно используются в служебном классе `Collectors` для группирующих и подчиненных коллекторов.

Методы `andThen` и `compose` обсуждаются в рецепте 5.8. Метод `identity` – это просто лямбда-выражение `e -> e`. Пример его применения приведен в рецепте 4.3.

См. также

Примеры применения методов `andThen` и `compose` интерфейса `Function` см. в рецепте 5.8. Пример метода `Function.identity` см. в рецепте 4.3. Примеры использования функций в качестве подчиненных коллекторов см. в рецепте 4.6. Метод `computeIfAbsent` обсуждается в рецепте 5.4. Бинарные операторы рассматриваются также в рецепте 3.3.

¹ Дополнительные сведения об использовании `BinaryOperator` в стандартной библиотеке см. в рецепте 3.3.

Глава 3

Потоки

В Java 8 введена новая метафора потоков, наглядно демонстрирующая преимущества функционального программирования. Поток – это последовательность элементов, в которой элементы не сохраняются, а их источник не модифицируется. В функциональных программах на Java часто порождаются потоки данных из некоторого источника, затем элементы подвергаются серии промежуточных операций (которая называется *конвейером*), после чего процесс завершается *терминальной операцией*.

Поток можно использовать только один раз. Поток, прошедший через нуль или более промежуточных операций и достигший терминальной операции, считается завершенным. Чтобы обработать значения еще раз, нужно создать новый поток.

Потоки по своей природе ленивые. Поток обрабатывает лишь столько данных, сколько необходимо для перехода в терминальное состояние. Как это работает, показано в рецепте 3.13.

В рецептах из этой главы демонстрируются типичные операции с потоками.

3.1. СОЗДАНИЕ ПОТОКОВ

Проблема

Требуется создать поток из источника данных.

Решение

Использовать статические фабричные методы интерфейса `Stream` или метод `stream` интерфейса `Iterable` или класса `Arrays`.

Обсуждение

Новый интерфейс `java.util.stream.Stream` в Java 8 предоставляет несколько статических методов создания потоков, а именно: `Stream.of`, `Stream.iterate` и `Stream.generate`.

Метод `Stream.of` принимает переменное число элементов:

```
static <T> Stream<T> of(T... values)
```

Реализация метода `of` в стандартной библиотеке в действительности делегирует работу методу `stream` класса `Arrays`, как показано в примере 3.1.

Пример 3.1 ❖ Эталонная реализация метода `Stream.of`

```
@SafeVarargs
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```

- ❑ Аннотация `@SafeVarargs` – часть механизма универсальных типов в Java. Она нужна, когда в качестве аргумента используется массив, поскольку можно присвоить типизированный массив массиву `Object` и затем нарушить типобезопасность при добавлении элемента. Аннотация `@SafeVarargs` сообщает компилятору, что разработчик обещает этого не делать. Дополнительные сведения см. в приложении А.

Тривиальный пример приведен в примере 3.2.

- ❗ Поскольку поток не начинает обработку данных, пока не будет достигнута терминальная операция, во всех примерах из этого рецепта в конце присутствует терминальный метод, например `collect` или `forEach`.

Пример 3.2 ❖ Создание потока методом `Stream.of`

```
String names = Stream.of("Gomez", "Morticia", "Wednesday", "Pugsley")
    .collect(Collectors.joining(", "));
System.out.println(names);
// печатается Gomez,Morticia,Wednesday,Pugsley
```

В API включен также перегруженный метод `of`, который принимает один элемент `T t`. Этот метод возвращает последовательный поток, состоящий из одного элемента.

В примере 3.3 демонстрируется метод `Arrays.stream`.

Пример 3.3 ❖ Создание потока методом `Arrays.stream`

```
String[] munsters = { "Herman", "Lily", "Eddie", "Marilyn", "Grandpa" };
names = Arrays.stream(munsters)
    .collect(Collectors.joining(", "));
System.out.println(names);
// печатается Herman,Lily,Eddie,Marilyn,Grandpa
```

Поскольку приходится предварительно создавать массив, этот подход менее удобен, но хорошо работает для динамических списков аргументов. В API включены перегруженные варианты метода `Arrays.stream` для массивов элементов типа `int`, `long` и `double`, а также показанный выше вариант для универсальных типов.

В интерфейсе `Stream` имеется также статический фабричный метод `iterate` с такой сигнатурой:

```
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
```

Согласно документации, этот метод «возвращает бесконечный последовательный упорядоченный поток `Stream`, порождаемый итеративным примене-

нием функции f к начальному элементу $seed$ ». Напомним, что `UnaryOperator` – это функция, для которой типы входа и выхода одинаковы (см. рецепт 2.4). Это полезно, если существует способ породить следующее значение по текущему, как в примере 3.4.

Пример 3.4 ❖ Создание потока методом `Stream.iterate`

```
List<BigDecimal> nums =
    Stream.iterate(BigDecimal.ONE, n -> n.add(BigDecimal.ONE) )
        .limit(10)
        .collect(Collectors.toList());
System.out.println(nums);
// печатается [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Stream.iterate(LocalDate.now(), ld -> ld.plusDays(1L))
    .limit(10)
    .forEach(System.out::println)
// печатается 10 дней, начиная с сегодняшнего
```

В первом примере производится подсчет значений типа `BigDecimal`, начиная с единицы, а во втором используется новый класс `LocalDate` из пакета `java.time` и к дате итеративно прибавляется один день. Поскольку оба потока неограниченные, включена промежуточная операция `limit`.

В интерфейсе `Stream` имеется также фабричный метод `generate` с такой сигнатурой:

```
static <T> Stream<T> generate(Supplier<T> s)
```

Он порождает последовательный неупорядоченный поток, повторно вызывая поставщик `Supplier`. В стандартной библиотеке есть простой пример `Supplier` (метод без аргументов, возвращающий значение) – метод `Math.random`, его использование демонстрируется в примере 3.5.

Пример 3.5 ❖ Создание потока случайных чисел типа `double`

```
long count = Stream.generate(Math::random)
    .limit(10)
    .forEach(System.out::println)
```

Если уже имеется коллекция, то можно воспользоваться методом по умолчанию `stream`, добавленным в интерфейс `Collection`, как показано в примере 3.6.

Пример 3.6 ❖ Создание потока из коллекции

```
List<String> bradyBunch = Arrays.asList("Greg", "Marcia", "Peter", "Jan",
    "Bobby", "Cindy");
names = bradyBunch.stream()
    .collect(Collectors.joining(", "));
System.out.println(names);
// печатается Greg,Marcia,Peter,Jan,Bobby,Cindy
```

Для работы с примитивными типами предусмотрены еще три родственных интерфейса: `IntStream`, `LongStream` и `DoubleStream`. В интерфейсах `IntStream` и `LongStream` имеются по два дополнительных фабричных метода создания потоков:

`range` и `rangeClosed`. Приведем их сигнатуры для `IntStream` (в `LongStream` все обстоит аналогично):

```
static IntStream range(int startInclusive, int endExclusive)
static IntStream rangeClosed(int startInclusive, int endInclusive)
static LongStream range(long startInclusive, long endExclusive)
static LongStream rangeClosed(long startInclusive, long endInclusive)
```

Различия понятны из имен аргументов: поток `rangeClosed` включает конечное значение, а поток `range` – нет. Оба метода возвращают последовательный упорядоченный поток, который начинается со значения первого аргумента, и каждый следующий элемент на 1 больше предыдущего. В примере 3.7 показано, как они используются.

Пример 3.7 ❖ Методы `range` и `rangeClosed`

```
List<Integer> ints = IntStream.range(10, 15)
    .boxed() ❶
    .collect(Collectors.toList());
System.out.println(ints);
// печатается [10, 11, 12, 13, 14]

List<Long> longs = LongStream.rangeClosed(10, 15)
    .boxed() ❶
    .collect(Collectors.toList());
System.out.println(longs);
// печатается [10, 11, 12, 13, 14, 15]
```

❶ Необходимо, чтобы коллектор мог преобразовать примитивы в `List<T>`

Единственная тонкость в этом примере – использование метода `boxed` для преобразования значений типа `int` в экземпляры класса `Integer`, эта тема будет рассмотрена подробнее в рецепте 3.2.

Итак, существуют следующие методы создания потоков:

- `Stream.of(T... values)` и `Stream.of(T t)`;
- `Arrays.stream(T[] array)` с перегруженными вариантами для `int[]`, `double[]` и `long[]`;
- `Stream.iterate(T seed, UnaryOperator<T> f)`;
- `Stream.generate(Supplier<T> s)`;
- `Collection.stream()`;
- `range` и `rangeClosed`:
 - `IntStream.range(int startInclusive, int endExclusive)`;
 - `IntStream.rangeClosed(int startInclusive, int endInclusive)`;
 - `LongStream.range(long startInclusive, long endExclusive)`;
 - `LongStream.rangeClosed(long startInclusive, long endInclusive)`.

См. также

Потоки используются на протяжении всей книги. Процедура преобразования потоков значений примитивных типов в обернутые экземпляры обсуждается в рецепте 3.2.

3.2. ОБЕРНУТЫЕ ПОТОКИ

Проблема

Требуется создать коллекцию из потока значений примитивных типов.

Решение

Использовать метод `boxed` интерфейса `Stream` для обертывания элементов. Можно вместо этого отобразить значения, пользуясь подходящим классом-оберткой, или воспользоваться вариантом метода `collect` с тремя аргументами.

Обсуждение

Для преобразования потока объектов в коллекцию существуют статические методы класса `Collectors`. Например, имея поток строк, мы можем создать список `List<String>`, как показано в примере 3.8.

Пример 3.8 ❖ Преобразование потока строк в список

```
List<String> strings = Stream.of("this", "is", "a", "list", "of", "strings")
    .collect(Collectors.toList());
```

Но для потока значений примитивных типов этот подход не годится. Код в примере 3.9 не компилируется.

Пример 3.9 ❖ Преобразование потока `int` в список `Integer` (НЕ КОМПИЛИРУЕТСЯ)

```
IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(Collectors.toList()); // не компилируется
```

Обойти эту трудность можно тремя способами. Во-первых, воспользоваться методом `boxed` интерфейса `Stream`, чтобы преобразовать `IntStream` в `Stream<Integer>`, как показано в примере 3.10.

Пример 3.10 ❖ Использование метода `boxed`

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .boxed() ❶
    .collect(Collectors.toList());
```

❶ Преобразует `int` в `Integer`

Другой способ – воспользоваться методом `mapToObj` для преобразования каждого значения примитивного типа в экземпляр класса-обертки, как в примере 3.11.

Пример 3.11 ❖ Использование метода `mapToObj`

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .mapToObj(Integer::valueOf)
    .collect(Collectors.toList());
```

Если методы `mapToInt`, `mapToLong` и `mapToDouble` преобразуют потоки объектов в потоки соответствующих примитивов, то метод `mapToObj`, присутствующий

в интерфейсах `IntStream`, `LongStream` и `DoubleStream`, преобразует примитивы в экземпляры ассоциированных классов-оберткок. В этом примере аргументом `mapToObj` является конструктор класса `Integer`.



В JDK 9 конструктор `Integer(int val)` объявлен нерекомендуемым из соображений производительности. Вместо него рекомендуется использовать метод `Integer.valueOf(int)`.

Еще одна альтернатива – использовать вариант метода `collect` с тремя аргументами:

```
<R> R collect(Supplier<R> supplier,
             ObjIntConsumer<R> accumulator,
             BiConsumer<R,R> combiner)
```

Пример 3.12 ❖ Использование варианта `collect` с тремя аргументами

```
List<Integer> ints = IntStream.of(3, 1, 4, 1, 5, 9)
    .collect(ArrayList<Integer>::new, ArrayList::add, ArrayList::addAll);
```

В этом примере поставщиком является конструктор класса `ArrayList<Integer>`, аккумулятором – метод `add`, описывающий, как добавить в список один элемент, а комбинатором (используется только в случае параллельных операций) – метод `addAll`, который объединяет два списка в один. Вариант `collect` с тремя аргументами применяется не часто, но понимать, как он работает, полезно.

Годятся все три подхода, какой выбрать – дело вкуса.

Кстати говоря, если вы хотите преобразовать поток в массив, а не в список, то метод `toArray` работает ничуть не хуже, а то и лучше. См. пример 3.13.

Пример 3.13 ❖ Преобразование `IntStream` в массив

```
int[] intArray = IntStream.of(3, 1, 4, 1, 5, 9).toArray();
// или
int[] intArray = IntStream.of(3, 1, 4, 1, 5, 9).toArray(int[]::new);
```

В первом случае используется вариант `toArray` по умолчанию, который возвращает `Object[]`, а во втором `IntFunction<int[]>` используется как генератор, который создает массив `int[]` подходящего размера и заполняет его.

Необходимость описанного выше преобразования – еще одно следствие принятого при проектировании Java решения рассматривать примитивы иначе, чем объекты, осложненного впоследствии добавлением в язык универсальных типов. Тем не менее методы `boxed` или `mapToObj` не вызывают трудностей, коль скоро вы о них знаете.

См. также

Коллекторы обсуждаются в главе 4, а ссылки на конструкторы – в рецепте 1.3.

3.3. ОПЕРАЦИИ РЕДУКЦИИ

Проблема

Требуется получить одно значение в результате операций с потоком.

Решение

Использовать метод `reduce` для аккумуляирования результатов вычислений.

Обсуждение

Функциональная парадигма в Java часто принимает форму процесса, который называется отображение-фильтрация-редукция. Операция `map` преобразует поток значений одного типа (например, `String`) в поток значений другого типа (например, `int` – путем вызова метода `length`). Затем к потоку применяется операция `filter`, порождающая новый поток, который содержит только интересующие элементы (например, строки не длиннее определенной величины). Наконец, терминальная операция порождает из потока единственное значение (например, суммарную или среднюю длину).

Встроенные операции редукции

В примитивных потоках `IntStream`, `LongStream` и `DoubleStream` есть несколько операций редукции, являющихся частью API.

В табл. 3.1 перечислены операции редукции в интерфейсе `IntStream`.

Таблица 3.1. Операции редукции в интерфейсе `IntStream`

Метод	Тип возвращаемого значения
<code>average</code>	<code>OptionalDouble</code>
<code>count</code>	<code>Long</code>
<code>max</code>	<code>OptionalInt</code>
<code>min</code>	<code>OptionalInt</code>
<code>sum</code>	<code>Int</code>
<code>summaryStatistics</code>	<code>IntSummaryStatistics</code>
<code>collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R,R> combiner)</code>	<code>R</code>
<code>reduce</code>	<code>int, OptionalInt</code>

Операции редукции `sum`, `count`, `max`, `min` и `average` делают ровно то, что можно ожидать, судя по названию. Интересен лишь тот факт, что некоторые из них возвращают значение типа `Optional`, поскольку если в потоке вообще нет элементов (возможно, в результате фильтрации), то результат не определен или равен `null`.

В примере 3.14 показаны операции редукции, основанные на длинах строк из коллекции.

Пример 3.14 ❖ Операции редукции для потока `IntStream`

```
String[] strings = "this is an array of strings".split(" ");
long count = Arrays.stream(strings)
    .map(String::length)    ❶
    .count();
System.out.println("Всего существует " + count + " строк");

int totalLength = Arrays.stream(strings)
    .mapToInt(String::length)  ❷
    .sum();
System.out.println("Суммарная длина равна " + totalLength);

OptionalDouble ave = Arrays.stream(strings)
    .mapToInt(String::length)  ❷
    .average();
System.out.println("Средняя длина равна " + ave);

OptionalInt max = Arrays.stream(strings)
    .mapToInt(String::length)  ❷
    .max();                    ❸

OptionalInt min = Arrays.stream(strings)
    .mapToInt(String::length)  ❷
    .min();                    ❸

System.out.println("Максимальная и минимальная длины равны " + max + " и " + min);
```

- ❶ `count` – метод `Stream`, поэтому необходимо преобразовать в `IntStream`
- ❷ `sum` и `average` определены только для потоков значений примитивного типа
- ❸ `max` и `min` без компаратора определены только для потоков значений примитивного типа

Программа печатает:

```
Всего существует 6 строк
Суммарная длина равна 22
Средняя длина равна OptionalDouble[3.6666666666666665]
Максимальная и минимальная длины равны OptionalInt[7] и OptionalInt[2]
```

Обратите внимание, что методы `average`, `max` и `min` возвращают `Optional`, поскольку теоретически к потоку мог быть применен фильтр, отбрасывающий все элементы.

Метод `count` весьма интересен и будет рассмотрен в рецепте 3.7.

В интерфейсе `Stream` определены методы `max(Comparator)` и `min(Comparator)`, в которых для определения максимального и минимального элементов используются компараторы. В интерфейсе `IntStream` есть перегруженные варианты этих методов, не принимающие аргументов, поскольку сравнение производится в естественном порядке целых чисел.

Метод `summaryStatistics` обсуждается в рецепте 3.8.

Последние две операции в таблице, `collect` и `reduce`, заслуживают отдельного обсуждения. Метод `collect` часто используется в этой книге для преобразования потока в коллекцию, обычно в сочетании с каким-либо из вспомогатель-

ных методов класса `Collectors`, например `toList` или `toSet`. Такого варианта `collect` нет в потоках значений примитивных типов. Показанный выше вариант с тремя аргументами принимает коллекцию, которую нужно заполнить, способ добавления одного элемента в коллекцию и способ добавления в нее нескольких элементов. Пример приведен в рецепте 3.2.

Базовые реализации редукции

Поведение метода `reduce` может показаться интуитивно непонятным, пока не увидишь его в действии.

В интерфейсе `IntStream` есть два перегруженных варианта `reduce`:

```
OptionalInt reduce(IntBinaryOperator op)
int reduce(int identity, IntBinaryOperator op)
```

Первый принимает `IntBinaryOperator` и возвращает `OptionalInt`, второй – целое число `identity` и `IntBinaryOperator`.

Напомним, что метод `java.util.function.BiFunction` принимает два аргумента и возвращает одно значение, причем типы всех трех значений могут быть различны. Если все три типа совпадают, то функция представляет собой бинарный оператор `BinaryOperator` (к примеру, `Math.max`). А тип `IntBinaryOperator` – это `BinaryOperator`, в котором оба входных значения и выходное значение имеют тип `int`.

Предположим на секунду, что мы забыли о существовании метода `sum`. Тогда для суммирования последовательности целых чисел можно было бы воспользоваться методом `reduce`, как показано в примере 3.15.

Пример 3.15 ❖ Суммирование чисел с помощью `reduce`

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + y).orElse(0); ❶
```

❶ Значение `sum` равно 55

i Обычно операции потокового конвейера записываются друг под другом, поскольку API потоков *текущий*, т. е. результат предыдущего метода подается на вход следующего. Но в данном случае метод `reduce` возвращает не поток, поэтому метод `orElse` записывается в той же строке, а не ниже, т. к. не является частью конвейера. Это просто для удобства, вы можете выбрать стиль форматирования, который вам больше нравится.

Здесь в качестве `IntBinaryOperator` используется лямбда-выражение, которое принимает два целых числа и возвращает их сумму. Поскольку поток может оказаться пустым, если в конвейер добавлен фильтр, результат имеет тип `OptionalInt`. Сцепление его с методом `orElse` означает, что если в потоке нет элементов, то следует вернуть нуль.

Первый аргумент лямбда-выражения можно рассматривать как аккумулятор, а во втором передаются значения элементов потока. Это становится очевидно, если распечатать значения аргументов, как показано в примере 3.6.

Пример 3.16 ❖ Печать значений `x` и `y`

```
int sum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> {
```

```

    System.out.printf("x=%d, y=%d\n", x, y);
    return x + y;
}).orElse(0);

```

В примере 3.17 показан результат.

Пример 3.17 ❖ Распечатка передаваемых значений

```

x=1, y=2
x=3, y=3
x=6, y=4
x=10, y=5
x=15, y=6
x=21, y=7
x=28, y=8
x=36, y=9
x=45, y=10
sum=55

```

Как видим, начальными значениями x и y являются первые два значения из диапазона. Значение, возвращенное бинарным оператором, становится новым значением x на следующей итерации (т. е. мы имеем аккумулятор), тогда как y принимает значение следующего элемента потока.

Все это замечательно, но что, если мы захотим обработать каждый элемент перед суммированием? Допустим, к примеру, что требуется сначала умножить каждое число на два, а затем прибавить к сумме¹. В примере 3.18 показана наивная попытка.

Пример 3.18 ❖ Удвоение значений в процессе суммирования (НЕПРАВИЛЬНО)

```

int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce((x, y) -> x + 2 * y).orElse(0); ❶

```

❶ Значение `doubleSum` равно 109 (плохо! ошибка на 1!)

Поскольку сумма целых чисел от 1 до 10 равна 55, должен получиться результат 110, а в действительности получилось 109. Дело в том, что в лямбда-выражении в методе `reduce` начальные значения x и y равны 1 и 2 (первые два элемента потока), и первое из них не было удвоено.

Именно поэтому существует перегруженный вариант `reduce`, который принимает начальное значение аккумулятора. Итоговый код показан в примере 3.19.

Пример 3.19 ❖ Удвоение значений в процессе суммирования (ПРАВИЛЬНО)

```

int doubleSum = IntStream.rangeClosed(1, 10)
    .reduce(0, (x, y) -> x + 2 * y); ❶

```

❶ Значение `doubleSum` равно 110, как и должно быть

¹ Эту задачу можно решить разными способами, в т. ч. просто удвоить значение, возвращенное методом `sum`. Выбранный нами подход позволяет проиллюстрировать вариант `reduce` с двумя аргументами.

Поскольку мы задали нулевое начальное значение аккумулятора x , переменной y поочередно присваиваются все элементы потока, и все они удваиваются. Значения x и y на каждой итерации показаны в примере 3.20.

Пример 3.20 ❖ Значения параметров лямбда-выражения на каждой итерации

```
Acc=0, n=1
Acc=2, n=2
Acc=6, n=3
Acc=12, n=4
Acc=20, n=5
Acc=30, n=6
Acc=42, n=7
Acc=56, n=8
Acc=72, n=9
Acc=90, n=10
sum=110
```

Отметим также, что в варианте `reduce` с начальным значением аккумулятора возвращаемое значение имеет тип `int`, а не `OptionalInt`.

Нейтральные значения бинарных операторов

В примерах из этого рецепта мы называем первый аргумент начальным значением аккумулятора, хотя в сигнатуре метода он называется `identity`. Слово `identity` (нейтральный элемент) означает, что если первый операнд бинарного оператора принимает такое значение, то результатом оператора будет значение второго операнда. Так, для сложения нейтральным элементом является 0, а для умножения – 1. Для конкатенации строк нейтральным элементом будет пустая строка.

Следует помнить об этом требовании к `reduce`: первый аргумент должен быть нейтральным элементом той операции, которую реализует бинарный оператор. Это значение становится начальным значением аккумулятора.

В стандартной библиотеке имеется много методов редукции, но если ни один из них не подходит для решаемой задачи, то могут оказаться полезны два показанных выше варианта метода `reduce`.

Библиотечные бинарные операторы

В стандартную библиотеку добавлено несколько методов, существенно упрощающих операции редукции. Например, в классах `Integer`, `Long` и `Double` имеется метод `sum`, который работает именно так, как и следовало ожидать. Ниже показана реализация метода `sum` в классе `Integer`:

```
public static int sum(int a, int b) {
    return a + b;
}
```

Но зачем вообще создавать метод для сложения двух целых чисел? Затем, что метод `sum` – это реализация интерфейса `BinaryOperator` (точнее, `IntBinary-`

operator), поэтому его можно использовать в операции reduce, как показано в примере 3.21.

Пример 3.21 ❖ Выполнение редукции с помощью бинарного оператора

```
int sum = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                .reduce(0, Integer::sum);
System.out.println(sum);
```

Здесь нам даже IntStream не понадобился, а результат тот же самый. В классе Integer есть еще методы max и min, также являющиеся бинарными операторами, и использовать их можно точно так же.

Пример 3.22 ❖ Нахождение максимума с помощью редукции

```
Integer max = Stream.of(3, 1, 4, 1, 5, 9)
                    .reduce(Integer.MIN_VALUE, Integer::max); ❶
System.out.println("Максимальное значение равно " + max);
```

❶ Нейтральным элементом max является минимальное целое число

Еще один интересный пример – метод concat класса String, который, правда, не выглядит как BinaryOperator, потому что принимает только один аргумент:

```
String concat(String str)
```

Тем не менее его можно использовать в операции reduce, как показано в примере 3.23.

Пример 3.23 ❖ Конкатенация потока строк с помощью редукции

```
String s = Stream.of("this", "is", "a", "list")
                 .reduce("", String::concat);
System.out.println(s); ❶
```

❶ Печатается thisisalist

Это работает, потому что в случае, когда в ссылке на метод фигурирует имя класса (как в String::concat), первый параметр становится объектом, от имени которого вызывается метод concat, а второй – аргументом concat. Поскольку тип возвращаемого значения String, то объект, параметр и возвращаемое значение имеют одинаковый тип, и, следовательно, всю конструкцию можно рассматривать как бинарный оператор с точки зрения метода reduce.

Этот прием заметно уменьшает размер кода, так что помните о нем, изучая документацию API.

Использование коллекторов

Использовать метод concat подобным образом возможно, но не эффективно, потому что в процессе конкатенации строк создаются и уничтожаются объекты. Гораздо лучше применить метод collect в сочетании с коллектором.

Один из перегруженных вариантов метода collect интерфейса Stream принимает поставщика коллекции (Supplier), объект типа BiConsumer, который добавляет

в коллекцию один элемент, и объект `BiConsumer`, который объединяет две коллекции. В случае строк естественным аккумулятором будет `StringBuilder`. Соответствующая реализация на базе `collect` показана в примере 3.24.

Пример 3.24 ❖ Собираение строк с помощью `StringBuilder`

```
String s = Stream.of("this", "is", "a", "list")
    .collect(() -> new StringBuilder(),           ❶
            (sb, str) -> sb.append(str),        ❷
            (sb1, sb2) -> sb1.append(sb2))     ❸
    .toString();
```

- ❶ Поставщик результата
- ❷ Добавить к результату одно значение
- ❸ Объединить два результата

Этот подход проще выразить с помощью ссылок на методы, как в примере 3.25.

Пример 3.25 ❖ Собираение строк с помощью ссылок на методы

```
String s = Stream.of("this", "is", "a", "list")
    .collect(StringBuilder::new,
            StringBuilder::append,
            StringBuilder::append)
    .toString();
```

Но самое простое – воспользоваться методом `joining` из служебного класса `Collectors`, как в примере 3.26.

Пример 3.26 ❖ Соединение строк с помощью класса `Collectors`

```
String s = Stream.of("this", "is", "a", "list")
    .collect(Collectors.joining());
```

У метода `joining` имеется перегруженный вариант, принимающий еще разделитель строк. Трудно придумать что-то более простое. Дополнительные сведения и примеры см. в рецепте 4.2.

Самая общая форма редукции

У метода `reduce` есть и третий вариант:

```
<U> U reduce(U identity,
            BiFunction<U,? super T,U> accumulator,
            BinaryOperator<U> combiner)
```

Он несколько сложнее, и обычно есть более простые пути добиться той же цели, но все же покажем, как этот вариант можно использовать.

Рассмотрим класс `Book`, содержащий целочисленный идентификатор и строковое название книги.

Пример 3.27 ❖ Простой класс `Book`

```
public class Book {
    private Integer id;
```



```
private String title;

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

Пусть имеется список книг, и мы хотим поместить их в структуру Map, сделав ключами идентификаторы, а значениями – сами объекты Book. Один из возможных способов продемонстрирован в примере 3.28.



Задачу можно решить гораздо проще с помощью метода `Collectors.toMap`, показанного в рецепте 4.3. Здесь же мы хотим акцентировать внимание на более сложном варианте `reduce`.

Пример 3.28 ❖ Помещение объектов Book в структуру Map с помощью аккумулятора

```
HashMap<Integer, Book> bookMap = books.stream()
    .reduce(new HashMap<Integer, Book>(),
        (map, book) -> {
            map.put(book.getId(), book);
            return map;
        },
        (map1, map2) -> {
            map1.putAll(map2);
            return map1;
        });

bookMap.forEach((k,v) -> System.out.println(k + ": " + v));
```

- ❶ Нейтральное значение для `putAll`
- ❷ Поместить одну книгу в Map методом `put`
- ❸ Объединить две структуры Map методом `putAll`

Объяснять, что означают аргументы метода `reduce` проще, начав с конца.

Последний аргумент, комбинатор `combiner`, должен быть бинарным оператором. В данном случае мы написали лямбда-выражение, которое принимает два отображения, копирует все ключи из второго в первое и возвращает первое отображение. Это лямбда-выражение было бы проще, если бы метод `putAll` возвращал отображение, но не повезло. Комбинатор важен только в том случае, когда операция `reduce` выполняется параллельно, поскольку тогда необходимо объединять отображения, порождаемые в процессе обработки разных частей диапазона.

Второй аргумент – функция, которая добавляет одну книгу в Map. Она тоже была бы проще, если бы метод `put` интерфейса Map возвращал Map после добавления новой записи.

Первый аргумент метода `reduce` – нейтральное значение оператора `combiner`. В данном случае таковым будет пустое отображение, поскольку его комбинация с любым другим отображением не меняет последнего.

Результат работы программы выглядит следующим образом:

```
1: Book{id=1, title='Modern Java Recipes'}
2: Book{id=2, title='Making Java Groovy'}
3: Book{id=3, title='Gradle Recipes for Android'}
```

Операции редукции играют очень важную роль в функциональном программировании. Для многих типичных случаев интерфейс `Stream` предоставляет готовые методы, например `sum` или `collect(Collectors.joining(', '))`. Но если потребуется написать свой собственный, то из этого рецепта вы узнали, как использовать операцию `reduce` непосредственно.

Кстати, хочу вас обрадовать: разобравшись, как использовать `reduce` в Java 8, вы легко сможете сделать то же самое в других языках, даже если операция в них называется по-другому (`inject` в Groovy или `fold` в Scala).

См. также

Гораздо более простой способ преобразовать список Java-объектов в отображение `Map` представлен в рецепте 4.3. Сводные статистики обсуждаются в рецепте 3.8, коллекторы – в главе 4.

3.4. ПРОВЕРКА ПРАВИЛЬНОСТИ СОРТИРОВКИ С ПОМОЩЬЮ РЕДУКЦИИ

Проблема

Требуется проверить, отсортирован ли поток.

Решение

Использовать метод `reduce` для проверки пар соседних элементов.

Обсуждение

Метод `reduce` интерфейса `Stream` принимает в качестве аргумента `BinaryOperator`:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

`BinaryOperator` – это функция `Function`, для которой типы обоих входных аргументов и тип возвращаемого значения одинаковы. Как показано в рецепте 3.3, первым элементом `BinaryOperator` обычно является аккумулятор, а второму поочередно передается значение каждого элемента потока, как в примере 3.29.

Пример 3.29 ❖ Суммирование `BigDecimal` с помощью `reduce`

```
BigDecimal total = Stream.iterate(BigDecimal.ONE, n -> n.add(BigDecimal.ONE))
    .limit(10)
    .reduce(BigDecimal.ZERO, (acc, val) -> acc.add(val)); ❶
System.out.println("Сумма равна " + total);
```

❶ Использование метода `add` класса `BigDecimal` в качестве `BinaryOperator`

Как обычно, значение лямбда-выражения становится значением переменной `acc` на следующей итерации. Следовательно, результатом этого вычисления является сумма первых 10 объектов типа `BigDecimal`.

Это самый распространенный способ использования метода `reduce`, но из того, что здесь `acc` играет роль аккумулятора, вовсе не следует, что так должно быть всегда. Рассмотрим вместо этого подход к сортировке строк, описанный в рецепте 4.1. В примере 3.30 показано, как отсортировать строки по длине.

Пример 3.30 ❖ Сортировка строк по длине

```
List<String> strings = Arrays.asList(
    "this", "is", "a", "list", "of", "strings");

List<String> sorted = strings.stream()
    .sorted(Comparator.comparingInt(String::length))
    .collect(toList()); ❶
```

❶ Получается [“a”, “is”, “of”, “this”, “list”, “strings”]

Вопрос заключается в том, как протестировать это решение. Требуется сравнить каждую пару соседних элементов и убедиться, что длина первого меньше или равна длине второго. Для этого можно воспользоваться методом `reduce`, как показано в примере 3.31 (это часть теста Junit).

Пример 3.31 ❖ Проверка правильности сортировки потока строк

```
strings.stream()
    .reduce((prev, curr) -> {
        assertTrue(prev.length() <= curr.length()); ❶
        return curr; ❷
    });
```

❶ Проверить, что пара отсортирована правильно

❷ `curr` становится следующим значением `prev`

Для каждой пары соседних элементов предыдущий элемент присваивается переменной `prev`, а текущий – переменной `curr`. В утверждении проверяется, что длина предыдущей строки меньше или равна длине текущей. Важно здесь то, что лямбда-выражение, являющееся аргументом `reduce`, возвращает значение текущей строки, `curr`, которое становится значением `prev` на следующей итерации.

Чтобы этот код работал, нужно лишь, чтобы поток был последовательным и упорядоченным, а эти условия здесь выполняются.

См. также

Метод `reduce` обсуждается в рецепте 3.3, а сортировка – в рецепте 4.1.

3.5. Отладка потоков с помощью РЕЕК

Проблема

Требуется видеть отдельные элементы потока по мере их обработки.

Решение

Вставить в нужное место потока промежуточную операцию peek.

Обсуждение

Обработка потока состоит из нуля или более промежуточных операций, за которым следует терминальная операция. Каждая промежуточная операция возвращает новый поток. Терминальная операция возвращает нечто, не являющееся потоком.

Начинающих знакомить с Java 8 иногда раздражает последовательность промежуточных операций, потому что они не могут увидеть, как обрабатываются элементы потока.

Рассмотрим простой метод, который принимает начало и конец диапазона целых чисел, умножает каждое число из этого диапазона на два и вычисляет сумму тех результатов, которые делятся на 3.

Пример 3.32 ❖ Удвоение, фильтрация и суммирование целых чисел

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

Для проверки напишем простой тест:

```
@Test
public void sumDoublesDivisibleBy3() throws Exception {
    assertEquals(1554, demo.sumDoublesDivisibleBy3(100, 120));
}
```

Это вселяет некую уверенность, но не помогает заглянуть внутрь. Если бы код не работал, то было бы очень трудно понять, в чем проблема.

Добавим в конвейер операцию map, которая принимает значение, печатает его и возвращает без изменения, как показано в примере 3.33.

Пример 3.33 ❖ Добавление тождественного отображения для печати

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .map(n -> { ❶
            System.out.println(n);
            return n;
        })
        .map(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
}
```

❶ Тождественное отображение, которое печатает и возвращает элемент

В результате будут напечатаны числа от `start` до `end` включительно, по одному на строку. В производственном коде это ни к чему, а в отладочном дает возможность заглянуть внутрь потока, не вмешиваясь в его работу.

Именно так работает метод `peek` интерфейса `Stream`. Вот как выглядит его объявление:

```
Stream<T> peek(Consumer<? super T> action)
```

В документации написано, что метод `peek` «возвращает поток, состоящий из элементов данного потока, и дополнительно выполняет над каждым элементом указанное действие при его потреблении из результирующего потока». Напомним, что потребитель `Consumer` принимает один аргумент и ничего не возвращает, поэтому никакой потребитель не сможет испортить проходящие мимо него значения.

Поскольку `peek` – промежуточная операция, ее можно вставлять в конвейер сколько угодно раз.

Пример 3.34 ❖ Несколько методов `peek`

```
public int sumDoublesDivisibleBy3(int start, int end) {
    return IntStream.rangeClosed(start, end)
        .peek(n -> System.out.printf("original: %d\n", n)) ❶
        .map(n -> n * 2)
        .peek(n -> System.out.printf("doubled : %d\n", n)) ❷
        .filter(n -> n % 3 == 0)
        .peek(n -> System.out.printf("filtered: %d\n", n)) ❸
        .sum();
}
```

- ❶ Печатать значение до удвоения
- ❷ Печатать значение после удвоения, но до фильтрации
- ❸ Печатать значение после фильтрации, но до суммирования

В результате мы увидим сначала исходный элемент, затем результат его удвоения и еще раз, если он прошел фильтр. Вот что печатает этот код:

```
original: 100
doubled : 200
original: 101
doubled : 202
original: 102
doubled : 204
filtered: 204
...
original: 119
doubled : 238
original: 120
doubled : 240
filtered: 240
```

К сожалению, никак нельзя сделать вызовы `peek` факультативными, так что этот шаг удобен для отладки, но из производственного кода его нужно удалять.

3.6. ПРЕОБРАЗОВАНИЕ СТРОК В ПОТОКИ И НАОБОРОТ

Проблема

Требуется более идиоматичный способ обработки строк, чем цикл по отдельным символам.

Решение

Использовать методы по умолчанию `chars` и `codePoints` интерфейса `java.lang.CharSequence` для преобразования `String` в `IntStream`. Для обратного преобразования потока в строку использовать перегруженный вариант метода `collect` из интерфейса `IntStream`, который принимает поставщик `Supplier`, объект `BiConsumer`, представляющий аккумулятор, и объект `BiConsumer`, представляющий комбинатор.

Обсуждение

Строки – это коллекции символов, поэтому, в принципе, преобразовать строку в поток должно быть так же просто, как для любой другой коллекции или массива. Но, к сожалению, класс `String` не является частью подсистемы коллекций, не реализует интерфейс `Iterable` и, следовательно, в нем отсутствует фабричный метод `stream` для преобразования в `Stream`. Можно было бы попробовать обратиться к статическим методам `stream` из класса `java.util.Arrays`, но, увы, существуют `Arrays.stream` для `int[]`, `long[]`, `double[]` и даже `T[]`, но не для `char[]`. Похоже, проектировщики API не хотели, чтобы мы применяли к `String` приемы работы с потоками.

И все-таки есть подход, который работает. Класс `String` реализует интерфейс `CharSequence`, в котором появились два новых метода, порождающих `IntStream`. Оба они являются методами по умолчанию, т. е. для них предоставлена реализация. Их сигнатуры показаны в примере 3.35.

Пример 3.35 ❖ Поточковые методы интерфейса `java.lang.CharSequence`

```
default IntStream chars()
default IntStream codePoints()
```

Разница между ними в том, что один имеет дело с кодировкой UTF-16, а другой – с полным набором кодовых позиций Юникода. Все тонкости объяснены в документации по классу `java.lang.Character`. С точки зрения рассматриваемых здесь методов, единственное отличие – тип возвращаемых целых чисел. Первый метод возвращает поток `IntStream`, содержащий значения типа `char`, соответствующие элементам последовательности, а второй – `IntStream`, содержащий кодовые позиции Юникода.

Противоположный вопрос: как преобразовать поток символов в строку? Использовать метод `Stream.collect`, который производит изменяющую редукцию элементов потока с целью порождения коллекции. Чаще всего применяется вариант `collect`, принимающий объект `Collector`, поскольку в служебном классе

Collectors много статических методов (toList, toSet, toMap, joining и другие – все они рассматриваются в этой книге), которые порождают нужный коллектор.

Но выделяется своим отсутствием коллектор, который принимал бы поток символов и формировал из них строку. К счастью, такой код нетрудно написать, воспользовавшись другим перегруженным вариантом collect, который принимает один аргумент типа Supplier и два аргумента типа BiConsumer, играющих соответственно роль аккумулятора и комбинатора.

Все это звучит сложнее, чем есть на самом деле. Попробуем написать метод, который проверяет, является ли строка палиндромом, т. е. читается слева направо так же, как справа налево. При проверке регистр букв не учитывается и все знаки препинания предварительно удаляются. В примере 3.36 показано, как это можно было бы сделать в Java 7 и более ранних версиях.

Пример 3.36 ❖ Проверка на палиндром в Java 7 и более ранних версиях

```
public boolean isPalindrome(String s) {
    StringBuilder sb = new StringBuilder();
    for (char c : s.toCharArray()) {
        if (Character.isLetterOrDigit(c)) {
            sb.append(c);
        }
    }
    String forward = sb.toString().toLowerCase();
    String backward = sb.reverse().toString().toLowerCase();
    return forward.equals(backward);
}
```

Как принято в нефункциональном коде, метод объявляет отдельный объект с изменяемым состоянием (экземпляр класса StringBuilder), затем обходит коллекцию (массив char[], возвращенный методом toCharArray класса String), проверяя в условии if, следует ли добавлять символ в буфер. Ко всему прочему, в классе StringBuilder имеется метод reverse, который упрощает проверку на палиндром; в классе String такого метода нет. Комбинация изменяемого состояния, итерирования и принятия решений просто вопиет о необходимости другого подхода – на основе потоков.

Этот альтернативный подход показан в примере 3.37.

Пример 3.37 ❖ Проверка на палиндром с помощью потоков Java 8

```
public boolean isPalindrome(String s) {
    String forward = s.toLowerCase().codePoints() ❶
        .filter(Character::isLetterOrDigit)
        .collect(StringBuilder::new,
            StringBuilder::appendCodePoint,
            StringBuilder::append)
        .toString();

    String backward = new StringBuilder(forward).reverse().toString();
    return forward.equals(backward);
}
```

❶ Возвращается IntStream

Метод `codePoints` возвращает поток `IntStream`, который затем можно профильтровать по тому же условию. Здесь особенно интересен метод `collect` с такой сигнатурой:

```
<R> R collect(Supplier<R> supplier,
             BiConsumer<R,? super T> accumulator,
             BiConsumer<R,R> combiner)
```

Метод принимает следующие аргументы:

- поставщик `Supplier`, который порождает результирующий редуцированный объект, в данном случае `StringBuilder`;
- потребитель `BiConsumer`, который аккумулирует каждый элемент потока в результирующую структуру данных; в данном случае в этом качестве используется метод `appendCodePoint`;
- потребитель `BiConsumer`, представляющий комбинатор, т. е. «ничего не изменяющую функцию без состояния» для объединения двух значений, которые должны быть совместимы с аккумулятором. Здесь в этом качестве используется метод `append`. Отметим, что комбинатор применяется только в случае, когда операция выполняется параллельно.

Выглядит это всё довольно сложно, но у такого кода есть то преимущество, что не нужно проводить различия между символами и целыми числами, а это часто бывает проблемой при работе с элементами строк.

В примере 3.38 показан простой тест этого метода.

Пример 3.38 ❖ Тестирование метода проверки на палиндром

```
private PalindromeEvaluator demo = new PalindromeEvaluator();

@Test
public void isPalindrome() throws Exception {
    assertTrue(
        Stream.of("Madam, in Eden, I'm Adam",
                 "Go hang a salami; I'm a lasagna hog",
                 "Flee to me, remote elf!",
                 "A Santa pets rats as Pat taps a star step at NASA")
              .allMatch(demo::isPalindrome));
    assertFalse(demo.isPalindrome("Это НЕ палиндром"));
}
```

Интерпретация строк как массивов символов не вполне укладывается в функциональные идиомы Java 8, но надеемся, что механизмы, изложенные в этом рецепте, покажут, как можно решить эту проблему.

См. также

Мы продолжим обсуждать коллекторы в главе 4 и реализуем собственный коллектор в рецепте 4.9. Метод `allMatch` обсуждается в рецепте 3.10.

3.7. ПОДСЧЕТ ЭЛЕМЕНТОВ

Проблема

Требуется узнать, сколько элементов в потоке.

Решение

Использовать метод `Stream.count` или `Collectors.counting`.

Обсуждение

Несмотря на простоту, этот рецепт все же демонстрирует технику, к которой мы еще вернемся в рецепте 4.6.

В интерфейсе `Stream` имеется метод по умолчанию `count`, который возвращает число типа `long`, как показано в примере 3.39.

Пример 3.39 ❖ Подсчет элементов в потоке

```
long count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5).count();
System.out.printf("В потоке %d элементов%n", count); ❶
```

❶ Печатается В потоке 9 элементов

В документации отмечена интересная особенность реализации метода `count`: это частный случай редукции, эквивалентный вызову

```
return mapToLong(e -> 1L).sum();
```

Сначала каждый элемент отображается на 1 типа `long`. Затем метод `mapToLong` порождает поток `LongStream`, в котором имеется метод `sum`. Иными словами, каждому элементу сопоставляется 1, а затем эти единицы суммируются. Просто и изящно.

Есть и альтернатива – метод `counting` класса `Collectors`, показанный в примере 3.40.

Пример 3.40 ❖ Подсчет элементов с помощью метода `Collectors.counting`

```
count = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .collect(Collectors.counting());
System.out.printf("В потоке %d элементов%n", count);
```

Результат тот же самый. Спрашивается, зачем так делать? Не проще ли использовать метод `count` интерфейса `Stream`?

Можно, конечно, и, пожалуй, даже нужно. Но этот прием оказывается полезным при использовании подчиненного коллектора, который обсуждается в рецепте 4.6. А чтобы разжечь аппетит, рассмотрим пример 3.41.

Пример 3.41 ❖ Подсчет строк в группах по длине

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(
        s -> s.length() % 2 == 0, ❶
```

```

Collectors.counting()); ❷
numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n", k, v));
//
// false: 4
// true: 8

```

- ❶ Предикат
- ❷ Подчиненный коллектор

В качестве первого аргумента метод `partitioningBy` принимает предикат, который разбивает строки на две категории: удовлетворяющие и не удовлетворяющие предикату. Если бы это был единственный аргумент, то в результате получилось бы отображение `Map<Boolean, List<String>>` с двумя ключами, `true` и `false`, которым соответствуют списки строк четной и нечетной длины.

Но использованный здесь перегруженный вариант метода `partitioningBy` с двумя аргументами принимает еще объект типа `Collector`, называемый подчиненным коллектором (`downstream collector`), который дополнительно обрабатывает возвращенный список строк. Это как раз подходящая ситуация для метода `Collectors.counting`. Теперь на выходе получается отображение `Map<Boolean, Long>`, в котором значениями являются счетчики строк четной и нечетной длины.

В этом разделе мы обсудим еще несколько методов интерфейса `Stream`, имеющих аналоги в классе `Collectors`. Если вы работаете непосредственно с потоком, то лучше использовать методы `Stream`. А методы `Collectors` предназначены для вложенной постобработки результатов операций `partitioningBy` или `groupingBy`.

См. также

Подчиненные коллекторы обсуждаются в рецепте 4.6, а коллекторы вообще – в нескольких рецептах из главы 4. Подсчет как встроенная операция редукции рассмотрен в рецепте 3.3.

3.8. СВОДНЫЕ СТАТИСТИКИ

Проблема

Требуется найти количество, сумму, минимум, максимум и среднее значение потока числовых элементов.

Решение

Использовать метод `summaryStatistics`, имеющийся в интерфейсах `IntStream`, `DoubleStream` и `LongStream`.

Обсуждение

В интерфейсы `IntStream`, `DoubleStream` и `LongStream` добавлены методы для работы с примитивными типами. Один из них, `summaryStatistics`, показан в примере 3.42.

Пример 3.42 ❖ Сводные статистики

```

DoubleSummaryStatistics stats = DoubleStream.generate(Math::random)
    .limit(1_000_000)
    .summaryStatistics();

System.out.println(stats);

System.out.println("count: " + stats.getCount());
System.out.println("min : " + stats.getMin());
System.out.println("max : " + stats.getMax());
System.out.println("sum : " + stats.getSum());
System.out.println("ave : " + stats.getAverage());

```

❶ Печатать методом toString

✓ В Java 7 добавлена возможность использовать знак подчеркивания в числовых литералах, например 1_000_000.

Вот результат типичного прогона:

```

DoubleSummaryStatistics{count=1000000, sum=499608.317465, min=0.000001,
    average=0.499608, max=0.999999}
count: 1000000
min   : 1.3938598313334438E-6
max   : 0.9999988915490642
sum   : 499608.31746475823
ave   : 0.49960831746475826

```

Реализация метода toString в классе DoubleSummaryStatistics показывает все значения, но имеются также методы чтения каждого значения по отдельности: getCount, getSum, getMax, getMin и getAverage. При миллионе чисел типа double неудивительно, что минимум близок к нулю, максимум – к единице, сумма приблизительно равна 500 000, а среднее близко к 0.5.

В классе DoubleSummaryStatistics есть еще два интересных метода:

```

void accept(double value)
void combine(DoubleSummaryStatistics other)

```

Метод accept добавляет новое значение во множество чисел, по которым вычисляется сводная статистика, а метод combine объединяет два объекта DoubleSummaryStatistics в один. Они используются, когда перед вычислением результатов в экземпляре класса добавляются данные.

Например, сайт Spotrac (www.spotrac.com) отслеживает статистику выплат спортивным командам. В исходном коде, прилагаемом к этой книге, имеется файл, содержащий платежные ведомости всех 30 команд главной лиги бейсбола за сезон 2017 года, взятый с этого сайта¹.

В примере 3.43 определен класс Team, содержащий идентификатор команды id, ее название name и суммарную зарплату salary.

¹ Источник: <http://www.spotrac.com/mlb/payroll/>, где можно задать год или другую информацию.

Пример 3.43 ❖ Класс `Team` с полями `id`, `name` и `salary`

```
public class Team {
    private static final NumberFormat nf = NumberFormat.getCurrencyInstance();

    private int id;
    private String name;
    private double salary;

    // ... конструкторы, методы чтения и установки ...

    @Override
    public String toString() {
        return "Team{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", salary=" + nf.format(salary) +
            '}';
    }
}
```

Разбор файла с зарплатами команд дает такие результаты:

```
Team{id=1, name='Los Angeles Dodgers', salary=$245,269,535.00}
Team{id=2, name='Boston Red Sox', salary=$202,135,939.00}
Team{id=3, name='New York Yankees', salary=$202,095,552.00}
...
Team{id=28, name='San Diego Padres', salary=$73,754,027.00}
Team{id=29, name='Tampa Bay Rays', salary=$73,102,766.00}
Team{id=30, name='Milwaukee Brewers', salary=$62,094,433.00}
```

Теперь есть два способа вычислить сводную статистику по всем командам. Первый – воспользоваться методом `collect` с тремя аргументами, как показано в примере 3.44.

Пример 3.44 ❖ Метод `collect` с поставщиком, аккумулятором и комбинатором

```
DoubleSummaryStatistics teamStats = teams.stream()
    .mapToDouble(Team::getSalary)
    .collect(DoubleSummaryStatistics::new,
            DoubleSummaryStatistics::accept,
            DoubleSummaryStatistics::combine);
```

Этот вариант метода `collect` обсуждается в рецепте 4.9. В данном случае в нем используются ссылка на конструктор класса `DoubleSummaryStatistics`, метод `accept` для добавления нового значения в уже имеющийся объект `DoubleSummaryStatistics` и метод `combine` для объединения двух объектов `DoubleSummaryStatistics` в один.

Результаты (после форматирования) выглядят так:

```
30 teams
sum = $4,232,271,100.00
min = $62,094,433.00
max = $245,269,535.00
ave = $141,075,703.33
```

В рецепте 4.6, посвященном подчиненным коллекторам, объясняется другой способ вычисления тех же данных. В данном случае сводка вычисляется, как показано в примере 3.45.

Пример 3.45 ❖ Метод `collect` с использованием `summarizingDouble`

```
teamStats = teams.stream()
    .collect(Collectors.summarizingDouble(Team::getSalary));
```

В качестве аргумента методу `Collectors.summarizingDouble` передается суммарная зарплата каждой команды. Результат получается такой же, как и раньше.

Классы сводной статистики – это, по существу, статистика «для бедных». Они умеют вычислять только пять свойств (общее число, максимум, минимум, сумму и среднее значение), но если больше вам ничего и не нужно, то вы будете рады, что такие классы есть в библиотеке.

См. также

Сводная статистика – частный случай операции редукции. Другие операции описаны в рецепте 3.3. Подчиненные коллекторы рассматриваются в рецепте 4.6. Метод `collect` с тремя аргументами обсуждается в рецепте 4.9.

3.9. НАХОЖДЕНИЕ ПЕРВОГО ЭЛЕМЕНТА В ПОТОКЕ

Проблема

Требуется найти в потоке первый элемент, удовлетворяющий заданному условию.

Решение

Воспользоваться методом `findFirst` или `findAny`, предварительно применив фильтр.

Обсуждение

Методы `findFirst` и `findAny` интерфейса `java.util.stream.Stream` возвращают объект типа `Optional`, описывающий первый элемент потока. Ни тот, ни другой не принимают аргументов, т. е. операции отображения или фильтрации должны быть произведены предварительно.

Например, чтобы найти первое четное число в списке целых чисел, нужно сначала применить фильтр четности, а затем метод `findFirst`, как показано в примере 3.46.

Пример 3.46 ❖ Нахождение первого четного числа

```
Optional<Integer> firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .filter(n -> n % 2 == 0)
```

```
.findFirst();
```

```
System.out.println(firstEven);
```

❶ Печатается `Optional[4]`

Если поток пустой, то будет возвращен пустой объект `Optional` (см. пример 3.47).

Пример 3.47 ❖ Применение `findFirst` к пустому потоку

```
Optional<Integer> firstEvenGT10 = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .filter(n -> n > 10)
    .filter(n -> n % 2 == 0)
    .findFirst();
```

```
System.out.println(firstEvenGT10);
```

❶ Печатается `Optional.empty`

Поскольку этот код возвращает первый элемент после фильтрации, может закрасться мысль, что мы делаем уйму работы впустую. Зачем применять деление по модулю ко всем элементам потока, с тем чтобы потом выбрать только первый? Но поскольку на самом деле элементы потока обрабатываются по одному, никакой проблемы нет. Этот вопрос обсуждается в рецепте 3.13.

Если у потока нет порядка следования, то может быть возвращен любой элемент. В данном случае порядок следования есть, поэтому «первым» четным числом будет 4 *вне зависимости от того, производится поиск в последовательном или параллельном потоке*. См. пример 3.48.

Пример 3.48 ❖ Использование `firstEven` с параллельным потоком

```
firstEven = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .parallel()
    .filter(n -> n % 2 == 0)
    .findFirst();
```

```
System.out.println(firstEven);
```

❶ Всегда печатается `Optional[4]`

На первый взгляд, кажется странным. Почему всегда должно получаться одно и то же значение, если несколько чисел могут обрабатываться одновременно? Ответ кроется в понятии *порядка следования* (*encounter order*).

В API порядок следования определяется как порядок, в котором источник данных делает элементы доступными. Для списка `List` и для массива порядок следования определен, а для множества `Set` – нет.

В интерфейсе `BaseStream` (который `Stream` расширяет) имеется также метод `unordered`, который возвращает эквивалентный неупорядоченный поток, хотя иногда этот поток может совпадать с исходным. Это промежуточная (не терминальная) операция.

Множества и порядок следования

В объектах класса `HashSet` порядка следования нет, но если несколько раз инициализировать такой объект одними и теми же данными, то (в Java 8) элементы всегда будут возвращаться в одном и том же порядке. Это означает, что `findFirst` всякий раз будет давать один и тот же результат. В документации сказано, что `findFirst` может давать разные результаты для неупорядоченных потоков, но в текущей реализации поведение не меняется только потому, что поток неупорядоченный. Чтобы получить множество с другим порядком следования, можно произвести настолько много операций добавления и удаления элементов, чтобы вызвать повторное хэширование. Например:

```
List<String> wordList = Arrays.asList(
    "this", "is", "a", "stream", "of", "strings");
Set<String> words = new HashSet<>(wordList);
Set<String> words2 = new HashSet<>(words);

// Теперь добавим и удалим столько элементов, чтобы вызвать повторное хэширование
IntStream.rangeClosed(0, 50).forEachOrdered(i ->
    words2.add(String.valueOf(i)));
words2.retainAll(wordList);

// Множества равны, но порядок элементов различен
System.out.println(words.equals(words2));
System.out.println("До : " + words);
System.out.println("После: " + words2);
```

Результат будет выглядеть примерно так:

```
true
До : [a, strings, stream, of, this, is]
После: [this, is, strings, stream, of, a]
Порядок различается, поэтому findFirst дает разные результаты.
```

Появившиеся в Java 9 неизменяемые множества (и отображения) рандомизированы, поэтому порядок их обхода меняется при каждом прогоне, даже если каждый раз они инициализируются одинаково¹.

Метод `findAny` возвращает объект `Optional`, описывающий некоторый элемент потока, или пустой `Optional`, если поток пуст. В данном случае поведение операции явно *недетерминировано*, т. е. она вправе выбрать абсолютно любой элемент. Это открывает возможность оптимизации параллельных операций.

Чтобы продемонстрировать это, рассмотрим возврат произвольного элемента из неупорядоченного параллельного потока целых чисел. В примере 3.49 введена искусственная задержка – мы отображаем каждый элемент на себя, добавив случайную задержку величиной не более 100 миллисекунд.

¹ Спасибо Стюарту Марксу за это разъяснение.

Пример 3.49 ❖ Применение `findAny` к параллельному потоку со случайной задержкой

```
public Integer delay(Integer n) {
    try {
        Thread.sleep((long) (Math.random() * 100));
    } catch (InterruptedException ignored) { ❶
    }
    return n;
}

// ...

Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()           ❷
    .parallel()           ❸
    .map(this::delay)     ❹
    .findAny();           ❺

System.out.println("Any: " + any);
```

- ❶ Единственное исключение в Java, которое можно безопасно перехватить и проигнорировать¹
- ❷ Порядок нас не волнует
- ❸ Для распараллеливания используем обычный пул разветвления-соединения
- ❹ Вводим случайную задержку
- ❺ Возвращаем первый элемент независимо от порядка следования

Теперь на выходе может появиться любое из заданных чисел в зависимости от того, какой поток первым получит управление.

Методы `findFirst` и `findAny` являются *укорачивающими терминальными* операциями. Укорачивающая (*short-circuiting*) операция может породить конечный поток, получив на входе бесконечный. Терминальная операция является укорачивающей, если она может завершиться за конечное время, будучи применена к бесконечным входным данным.

Примеры из этого рецепта показывают, что иногда распараллеливание может навредить, а не повысить производительность. Потоки ленивые в том смысле, что обрабатывают лишь столько элементов, сколько необходимо для работы конвейера. В данном случае требуется всего-то вернуть первый элемент, потому запуск пула разветвления-соединения – явный перебор. См. пример 3.50.

Пример 3.50 ❖ Применение `findAny` к последовательному и к параллельному потокам

```
Optional<Integer> any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .map(this::delay)
    .findAny(); ❶
```

¹ Если говорить серьезно, то никакое исключение не стоит перехватывать и игнорировать. Просто в случае `InterruptedException` так поступают довольно часто. Но все равно это порочная практика.


```
System.out.println("Sequential Any: " + any);
any = Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
    .unordered()
    .parallel()
    .map(this::delay)
    .findAny(); ❷
System.out.println("Parallel Any: " + any);
```

- ❶ Последовательный поток (по умолчанию)
- ❷ Параллельный поток

Ниже показано, как выглядит типичный результат (на восьмиядерной машине, где пул разветвления-соединения по умолчанию содержит восемь потоков выполнения)¹.

В случае последовательной обработки:

```
main // sequential, so only one thread
Sequential Any: Optional[3]
```

В случае параллельной обработки:

```
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-5
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-6
ForkJoinPool.commonPool-worker-7
main
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-4
Parallel Any: Optional[1]
```

Последовательному конвейеру необходим доступ только к одному элементу, который он и возвращает, укорачивая тем самым процесс. Параллельный конвейер запускает до восьми потоков выполнения, находит один элемент, а затем завершает все потоки. Поэтому параллельный конвейер обрабатывает больше значений, чем необходимо.

Еще раз подчеркнем важность порядка следования в потоке. Если для потока определен порядок следования, то `findFirst` всегда возвращает одно и то же значение. Методу `findAny` разрешено возвращать любой элемент, поэтому он лучше подходит для параллельных операций.

См. также

Ленивые потоки обсуждаются в рецепте 3.13, параллельные потоки – в главе 9.

¹ В этом примере предполагается, что метод `delay` модифицирован и печатает не только обрабатываемое значение, но и имя текущего потока.

3.10. МЕТОДЫ ANYMATCH, ALLMATCH И NONEMATCH

Проблема

Требуется определить, существуют ли в потоке элементы, удовлетворяющие предикату, верно ли, что все элементы удовлетворяют предикату или что таких элементов нет вовсе.

Решение

Воспользоваться методами anyMatch, allMatch и noneMatch интерфейса Stream. Все они возвращают булево значение.

Обсуждение

Ниже приведены сигнатуры методов anyMatch, allMatch и noneMatch интерфейса Stream:

```
boolean anyMatch(Predicate<? super T> predicate)
boolean allMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

В качестве примера рассмотрим калькулятор простых чисел. Целое число называется простым, если его единственными делителями являются оно само и 1.

Тривиальный способ проверить, является ли число простым, – вычислить остаток от деления на все числа, начиная с 2 и заканчивая квадратным корнем из него, как показано в примере 3.51.

Пример 3.51 ❖ Проверка простоты числа

```
public boolean isPrime(int num) {
    int limit = (int) (Math.sqrt(num) + 1);           ❶
    return num == 2 || num > 1 && IntStream.range(2, limit)  ❷
        .noneMatch(divisor -> num % divisor == 0);
}
```

- ❶ Максимальное проверяемое число
- ❷ Использование noneMatch

Метод noneMatch весьма упрощает вычисление.

BigInteger и простые числа

Интересно, что в классе java.math.BigInteger есть метод isProbablyPrime с такой сигнатурой:

```
boolean isProbablyPrime(int certainty)
```

Если он возвращает false, то число заведомо составное, а если true, то в игру вступает аргумент certainty.

Величина `certainty` определяет допустимую недостоверность. Если метод возвращает `true`, то число является простым с вероятностью не менее $1 - 1/2^{\{certainty\}}$, т. е. если `certainty` равно 2, то вероятность равна 0.5, если `certainty` равно 3, то 0.75, если 4, то 0.875, если 5, то 0.9375 и т. д.

Чем выше величина `certainty`, тем дольше работает алгоритм.

В примере 3.52 приведены два теста калькулятора.

Пример 3.52 ❖ Тесты калькулятора простых чисел

```
private Primes calculator = new Primes();

@Test ❶
public void testIsPrimeUsingAllMatch() throws Exception {
    assertTrue(IntStream.of(2, 3, 5, 7, 11, 13, 17, 19)
        .allMatch(calculator::isPrime));
}

@Test ❷
public void testIsPrimeWithComposites() throws Exception {
    assertFalse(Stream.of(4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20)
        .anyMatch(calculator::isPrime));
}
```

❶ Для простоты используем `allMatch`

❷ Тест с составными числами

В первом тесте для потока заведомо простых чисел вызывается метод `allMatch`, аргумент которого является предикатом. Этот метод возвращает `true`, только если все числа простые.

Во втором тесте для потока составных чисел вызывается метод `anyMatch` и утверждается, что ни одно число не удовлетворяет предикату.

Методы `anyMatch`, `allMatch` и `noneMatch` дают удобный способ проверить выполнение условия для потока значений.

Но следует иметь в виду один граничный случай. Поведение методов `anyMatch`, `allMatch` и `noneMatch` для пустых потоков может показаться интуитивно неочевидным, как явствует из примера 3.53.

Пример 3.53 ❖ Тестирование для пустых потоков

```
@Test
public void emptyStreamsDanger() throws Exception {
    assertTrue(Stream.empty().allMatch(e -> false));
    assertTrue(Stream.empty().noneMatch(e -> true));
    assertFalse(Stream.empty().anyMatch(e -> true));
}
```

Для методов `allMatch` и `noneMatch` в документации сказано «если поток пустой, то возвращается `true` и предикат не вычисляется», т. е. в этих случаях предикат может быть любым. Метод `anyMatch` для пустого потока возвращает `false`. Это может стать причиной трудно вылавливаемых ошибок, так что будьте осторожны.



Для пустого потока методы `allMatch` и `noneMatch` возвращают `true`, а метод `anyMatch` – `false` независимо от переданного предиката. Предикат вообще не вычисляется, если поток пустой.

См. также

Предикаты обсуждаются в рецепте 2.3.

3.11. МЕТОДЫ FLATMAP И MAP

Проблема

Имеется поток и требуется некоторым образом преобразовать его элементы, но вы не знаете, каким методом воспользоваться: `map` или `flatMap`.

Решение

Использовать `map`, если каждый элемент преобразуется в одно значение. Использовать `flatMap`, если каждый элемент преобразуется в несколько значений и получившийся поток нужно «разгладить».

Обсуждение

Оба метода, `map` и `flatMap`, интерфейса `Stream` принимают в качестве аргумента объект типа `Function`. Сигнатура `map` имеет вид:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Функция `Function` принимает один входной аргумент типа `T` и преобразует его в одно выходное значение типа `R`.

Рассмотрим класс `Customer`, описывающий заказчика, с которым ассоциированы имя и коллекция заказов `Order`. Для простоты будем считать, что в классе `Order` имеется только целый идентификатор и больше ничего. Оба класса приведены в примере 3.54.

Пример 3.54 ❖ Отношение один ко многим

```
public class Customer {
    private String name;
    private List<Order> orders = new ArrayList<>();

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

    public String getName() { return name; }
    public List<Order> getOrders() { return orders; }

    public Customer addOrder(Order order) {
        orders.add(order);
        return this;
    }
}
```

```
public class Order {
    private int id;

    public Order(int id) {
        this.id = id;
    }

    public int getId() { return id; }
}
```

Теперь создадим несколько заказчиков и заказов.

Пример 3.55 ❖ Создание заказчиков и заказов

```
Customer sheridan = new Customer("Sheridan");
Customer ivanova = new Customer("Ivanova");
Customer garibaldi = new Customer("Garibaldi");

sheridan.addOrder(new Order(1))
    .addOrder(new Order(2))
    .addOrder(new Order(3));
ivanova.addOrder(new Order(4))
    .addOrder(new Order(5));

List<Customer> customers = Arrays.asList(sheridan, ivanova, garibaldi);
```

Операция `map` выполняется, когда каждому входу соответствует один и только один выход. В примере 3.56 мы отображаем заказчиков на имена и печатаем имена.

Пример 3.56 ❖ Применение `map` для отображения `Customer` на `name`

```
customers.stream()           ❶
    .map(Customer::getName)   ❷
    .forEach(System.out::println); ❸
```

- ❶ `Stream<Customer>`
- ❷ `Stream<String>`
- ❸ Sheridan, Ivanova, Garibaldi

Если отображать заказчиков не на имена, а на заказы, то получится коллекция коллекций, как в примере 3.57.

Пример 3.57 ❖ Применение `map` для отображения `Customer` на `orders`

```
customers.stream()           ❶
    .map(Customer::getOrders)  ❷
    .forEach(System.out::println);

customers.stream()
    .map(customer -> customer.getOrders().stream()) ❸
    .forEach(System.out::println);
```

- ❶ `Stream<List<Order>>`
- ❷ `[Order{id=1}, Order{id=2}, Order{id=3}], [Order{id=4}, Order{id=5}], []`
- ❸ `Stream<Stream<Order>>`

Операция отображения порождает поток `Stream<List<Order>>`, в котором последний список пуст. Если вызвать метод `stream` для списка заказов, то получится `Stream<Stream<Order>>`, где последний внутренний поток пуст.

Именно тут вступает в игру метод `flatMap`, имеющий следующую сигнатуру:

```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

Для аргумента универсального типа `T` функция порождает поток `Stream<R>`, а не просто `R`. Затем метод `flatMap` «разглаживает» результирующий поток – удаляет все элементы из отдельных потоков и добавляет их в выходной поток.

☑ Функция `Function`, переданная методу `flatMap`, принимает входной аргумент универсального типа и порождает поток значений выходного типа.

Пример 3.58 ❖ Применение `flatMap` к заказам

```
customers.stream()           ❶
    .flatMap(customer -> customer.getOrders().stream()) ❷
    .forEach(System.out::println); ❸
```

❶ `Stream<Customer>`

❷ `Stream<Order>`

❸ `Order{id=1}, Order{id=2}, Order{id=3}, Order{id=4}, Order{id=5}`

В результате операции `flatMap` получается разглаженный поток `Stream<Order>`, так что вложенные потоки уже не будут доставлять хлопот.

Сформулируем две ключевые идеи `flatMap`:

- аргумент `Function` порождает поток выходных значений;
- результирующий поток потоков разглаживается, так что получается один выходной поток.

Если не забывать об этих моментах, то у метода `flatMap` найдется немало полезных применений.

Напоследок отметим, что в классе `Optional` также имеются методы `map` и `flatMap`. Дополнительные сведения см. в рецептах 6.4 и 6.5.

См. также

Метод `flatMap` демонстрируется также в рецепте 6.5. Метод `flatMap` класса `Optional` обсуждается в рецепте 6.4.

3.12. КОНКАТЕНАЦИЯ ПОТОКОВ

Проблема

Требуется объединить два или более потоков в один.

Решение

Метод `concat` интерфейса `Stream` объединяет два потока. Его стоит применять, если количество потоков невелико. В противном случае используйте `flatMap`.

Обсуждение

Допустим, вы собираете данные из нескольких мест и хотите обработать каждый элемент, применяя потоки. Для этого можно, в частности, воспользоваться методом `concat` интерфейса `Stream` с такой сигнатурой:

```
static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
```

Этот метод создает лениво конкатенируемый поток, который сначала читает все элементы первого потока, а затем все элементы второго. В документации сказано, что результирующий поток упорядочен, если входные потоки упорядочены, и параллелен, если *хотя бы один* из входных потоков параллелен. При закрытии возвращенного потока закрываются оба входных потока.



Оба входных потока должны содержать элементы одного и того же типа.

Пример 3.59 ❖ Конкатенация двух потоков

```
@Test
public void concat() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    List<String> strings = Stream.concat(first, second) ❶
        .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c", "X", "Y", "Z");
    assertEquals(stringList, strings);
}
```

❶ Сначала элементы потока `first`, потом элементы потока `second`

Чтобы добавить еще и третий поток, следует воспользоваться вложенной конкатенацией.

Пример 3.60 ❖ Конкатенация нескольких потоков

```
@Test
public void concatThree() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    List<String> strings = Stream.concat(Stream.concat(first, second), third)
        .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

Этот подход работает, но в документации есть такое примечание:

Будьте осторожны при конструировании потоков повторной конкатенацией. Доступ к элементу глубоко конкатенированного потока ведет к глубоким цепочкам вызовов и может даже закончиться исключением `StackOverflowException`.

Дело в том, что метод `concat`, по существу, строит двоичное дерево потоков, которое может оказаться большим, если потоков слишком много.

Альтернативный подход к конкатенации нескольких потоков – воспользоваться методом `reduce`, как показано в разделе 3.61.

Пример 3.61 ❖ Конкатенация с помощью метода `reduce`

```
@Test
public void reduce() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    List<String> strings = Stream.of(first, second, third, fourth)
        .reduce(Stream.empty(), Stream::concat) ❶
        .collect(Collectors.toList());

    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

❶ Вызов `reduce` с пустым потоком и бинарным оператором

Это работает, потому что метод `concat`, используемый как ссылка на метод, является бинарным оператором. Такой код выглядит проще, но не устраняет возможности переполнения стека. Для объединения потоков более естественно использовать метод `flatMap`, как показано в примере 3.62.

Пример 3.62 ❖ Применение `flatMap` для конкатенации потоков

```
@Test
public void flatMap() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    List<String> strings = Stream.of(first, second, third, fourth)
        .flatMap(Function.identity())
        .collect(Collectors.toList());
    List<String> stringList = Arrays.asList("a", "b", "c",
        "X", "Y", "Z", "alpha", "beta", "gamma");
    assertEquals(stringList, strings);
}
```

У этого подхода тоже есть свои тонкости. Метод `concat` создает параллельный поток, если хотя бы один из входных потоков параллелен, но для `flatMap` это не так (см. пример 3.63).

Пример 3.63 ❖ Параллельный или нет?

```
@Test
public void concatParallel() throws Exception {
```



```

Stream<String> first = Stream.of("a", "b", "c").parallel();
Stream<String> second = Stream.of("X", "Y", "Z");
Stream<String> third = Stream.of("alpha", "beta", "gamma");

Stream<String> total = Stream.concat(Stream.concat(first, second), third);

assertTrue(total.isParallel());
}

@Test
public void flatMapNotParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    Stream<String> total = Stream.of(first, second, third, fourth)
        .flatMap(Function.identity());
    assertFalse(total.isParallel());
}

```

Правда, при желании поток можно сделать параллельным, вызвав метод `parallel`, если только обработка данных еще не началась (см. пример 3.64).

Пример 3.64 ❖ Распараллеливание потока, являющегося результатом `flatMap`

```

@Test
public void flatMapParallel() throws Exception {
    Stream<String> first = Stream.of("a", "b", "c").parallel();
    Stream<String> second = Stream.of("X", "Y", "Z");
    Stream<String> third = Stream.of("alpha", "beta", "gamma");
    Stream<String> fourth = Stream.empty();

    Stream<String> total = Stream.of(first, second, third, fourth)
        .flatMap(Function.identity());
    assertFalse(total.isParallel());

    total = total.parallel();
    assertTrue(total.isParallel());
}

```

Поскольку `flatMap` – промежуточная операция, поток еще можно модифицировать с помощью метода `parallel`.

Короче говоря, метод `concat` эффективен для двух потоков, и его можно использовать в составе общей операции редукции, но `flatMap` является более естественным решением.

См. также

См. замечательную статью по адресу <http://bit.ly/efficient-multistream-concat-entation>, где рассмотрены технические детали, вопросы производительности и многое другое.

Метод `flatMap` интерфейса `Stream` обсуждается в рецепте 3.11.

3.13. ЛЕНИВЫЕ ПОТОКИ

Проблема

Требуется обработать минимальное число элементов потока, необходимое для решения задачи.

Решение

Потоки изначально являются ленивыми и не начинают обработку элементов, пока не встретится терминальная операция. После этого каждый элемент обрабатывается по отдельности. Если в конце находится укорачивающая операция, то обработка потока завершается, как только будут удовлетворены все условия.

Обсуждение

У человека, только приступающего к изучению потоков, часто возникает ощущение, что производится гораздо больше работы, чем необходимо. Возьмем, к примеру, диапазон чисел между 100 и 200, и пусть требуется умножить каждое из них на два, а затем найти первое значение, делящееся на три, как показано в примере 3.65¹.

Пример 3.65 ❖ Первое число от 200 до 400, делящееся на 3

```
OptionalInt firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(n -> n * 2)
    .filter(n -> n % 3 == 0)
    .findFirst();
```

```
System.out.println(firstEvenDoubleDivBy3); ❶
```

❶ Печатается Optional[204]

Неискушенному читателю может показаться, что многое здесь делается зря:

- создается диапазон чисел от 100 до 199 (100 операций);
- каждое число умножается на два (100 операций);
- для каждого числа проверяется, делится ли оно на три (100 операций);
- возвращается первый элемент результирующего потока (1 операция).

Но коль скоро первое число, удовлетворяющее всем требованиям, равно 204, то к чему обрабатывать все остальные?

По счастью, обработка потоков работает иначе. Потоки *ленивые*, т. е. никакая работа не выполняется, пока не встретится терминальная операция, а затем каждый элемент обрабатывается индивидуально. Чтобы продемонстрировать это, в примере 3.66 показан тот же самый код, но переработанный так, чтобы было видно каждый элемент, проходящий по конвейеру.

¹ Благодарю неподражаемого Венката Субраманиама за идею этого примера.

Пример 3.66 ❖ Явная обработка каждого элемента потока

```

public int multByTwo(int n) {           ❶
    System.out.printf("В multByTwo с аргументом %d\n", n);
    return n * 2;
}

public boolean divByThree(int n) {     ❷
    System.out.printf("В divByThree с аргументом %d\n", n);
    return n % 3 == 0;
}

// ...

firstEvenDoubleDivBy3 = IntStream.range(100, 200)
    .map(this::multByTwo)               ❶
    .filter(this::divByThree)          ❷
    .findFirst();

```

- ❶ Ссылка на метод умножения на 2 с печатью
- ❷ Ссылка на метод деления на 3 по модулю с печатью

Теперь печатается вот что:

```

В multByTwo с аргументом 100
В divByThree с аргументом 200
В multByTwo с аргументом 101
В divByThree с аргументом 202
В multByTwo с аргументом 102
В divByThree с аргументом 204
First even divisible by 3 is Optional[204]

```

Значение 100 проходит через `map` и преобразуется в 200, но отбрасывается фильтром, поэтому поток переходит к значению 101. Оно преобразуется в 202, которое также отбрасывается фильтром. Следующее значение отображается в 204, это число делится на 3, поэтому пропускается фильтром. Обработка потока завершается *после обработки всего трех значений*, для чего требуются шесть операций.

Это одно из главных преимуществ потоков, по сравнению с коллекциями. При работе с коллекцией все операции пришлось бы выполнить до перехода к следующему шагу. В случае потоков промежуточные операции образуют конвейер, но, пока не будет достигнута терминальная операция, ничего не происходит. А затем обрабатывается лишь столько элементов потока, сколько необходимо.

Так бывает не всегда – если какая-то операция обладает состоянием, как, например, сортировка или суммирование всех элементов, то, хочешь не хочешь, придется обработать все элементы. Но если последовательность операций без состояния завершается укорачивающей терминальной операцией, то выгода очевидна.

См. также

Различия между методами `findFirst` и `findAny` обсуждаются в рецепте 3.9.

Глава 4

Компараторы и коллекторы

В Java 8 интерфейс `Comparator` дополнен несколькими статическими методами и методами по умолчанию, которые значительно упрощают операции сортировки. Теперь можно сортировать коллекцию объектов сначала по одному свойству, в случае совпадения – по другому, в случае совпадения двух – по третьему и т. д. Для этого достаточно просто написать последовательность библиотечных вызовов.

В Java 8 также добавлен новый служебный класс `java.util.stream.Collectors`, содержащий методы для преобразования потоков в различные коллекции. Возможны подчиненные коллекторы, обрабатывающие результаты операции группировки или разбиения.

В этой главе будут проиллюстрированы все эти новшества.

4.1. СОРТИРОВКА С ПОМОЩЬЮ КОМПАРАТОРА

Проблема

Требуется отсортировать объекты.

Решение

Воспользоваться методом `sorted` интерфейса `Stream`, передав ему компаратор, реализованный в виде лямбда-выражения или сгенерированный одним из статических методов интерфейса `Comparator`.

Обсуждение

Метод `sorted` интерфейса `Stream` порождает новый поток, отсортированный в естественном для класса порядке. Естественный порядок задается с помощью реализации интерфейса `java.util.Comparable`.

Рассмотрим сортировку строк, показанную в примере 4.1.

Пример 4.1 ❖ Сортировка строк в лексикографическом порядке

```
private List<String> sampleStrings =  
    Arrays.asList("this", "is", "a", "list", "of", "strings");
```

```

public List<String> defaultSort() {
    Collections.sort(sampleStrings); ❶
    return sampleStrings;
}

public List<String> defaultSortUsingStreams() {
    return sampleStrings.stream()
        .sorted() ❷
        .collect(Collectors.toList());
}

```

- ❶ Сортировка по умолчанию, принятая в Java 7 и предыдущих версиях
- ❷ Сортировка по умолчанию, принятая в Java 8 и последующих версиях

Служебный класс `Collections` существовал в Java еще со времен версии 1.2, когда в язык были добавлены коллекции. Статический метод `sort` класса `Collections` принимает в качестве аргумента `List` и возвращает `void`. В процессе сортировки коллекция модифицируется. Этот подход не согласуется с функциональными принципами, провозглашенными в Java 8 и приветствующими неизменяемость.

В Java 8 для такой же сортировки к потоку применяется метод `sorted`, но при этом порождается новый поток, а не модифицируется исходная коллекция. В данном примере возвращенный список будет отсортирован в естественном для класса порядке. Для строк естественным является лексикографический порядок, который сводится к алфавитному в случае, когда все строки состоят из символов в нижнем регистре.

Чтобы отсортировать строки по-другому, следует воспользоваться перегруженным вариантом метода `sorted`, который принимает в качестве аргумента объект типа `Comparator`.

В примере 4.2 показана сортировка строк по длине двумя способами.

Пример 4.2 ❖ Сортировка строк по длине

```

public List<String> lengthSortUsingSorted() {
    return sampleStrings.stream()
        .sorted((s1, s2) -> s1.length() - s2.length()) ❶
        .collect(toList());
}

public List<String> lengthSortUsingComparator() {
    return sampleStrings.stream()
        .sorted(Comparator.comparingInt(String::length)) ❷
        .collect(toList());
}

```

- ❶ Лямбда-выражение в качестве компаратора для сортировки по длине
- ❷ Метод `comparingInt` в качестве компаратора

Аргументом метода `sorted` является объект, реализующий функциональный интерфейс `java.util.Comparator`. В методе `lengthSortUsingSorted` метод `compare` интерфейса `Comparator` реализован лямбда-выражением. В Java 7 и более ранних

версиях это делалось бы с помощью анонимного внутреннего класса, но теперь вполне достаточно лямбда-выражения.

i В Java 8 в интерфейс добавлен метод по умолчанию `sort(Comparator)`, эквивалентный статическому методу `static void sort(List, Comparator)` интерфейса `Collections`. Тот и другой модифицируют переданную коллекцию и возвращают `void`, поэтому следует все же предпочесть обсуждаемый в этом рецепте метод потока `sorted(Comparator)` (который возвращает новый отсортированный поток).

Во втором методе, `lengthSortUsingComparator`, используются статические методы, добавленные в интерфейс `Comparator`. Метод `comparingInt` принимает аргумент типа `ToIntFunction`, который преобразует строку в целочисленный ключ (в документации он называется `keyExtractor`) и генерирует компаратор, сортирующий коллекцию по этому ключу.

Добавленные в интерфейс `Comparator` методы по умолчанию чрезвычайно полезны. Написать компаратор для сортировки строк по длине легко, но если требуется сортировать по нескольким полям, то задача усложняется. Взять, к примеру, сортировку строк по длине, а в случае равной длины – по алфавиту. Благодаря методам интерфейса `Comparator` – статическим и по умолчанию – эта задача оказывается почти тривиальной.

Пример 4.3 ❖ Сортировка сначала по длине, затем лексикографически

```
public List<String> lengthSortThenAlphaSort() {
    return sampleStrings.stream()
        .sorted(comparing(String::length)) ❶
        .thenComparing(naturalOrder())
        .collect(toList());
}
```

❶ Сортировка строк по длине, а в случае равной длины – по алфавиту

В интерфейсе `Comparator` имеется метод по умолчанию `thenComparing`. Как и `comparing`, он принимает аргумент `Function`, который также называется в документации `keyExtractor`. Его сцепление с методом `comparing` возвращает компаратор, который сравнивает объекты сначала по первому полю, в случае равенства – по второму и т. д.

Статический импорт часто упрощает чтение кода. Привыкнув к статическим методам `Comparator` и `Collectors`, вы будете часто прибегать к этому способу. В данном случае методы `comparing` и `naturalOrder` были статически импортированы.

Этот подход работает для любого класса, даже если он не реализует интерфейс `Comparable`. Рассмотрим класс `Golfer`, показанный в примере 4.4.

Пример 4.4 ❖ Класс, описывающий игрока в гольф

```
public class Golfer {
    private String first;
    private String last;
```

```
private int score;
// ... прочие методы ...
}
```

Для создания турнирной таблицы имеет смысл отсортировать игроков сначала по рейтингу, затем по фамилии и, наконец, по имени. В примере 4.5 показано, как это сделать.

Пример 4.5 ❖ Сортировка гольфистов

```
private List<Golfer> golfers = Arrays.asList(
    new Golfer("Джек", "Никлаус", 68),
    new Golfer("Тайгер", "Вудс", 70),
    new Golfer("Том", "Уотсон", 70),
    new Golfer("Тай", "Уэбб", 68),
    new Golfer("Бубба", "Уотсон", 70)
);

public List<Golfer> sortByScoreThenLastThenFirst() {
    return golfers.stream()
        .sorted(comparingInt(Golfer::getScore)
            .thenComparing(Golfer::getLast)
            .thenComparing(Golfer::getFirst))
        .collect(toList());
}
```

Результат вызова `sortByScoreThenLastThenFirst` показан в примере 4.6.

Пример 4.6 ❖ Отсортированные гольфисты

```
Golfer{first='Джек', last='Никлаус', score=68}
Golfer{first='Тай', last='Уэбб', score=68}
Golfer{first='Тайгер', last='Вудс', score=70}
Golfer{first='Бубба', last='Уотсон', score=70}
Golfer{first='Том', last='Уотсон', score=70}
```

Гольфисты сортируются сначала по рейтингу, поэтому Никлаус и Уэбб оказались раньше Вудса и обоих Уотсонов¹. Затем игроки с равным рейтингом сортируются по фамилии, так что Никлаус оказывается раньше Уэбба, а Вудс – раньше Уотсонов. Наконец, игроки с одинаковыми рейтингами и фамилиями сортируются по имени, так что Бубба Уотсон оказывается раньше Тома Уотсона.

Добавленные в интерфейс `Comparator` статические методы и методы по умолчанию в сочетании с новым методом `sorted` интерфейса `Stream` заметно упрощают выполнение сложных сортировок.

¹ Тай Уэбб, конечно, персонаж фильма «Гольф-клуб». Судья Смайлз: «Тай, сколько ты набрал сегодня?» Тай Уэбб: «Да ну, судья, я счет не веду». Смайлз: «Тогда как же ты меряешься с другими игроками?» Уэбб: «По росту». Добавление сортировки по росту оставляю читателям в качестве упражнения.

4.2. ПРЕОБРАЗОВАНИЕ ПОТОКА В КОЛЛЕКЦИЮ

Проблема

Обработанный поток требуется преобразовать в `List`, `Set` или другую линейную коллекцию.

Решение

Воспользоваться методами `toList`, `toSet` или `toCollection` служебного класса `Collectors`.

Обсуждение

В Java 8 часто встречается идиома передачи элементов потока по конвейеру промежуточных операций, заканчивающемуся терминальной операцией. Одной из терминальных операций является метод `collect`, который служит для преобразования потока в коллекцию.

У метода `collect` есть два перегруженных варианта в интерфейсе `Stream`, они приведены в примере 4.7.

Пример 4.7 ❖ Метод `collect` интерфейса `Stream<T>`

```
<R,A> R collect(Collector<? super T,A,R> collector)
<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```

В этом рецепте мы рассмотрим первый вариант, принимающий аргумент типа `Collector`. Коллекторы выполняют «изменяющую операцию редукции», которая аккумулирует элементы в результирующем контейнере. В данном случае результатом будет коллекция.

`Collector` – интерфейс, поэтому создать объект такого типа невозможно. В интерфейсе имеется статический метод `of` для порождения объектов, но часто существует лучший или, по крайней мере, более простой способ.



В Java 8 API нередко встречаются статические методы с именем `of`, используемые в качестве фабричных.

Здесь мы воспользуемся статическими методами класса `Collectors` с целью порождения экземпляров `Collector`, которые будут переданы методу `Stream.collect` для заполнения коллекции.

В примере 4.8 показано, как создается список `List`¹.

¹ Имена в этом рецепте взяты из одного из самых недооцененных фильмов 1990-х годов «Таинственные люди». (М-р Яростный: «Лэнс Хант и есть Капитан Изумительный». Землекоп: «Лэнс Хант носит очки. Капитан Изумительный не носит очков». М-р Яростный: «Он их снимает, когда преображается». Землекоп: «Но это же бессмысленно! Тогда он бы ничего не *видел!*»)

Пример 4.8 ❖ Создание списка

```
List<String> superHeroes =
    Stream.of("Mr. Furious", "The Blue Raja", "The Shoveler",
             "The Bowler", "Invisible Boy", "The Spleen", "The Sphinx")
        .collect(Collectors.toList());
```

Этот метод создает и заполняет список `ArrayList` элементами заданного потока. В примере 4.9 показано столь же простое создание множества `Set`.

Пример 4.9 ❖ Создание множества

```
Set<String> villains =
    Stream.of("Casanova Frankenstein", "The Disco Boys",
             "The Not-So-Goodie Mob", "The Suits", "The Suzies",
             "The Furriers", "The Furriers")
        .collect(Collectors.toSet());
}
```

❶ Повторяющееся имя, в процессе преобразования во множество удаляется

Этот метод создает и заполняет экземпляр `HashSet`, устраняя дубликаты.

В обоих примерах используются реализации интерфейсов, подразумеваемые по умолчанию: `ArrayList` для `List` и `HashSet` для `Set`. Если вы хотите сами указать структуру данных, то должны использовать метод `Collectors.toCollection`, который принимает аргумент `Supplier`.

Пример 4.10 ❖ Создание связанного списка

```
List<String> actors =
    Stream.of("Hank Azaria", "Janeane Garofalo", "William H. Macy",
             "Paul Reubens", "Ben Stiller", "Kel Mitchell", "Wes Studi")
        .collect(Collectors.toCollection(LinkedList::new));
}
```

Аргументом метода `toCollection` является поставщик коллекции, поэтому в данном случае указана ссылка на конструктор класса `LinkedList`. Метод `collect` создает экземпляр `LinkedList` и заполняет его заданными именами.

В классе `Collectors` имеются также средства для создания массива объектов, а именно два перегруженных варианта метода `toArray`:

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

Первый вариант возвращает массив, содержащий элементы потока без указания типа. Второй вариант принимает функцию, которая порождает новый массив указанного типа, длина которого равна размеру потока; его проще всего использовать, передав ссылку на конструктор массива, как показано в примере 4.11.

Пример 4.11 ❖ Создание массива

```
String[] wannabes =
    Stream.of("The Waffler", "Reverse Psychologist", "PMS Avenger")
```

```
        .toArray(String[]::new); ❶
    }
```

❶ Ссылка на конструктор массива в качестве `Supplier`

Для преобразования в отображение `Map` применяется метод `Collectors.toMap`, которому передаются два экземпляра `Function`: для ключей и для значений.

Рассмотрим простой класс `Actor`, обертывающий имя актера `name` и роль `role`. Если имеется множество `Set` объектов `Actor`, описывающих актеров, занятых в фильме, то код в примере 4.12 создаст из них отображение `Map`.

Пример 4.12 ❖ Создание отображения

```
Set<Actor> actors = mysteryMen.getActors();
Map<String, String> actorMap = actors.stream()
    .collect(Collectors.toMap(Actor::getName, Actor::getRole)); ❶
actorMap.forEach((key, value) ->
    System.out.printf("%s played %s\n", key, value));
```

❶ Функции, порождающие ключи и значения

В результате будет напечатано:

```
Janeane Garofalo played The Bowler
Greg Kinnear played Captain Amazing
William H. Macy played The Shovelor
Paul Reubens played The Spleen
Wes Studi played The Sphinx
Kel Mitchell played Invisible Boy
Geoffrey Rush played Casanova Frankenstein
Ben Stiller played Mr. Furious
Hank Azaria played The Blue Raja
```

Аналогичный код работает для создания `ConcurrentMap` методом `toConcurrentMap`.

См. также

Поставщики `Supplier` обсуждаются в рецепте 2.2, а ссылки на конструкторы – в рецепте 1.3. Метод `toMap` демонстрируется также в рецепте 4.3.

4.3. ДОБАВЛЕНИЕ ЛИНЕЙНОЙ КОЛЛЕКЦИИ В ОТОБРАЖЕНИЕ

Проблема

Требуется добавить коллекцию объектов в отображение `Map`, так чтобы ключом было одно из свойств объекта, а значением – сам объект.

Решение

Использовать метод `toMap` класса `Collectors` в сочетании с функцией `Function.identity`.

Обсуждение

Это весьма частная задача, но когда она возникает, приведенное здесь решение очень удобно.

Пусть имеется список `List` объектов класса `Book`, в котором имеются идентификатор, название книги и цена. Часть класса `Book` приведена в примере 4.13.

Пример 4.13 ❖ Простой класс, представляющий книгу

```
public class Book {
    private int id;
    private String name;
    private double price;

    // ... прочие методы ...
}
```

В примере 4.14 показана коллекция таких объектов.

Пример 4.14 ❖ Коллекция книг

```
List<Book> books = Arrays.asList(
    new Book(1, "Modern Java Recipes", 49.99),
    new Book(2, "Java 8 in Action", 49.99),
    new Book(3, "Java SE8 for the Really Impatient", 39.99),
    new Book(4, "Functional Programming in Java", 27.64),
    new Book(5, "Making Java Groovy", 45.99)
    new Book(6, "Gradle Recipes for Android", 23.76)
);
```

Во многих ситуациях вместо списка нужно отображение, ключами которого являются идентификаторы книг, а значениями – сами книги. Это легко сделать с помощью метода `toMap` класса `Collectors` – и даже двумя способами, как показано в примере 4.15.

Пример 4.15 ❖ Добавление книг в отображение

```
Map<Integer, Book> bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, b -> b));           ❶

bookMap = books.stream()
    .collect(Collectors.toMap(Book::getId, Function.identity())); ❷
```

- ❶ Тожественное лямбда-выражение: возвращает тот же элемент, что получает
- ❷ Статический метод `identity` интерфейса `Function` делает то же самое

Метод `toMap` принимает два экземпляра `Function`: первый генерирует по заданному объекту ключи, второй – значения. В данном случае ключ возвращает метод `getId` объекта `Book`, а значением является сама книга.

В первом вызове `toMap` в примере 4.16 для получения ключа используется метод `getId`, а для получения значения – тождественное лямбда-выражение, которое просто возвращает свой параметр. Во втором вызове то же самое делается с помощью статического метода `identity` интерфейса `Function`.

Два статических метода identity

Статический метод `identity` интерфейса `Function` имеет сигнатуру

```
static <T> Function<T,T> identity()
```

В примере 4.16 показана его реализация в стандартной библиотеке.

Пример 4.16 ❖ Статический метод `identity` интерфейса `Function`

```
static <T> Function<T, T> identity() {
    return t -> t;
}
```

Интерфейс `UnaryOperator` расширяет `Function`, но переопределять статический метод нельзя. Согласно документации, в нем также определен статический метод `identity`:

```
static <T> UnaryOperator<T> identity()
```

Его реализация в стандартной библиотеке практически такая же, как в примере 4.17.

Пример 4.17 ❖ Статический метод `identity` интерфейса `UnaryOperator`

```
static <T> UnaryOperator<T> identity() {
    return t -> t;
}
```

Разница – лишь в способе вызова (с указанием разных имен интерфейсов) и в типе возвращаемого значения. В данном случае не важно, какой метод использовать, но знать, что их два, полезно.

Что предпочесть – лямбда-выражение или статический метод, – дело вкуса. Как бы то ни было, легко построить отображение, в котором ключом является некоторое свойство объекта, а значением – сам объект.

См. также

Функции рассматриваются в рецепте 2.4, где обсуждаются также унарные и бинарные операторы.

4.4. СОРТИРОВКА ОТОБРАЖЕНИЙ

Проблема

Требуется отсортировать отображение по ключу или по значению.

Решение

Использовать новые статические методы интерфейса `Map.Entry`.

Обсуждение

Интерфейс `Map` всегда содержал открытый статический внутренний интерфейс `Map.Entry`, представляющий пару ключ-значение. Метод `Map.entrySet` возвращает множество элементов типа `Map.Entry`. До версии Java 8 основными методами этого интерфейса были `getKey` и `getValue` – в полном соответствии с ожиданиями.

В Java 8 добавлены статические методы, перечисленные в табл. 4.1.

Таблица 4.1. Статические методы интерфейса `Map.Entry` (из документации Java 8)

Метод	Описание
<code>comparingByKey()</code>	Возвращает компаратор, который сравнивает <code>Map.Entry</code> в естественном порядке ключей
<code>comparingByKey(Comparator<? super K> cmp)</code>	Возвращает компаратор, который сравнивает <code>Map.Entry</code> по ключу с использованием заданного объекта <code>Comparator</code>
<code>comparingByValue()</code>	Возвращает компаратор, который сравнивает <code>Map.Entry</code> в естественном порядке значений
<code>comparingByValue(Comparator<? super V> cmp)</code>	Возвращает компаратор, который сравнивает <code>Map.Entry</code> по значению с использованием заданного объекта <code>Comparator</code>

Для демонстрации этих методов в примере 4.18 генерируется отображение длины слова на количество слов такой длины в словаре. В любой Unix-системе имеется файл `/usr/share/dict/words`, содержащий весь словарь Вебстера (2-е издание), по одному слову на строку. Для чтения этого файла можно воспользоваться методом `Files.lines`, который порождает поток строк, содержащих слова.

Пример 4.18 ❖ Чтение файла словаря в отображение

```
System.out.println("\nРаспределение числа слов по длинам:");
try (Stream<String> lines = Files.lines(dictionary)) {
    lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()))
        .forEach((len, num) -> System.out.printf("%d: %d\n", len, num));
} catch (IOException e) {
    e.printStackTrace();
}
```

Этот пример обсуждается в рецепте 7.1, но кратко опишем, что здесь происходит.

- Файл читается внутри блока `try` с ресурсами. `Stream` реализует интерфейс `AutoCloseable`, поэтому при выходе из блока автоматически вызывается метод `close` потока, который, в свою очередь, вызывает метод `close` объекта `File`.
- Фильтр ограничивает обработку только словами длиной не более 20 символов.

- Методу `groupingBy` класса `Collectors` в первом аргументе передается функция, представляющая классификатор. Здесь классификация производится по длине строки. Если передать только один аргумент, то результатом будет отображение `Map`, ключами которого являются выделенные классификатором категории, а значениями – списки элементов в каждой категории. В нашем примере вызов `groupingBy(String::length)` породил бы отображение `Map<Integer, List<String>>`, в котором ключами являются длины слов, а значениями – списки слов данной длины.
- Вариант `groupingBy` с двумя аргументами принимает еще один коллектор, называемый *подчиненным*, который осуществляет постобработку списка слов. В данном случае возвращается отображение `Map<Integer, Long>`, в котором ключом является длина слова, а значением – количество слов данной длины в словаре.

В результате будет напечатано:

Распределение числа слов по длинам:

```
21: 82
22: 41
23: 17
24: 5
```

Иными словами, существует 82 слова длины 21, 41 слово длины 22, 17 слов длины 23 и 5 слов длины 24¹.

Как видим, отображение печатается в порядке возрастания длины слова. Чтобы напечатать его в порядке убывания, следует воспользоваться методом `Map.Entry.comparingByKey`, как в примере 4.19.

Пример 4.19 ❖ Сортировка отображения по ключу

```
System.out.println("\nРаспределение числа слов по длинам (в порядке убывания):");
try (Stream<String> lines = Files.lines(dictionary)) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(
            String::length, Collectors.counting()));

    map.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
        .forEach(e -> System.out.printf("Длина %d: %2d слов%n",
            e.getKey(), e.getValue()));
} catch (IOException e) {
    e.printStackTrace();
}
```

После вычисления `Map<Integer, Long>` эта операция получает `entrySet` и порождает поток. Метод потока `sorted` порождает отсортированный поток, применяя переданный компаратор.

¹ Для справки приведем эти пять самых длинных слов: `formaldehydesulphoxylate`, `pathologicopsychological`, `scientificphilosophical`, `tetraiodophenolphthalein`, `thyroparathyroidectomize`. Успехов в проверке орфографии!

В данном случае метод `Map.Entry.comparingByKey` генерирует компаратор, который сортирует по ключу, а использование перегруженного варианта, принимающего компаратор, позволяет указать, что мы хотим отсортировать в обратном порядке.

i Метод `sorted` интерфейса `Stream` порождает новый отсортированный поток, не модифицируя исходный. Исходное отображение остается неизменным.

В результате будет напечатано:

Распределение числа слов по длинам (в порядке убывания):

Длина 24: 5 слов

Длина 23: 17 слов

Длина 22: 41 слов

Длина 21: 82 слов

Другие методы сортировки, представленные в табл. 4.1, работают аналогично.

См. также

Еще один пример сортировки отображения по ключу или значению приведен в приложении А. Подчиненные коллекторы обсуждаются в рецепте 4.6. Файловые операции со словарем рассматриваются в рецепте 7.1.

4.5. РАЗБИЕНИЕ И ГРУППИРОВКА

Проблема

Требуется распределить элементы коллекции по категориям.

Решение

Метод `Collectors.partitioningBy` разделяет элементы на те, что удовлетворяют предикату, и все остальные. Метод `Collectors.groupingBy` порождает отображение `Map`, ключами которого являются категории, а значениями – элементы, принадлежащие одной категории.

Обсуждение

Пусть имеется коллекция строк, и мы хотим разбить на две части: строки четной и нечетной длины. Для этого можно воспользоваться методом `Collectors.partitioningBy`, как в примере 4.20.

Пример 4.20 ❖ Разбиение строк по четности длины

```
List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
    "strings", "to", "use", "as", "a", "demo");
```

```
Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0)); i
```

```
lengthMap.forEach((key,value) -> System.out.printf("%5s: %s%n", key, value));
//
// false: [a, strings, use, a]
// true: [this, is, long, list, of, to, as, demo]
```

❶ Разбиение на строки четной и нечетной длины

Ниже приведены сигнатуры вариантов метода `partitioningBy`:

```
static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(
    Predicate<? super T> predicate)
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

Тип возвращаемого значения выглядит малопонятно из-за использования универсальных типов, но на практике вам редко придется выписывать его явно. Обычно результат операции передается методу `collect`, который применяет сгенерированный коллектор для создания выходного отображения, определяемого третьим универсальным аргументом.

Первый вариант метода `partitioningBy` принимает в качестве аргумента предикат и разделяет элементы коллекции на две группы: удовлетворяющие и не удовлетворяющие предикату. На выходе всегда получается отображение `Map`, содержащее ровно две записи: список значений, удовлетворяющих предикату, и список прочих значений.

Другой вариант метода принимает второй аргумент типа `Collector`, который называется *подчиненным коллектором* (`downstream collector`). Это позволяет выполнить постобработку списков, созданных в результате разбиения, и обсуждается в рецепте 4.6.

Метод `groupingBy` выполняет операцию, аналогичную команде SQL «group by». Он возвращает отображение, ключами которого являются группы, а значениями – списки элементов в каждой группе.

i Если вы получаете данные из базы данных, то вполне можете выполнить операцию группировки средствами базы. Новые методы предназначены для того, чтобы было удобно делать это с данными в памяти.

Сигнатура метода `groupingBy` имеет вид:

```
static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(
    Function<? super T,? extends K> classifier)
```

Переданная функция принимает элементы потока и извлекает из них свойство, по которому нужно группировать. В качестве примера рассмотрим не просто разбиение строк на две категории, а группировку их по длине, как в примере 4.21.

Пример 4.21 ❖ Группировка строк по длине

```
List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
    "strings", "to", "use", "as", "a", "demo");
Map<Integer, List<String>> lengthMap = strings.stream()
```



```

    .collect(Collectors.groupingBy(String::length)); ❶
lengthMap.forEach((k,v) -> System.out.printf("%d: %s\n", k, v));
//
// 1: [a, a]
// 2: [is, of, to, as]
// 3: [use]
// 4: [this, long, list, demo]
// 7: [strings]

```

❶ Группировка строк по длине

Ключами результирующего отображения являются длины строк (1, 2, 3, 4, 7), а значениями – списки строк каждой длины.

См. также

В рецепте 4.6 обсуждается продолжение этого рецепта – как произвести пост-обработку списков, возвращенных операцией `groupingBy` или `partitioningBy`.

4.6. ПОДЧИНЕННЫЕ КОЛЛЕКТОРЫ

Проблема

Требуется произвести постобработку коллекций, возвращенных операцией `groupingBy` или `partitioningBy`.

Решение

Воспользоваться одним из статических методов служебного класса `java.util.stream.Collectors`.

Обсуждение

В рецепте 4.5 мы рассмотрели, как разбить элементы на несколько категорий. Методы `partitioningBy` и `groupingBy` возвращают отображение `Map`, ключами которого являются категории (булевы значения `true` и `false` в случае `partitioningBy`, объекты в случае `groupingBy`), а значениями – списки элементов, попадающих в категорию. В примере 4.22 для удобства повторено разбиение на строки четной и нечетной длины из примера 4.20.

Пример 4.22 ❖ Разбиение строк по четности длины

```

List<String> strings = Arrays.asList("this", "is", "a", "long", "list", "of",
    "strings", "to", "use", "as", "a", "demo");

Map<Boolean, List<String>> lengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0));

lengthMap.forEach((key,value) -> System.out.printf("%5s: %s\n", key, value));
//
// false: [a, strings, use, a]
// true: [this, is, long, list, of, to, as, demo]

```

Но, возможно, нас интересуют не сами списки, а количество элементов в каждой категории. Иными словами, требуется построить отображение, в котором значением будет не список `List<String>`, а просто число элементов в списке. У метода `partitioningBy` имеется перегруженный вариант, принимающий второй аргумент типа `Collector`:

```
static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(
    Predicate<? super T> predicate, Collector<? super T,A,D> downstream)
```

Тут-то и приходит на помощь метод `Collectors.counting`. В примере 4.23 показано, как он работает.

Пример 4.23 ❖ Подсчет количества строк в каждой категории

```
Map<Boolean, Long> numberLengthMap = strings.stream()
    .collect(Collectors.partitioningBy(s -> s.length() % 2 == 0,
        Collectors.counting())); ❶

numberLengthMap.forEach((k,v) -> System.out.printf("%5s: %d%n", k, v));
//
// false: 4
// true: 8
```

❶ Подчиненный коллектор

Мы имеем так называемый *подчиненный коллектор*, который производит постобработку результирующих списков (после завершения операции разбиения).

У метода `groupingBy` также имеется перегруженный вариант, принимающий подчиненный коллектор:

```
/**
 * @param <T> тип входных элементов
 * @param <K> тип ключей
 * @param <A> тип промежуточного аккумулятора в подчиненном коллекторе
 * @param <D> тип результата подчиненной редукции
 * @param classifier функция классификации, отображающая входные элементы на ключи
 * @param downstream {@code Collector}, реализующий подчиненную редукцию
 * @return {@code Collector}, реализующий каскадную операцию group-by
 */
static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(
    Function<? super T,? extends K> classifier,
    Collector<? super T,A,D> downstream)
```

В сигнатуре включена часть документации из исходного кода, согласно которой `T` – тип элемента коллекции, `K` – тип ключа результирующего отображения, `A` – аккумулятор, а `D` – тип подчиненного коллектора. Знак `?` означает «неизвестно». Дополнительные сведения об универсальных типах в Java 8 см. в приложении А.

У некоторых методов интерфейса `Stream` есть аналоги в классе `Collectors`. В табл. 4.2 приведено соответствие между ними.

Таблица 4.2. Соответствие между методами класса *Collectors* и интерфейса *Stream*

Stream	Collectors
count	counting
map	mapping
min	minBy
max	maxBy
IntStream.sum	summingInt
DoubleStream.sum	summingDouble
LongStream.sum	summingLong
IntStream.summarizing	summarizingInt
DoubleStream.summarizing	summarizingDouble
LongStream.summarizing	summarizingLong

И снова напомним, что назначение подчиненного коллектора – постобработка коллекции объектов, порожденной предшествующей операцией, например разбиения или группировки.

См. также

В рецепте 7.1 приведен пример подчиненного коллектора для нахождения самых длинных слов в словаре. В рецепте 4.5 более подробно обсуждаются методы `partitionBy` и `groupingBy`. Универсальные типы в целом рассматриваются в приложении А.

4.7. НАХОЖДЕНИЕ МИНИМАЛЬНОГО И МАКСИМАЛЬНОГО ЗНАЧЕНИЙ

Проблема

Требуется найти минимальное и максимальное значения в потоке.

Решение

Есть несколько возможностей: методы `maxBy` и `minBy` интерфейса `BinaryOperator`, методы `max` и `min` интерфейса `Stream` или методы `maxBy` и `minBy` служебного класса `Collectors`.

Обсуждение

`BinaryOperator` – один из функциональных интерфейсов в пакете `java.util.function`. Он расширяет `BiFunction` и применяется, когда оба аргумента функции и возвращаемое ей значение имеют один и тот же тип.

Интерфейс `BinaryOperator` добавляет два статических метода:

```
static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
```

Оба они возвращают `BinaryOperator`, который использует переданный компаратор. Для демонстрации различных способов нахождения максимального значения в потоке рассмотрим класс `Employee` с тремя атрибутами: `name`, `salary` и `department`.

Пример 4.24 ❖ Класс `Employee`

```
public class Employee {
    private String name;
    private Integer salary;
    private String department;

    // ... прочие методы ...
}
```

```
List<Employee> employees = Arrays.asList( ❶
    new Employee("Cersei", 250_000, "Lannister"),
    new Employee("Jamie", 150_000, "Lannister"),
    new Employee("Tyrion", 1_000, "Lannister"),
    new Employee("Tywin", 1_000_000, "Lannister"),
    new Employee("Jon Snow", 75_000, "Stark"),
    new Employee("Robb", 120_000, "Stark"),
    new Employee("Eddard", 125_000, "Stark"),
    new Employee("Sansa", 0, "Stark"),
    new Employee("Arya", 1_000, "Stark"));
```

```
Employee defaultEmployee = ❷
    new Employee("A man (or woman) has no name", 0, "Black and White");
```

- ❶ Коллекция работников
- ❷ Значение по умолчанию в случае, когда поток пуст

Имея коллекцию работников, мы можем воспользоваться методом `reduce` интерфейса `Stream`, который принимает `BinaryOperator`. Фрагмент кода в примере 4.25 показывает, как найти работника с максимальной зарплатой.

Пример 4.25 ❖ Использование метода `BinaryOperator.maxBy`

```
Optional<Employee> optionalEmp = employees.stream()
    .reduce(BinaryOperator.maxBy(Comparator.comparingInt(Employee::getSalary)));

System.out.println("Работник с максимальной зарплатой: " +
    optionalEmp.orElse(defaultEmployee));
```

Методу `reduce` нужен объект типа `BinaryOperator`. Статический метод `maxBy` порождает бинарный оператор на основе переданного ему компаратора, который в данном случае сравнивает работников по зарплате.

Это работает, но можно поступить проще – воспользоваться вспомогательным методом `max`, который применим непосредственно к потоку:

```
Optional<T> max(Comparator<? super T> comparator)
```

Этот подход демонстрируется в примере 4.26.

Пример 4.26 ❖ Использование метода `Stream.max`

```
optionalEmp = employees.stream()
    .max(Comparator.comparingInt(Employee::getSalary));
```

Результат получается точно такой же.

Отметим, что методы `max` имеются и в потоках примитивных типов (`IntStream`, `LongStream` и `DoubleStream`), они не нуждаются ни в каких аргументах (см. пример 4.27).

Пример 4.27 ❖ Нахождение наибольшей зарплаты

```
OptionalInt maxSalary = employees.stream()
    .mapToInt(Employee::getSalary)
    .max();
System.out.println("Наибольшая зарплата равна " + maxSalary);
```

В этом случае метод `mapToInt` используется, чтобы преобразовать поток работников в поток целых чисел `IntStream` путем вызова метода `getSalary`. Затем вызывается метод `max`, который возвращает `OptionalInt`.

В служебном классе `Collectors` также существует статический метод `maxBy`. Его можно вызвать напрямую, как показано в примере 4.28.

Пример 4.28 ❖ Использование метода `Collectors.maxBy`

```
optionalEmp = employees.stream()
    .collect(Collectors.maxBy(Comparator.comparingInt(Employee::getSalary)));
```

Но это громоздко, лучше воспользоваться методом `max` интерфейса `Stream`, как показано в предыдущем примере. Метод `maxBy` класса `Collectors` полезен в роли подчиненного коллектора (т. е. для постобработки результата операций группировки или разбиения). В примере 4.29 метод `groupingBy` интерфейса `Stream` вызывается для создания отображения отделов на списки работников, а затем в каждом отделе ищется работник с наибольшей зарплатой.

Пример 4.29 ❖ Использование метода `Collectors.maxBy` в роли подчиненного коллектора

```
Map<String, Optional<Employee>> map = employees.stream()
    .collect(Collectors.groupingBy(
        Employee::getDepartment,
        Collectors.maxBy(
            Comparator.comparingInt(Employee::getSalary))));
map.forEach((house, emp) ->
    System.out.println(house + ": " + emp.orElse(defaultEmployee)));
```

Во всех этих классах метод `minBy` работает аналогично.

См. также

Функции обсуждаются в рецепте 2.4, подчиненные коллекторы – в рецепте 4.6.

4.8. СОЗДАНИЕ НЕИЗМЕНЯЕМЫХ КОЛЛЕКЦИЙ

Проблема

Требуется создать неизменяемый список, множество или отображение средствами API потоков.

Решение

Воспользоваться новым статическим методом `collectingAndThen` класса `Collectors`.

Обсуждение

Функциональное программирование с его акцентом на распараллеливании и достижении ясности кода отдает предпочтение неизменяемым объектам. В подсистеме коллекций, добавленной в Java 1.2, всегда существовали методы для создания неизменяемых коллекций на основе существующих, хотя делалось это несколько неуклюже.

В служебном классе `Collections` имеются методы `unmodifiableList`, `unmodifiableSet` и `unmodifiableMap` (и ряд других с префиксом `unmodifiable`).

Пример 4.30 ❖ Методы с префиксом `unmodifiable` в классе `Collections`

```
static <T> List<T> unmodifiableList(List<? extends T> list)
static <T> Set<T> unmodifiableSet(Set<? extends T> s)
static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
```

Во всех случаях аргументом метода является существующая коллекция (список, множество или отображение), и в результате получается коллекция того же вида и с теми же самыми элементами, но обладающая важным отличием: все методы, которые могли бы модифицировать коллекцию, например `add` или `remove`, теперь возбуждают исключение `UnsupportedOperationException`.

До версии Java 8, если передавались отдельные значения в виде списка аргументов переменной длины, то немодифицируемый список или множество создавались, как показано в примере 4.31.

Пример 4.31 ❖ Создание немодифицируемого списка или множества до Java 8

```
@SafeVarargs ❶
public final <T> List<T> createImmutableListJava7(T... elements) {
    return Collections.unmodifiableList(Arrays.asList(elements));
}

@SafeVarargs ❶
public final <T> Set<T> createImmutableSetJava7(T... elements) {
    return Collections.unmodifiableSet(new HashSet<>(Arrays.asList(elements)));
}
```

❶ Мы обещаем не портить тип входного массива. Детали см. в приложении A

В обоих случаях идея заключается в том, чтобы сначала преобразовать входные аргументы в список `List`. Затем этот список можно обернуть немодифицируемым списком `unmodifiableList` или – в случае `Set` – передать список конструктору множества, которое затем обортывается объектом `unmodifiableSet`.

Новый потоковый API в Java 8 позволяет вместо этого воспользоваться статическим методом `Collectors.collectingAndThen`, как показано в примере 4.32.

Пример 4.32 ❖ Создание немодифицируемых списков и множеств в Java 8

```
import static java.util.stream.Collectors.collectingAndThen;
import static java.util.stream.Collectors.toList;
import static java.util.stream.Collectors.toSet;

// ... определить класс со следующими методами ...

@SafeVarargs
public final <T> List<T> createImmutableList(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toList(),
            Collections::unmodifiableList)); ❶
}

@SafeVarargs
public final <T> Set<T> createImmutableSet(T... elements) {
    return Arrays.stream(elements)
        .collect(collectingAndThen(toSet(),
            Collections::unmodifiableSet)); ❶
}
```

❶ «Завершитель» обортывает сгенерированную коллекцию

Метод `Collectors.collectingAndThen` принимает два аргумента: подчиненный коллектор и функцию `Function`, называемую *завершителем* (*finisher*). Идея в том, чтобы сначала создать поток входных элементов, затем собрать их в список или множество и, наконец, применить к результирующей коллекции функцию, делающую ее немодифицируемой.

Преобразование последовательности входных элементов в немодифицируемое отображение `Map` не столь очевидно, отчасти потому что не ясно, какие из входных элементов должны быть ключами, а какие – значениями. Код в примере 4.33¹ создает немодифицируемое отображение очень коряво, применяя инициализатор экземпляра.

Пример 4.33 ❖ Создание немодифицируемого отображения

```
Map<String, Integer> map = Collections.unmodifiableMap(
    new HashMap<String, Integer>() {{
        put("have", 1);
    }});
```

¹ Из статьи Карла Мартенсена «Java 9's Immutable Collections Are Easier To Create But Use With Caution» (<http://carlmartensen.com/immutability-made-easy-in-java-9>).

```

    put("the", 2);
    put("high", 3);
    put("ground", 4);
  });

```

Читатели, знакомые с Java 9, вероятно, знают, что весь этот рецепт можно было заменить очень простым набором фабричных методов: `List.of`, `Set.of` и `Map.of`.

См. также

В рецепте 10.3 демонстрируются новые фабричные методы в Java 9, которые автоматически создают немодифицируемые коллекции.

4.9. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА COLLECTOR

Проблема

Требуется вручную реализовать интерфейс `java.util.stream.Collector`, поскольку ни один из фабричных методов класса `java.util.stream.Collectors` не делает в точности то, что нужно.

Решение

Предоставить лямбда-выражения или ссылки на методы, реализующие функции поставщика, аккумулятора, комбинатора и завершителя, которые используются в фабричных методах `Collector.of`, а также функцию `characteristics`, описывающую требуемые характеристики.

Обсуждение

В служебном классе `java.util.stream.Collectors` имеется несколько удобных статических методов, возвращающих значение типа `Collector`, например: `toList`, `toSet`, `toMap` и даже `toCollection`. Все они будут продемонстрированы в этой книге. Экземпляры классов, реализующих интерфейс `Collector`, передаются методу `collect` интерфейса `Stream`. Так, в примере 4.34 метод принимает строки и возвращает список, содержащий только строки четной длины.

Пример 4.34 ❖ Использование метода `collect` для возврата списка

```

public List<String> evenLengthStrings(String... strings) {
    return Stream.of(strings)
        .filter(s -> s.length() % 2 == 0)
        .collect(Collectors.toList()); ❶
}

```

❶ Собрать строки четной длины в список

Но если вы хотите написать собственный коллектор, то процедура несколько усложняется. В коллекторах используются пять функций, которые совместно

работают, чтобы аккумулировать данные в изменяемом контейнере и факультативно преобразовать результат. Эти функции называются `supplier`, `accumulator`, `combiner`, `finisher` и `characteristics`.

Начнем с функции `characteristics`. Она представляет неизменяемое множество элементов перечисляемого типа `Collector.Characteristics` с тремя возможными значениями: `CONCURRENT`, `IDENTITY_FINISH` и `UNORDERED`. Значение `CONCURRENT` означает, что результирующий контейнер поддерживает одновременные вызовы функции `accumulator` из разных потоков. Значение `UNORDERED` означает, что операция собирания не обязана сохранять порядок следования элементов. Наконец, `IDENTITY_FINISH` означает, что функция-завершитель просто возвращает свой аргумент без изменения.

Отметим, что можно не задавать характеристики, если умолчания вас удовлетворяют.

Опишем назначение каждой функции.

`supplier()`

Создает контейнер-аккумулятор с помощью объекта типа `Supplier<A>`.

`accumulator()`

Добавляет один элемент данных в аккумулятор с помощью объекта типа `BiConsumer<A,T>`.

`combiner()`

Объединяет два аккумулятора с помощью объекта типа `BinaryOperator<A>`.

`finisher()`

Преобразует аккумулятор в результирующий контейнер с помощью объекта типа `Function<A,R>`.

`characteristics()`

Множество `Set<Collector.Characteristics>`, состоящее из объектов перечисляемого типа.

Как обычно, все становится проще, если понимаешь функциональные интерфейсы, определенные в пакете `java.util.function`. Объект типа `Supplier` используется для создания контейнера, в котором аккумулируются временные результаты. Объект типа `BiConsumer` добавляет в аккумулятор один элемент. Интерфейс `BinaryOperator` подразумевает, что типы обоих входных аргументов и возвращаемого значения одинаковы, так что его назначение – объединить два аккумулятора в один. Объект типа `Function` осуществляет завершающее преобразование аккумулятора в требуемый результирующий контейнер.

Все эти методы вызываются в процессе собирания, который запускается, например, вызовом метода `collect` интерфейса `Stream`. Концептуально процесс собирания эквивалентен взятому из официальной документации коду, который приведен в примере 4.35.

Пример 4.35 ❖ Как используются методы коллектора

```
R container = collector.supplier.get();           ❶
for (T t : data) {
    collector.accumulator().accept(container, t);  ❷
}
return collector.finisher().apply(container);    ❸
```

- ❶ Создать контейнер-аккумулятор
- ❷ Добавить элементы в аккумулятор по одному
- ❸ Преобразовать аккумулятор в результирующий контейнер с помощью завершителя

Интересно, что функция `combiner` здесь вообще не упоминается. Если поток последовательный, то она и не нужна – алгоритм работает, как описано выше. Но при обработке параллельного потока работа разбивается на несколько частей, каждая из которых порождает свой аккумулятор. Тогда комбинатор используется на этапе соединения, чтобы объединить все аккумуляторы в один до вызова функции-завершителя.

В примере 4.36 приведен фрагмент кода, аналогичный примеру 4.34.

Пример 4.36 ❖ Использование метода `collect` для возврата немодифицируемой коллекции `SortedSet`

```
public SortedSet<String> oddLengthStringSet(String... strings) {
    Collector<String, ?, SortedSet<String>> intoSet =
        Collector.of(TreeSet<String>::new,           ❶
            SortedSet::add,                          ❷
            (left, right) -> {                       ❸
                left.addAll(right);
                return left;
            },
            Collections::unmodifiableSortedSet);    ❹
    return Stream.of(strings)
        .filter(s -> s.length() % 2 != 0)
        .collect(intoSet);
}
```

- ❶ `Supplier`, возвращающий новый контейнер `TreeSet`
- ❷ `BiConsumer`, добавляющий одну строку в `TreeSet`
- ❸ `BinaryOperator`, объединяющий два объекта `SortedSet` в один
- ❹ Функция `finisher`, создающая немодифицируемое множество

В результате получается отсортированное немодифицируемое множество лексикографически упорядоченных строк.

В этом примере используется один из двух перегруженных вариантов статического метода `of` для порождения коллекторов. Приведем сигнатуры обоих вариантов:

```
static <T,A,R> Collector<T,A,R> of(Supplier<A> supplier,
    BiConsumer<A,T> accumulator,
    BinaryOperator<A> combiner,
    Function<A,R> finisher,
```

```
Collector.Characteristics... characteristics)  
static <T,R> Collector<T,R,R> of(Supplier<R> supplier,  
    BiConsumer<R,T> accumulator,  
    BinaryOperator<R> combiner,  
    Collector.Characteristics... characteristics)
```

Учитывая наличие в классе `Collectors` вспомогательных методов, порождающих коллекторы, вам вряд ли когда-нибудь придется писать свой собственный. Тем не менее это полезный навык и лишняя иллюстрация того, как слаженная работа функциональных интерфейсов из пакета `java.util.function` позволяет создавать интересные объекты.

См. также

Функция `finisher` – пример подчиненного коллектора, их подробное обсуждение содержится в рецепте 4.6. Функциональные интерфейсы `Supplier`, `Function` и `BinaryOperator` рассматриваются в различных рецептах из главы 2. Статические служебные методы класса `Collectors` обсуждаются в рецепте 4.2.

Глава 5

Применение потоков, лямбда-выражений и ссылок на методы

Итак, мы познакомились с основами лямбда-выражений и ссылок на методы и с тем, как они используются вместе с потоками. Но существует ряд дополнительных вопросов, связанных с этой комбинацией. Например, теперь, когда интерфейсы могут иметь методы по умолчанию, что происходит, если класс реализует несколько интерфейсов, в которых определены методы по умолчанию с одинаковыми сигнатурами, но разными реализациями? Или другой пример – что происходит, когда код лямбда-выражения пытается прочитать или модифицировать переменную, определенную вне него? А как насчет исключений? Как они обрабатываются в лямбда-выражениях, где нет сигнатуры метода, в которую можно было поместить фразу `throws`?

Эта глава содержит ответы на эти и другие вопросы.

5.1. КЛАСС `JAVA.UUTIL.OBJECTS`

Проблема

Нужны статические служебные методы для проверки на `null`, сравнения и т. п.

Решение

Воспользоваться классом `java.util.Objects`, который был добавлен в Java 7, но полезен и для потоковой обработки.

Обсуждение

`java.util.Objects` – один из малоизвестных классов, добавленных в Java 7. Он содержит статические методы для различных надобностей, а именно:

`static boolean deepEquals(Object a, Object b)`

Производит «глубокое» сравнение на равенство, что особенно полезно при сравнении массивов.

`static boolean equals(Object a, Object b)`

Вызывает метод `equals` от имени первого аргумента, но при этом безопасен, когда этот аргумент равен `null`.

`static int hash(Object... values)`

Генерирует хэш-код последовательности входных значений.

`static String toString(Object o)`

Возвращает результат вызова `toString` от имени аргумента, если он не равен `null`, и `null` в противном случае.

`static String toString(Object o, String nullDefault)`

Возвращает результат вызова `toString` от имени первого аргумента или второй аргумент, если первый равен `null`.

Следующий метод имеет несколько перегруженных вариантов, полезных для проверки аргументов:

`static <T> T requireNotNull(T obj)`

Возвращает `obj`, если аргумент не равен `null`, в противном случае возбуждает исключение `NullPointerException`.

`static <T> T requireNotNull(T obj, String message)`

То же, что и выше, но исключение `NullPointerException` будет содержать указанное сообщение.

`static <T> T requireNotNull(T obj, Supplier<String> messageSupplier)`

То же, что и выше, но при возбуждении исключения `NullPointerException` вызывается указанный поставщик, который генерирует сообщение.

Последний вариант метода принимает в качестве аргумента объект типа `Supplier<String>`, что и объясняет причину включения этого класса в книгу, посвященную версии Java 8 и выше. Но, пожалуй, еще более основательную причину дают методы `isNull` и `nonNull`, возвращающие значение типа `boolean`:

`static boolean isNull(Object obj)`

Возвращает `true`, если переданная ссылка равна `null`, и `false` в противном случае.

`static boolean nonNull(Object obj)`

Возвращает `true`, если переданная ссылка не равна `null`, и `false` в противном случае.

Полезность этих методов – в том, что они могут выступать в роли предикатов в фильтрах.

Пусть, например, имеется класс, возвращающий коллекцию. В примере 5.1 показаны метод для возврата всей коллекции и метод для возврата только элементов коллекции, отличных от `null`.

Пример 5.1 ❖ Возврат полной коллекции и фильтрация `null`

```
List<String> strings = Arrays.asList(
    "this", null, "is", "a", null, "list", "of", "strings", null);

List<String> nonNullStrings = strings.stream()
    .filter(Objects::nonNull)           ❶
    .collect(Collectors.toList());
```

❶ Фильтрация элементов, равных `null`

Для тестирования этого кода можно использовать метод `Objects.deepEquals`, как показано в примере 5.2.

Пример 5.2 ❖ Тестирование фильтра

```
@Test
public void testNonNulls() throws Exception {
    List<String> strings =
        Arrays.asList("this", "is", "a", "list", "of", "strings");
    assertTrue(Objects.deepEquals(strings, nonNullStrings);
}
```

Эту процедуру можно обобщить со строк на другие типы. Код в примере 5.3 отфильтровывает значения `null` из любого списка.

Пример 5.3 ❖ Фильтрация `null` в любом списке

```
public <T> List<T> getNonNullElements(List<T> list) {
    return list.stream()
        .filter(Objects::nonNull)
        .collect(Collectors.toList());
}
```

Теперь можно без труда профильтровать любой список, который может содержать значения `null`.

5.2. ЛЯМБДА-ВЫРАЖЕНИЯ И ЭФФЕКТИВНАЯ ФИНАЛЬНОСТЬ

Проблема

Внутри лямбда-выражения требуется получить доступ к переменной, определенной вне его.

Решение

Локальные переменные, к которым производится доступ из лямбда-выражения, должны быть финальными или «эффективно финальными». Атрибуты можно читать и модифицировать.

Обсуждение

В конце 1990-х годов, когда язык Java еще сверкал новизной, разработчики иногда писали клиентские приложения, применяя библиотеку Swing для построения пользовательского интерфейса. Как и во всех подобных библиотеках, компоненты Swing управлялись событиями: компонент генерирует события, а прослушиватель реагирует на них.

Поскольку считалось хорошим стилем писать отдельные прослушиватели для каждого компонента, для реализации прослушивателей часто использовали анонимные внутренние классы. Это позволяло обеспечить их модульность, но у внутренних классов было и дополнительное преимущество: код, находящийся во внутреннем классе, мог читать и модифицировать закрытые атрибуты внешнего класса. Например, объект JButton генерирует событие ActionEvent, а интерфейс ActionListener содержит единственный метод actionPerformed, который вызывается после того, как его реализация зарегистрирована в качестве прослушивателя. Как это выглядит, показано в примере 5.4.

Пример 5.4 ❖ Тривиальный пользовательский интерфейс на Swing

```
public class MyGUI extends JFrame {
    private JTextField name = new JTextField("Please enter your name");
    private JTextField response = new JTextField("Greeting");
    private JButton button = new JButton("Say Hi");

    public MyGUI() {
        // ... не относящийся к GUI код инициализации ...
        String greeting = "Hello, %s!";
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                response.setText(
                    String.format(greeting, name.getText());
                // greeting = "Anything else";
            }
        });
    }
}
```

- ❶ Локальная переменная
- ❷ Доступ к локальной переменной и атрибутам
- ❸ Модификация локальной переменной (не компилируется)

Строка `greeting` – локальная переменная, определенная внутри конструктора. Переменные `name` и `response` – атрибуты класса. Интерфейс `ActionListener` реализован в виде анонимного внутреннего класса с единственным методом `actionPerformed`. Во внутреннем классе разрешается:

- обращаться к атрибутам, например `name` и `response`;
- модифицировать атрибуты (хотя здесь это и не показано);
- обращаться к локальной переменной `greeting`.

Но модифицировать локальную переменную *запрещается*.

На самом деле до выхода Java 8 компилятор потребовал бы, чтобы переменная `greeting` была объявлена с ключевым словом `final`. В Java 8 объявлять переменную финальной необязательно, но она должна быть *эффективно финальной*. Иными словами, любая попытка изменить значение локальной переменной не откомпилируется.

Разумеется, в Java 8 анонимный внутренний класс следовало бы заменить лямбда-выражением, как показано в примере 5.5.

Пример 5.5 ❖ Прослушиватель в виде лямбда-выражения

```
String greeting = "Hello, %s!";
button.addActionListener(e ->
    response.setText(String.format(greeting, name.getText())));
```

Действуют все те же правила. В объявлении переменной `greeting` может отсутствовать слово `final`, но она должна быть эффективно финальной, иначе код не откомпилируется.

Если Swing вам не по вкусу, то вот вам другой пример на ту же тему. Допустим, требуется вычислить сумму чисел в списке, как в примере 5.6.

Пример 5.6 ❖ Вычисление суммы чисел в списке

```
List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9);

int total = 0;                                ❶
for (int n : nums) {                          ❷
    total += n;
}

total = 0;
nums.forEach(n -> total += n);                ❸

total = nums.stream()                        ❹
    .mapToInt(Integer::valueOf)
    .sum();
```

- ❶ Локальная переменная `total`
- ❷ Традиционный цикл `for-each`
- ❸ Модификация локальной переменной в лямбда-выражении: НЕ КОМПИЛИРУЕТСЯ
- ❹ Преобразование потока в `IntStream` и вызов `sum`

Здесь объявлена локальная переменная `total`. Суммирование значений в традиционном цикле `for-each` работает прекрасно.

Метод `forEach` интерфейса `Iterable` принимает аргумент типа `Consumer`, но если потребитель попытается модифицировать переменную `total`, то код не откомпилируется.

Конечно, правильный подход к решению задачи – преобразовать поток в `IntStream`, где есть метод `sum`, – тогда никаких локальных переменных не понадобится.

Технически функция вместе со всеми доступными из нее переменными называется *замыканием*. В этом смысле Java пребывает в серой зоне – локальные

переменные доступны, но модифицировать их нельзя. Можно было бы возразить, что лямбда-выражения в Java 8 все-таки являются замыканиями, но только замыкают они значения, а не переменные¹.

См. также

В других языках действуют другие правила доступа к переменным из замыканий. Например, в Groovy их разрешено модифицировать, хотя это считается дурным тоном.

5.3. Потоки случайных чисел

Проблема

Требуется поток случайных чисел типа `int`, `long` или `double` в определенном диапазоне.

Решение

Использовать статические методы `ints`, `longs` и `doubles` класса `java.util.Random`.

Обсуждение

Если вам нужно всего одно случайное число типа `double`, то подойдет статический метод `Math.random`, который возвращает число в диапазоне от 0.0 до 1.0². Это то же самое, что создать экземпляр класса `java.util.Random` и вызвать его метод `nextDouble`.

В классе `Random` имеется конструктор, позволяющий задать начальное значение. При одном и том же начальном значении получается одна и та же последовательность случайных чисел, что полезно для тестирования.

На случай, когда требуется последовательный поток случайных чисел, в Java 8 добавлено несколько методов в класс `Random`, а именно `ints`, `longs` и `doubles` с такими сигнатурами (без учета перегруженных вариантов):

```
IntStream ints()  
LongStream longs()  
DoubleStream doubles()
```

Перегруженные варианты позволяют задать размер потока, а также минимальное и максимальное значения. Например:

¹ Но почему тогда лямбда-выражения в Java 8 не называются замыканиями? Согласно Брюсу Эккелю, термин «замыкание» так сильно перегружен, что это ведет к спорам. «Говоря о настоящих замыканиях, человек очень часто имеет в виду замыкания в том первом для него языке программирования, в котором замыкания присутствовали». За деталями отсылаем к статье в его блоге «Are Java 8 Lamdas Closures?».

² В документации написано, что возвращается «псевдослучайное число с приблизительно равномерным распределением в этом диапазоне», что говорит об ограничениях, которые нужно иметь в виду при обсуждении генераторов случайных чисел.

```
DoubleStream doubles(long streamSize, double randomNumberOrigin,
    double randomNumberBound)
```

Возвращенный поток будет содержать `streamSize` псевдослучайных чисел типа `double`, каждое из которых больше или равно `randomNumberOrigin` и строго меньше `randomNumberBound`.

В тех вариантах, где `streamSize` не задается, возвращаемый поток «эффективно не ограничен».

Если не задан минимум или максимум, то по умолчанию подразумеваются 0 и 1 для метода `doubles`, полный диапазон типа `Integer` для метода `ints` и полный диапазон типа `Long` для метода `longs`. В любом случае, получение элемента из потока эквивалентно повторяющимся вызовам метода `nextDouble`, `nextInt` или `nextLong` соответственно.

Пример 5.7 ❖ Порождение потока случайных чисел

```
Random r = new Random();
r.ints(5)                                     ❶
    .sorted()
    .forEach(System.out::println);

r.doubles(5, 0, 0.5)                         ❷
    .sorted()
    .forEach(System.out::println);

List<Long> longs = r.longs(5)
    .boxed()                                  ❸
    .collect(Collectors.toList());
System.out.println(longs);

List<Integer> listOfInts = r.ints(5, 10, 20)
    .collect(LinkedList::new, LinkedList::add, LinkedList::addAll); ❹
System.out.println(listOfInts);
```

- ❶ Пять случайных целых чисел
- ❷ Пять случайных чисел типа `double` от 0 (включая) до 0.5 (не включая)
- ❸ Обертывание `long` типом `Long`, чтобы числа можно было собрать в коллекцию
- ❹ Другая форма `collect`, без вызова `boxed`

В последних двух примерах демонстрируется мелкое неудобство при создании коллекции значений примитивных типов. Для потока примитивов нельзя вызвать метод `collect(Collectors.toList())`, о чем говорилось в рецепте 3.2. Там же рекомендовалось либо вызывать метод `boxed` для преобразования `long` в `Long`, либо использовать вариант `collect` с тремя аргументами и задавать поставщик `Supplier`, аккумулятор и комбинатор самостоятельно. Оба подхода показаны в этом примере.

Отметим, что у класса `Random` есть подкласс `SecureRandom`. Он реализует криптографически стойкий генератор случайных чисел. Все вышеупомянутые методы (`ints`, `longs`, `doubles` и их перегруженные варианты) работают также для класса `SecureRandom`, просто используется другой генератор.

См. также

Метод `boxed` интерфейса `Stream` обсуждается в рецепте 3.2.

5.4. МЕТОДЫ ПО УМОЛЧАНИЮ ИНТЕРФЕЙСА `MAP`

Проблема

Требуется добавить или заменить элемент отображения `Map`, но только если он соответственно отсутствует или присутствует, или выполнить другую похожую операцию.

Решение

Воспользоваться одним из многочисленных новых методов интерфейса `java.util.Map`: `computeIfAbsent`, `computeIfPresent`, `replace`, `merge` и т. д.

Обсуждение

Интерфейс `Map` существует в Java с момента добавления подсистемы коллекций в версии 1.2. С появлением в Java 8 методов по умолчанию в интерфейсах `Map` обрел несколько новых методов. Все они перечислены в табл. 5.1.

Таблица 5.1. Методы по умолчанию интерфейса `Map`

Метод	Назначение
<code>compute</code>	Вычислить новое значение по существующему ключу и значению
<code>computeIfAbsent</code>	Вернуть значение заданного ключа, если он присутствует, или использовать заданную функцию для вычисления нового значения и сохранения его в отображении
<code>computeIfPresent</code>	Вычислить новое значение вместо существующего
<code>forEach</code>	Обойти отображение, передавая потребителю каждую пару ключ-значение
<code>getOrDefault</code>	Если ключ присутствует в отображении, вернуть его значение, иначе вернуть значение по умолчанию
<code>merge</code>	Если ключ отсутствует в отображении, вернуть переданное значение, иначе вычислить новое
<code>putIfAbsent</code>	Если ключ отсутствует в отображении, ассоциировать его с заданным значением
<code>remove</code>	Удалить запись, связанную с данным ключом, только если значение в ней совпадает с заданным
<code>replace</code>	Заменить значение, ассоциированное с ключом
<code>replaceAll</code>	Заменить значения во всех записях отображения результатами применения заданной функции к текущей записи

Как видим, в интерфейсе, которым мы пользуемся вот уже больше десяти лет, появилось немало новых методов. И некоторые из них очень удобны.

`computeIfAbsent`

Полная сигнатура метода `computeIfAbsent` выглядит так:

```
V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
```

Этот метод особенно удобен для кэширования результатов вызова метода. Рассмотрим, к примеру, классическое рекурсивное вычисление чисел Фибоначчи. Первые два числа Фибоначчи равны 0 и 1, а значение каждого следующего равно сумме двух предыдущих¹.

Пример 5.8 ❖ Рекурсивное вычисление чисел Фибоначчи

```
long fib(long i) {
    if (i == 0) return 0;
    if (i == 1) return 1;
    return fib(i - 1) + fib(i - 2); ❶
}
```

❶ В высшей степени неэффективно

Проблема в том, что $\text{fib}(5) = \text{fib}(4) + \text{fib}(3) = \text{fib}(3) + \text{fib}(2) + \text{fib}(2) + \text{fib}(1) = \dots$, так что производится очень много повторных вычислений. Исправить это можно путем хэширования, в функциональном программировании эта техника называется *запоминанием* (memoization). Новая программа, модифицированная для запоминания объектов типа `BigInteger`, показана в примере 5.9.

Пример 5.9 ❖ Вычисление чисел Фибоначчи с использованием кэша

```
private Map<Long, BigInteger> cache = new HashMap<>();

public BigInteger fib(long i) {
    if (i == 0) return BigInteger.ZERO;
    if (i == 1) return BigInteger.ONE;

    return cache.computeIfAbsent(i, n -> fib(n - 2).add(fib(n - 1))); ❶
}
```

❶ Возвращается значение из кэша, если оно существует, в противном случае вычисляется и сохраняется новое значение

При вычислении используется кэш, ключами которого являются натуральные числа, а значениями – соответствующие числа Фибоначчи. Метод `computeIfAbsent` ищет число с указанным номером в кэше. Если оно существует, то возвращается соответствующее значение. В противном случае переданная функция вызывается для вычисления нового значения, оно сохраняется в кэше и возвращается.

computeIfPresent

Полная сигнатура метода `computeIfPresent` выглядит так:

```
V computeIfPresent(K key,
    BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

¹ Напомню шутку, которую вы, скорее всего, слышали: «Ходят слухи, что в этом году конференция по числам Фибоначчи будет такой же успешной, как две предыдущие вместе взятые».

Метод `computeIfPresent` обновляет значение, только если ассоциированный с ним ключ уже присутствует в отображении. Допустим, мы анализируем текст и подсчитываем количество вхождений каждого слова. Это вычисление, известное под названием конкорданса, встречается довольно часто. Но если нас интересуют только определенные ключевые слова, то можно воспользоваться методом `computeIfPresent`.

Пример 5.10 ❖ Обновление счетчика встречаемости только для определенных слов

```
public Map<String,Integer> countWords(String passage, String... strings) {
    Map<String, Integer> wordCounts = new HashMap<>();

    Arrays.stream(strings).forEach(s -> wordCounts.put(s, 0)); ❶

    Arrays.stream(passage.split(" ")).forEach(word ->
        wordCounts.computeIfPresent(word, (key, val) -> val + 1)); ❷

    return wordCounts;
}
```

- ❶ Поместить в отображение интересующие нас слова, сопоставив им счетчик 0
- ❷ Читать текст и обновлять счетчики только для интересующих нас слов

Если предварительно поместить в отображение только интересующие нас слова с нулевым начальным значением счетчика, то метод `computeIfPresent` будет обновлять счетчик только для этих слов.

Запустив этот метод для заданного текста и списка слов через запятую, как показано в примере 5.11, мы получим искомый результат.

Пример 5.11 ❖ Вызов метода `countWords`

```
String passage = "NSA agent walks into a bar. Bartender says, " +
    "'Hey, I have a new joke for you.' Agent says, 'heard it.'";

Map<String, Integer> counts = demo.countWords(passage, "NSA", "agent", "joke");
counts.forEach((word, count) -> System.out.println(word + "=" + count));

// Выводится agent=1, NSA=2, joke=1
```

Лишь интересующие нас слова являются ключами отображения, поэтому только их счетчики и обновляются. Как обычно, для печати значений используется метод по умолчанию `forEach` интерфейса `Map`, принимающий объект типа `BiConsumer`, которому передаются ключ и значение.

Прочие методы

Метод `replace` работает, как `put`, но только в случае, когда ключ уже существует. Иначе `replace` не делает ничего, тогда как `put` добавляет ключ и возвращает `null`, что может быть нежелательно.

У метода `replace` есть два перегруженных варианта:

```
V replace(K key, V value)
boolean replace(K key, V oldValue, V newValue)
```

Первый заменяет значение, если ключ вообще присутствует в отображении. Второй производит замену, только если текущее значение совпадает с заданным.

Метод `getOrDefault` решает назойливую проблему, связанную с тем, что вызов метода `get` с несуществующим ключом возвращает `null`. Это полезно, но следует помнить, что этот метод только возвращает значение по умолчанию, а не добавляет его в отображение.

Сигнатура метода `getOrDefault` имеет вид:

```
V getOrDefault(Object key, V defaultValue)
```



Метод `getOrDefault` возвращает значение по умолчанию, если ключ отсутствует в отображении, но не добавляет этот ключ.

Метод `merge` очень полезен. Его полная сигнатура имеет вид:

```
V merge(K key, V value,
        BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

Пусть требуется вычислить счетчики вхождения всех, а не только избранных слов в данный текст. Обычно требуется рассмотреть два случая: если слово уже есть в отображении, обновить его счетчик, иначе поместить слово в отображение со счетчиком 1. Метод `merge` упрощает эту процедуру, как показано в примере 5.12.

Пример 5.12 ❖ Использование метода `merge`

```
public Map<String, Integer> fullWordCounts(String passage) {
    Map<String, Integer> wordCounts = new HashMap<>();
    String testString = passage.toLowerCase().replaceAll("\\W", " "); ❶

    Arrays.stream(testString.split("\\s+")).forEach(word ->
        wordCounts.merge(word, 1, Integer::sum)); ❷

    return wordCounts;
}
```

- ❶ Не учитываем регистра букв и знаков препинания
- ❷ Добавить или обновить счетчик для данного слова

Метод `merge` принимает ключ и значение по умолчанию, которое будет вставлено, если ключа еще нет в отображении. Если же ключ есть, то `merge` применяет бинарный оператор (в данном случае метод `sum` класса `Integer`) для вычисления нового значения на основе старого.

Хочется надеяться, из этого рецепта стало понятно, что новые методы по умолчанию интерфейса `Map` являются удобным полезным подспорьем при написании кода.

5.5. КОНФЛИКТ МЕЖДУ МЕТОДАМИ ПО УМОЛЧАНИЮ

Проблема

Имеется класс, реализующий два интерфейса, в которых встречаются методы по умолчанию с одинаковой сигнатурой, но разными реализациями.

Решение

Реализовать метод в этом классе. В вашей реализации можно использовать имеющиеся в интерфейсах методы по умолчанию с помощью ключевого слова `super`.

Обсуждение

Java 8 поддерживает методы по умолчанию и статические методы в интерфейсах. Метод по умолчанию предоставляет реализацию, наследуемую классом. Это позволяет добавлять в интерфейсы новые методы, не нарушая работоспособности существующих классов.

Поскольку класс может реализовывать несколько интерфейсов, то он может унаследовать методы по умолчанию, имеющие одинаковую сигнатуру, но реализованные по-разному. Не исключено также, что в самом классе уже имеется собственный вариант метода по умолчанию.

В такой ситуации существуют три возможности:

- в случае конфликта между методом класса и методом по умолчанию интерфейса всегда побеждает класс;
- в случае конфликта между двумя интерфейсами, один из которых расширяет другой, побеждает расширяющий интерфейс, как и в случае классов;
- если между интерфейсами нет отношения наследования, то класс не откомпилируется.

В последнем случае нужно просто реализовать метод в самом классе, и все заработает. Тем самым третий случай сводится к первому.

В качестве примера рассмотрим интерфейс `Company` из примера 5.13 и интерфейс `Employee` из примера 5.14.

Пример 5.13 ❖ Интерфейс `Company` с методом по умолчанию

```
public interface Company {
    default String getName() {
        return "Initech";
    }
    // прочие методы
}
```

Ключевое слово `default` говорит, что `getName` является методом по умолчанию. Он возвращает название компании.

Пример 5.14 ❖ Интерфейс `Employee` с методом по умолчанию

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

    String getLast();

    void convertCaffeineToCodeForMoney();

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

Интерфейс `Employee` также содержит метод по умолчанию `getName` с такой же сигнатурой, как в интерфейсе `Company`, но с другой реализацией. В примере 5.15 приведен класс `CompanyEmployee`, реализующий оба интерфейса, что приводит к конфликту.

Пример 5.15 ❖ Первая попытка написать класс `CompanyEmployee` (НЕ КОМПИЛИРУЕТСЯ)

```
public class CompanyEmployee implements Company, Employee {
    private String first;
    private String last;

    @Override
    public void convertCaffeineToCodeForMoney() {
        System.out.println("Coding...");
    }

    @Override
    public String getFirst() {
        return first;
    }

    @Override
    public String getLast() {
        return last;
    }
}
```

Поскольку класс `CompanyEmployee` наследует метод по умолчанию от не связанных отношением наследования интерфейсов, то он не откомпилируется. Чтобы исправить ошибку, добавим в класс свой вариант `getName`, который переопределит обе реализации.

При этом имеющиеся методы по умолчанию можно использовать благодаря ключевому слову `super`, как показано в примере 5.16.

Пример 5.16 ❖ Исправленная версия класса `CompanyEmployee`

```
public class CompanyEmployee implements Company, Employee {

    @Override
    public String getName() {
        return String.format("%s работает в компании %s",
```



```

    Employee.super.getName(), Company.super.getName()); ❷
}
// ... остальное не меняется ...
}

```

- ❶ Реализовать `getName`
- ❷ Получить доступ к реализациям по умолчанию с помощью `super`

В этой версии метод `getName` строит строку из того, что возвращают методы по умолчанию интерфейсов `Company` и `Employee`.

Хорошая новость заключается в том, что ничего более сложного в методах по умолчанию нет. Теперь вы знаете о них всё.

Впрочем, имеется еще один граничный случай. Допустим, что интерфейс `Company` содержит метод `getName`, не помеченный ключевым словом `default` (и не имеющий реализации, т. е. абстрактный). Спрашивается: возник бы конфликт с одноименным методом в интерфейсе `Employee`? Как ни странно, возник бы, поэтому в классе `CompanyEmployee` все равно пришлось бы предоставить реализацию.

Разумеется, если один и тот же метод встречается в двух интерфейсах и ни в одном из них не является методом по умолчанию, то мы имеем ситуацию, которая существовала до выхода Java 8. Конфликта нет, но класс обязан предоставить реализацию.

См. также

Методы по умолчанию в интерфейсах обсуждаются в рецепте 1.5.

5.6. ОБХОД КОЛЛЕКЦИЙ И ОТОБРАЖЕНИЙ

Проблема

Требуется обойти коллекцию или отображение.

Решение

Воспользоваться методом `forEach`, добавленным в интерфейсы `Iterable` и `Map` в виде метода по умолчанию.

Обсуждение

Вместо того чтобы обходить линейную коллекцию (т. е. класс, реализующий интерфейс `Collection` или производный от него) в цикле, можно воспользоваться методом `forEach`, добавленным в интерфейс `Iterable` в виде метода по умолчанию.

Согласно документации, он имеет следующую сигнатуру:

```
default void forEach(Consumer<? super T> action)
```

Аргумент `forEach` имеет тип `Consumer` – один из функциональных интерфейсов в пакете `java.util.function`. Тип `Consumer` представляет операцию, которая

принимает один параметр универсального типа и не возвращает никакого результата. Как написано в документации, «в отличие от большинства функциональных интерфейсов, ожидается, что `Consumer` имеет побочные эффекты».

i *Чистой* называется функция без побочных эффектов, т. е. все ее вызовы с одними и теми же параметрами дают одинаковый результат. В функциональном программировании это называется *ссылочной прозрачностью* и означает, что функцию можно заменить ее значением.

Поскольку `java.util.Collection` – подынтерфейс `Iterable`, метод `forEach` имеется во всех линейных коллекциях, от `ArrayList` до `LinkedHashSet`. Поэтому обойти их очень просто (см. пример 5.17).

Пример 5.17 ❖ Обход линейной коллекции

```
List<Integer> integers = Arrays.asList(3, 1, 4, 1, 5, 9);

integers.forEach(new Consumer<Integer>() {           ❶
    @Override
    public void accept(Integer integer) {
        System.out.println(integer);
    }
});

integers.forEach((Integer n) -> {                   ❷
    System.out.println(n);
});

integers.forEach(n -> System.out.println(n));       ❸

integers.forEach(System.out::println);             ❹
}
```

- ❶ Реализация с помощью анонимного внутреннего класса
- ❷ Полное блочное лямбда-выражение
- ❸ Сокращенное лямбда-выражение
- ❹ Ссылка на метод

Версия с анонимным внутренним классом приведена просто для того, чтобы напомнить о сигнатуре метода `accept` интерфейса `Consumer`. Как видим, метод `accept` принимает один аргумент и возвращает `void`. Версии с лямбда-выражениями совместимы с этой сигнатурой, поскольку обе они содержат единственное обращение к методу `println` объекта `System.out`. А в последней версии мы указали ссылку на этот самый метод.

В интерфейс `Map` также добавлен метод по умолчанию `forEach`, только в этом случае он принимает аргумент типа `BiConsumer`:

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

`BiConsumer` – еще один из новых интерфейсов в пакете `java.util.function`. Он представляет функцию, принимающую два аргумента универсального типа и возвращающую `void`. В случае метода `forEach` интерфейса `Map` потребителю передаются ключ и значение, входящие в объект `Map.Entry` из множества `entrySet`.

Таким образом, обход Map теперь ничуть не сложнее обхода List, Set или любой другой линейной коллекции (см. пример 5.18).

Пример 5.18 ❖ Обход Map

```
Map<Long, String> map = new HashMap<>();
map.put(86L, "Don Adams (Maxwell Smart)");
map.put(99L, "Barbara Feldon");
map.put(13L, "David Ketchum");

map.forEach((num, agent) ->
    System.out.printf("Агента %d играл(а) %s%n", num, agent));
```

Результат этого обхода приведен в примере 5.19¹.

Пример 5.19 ❖ Результат обхода Map

```
Агента 99 играл(а) Barbara Feldon
Агента 86 играл(а) Don Adams (Maxwell Smart)
Агента 13 играл(а) David Ketchum
```

До выхода Java 8, чтобы обойти отображение, нужно было сначала с помощью метода `keySet` или `entrySet` получить множество ключей или объектов `Map.Entry`, а затем обойти это множество. Новый метод `forEach` существенно упростил обход.



Имейте в виду, что не существует простого способа выйти из цикла, организованного методом `forEach`. Подумайте о том, чтобы переписать код потоковой обработки с применением метода `filter` и (или) `sorted`, за которым следует `findFirst`.

См. также

Функциональные интерфейсы `Consumer` и `BiConsumer` обсуждаются в рецепте 2.1.

5.7. ПРОТОКОЛИРОВАНИЕ С ПОМОЩЬЮ SUPPLIER

Проблема

Требуется создать сообщение, но только если задан уровень протоколирования, при котором оно должно быть видно.

Решение

Использовать новые перегруженные методы класса `Logger`, принимающие `Supplier`.

¹ Пример взят из уже забытого телевизионного сериала «Напряги извилины» (Get Smart), выходявшего с 1965 по 1970 год. Максвелл Сمارт – это пародийное сочетание Джеймса Бонда и инспектора Клузо, созданное продюсерами Мелом Бруксом и Бакком Генри.

Обсуждение

Методы протоколирования из класса `java.util.logging.Logger`, в т. ч. `info`, `warning` и `severe`, теперь имеют по два перегруженных варианта: один принимает `String`, другой – `Supplier<String>`.

В примере 5.20 показаны сигнатуры методов протоколирования¹.

Пример 5.20 ❖ Перегруженные методы протоколирования в классе `java.util.logging.Logger`

```
void config(String msg)
void config(Supplier<String> msgSupplier)

void fine(String msg)
void fine(Supplier<String> msgSupplier)

void finer(String msg)
void finer(Supplier<String> msgSupplier)

void finest(String msg)
void finest(Supplier<String> msgSupplier)

void info(String msg)
void info(Supplier<String> msgSupplier)

void warning(String msg)
void warning(Supplier<String> msgSupplier)

void severe(String msg)
void severe(Supplier<String> msgSupplier)
```

В каждом случае вариант, принимающий аргумент типа `String`, был частью оригинального API, появившегося в Java 1.4. Вариант с аргументом типа `Supplier` включен, начиная с Java 8. В стандартной библиотеке варианты, принимающие `Supplier`, написаны, как показано в примере 5.21.

Пример 5.21 ❖ Детали реализации класса `Logger`

```
public void info(Supplier<String> msgSupplier) {
    log(Level.INFO, msgSupplier);
}

public void log(Level level, Supplier<String> msgSupplier) {
    if (!isLoggable(level)) {
        return;
    }
    LogRecord lr = new LogRecord(level, msgSupplier.get());
    doLog(lr);
}
```

- ❶ Вернуться, если сообщение не должно быть видно
- ❷ Получить сообщение от `Supplier`, вызвав метод `get`

¹ Возможно, вам не понятно, почему проектировщики подсистемы протоколирования в Java не стали использовать те же уровни протоколирования (`trace`, `debug`, `info`, `warn`, `error` и `fatal`), что во всех остальных API протоколирования. Это интересный вопрос. Если найдете на него ответ, дайте мне знать.

Вместо того чтобы конструировать сообщение, которое никогда не будет показано, мы проверяем, является ли сообщение «протоколируемым». Если сообщение было задано в виде простой строки, то оно вычисляется в любом случае. Но вариант с `Supplier` позволяет разработчику поставить перед сообщением пустую пару круглых скобок и стрелку, тем самым превратив его в поставщика, который будет вызван только при подходящем уровне протоколирования. В примере 5.22 показано, как используются оба варианта 5.22.

Пример 5.22 ❖ Использование `Supplier` в методе `info`

```
private Logger logger = Logger.getLogger(this.getClass().getName());
private List<String> data = new ArrayList<>();

// ... заполнить список данными ...

logger.info("Данные " + data.toString());           ❶
logger.info(() -> "Данные " + data.toString());     ❷
```

- ❶ Аргумент конструируется в любом случае
- ❷ Аргумент конструируется, только если уровень протоколирования таков, что информационные сообщения видны

В этом примере для построения сообщения необходимо вызвать метод `toString` для каждого объекта в списке. В первом случае строка формируется вне зависимости от того, будет показано сообщение или нет. Но если преобразовать аргумент в объект `Supplier`, просто добавив в начале `() ->`, то метод `get` этого объекта будет вызван, только если сообщение действительно используется.

Техника замены аргумента объектом `Supplier` соответствующего типа называется *отложенным выполнением*, она применима в любом контексте, где создание объекта может обходиться дорого.

См. также

Отложенное выполнение – одно из основных применений поставщиков (объектов типа `Supplier`). Поставщики обсуждаются в рецепте 2.2.

5.8. Композиция замыканий

Проблема

Требуется последовательно применить ряд небольших независимых функций.

Решение

Использовать методы композиции, определенные как методы по умолчанию в интерфейсах `Function`, `Consumer` и `Predicate`.

Обсуждение

Одно из достоинств функционального программирования заключается в том, что можно создать набор небольших допускающих повторное использование

функций, которые можно комбинировать для решения более сложных задач. Для этого в пакете `java.util.function` имеются функциональные интерфейсы с методами, упрощающими композицию.

Например, в интерфейсе `Function` определены два метода по умолчанию, сигнатуры которых показаны в примере 5.23.

Пример 5.23 ❖ Методы композиции в интерфейсе `java.util.function.Function`

```
default <V> Function<V,R> compose(Function<? super V,? extends T> before)
default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
```

Имена аргументов в документации говорят, для чего предназначен каждый метод. Метод `compose` применяет свой аргумент *до* (before) исходной функции, а метод `andThen` – после нее.

Для демонстрации рассмотрим тривиальный код в примере 5.24.

Пример 5.24 ❖ Использование методов `compose` и `andThen`

```
Function<Integer, Integer> add2 = x -> x + 2;
Function<Integer, Integer> mult3 = x -> x * 3;

Function<Integer, Integer> mult3add2 = add2.compose(mult3); ❶
Function<Integer, Integer> add2mult3 = add2.andThen(mult3); ❷

System.out.println("mult3add2(1): " + mult3add2.apply(1));
System.out.println("add2mult3(1): " + add2mult3.apply(1));
```

- ❶ Сначала `mult3`, затем `add2`
- ❷ Сначала `add2`, затем `mult3`

Функция `add2` прибавляет 2 к своему аргументу. Функция `mult3` умножает свой аргумент на 3. Поскольку функция `mult3add2` построена с помощью `compose`, то сначала применяется `mult3`, а затем `add2`. С другой стороны, функция `add2mult3` построена с помощью `andThen` и, значит, ведет себя противоположно.

Ниже показаны результаты работы обеих функций:

```
mult3add2(1): 5 // поскольку (1 * 3) + 2 == 5
add2mult3(1): 9 // поскольку (1 + 2) * 3 == 9
```

Результатом композиции является функция, так что этот процесс создает новые операции, которые можно использовать впоследствии. Допустим, к примеру, что мы получаем данные в HTTP-запросе, т. е. они передаются в форме строк. У нас уже есть метод обработки данных, но только если они представлены в числовой форме. Если такое случается часто, то мы можем составить композицию функций, первая из которых разбирает строковые данные, а вторая применяет операцию к полученному числу. См. пример 5.25.

Пример 5.25 ❖ Выделение целого числа из строки и прибавление к нему 2

```
Function<Integer, Integer> add2 = x -> x + 2;
Function<String, Integer> parseThenAdd2 = add2.compose(Integer::parseInt);
System.out.println(parseThenAdd2.apply("1"));
// печатается 3
```

Новая функция, `parseThenAdd2`, сначала вызывает метод `Integer.parseInt`, а затем прибавляет к результату 2. Можно, наоборот, определить функцию, которая вызывает метод `toString` после выполнения числовой операции, как в примере 5.26.

Пример 5.26 ❖ Прибавить число, затем преобразовать в строку

```
Function<Integer, Integer> add2 = x -> x + 2;
Function<Integer, String> plus2toString = add2.andThen(Object::toString);
System.out.println(plus2toString.apply(1));
// печатается "3"
```

Эта операция возвращает функцию, которая принимает аргумент типа `Integer` и возвращает `String`.

В интерфейсе `Consumer` тоже есть метод, применяемый для композиции замыканий, как показано в примере 5.27.

Пример 5.27 ❖ Композиция замыканий с помощью потребителей

```
default Consumer<T> andThen(Consumer<? super T> after)
```

В документации по интерфейсу `Consumer` поясняется, что метод `andThen` возвращает композицию потребителя, выполняющего исходную операцию, с потребителем, заданным в аргументе. Если хотя бы одна операция возбуждает исключение, то это исключение распространяется до точки вызова композиции. См. пример 5.28.

Пример 5.28 ❖ Составной потребитель для печати и протоколирования

```
Logger log = Logger.getLogger(...);
Consumer<String> printer = System.out::println;
Consumer<String> logger = log::info;

Consumer<String> printThenLog = printer.andThen(logger);
Stream.of("this", "is", "a", "stream", "of", "strings").forEach(printThenLog);
```

В этом примере создаются два потребителя: один – для вывода на консоль, другой – для протоколирования. Составной потребитель распечатывает и протоколирует каждый элемент потока.

В интерфейсе `Predicate` есть три метода для композиции предикатов, показанные в примере 5.29.

Пример 5.29 ❖ Методы композиции в интерфейсе `Predicate`

```
default Predicate<T> and(Predicate<? super T> other)
default Predicate<T> negate()
default Predicate<T> or(Predicate<? super T> other)
```

Как легко понять, методы `and`, `or` и `negate` используются для композиции предикатов с применением логических операций И, ИЛИ и НЕ.

В качестве хоть сколько-нибудь интересного примера рассмотрим свойства целых чисел. Квадратным называется число, квадратный корень из которого

тоже является целым числом. Треугольное число – это число кружков, которые можно расположить в форме правильного треугольника¹.

В примере 5.30 приведены методы, проверяющие, является ли число квадратным или треугольным, и показано, как с помощью метода `and` определить, обладает ли число обоими свойствами.

Пример 5.30 ❖ Треугольные числа, являющиеся также квадратными

```
public static boolean isPerfect(int x) {           ❶
    return Math.sqrt(x) % 1 == 0;
}

public static boolean isTriangular(int x) {      ❷
    double val = (Math.sqrt(8 * x + 1) - 1) / 2;
    return val % 1 == 0;
}
// ...

IntPredicate triangular = CompositionDemo::isTriangular;
IntPredicate perfect = CompositionDemo::isPerfect;
IntPredicate both = triangular.and(perfect);

IntStream.rangeClosed(1, 10_000)
    .filter(both)
    .forEach(System.out::println);              ❸
```

❶ Примеры: 1, 4, 9, 16, 25, 36, 49, 64, 81, ...

❷ Примеры: 1, 3, 6, 10, 15, 21, 28, 36, 45, ...

❸ То и другое (в диапазоне от 1 до 10 000): 1, 36, 1225

Идею композиции можно использовать для построения сложных операций на основе небольшой библиотеки простых функций².

См. также

Функции обсуждаются в рецепте 2.4, потребители – в рецепте 2.1, а предикаты – в рецепте 2.3.

5.9. ПРИМЕНЕНИЕ ВЫНЕСЕННОГО МЕТОДА ДЛЯ ОБРАБОТКИ ИСКЛЮЧЕНИЙ

Проблема

Код внутри лямбда-выражения должен возбудить исключение, но мы не хотим загромождать лямбда-выражение кодом обработки исключения.

¹ Подробнее см. статью Википедии https://ru.wikipedia.org/wiki/Треугольное_число. По-другому треугольное число можно определить как число рукопожатий каждого с каждым.

² Операционная система Unix основана на той же идее, со всеми вытекающими из нее преимуществами.

Решение

Создать отдельный метод, который выполняет операцию и обрабатывает исключение, а затем вызвать этот вынесенный метод из лямбда-выражения.

Обсуждение

Лямбда-выражение, по существу, представляет собой реализацию единственного абстрактного метода функционального интерфейса. Как и в случае анонимных внутренних классов, лямбда-выражения могут возбуждать только исключения, объявленные в сигнатуре абстрактного метода.

Если требуется возбудить неконтролируемое исключение, то ситуация относительно проста. Предком всех неконтролируемых исключений является класс `java.lang.RuntimeException`¹. Как и любой код в Java, лямбда-выражение может возбуждать неконтролируемое исключение, не объявляя его и не обертывая код блоком `try/catch`. Тогда исключение распространяется до вызывающей стороны.

Рассмотрим метод, который делит все элементы коллекции на постоянный коэффициент, как в примере 5.31.

Пример 5.31 ❖ Лямбда-выражение, которое может возбуждать неконтролируемое исключение

```
public List<Integer> div(List<Integer> values, Integer factor) {
    return values.stream()
        .map(n -> n / factor) ❶
        .collect(Collectors.toList());
}
```

❶ Может возбуждать `ArithmeticException`

Целочисленное деление может возбуждать исключение `ArithmeticException` (неконтролируемое), если знаменатель равен нулю². Оно распространяется до вызывающей стороны, как показано в примере 5.32.

Пример 5.32 ❖ Клиентский код

```
List<Integer> values = Arrays.asList(30, 10, 40, 10, 50, 90);
List<Integer> scaled = demo.div(values, 10);
```

¹ Это, наверное, самое неудачное имя класса во всем Java API. Любое исключение возбуждается на этапе выполнения, иначе это была бы ошибка компиляции. Так не следовало бы назвать класс `UncheckedException`? Чтобы еще сильнее подчеркнуть нелепость ситуации, в Java 8 добавлен новый класс `java.io.UncheckedIOException` с целью избежать некоторых проблем, обсуждаемых в этом рецепте.

² Интересно, что если заменить `Integer` на `Double`, то исключения не возникнет, даже если делитель равен 0.0. Вместо этого все элементы будут равны «бесконечности». Хотите верить, хотите нет, это правильное поведение, согласующееся со спецификацией IEEE 754, описывающей обработку чисел с плавающей точкой в двоичном компьютере (которое попортило мне немало крови, когда я писал программы – не поверите – на Фортране: кошмары со временем ушли, но не сразу, не сразу).

```
System.out.println(scaled);
// печатается: [3, 1, 4, 1, 5, 9]
scaled = demo.div(values, 0);
System.out.println(scaled);
// возбуждает ArithmeticException: / деление на 0
```

Клиентский код вызывает метод `div`, и если делитель равен 0, то лямбда-выражение возбуждает исключение `ArithmeticException`. Клиент может добавить блок `try/catch` в метод `map`, чтобы обработать это исключение, но тогда код становится поистине уродливым (см. пример 5.33).

Пример 5.33 ❖ Лямбда-выражение с блоком `try/catch`

```
public List<Integer> div(List<Integer> values, Integer factor) {
    return values.stream()
        .map( n -> {
            try {
                return n / factor;
            } catch (ArithmeticException e) {
                e.printStackTrace();
            }
        })
        .collect(Collectors.toList());
}
```

Это работает и для контролируемых исключений, если только исключение объявлено в функциональном интерфейсе.

В общем случае рекомендуется оставлять код потоковой обработки как можно более простым – в идеале каждая промежуточная операция должна укладываться в одну строку. В данном случае код можно упростить, вынеся функцию, вызываемую из метода `map`, в отдельный метод; тогда потоковая обработка сведется к вызову этого метода (см. пример 5.34).

Пример 5.34 ❖ Вынесение лямбда-выражения в отдельный метод

```
private Integer divide(Integer value, Integer factor) {
    try {
        return value / factor;
    } catch (ArithmeticException e) { ❶
        e.printStackTrace();
    }
}

public List<Integer> divUsingMethod(List<Integer> values, Integer factor) {
    return values.stream()
        .map(n -> divide(n, factor)) ❷
        .collect(Collectors.toList());
}
```

❶ Здесь обрабатывается исключение

❷ Поточковый код упростился

Попутно отметим, что если бы вынесенный метод не нуждался в аргументе `factor`, то методу `map` можно было бы просто передать ссылку на метод.

У техники вынесения лямбда-выражения в отдельный метод есть свои преимущества. Для вынесенного метода можно писать тесты (применяя рефлексию, если метод закрытый), ставить в нем точки прерывания и вообще пользоваться любыми приемами, применимыми к методам.

См. также

Лямбда-выражения с контролируемыми исключениями обсуждаются в рецепте 5.10, использование универсального метода-обертки – в рецепте 5.11.

5.10. КОНТРОЛИРУЕМЫЕ ИСКЛЮЧЕНИЯ И ЛЯМБДА-ВЫРАЖЕНИЯ

Проблема

Имеется лямбда-выражение, которое возбуждает контролируемое исключение, а в объявлении абстрактного метода функционального интерфейса, который оно реализует, это исключение не упомянуто.

Решение

Добавить в лямбда-выражение блок `try/catch` или делегировать обработку исключения вынесенному методу.

Обсуждение

Лямбда-выражение – это, по существу, реализация единственного абстрактного метода функционального интерфейса. Поэтому оно может возбуждать только те контролируемые исключения, которые объявлены в сигнатуре абстрактного метода.

Допустим, мы собираемся вызывать службу по URL-адресу и хотим сформировать строку запроса из коллекции строковых параметров. Параметры должны быть закодированы, так чтобы их можно было включить в URL. Для этой цели в Java имеется класс, который, естественно, называется `java.net.URLEncoder`, а в этом классе имеется статический метод `encode`, который принимает строку и кодирует ее в соответствии с указанной схемой кодирования.

В таком случае нам следовало бы написать код, как в примере 5.35.

Пример 5.35 ❖ URL-кодирование коллекции строк (НЕ КОМПИЛИРУЕТСЯ)

```
public List<String> encodeValues(String... values) {
    return Arrays.stream(values)
        .map(s -> URLEncoder.encode(s, "UTF-8"))
        .collect(Collectors.toList());
}
```

❶ Возбуждает исключение `UnsupportedEncodingException`, которое должно быть обработано

Этот метод принимает список строковых аргументов переменной длины и пытается применить к каждому из них метод `URLEncoder.encode`, указав рекомендуемую кодировку UTF-8. К сожалению, код не компилируется, потому что этот метод возбуждает контролируемое исключение `UnsupportedEncodingException`.

Возникает соблазн просто объявить, что метод `encodeValues` возбуждает такое исключение, но это не годится (см. пример 5.36).

Пример 5.36 ❖ Объявление исключения (ТОЖЕ НЕ КОМПИЛИРУЕТСЯ)

```
public List<String> encodeValues(String... values)
    throws UnsupportedEncodingException { ❶
    return Arrays.stream(values)
        .map(s -> URLEncoder.encode(s, "UTF-8"))
        .collect(Collectors.toList());
}
```

❶ Возбуждение исключения из объемлющего метода тоже НЕ КОМПИЛИРУЕТСЯ

Проблема в том, что возбуждение исключения из лямбда-выражения можно интерпретировать как построение абсолютно независимого класса с одним методом, который и возбуждает исключение. Полезно рассматривать лямбда-выражение как реализацию анонимного внутреннего класса, поскольку тогда становится ясно, что исключение, возбужденное во внутреннем объекте, надо обработать или объявить там же, а не в объемлющем объекте. Подобный код показан в примере 5.37, где приведены как версия с анонимным внутренним классом, так и с лямбда-выражением.

Пример 5.37 ❖ URL-кодирование с использованием блока try/catch (ПРАВИЛЬНО)

```
public List<String> encodeValuesAnonInnerClass(String... values) {
    return Arrays.stream(values)
        .map(new Function<String, String>() { ❶
            @Override
            public String apply(String s) { ❷
                try {
                    return URLEncoder.encode(s, "UTF-8");
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                    return "";
                }
            }
        })
        .collect(Collectors.toList());
}

public List<String> encodeValues(String... values) { ❸
    return Arrays.stream(values)
        .map(s -> {
            try {
                return URLEncoder.encode(s, "UTF-8");
            }
        })
        .collect(Collectors.toList());
}
```

```

    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
        return "";
    }
})
.collect(Collectors.toList());
}

```

- ❶ Анонимный внутренний класс
- ❷ Содержит код, возбуждающий контролируемое исключение
- ❸ Вариант с лямбда-выражением

Поскольку в методе `apply` (единственный абстрактный метод интерфейса `Function`) не объявлено никаких контролируемых исключений, мы обязаны добавить блок `try/catch` внутрь реализующего его лямбда-выражения.

В примере 5.38 показан вариант, в котором для кодирования используется вынесенный метод.

Пример 5.38 ❖ Делегирование URL-кодирования отдельному методу

```

private String encodeString(String s) { ❶
    try {
        return URLEncoder.encode(s, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

public List<String> encodeValuesUsingMethod(String... values) {
    return Arrays.stream(values)
        .map(this::encodeString) ❷
        .collect(Collectors.toList());
}

```

- ❶ Вынесенный метод для обработки исключения
- ❷ Ссылка на вынесенный метод

Такой подход работает, и его просто реализовать. Заодно мы получаем метод, который можно тестировать и отлаживать автономно. Единственный недостаток заключается в том, что писать такой вынесенный метод нужно для каждой операции, которая может возбудить исключение. Однако, как отмечалось в предыдущем рецепте, зачастую это упрощает тестирование компонентов потоковой обработки.

См. также

Использование вынесенного метода для обработки исключений в лямбда-выражениях рассматривается в рецепте 5.9. Использование универсальной обертки исключений обсуждается в рецепте 5.11.

5.11. ИСПОЛЬЗОВАНИЕ УНИВЕРСАЛЬНОЙ ОБЕРТКИ ИСКЛЮЧЕНИЙ

Проблема

Имеется лямбда-выражение, возбуждающее исключение. Требуется универсальная обертка, которая перехватывает все контролируемые исключения и повторно возбуждает их как неконтролируемые.

Решение

Создать специальные классы исключений и добавить универсальный метод, который принимает их и возвращает лямбда-выражение без исключений.

Обсуждение

В рецептах 5.9 и 5.10 показано, как делегировать вынесенному методу обработку исключений, возбуждаемых внутри лямбда-выражения. К сожалению, для каждой операции, которая может возбудить исключение, приходится писать отдельный закрытый метод. Но существует более общее решение – *универсальная обертка*.

Для этого определим отдельный функциональный интерфейс, содержащий метод, в сигнатуре которого объявлено, что он возбуждает исключение `Exception`, и с помощью метода-обертки присоединим его к своему коду.

Например, метод `map` интерфейса `Stream` принимает `Function`, но в объявлении метода `apply` интерфейса `Function` нет никаких контролируемых исключений. Желая использовать в `map` лямбда-выражение, которое может возбуждать контролируемое исключение, мы начнем с создания отдельного функционального интерфейса, в котором объявлено, что он возбуждает `Exception`, как показано в примере 5.39.

Пример 5.39 ❖ Основанный на `Function` функциональный интерфейс, который возбуждает `Exception`

```
@FunctionalInterface
public interface FunctionWithException<T, R, E extends Exception> {
    R apply(T t) throws E;
}
```

Теперь можно добавить метод-обертку, который принимает `FunctionWithException` и возвращает `Function`, обернув метод `apply` блоком `try/catch`, как в примере 5.40.

Пример 5.40 ❖ Метод-обертка для обработки исключений

```
private static <T, R, E extends Exception>
Function<T, R> wrapper(FunctionWithException<T, R, E> fe) {
    return arg -> {
        try {
```

```

        return fe.apply(arg);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
};
}

```

Метод `wrapper` принимает код, возбуждающий любое исключение, производное от `Exception`, и строит необходимый блок `try/catch`, делегируя работу методу `apply`. В данном случае метод `wrapper` сделан статическим, но это необязательно. В результате обертку можно вызывать, передав ей любую функцию, возбуждающую исключение, как показано в примере 5.41.

Пример 5.41 ❖ Использование универсального метода-обертки

```

public List<String> encodeValuesWithWrapper(String... values) {
    return Arrays.stream(values)
        .map(wrapper(s -> URLEncoder.encode(s, "UTF-8")))
        .collect(Collectors.toList());
}

```

❶ Использование метода `wrapper`

Теперь операции `map` можно передать код, возбуждающий исключения, а метод `wrapper` перехватит его и повторно возбудит уже как неконтролируемое. Недостаток этого решения – в том, что для каждого функционального интерфейса, который вы планируете использовать, необходима отдельная универсальная обертка: `ConsumerWithException`, `SupplierWithException` и т. д.

Именно из-за таких осложнений некоторые Java-каркасы (например, Spring и Hibernate) и даже целые языки на платформе JVM (например, Groovy и Kotlin) перехватывают все контролируемые исключения и повторно возбуждают их как неконтролируемые.

См. также

Лямбда-выражения с контролируруемыми исключениями обсуждаются в рецепте 5.10, вынесенные методы – в рецепте 5.9.

Глава 6

Тип Optional


Ох, ну почему в любой теме, относящейся к Optional, не меньше 300 сообщений?

– Брайан Гётц,
список рассылки lambda-libs-спес-experts
(23 октября 2013 г.)

В Java 8 API появился новый класс `java.util.Optional<T>`. И хотя многие разработчики полагают, что цель Optional – избавить код от исключений `NullPointerException`, не в этом его истинное назначение. На самом деле Optional нужен для того, чтобы сообщить пользователю о том, что возвращенное значение на законных основаниях может быть равно `null`. Такая ситуация может возникнуть, когда после фильтрации потока в нем вообще не остается элементов.

Следующие методы потокового API возвращают Optional, когда в потоке нет элементов: `reduce`, `min`, `max`, `findFirst`, `findAny`.

Объект типа Optional может находиться в одном из двух состояний: ссылка на объект типа T или пустой. Первый случай называется *присутствует*, второй – *пуст* (в противоположность `null`).

 Хотя Optional – ссылочный тип, ему никогда не следует присваивать значение `null`. Это считается серьезной ошибкой.

В этой главе мы рассмотрим способы идиоматичного использования Optional. Хотя вопрос о том, как надо использовать Optional, вполне может вызвать оживленную дискуссию в кругу ваших коллег¹, хорошая новость состоит в том, что на эту тему существуют стандартные рекомендации. И если им следовать, то ваши намерения станут ясны, а код будет легко сопровождать.

6.1. СОЗДАНИЕ OPTIONAL

Проблема

Требуется вернуть объект Optional, содержащий известное значение.

¹ Это я очень дипломатично выражаюсь.

Решение

Использовать методы `Optional.of`, `Optional.ofNullable` или `Optional.empty`.

Обсуждение

Как и для многих новых классов в Java 8 API, экземпляры типа `Optional` неизменяемы. В документации `Optional` называется *классом на основе значений* (value-based class), это означает, что:

- его экземпляры финальные и неизменяемые (хотя они могут содержать ссылки на изменяемые объекты)¹;
- класс не имеет открытых конструкторов, и потому его экземпляры должны создаваться фабричными методами;
- реализации методов `equals`, `hashCode` и `toString` основаны только на состоянии экземпляров.

Optional и неизменяемость

Экземпляры класса `Optional` неизменяемы, но обертываемые ими объекты необязательно таковы. Если экземпляр `Optional` содержит ссылку на изменяемый объект, то этот объект можно модифицировать. См. пример 6.1.

Пример 6.1 ❖ Являются ли объекты `Optional` неизменяемыми?

```
AtomicInteger counter = new AtomicInteger();
Optional<AtomicInteger> opt = Optional.ofNullable(counter);

System.out.println(optional);    // Optional[0]

counter.incrementAndGet();      ❶
System.out.println(optional);    // Optional[1]

optional.get().incrementAndGet();  ❷
System.out.println(optional);    // Optional[2]

optional = Optional.ofNullable(new AtomicInteger());  ❸
```

- ❶ Инкрементировать, используя `counter` непосредственно
- ❷ Извлечь содержащееся внутри значение и инкрементировать его
- ❸ Ссылке на `Optional` можно присвоить новое значение

Содержащееся внутри значение можно модифицировать либо по исходной ссылке, либо предварительно достав его методом `get`. Можно даже переприсвоить значение самой ссылке, что еще раз подтверждает тот факт, что неизменяемость и финальность – вещи разные. Чего нельзя сделать, так это модифицировать сам экземпляр `Optional`, поскольку для этого просто нет методов.

Неопределенность слова «неизменяемый» – вещь, довольно типичная для Java, где нет встроенного способа создать класс, порождающий только объекты, которые нельзя изменить.

¹ См. ниже врезку, посвященную неизменяемости.

Перечислим статические фабричные методы создания объектов типа `Optional`:

```
static <T> Optional<T> empty()
static <T> Optional<T> of(T value)
static <T> Optional<T> ofNullable(T value)
```

Метод `empty`, естественно, возвращает пустой `Optional`. Метод `of` возвращает объект `Optional`, который обортывает заданное значение или возбуждает исключение, если аргумент равен `null`. Предполагается, что он будет использоваться, как показано в примере 6.2.

Пример 6.2 ❖ Создание `Optional` методом `of`

```
public static <T> Optional<T> createOptionalTheHardWay(T value) {
    return value == null ? Optional.empty() : Optional.of(value);
}
```

Способ, примененный в примере 6.2, назван «трудным путем» (`The Hard Way`) не потому, что он действительно такой уж трудный, а потому, что есть более простой метод `ofNullable`, показанный в примере 6.3.

Пример 6.3 ❖ Создание `Optional` методом `ofNullable`

```
public static <T> Optional<T> createOptionalTheEasyWay(T value) {
    return Optional.ofNullable(value);
}
```

На самом деле эталонная реализация метода `ofNullable` в Java 8 совпадает со строчкой, показанной в `createOptionalTheHardWay`: проверить, равно ли обернуваемое значение `null`, и если да, то вернуть пустой `Optional`, иначе обернуть его, вызвав `Optional.of`.

Кстати говоря, классы `OptionalInt`, `OptionalLong` и `OptionalDouble` обортывают примитивные типы, которые не могут быть равны `null`, поэтому в них определен только метод `of`:

```
static OptionalInt of(int value)
static OptionalLong of(long value)
static OptionalDouble of(double value)
```

Методы чтения в них называются не `get`, а `getAsInt`, `getAsLong` и `getAsDouble` соответственно.

См. также

В других рецептах, например 6.4 и 6.5, тоже создаются объекты `Optional`, но из имеющихся коллекций. В рецепте 6.3 методы из этого рецепта применяются для обортывания переданных значений.

6.2. ИЗВЛЕЧЕНИЕ ЗНАЧЕНИЯ ИЗ OPTIONAL

Проблема

Требуется извлечь значение, хранящееся внутри Optional.

Решение

Использовать метод `get`, но только если есть уверенность, что Optional не пуст. В противном случае использовать один из вариантов метода `orElse`. Можно также воспользоваться методом `ifPresent`, если вы хотите вызывать Consumer, только когда значение присутствует.

Обсуждение

Если некоторый метод возвращает Optional, то для получения находящегося внутри значения следует вызвать метод `get`. Но если Optional пуст, то метод `get` возбуждает исключение `NoSuchElementException`.

Рассмотрим показанный в примере 6.4 метод, который возвращает первую в потоке строку четной длины.

Пример 6.4 ❖ Получение первой строки четной длины

```
Optional<String> firstEven =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();
```

Метод `findFirst` возвращает объект типа `Optional<String>`, т. к. может случиться, что ни одна из строк в потоке не пройдет через фильтр. Чтобы напечатать возвращенное значение, можно было бы вызвать метод `get`:

```
System.out.println(firstEven.get()) // Не делайте так, даже если это работает
```

В данном случае все обошлось, но никогда не следует вызывать метод `get`, если нет уверенности, что объект Optional содержит значение, иначе вы рискуете получить исключение, как в примере 6.5.

Пример 6.5 ❖ Получение первой строки нечетной длины

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();
```

```
System.out.println(firstOdd.get()); // возбуждается NoSuchElementException
```

Как решить эту проблему? Есть несколько путей. Первый – проверить, что Optional содержит значение, до того как извлекать его (пример 6.6).

Пример 6.6 ❖ Получение первой строки четной длины с защитой вызова `get`

```
Optional<String> firstEven =
    Stream.of("five", "even", "length", "string", "values")
```

```
.filter(s -> s.length() % 2 == 0)
.findFirst();
```

```
System.out.println(
    first.isPresent() ? first.get() : "Нет строк четной длины"); ❷
```

- ❶ То же, что и раньше
- ❷ Вызывать `get`, только если `isPresent` вернул `true`

Это работает, но мы просто обменяли проверку на `null` на вызов `isPresent` – значительным улучшением это не назовешь.

По счастью, есть способ лучше – использовать очень удобный метод `orElse`, как показано в примере 6.7.

Пример 6.7 ❖ Использование `orElse`

```
Optional<String> firstOdd =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 != 0)
        .findFirst();

System.out.println(firstOdd.orElse("Нет строк нечетной длины"));
```

Метод `orElse` возвращает обернутое значение, если оно присутствует, а в противном случае заданное значение по умолчанию. Он удобен, если такое значение по умолчанию существует.

Есть несколько вариантов метода `orElse`:

- `orElse(T other)` возвращает обернутое значение, если оно присутствует, а в противном случае значение по умолчанию `other`;
- `orElseGet(Supplier<? extends T> other)` возвращает обернутое значение, если оно присутствует, а в противном случае вызывает поставщик `other` и возвращает результат его работы;
- `orElseThrow(Supplier<? extends X> exceptionSupplier)` возвращает обернутое значение, если оно присутствует, а в противном случае возбуждает исключение, созданное поставщиком.

Разница между `orElse` и `orElseGet` состоит в том, что первый возвращает строку, которая создается всегда – вне зависимости от того, хранится внутри `Optional` какое-то значение или нет, а второй вызывает `Supplier`, только когда `Optional` пуст.

В данном случае значением по умолчанию является пустая строка, так что различие между обоими методами минимально. Но если аргумент `orElse` – сложный объект, то `orElseGet` гарантирует, что этот объект создается только при необходимости, как в примере 6.8.

Пример 6.8 ❖ Использование `Supplier` в методе `orElseGet`

```
Optional<ComplexObject> val = values.stream().findFirst()

val.orElse(new ComplexObject()); ❶
val.orElseGet(() -> new ComplexObject()) ❷
```

- ❶ Всегда создается новый объект
- ❷ Объект создается только при необходимости

i Использование `Supplier` в качестве аргумента метода – пример *отложенного*, или *ленивого*, выполнения. Это позволяет избежать вызова метода `get` поставщика, если в нем нет необходимости¹.

Библиотечная реализация метода `orElseGet` показана в примере 6.9.

Пример 6.9 ❖ Реализация метода `Optional.orElseGet` в JDK

```
public T orElseGet(Supplier<? extends T> other) {
    return value != null ? value : other.get(); ❶
}
```

❶ `value` – финальный атрибут типа `T` в `Optional`

Метод `orElseThrow`, сигнатура которого показана ниже, также принимает объект типа `Supplier`.

```
<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
```

В примере 6.10 ссылка на конструктор, переданная в качестве поставщика, не выполняется, если объект `Optional` содержит значение.

Пример 6.10 ❖ Использование метода `orElseThrow`

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();
System.out.println(first.orElseThrow(NoSuchElementException::new));
```

Наконец, метод `ifPresent` принимает потребитель `Consumer`, который вызывается, только если `Optional` содержит значение, как в примере 6.11.

Пример 6.11 ❖ Использование метода `ifPresent`

```
Optional<String> first =
    Stream.of("five", "even", "length", "string", "values")
        .filter(s -> s.length() % 2 == 0)
        .findFirst();
first.ifPresent(val -> System.out.println("Найдена строка четной длины"));

first = Stream.of("five", "even", "length", "string", "values")
    .filter(s -> s.length() % 2 != 0)
    .findFirst();
first.ifPresent(val -> System.out.println("Найдена строка нечетной длины"));
```

В данном случае будет напечатано только сообщение «Найдена строка четной длины».

¹ Подробные объяснения см. в главе 6 книги Венката Субраманиама «Functional Programming in Java» (Pragmatic Programmers, 2014).

См. также

Поставщики обсуждаются в рецепте 2.2, ссылки на конструкторы – в рецепте 1.3. Методы `findAny` и `findFirst` интерфейса `Stream`, возвращающие `Optional`, рассматриваются в рецепте 3.9.

6.3. OPTIONAL В МЕТОДАХ ЧТЕНИЯ И УСТАНОВКИ

Проблема

Требуется использовать `Optional` в методах чтения и установки.

Решение

Результат метода чтения можно обернуть объектом `Optional`. Но не следует делать то же самое для методов установки и в особенности для атрибутов.

Обсуждение

Тип данных `Optional` служит для того, чтобы сообщить пользователю о том, что результат операции может принимать значение `null`, не возбуждая при этом исключения `NullPointerException`. Однако класс `Optional` специально *не* сделан сериализуемым, поэтому его не следует использовать для обертывания полей класса.

Следовательно, рекомендуемый механизм включения `Optional` в методы чтения и установки состоит в том, чтобы обертывать допускающие значение `null` атрибуты, возвращаемые методами чтения, но не делать этого в методах установки. См. пример 6.12.

Пример 6.12 ❖ Использование `Optional` на уровне DAO

```
public class Department {
    private Manager boss;

    public Optional<Manager> getBoss() {
        return Optional.ofNullable(boss);
    }
    public void setBoss(Manager boss) {
        this.boss = boss;
    }
}
```

Подразумевается, что в классе `Department` атрибут `boss` типа `Manager` может быть равен `null`¹. Может возникнуть искушение назначить этому атрибуту тип `Optional<Manager>`, но тогда, поскольку тип `Optional` несериализуемый, таким же окажется и тип `Department`.

¹ Наверное, это недостижимая мечта, но, согласитесь, идея соблазнительная.

Мы приняли другой подход: не требовать от пользователя обернуть значение объектом `Optional` при вызове метода установки, что было бы необходимо, если бы метод `setBoss` принимал `Optional<Manager>` в качестве аргумента. Цель `Optional` – указать, что значение может на законных основаниях принимать значение `null`, и клиент уже знает, равно оно `null` или нет, а внутренней реализации до этого нет дела.

Наконец, возврат `Optional<Manager>` из метода чтения – указание для вызывающей стороны на то, что в отделе может и не быть начальника, и это нормально.

Недостаток этого подхода в том, что в течение многих лет соглашение о «JavaBeans» требовало определять методы чтения и установки параллельно, на основе атрибута. Вообще, определение *свойства* в Java (в отличие от просто атрибута) подразумевает наличие методов чтения и установки, написанных стандартным образом. Предложенный в данном рецепте подход нарушает этот принцип – методы чтения и установки больше не симметричны.

Отчасти по этой причине некоторые разработчики считают, что типу `Optional` вообще не место в методах чтения и установки. Они полагают, что это деталь внутренней реализации, которую не следует раскрывать клиенту.

Однако предложенный здесь подход популярен в среде разработчиков средств объектно-реляционного отображения (ORM) типа Hibernate. Решающим здесь является стремление сообщить клиенту о том, что за полем столбца базы данных, допускающий `null`, не заставляя клиента обернуть ссылку, передаваемую методу установки.

Вроде бы разумный компромисс, но, как говорится, на вкус и цвет товарищей нет.

См. также

В рецепте 6.5 этот пример DAO используется для преобразования коллекции идентификаторов в коллекцию работников. В рецепте 6.1 обсуждается оберывание значения объектом `Optional`.

6.4. МЕТОДЫ `flatMap` И `map` КЛАССА `OPTIONAL`

Проблема

Требуется избежать оберывания `Optional` другим `Optional`.

Решение

Использовать метод `flatMap` класса `Optional`.

Обсуждение

Методы `map` и `flatMap` интерфейса `Stream` рассматриваются в рецепте 3.11. Однако идея `flatMap` носит общий характер, ее можно применить и к `Optional`.

Метод flatMap класса Optional имеет следующую сигнатуру:

```
<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)
```

Это похоже на метод map интерфейса Stream в том смысле, что переданная функция применяется к каждому элементу и порождает единственный результат, в данном случае типа Optional<U>. Точнее, если аргумент T существует, то flatMap применяет к нему функцию и возвращает обертывающий результат Optional. Если аргумент отсутствует, то метод возвращает пустой Optional.

В рецепте 6.3 мы говорили, что объекты доступа к данным (DAO) часто пишутся так, что методы чтения возвращают Optional (если свойство может быть равно null), но методы установки не обертывают свои аргументы объектом Optional. Рассмотрим класс Manager, в котором имеется отличное от null строковое поле name, и класс Department, в котором есть допускающее null поле boss типа Manager (пример 6.13).

Пример 6.13 ❖ Часть уровня DAO с участием Optional

```
public class Manager {
    private String name;           ❶

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

    public String getName() {
        return name;
    }
}

public class Department {
    private Manager boss;         ❷

    public Optional<Manager> getBoss() { ❷
        return Optional.ofNullable(boss);
    }

    public void setBoss(Manager boss) {
        this.boss = boss;
    }
}
```

- ❶ Предполагается, что не может быть равно null, поэтому Optional не нужен
- ❷ Может быть равно null, поэтому значение, возвращенное методом чтения, обернуто объектом Optional, но для метода установки это не так

Клиент, вызывающий метод getBoss класса Department, получает результат, обернутый Optional (см. пример 6.14).

Пример 6.14 ❖ Возврат Optional

```
Manager mrSlate = new Manager("Mr. Slate");
Department d = new Department();
```



```
d.setBoss(mrSlate); ❶
System.out.println("Начальник: " + d.getBoss()); ❷

Department d1 = new Department(); ❸
System.out.println("Начальник: " + d1.getBoss()); ❹
```

- ❶ Отдел, в котором есть начальник
- ❷ Печатается Начальник: Optional[Manager{name='Mr. Slate'}]
- ❸ Отдел без начальника
- ❹ Печатается Начальник: Optional.empty

Пока все хорошо. Если в отделе есть начальник, то метод чтения возвращает его, обернутого объектом Optional. Если нет, возвращается пустой Optional.

Проблема в том, что если мы захотим получить имя начальника, то не сможем вызвать метод getName для объекта типа Optional. Нужно будет либо вытащить находящееся внутри Optional значение, либо использовать метод map (пример 6.15).

Пример 6.15 ❖ Извлечь имя начальника, обернутого объектом Optional

```
System.out.println("Имя: " +
    d.getBoss().orElse(new Manager("Unknown")).getName()); ❶

System.out.println("Имя: " +
    d1.getBoss().orElse(new Manager("Unknown")).getName());

System.out.println("Имя: " + d.getBoss().map(Manager::getName)); ❷
System.out.println("Имя: " + d1.getBoss().map(Manager::getName));
```

- ❶ Сначала извлечь из Optional начальника, затем вызвать getName
- ❷ Использовать Optional.map, чтобы применить getName к обернутому объекту Manager

Метод map (обсуждается далее в рецепте 6.5) применяет заданную функцию, только если объект Optional, от имени которого он вызывается, не пуст, поэтому это наиболее простой из двух показанных подходов.

Дело усложняется, если несколько объектов Optional может быть сцеплено. Допустим, в классе Company имеется поле Department (для простоты будем считать, что отдел только один), как в примере 6.16.

Пример 6.16 ❖ В компании может быть отдел (для простоты только один)

```
public class Company {
    private Department department;

    public Optional<Department> getDepartment() {
        return Optional.ofNullable(department);
    }

    public void setDepartment(Department department) {
        this.department = department;
    }
}
```

Результат метода `getDepartment` класса `Company` обертывается объектом `Optional`. Если мы затем хотим получить начальника отдела, то вроде бы следует воспользоваться методом `map`, как в примере 6.15. Но тут возникает проблема, поскольку в результате мы получим `Optional`, обернутый другим `Optional` (пример 6.17).

Пример 6.17 ❖ `Optional`, обернутый другим `Optional`

```
Company co = new Company();
co.setDepartment(d);

System.out.println("Отдел компании: " + co.getDepartment());    ❶
System.out.println("Начальник отдела компании: " + co.getDepartment()
    .map(Department::getBoss));                                ❷
```

- ❶ Печатается Отдел компании: `Optional[Department{boss=Manager{name='Mr. Slate'}}]`
- ❷ Печатается Начальник отдела компании: `Optional[Optional[Manager{name='Mr. Slate'}]]`

На помощь приходит метод `flatMap` класса `Optional`. Он разглаживает структуру, так что остается только один `Optional`. В примере 6.18 предполагается, что компания создана, как в предыдущем примере.

Пример 6.18 ❖ Использование метода `flatMap` для компании

```
System.out.println(
    co.getDepartment()    ❶
    .flatMap(Department::getBoss)    ❷
    .map(Manager::getName));    ❸
```

- ❶ `Optional<Department>`
- ❷ `Optional<Manager>`
- ❸ `Optional<String>`

Теперь обернем также компанию объектом `Optional`, как в примере 6.19.

Пример 6.19 ❖ Использование метода `flatMap` для компании, обернутой `Optional`

```
Optional<Company> company = Optional.of(co);
```

```
System.out.println(
    company    ❶
    .flatMap(Company::getDepartment)    ❷
    .flatMap(Department::getBoss)    ❸
    .map(Manager::getName)    ❹
);
```

- ❶ `Optional<Company>`
- ❷ `Optional<Department>`
- ❸ `Optional<Manager>`
- ❹ `Optional<String>`

О как! Выходит, мы можем даже компанию обернуть объектом `Optional`, а затем просто несколько раз вызвать `Optional.flatMap`, чтобы добраться до нужного свойства, после чего закончить операцию вызовом `Optional.map`.

См. также

Обертывание значения объектом `Optional` обсуждается в рецепте 6.1. Метод `flatMap` интерфейса `Stream` рассматривается в рецепте 3.11, использование `Optional` на уровне DAO – в рецепте 6.3. Метод `map` класса `Optional` – тема рецепта 6.5.

6.5. ОТОБРАЖЕНИЕ ОБЪЕКТОВ OPTIONAL

Проблема

Требуется применить функцию к коллекции объектов `Optional`, но только к тем, что содержат значение.

Решение

Использовать метод `map` класса `Optional`.

Обсуждение

Пусть имеется список идентификаторов работников и требуется построить коллекцию соответствующих объектов `Employee`. Если метод `findEmployeeById` имеет сигнатуру

```
public Optional<Employee> findEmployeeById(int id)
```

то поиск всех работников вернет коллекцию объектов `Optional`, некоторые из которых могут быть пустыми. В примере 6.20 показано, как отфильтровать пустые `Optional`.

Пример 6.20 ❖ Поиск работников по идентификатору

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById)    ❶
        .filter(Optional::isPresent)  ❷
        .map(Optional::get)            ❸
        .collect(Collectors.toList());
}
```

- ❶ `Stream<Optional<Employee>>`
- ❷ Удалить пустые `Optional`
- ❸ Извлечь заведомо существующие значения

Результатом первой операции `map` является поток объектов `Optional`, которые могут содержать или не содержать объект `Employee`. Для извлечения находящегося внутри значения кажется естественным вызвать метод `get`, однако этого нельзя делать, если нет уверенности, что значение присутствует. Поэтому мы сначала производим операцию `filter` со ссылкой на метод `Optional::isPresent` в качестве предиката, чтобы отфильтровать пустые `Optional`. Вот теперь

можно отобразить `Optional` на содержащиеся в них значения, вызвав метод `Optional::get`.

В этом примере мы использовали метод `map` интерфейса `Stream`. Но есть и другой подход – воспользоваться методом `map` класса `Optional` с сигнатурой

```
<U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Этому методу передается функция. Если объект `Optional` не пуст, то `map` извлекает обернутое им значение, применяет к нему функцию и возвращает `Optional`, содержащий результат. В противном случае возвращается пустой `Optional`.

Операцию поиска из примера 6.20 можно переписать с использованием этого метода, как показано в примере 6.21.

Пример 6.21 ❖ Использование метода `Optional.map`

```
public List<Employee> findEmployeesByIds(List<Integer> ids) {
    return ids.stream()
        .map(this::findEmployeeById)    ❶
        .flatMap(optional ->          ❷
            optional.map(Stream::of)    ❸
                .orElseGet(Stream::empty))
        .collect(Collectors.toList());
}
```

❶ `Stream<Optional<Employee>>`

❷ Преобразовывает непустой `Optional<Employee>` в `Optional<Stream<Employee>>`

❸ Извлекает `Stream<Employee>` из `Optional`

Идея в том, что если `Optional`, содержащий `Employee`, не пуст, то нужно вызвать метод `Stream::of` для находящегося внутри значения и тем самым преобразовать его в одноэлементный поток, содержащий это значение, который затем обернуть объектом `Optional`. В противном случае возвращается пустой `Optional`.

Предположим, что работник с данным идентификатором найден. Метод `findEmployeeById` возвращает объект `Optional<Employee>`, содержащий соответствующее значение. Затем метод `optional.map(Stream::of)` возвращает `Optional`, содержащий одноэлементный поток для этого работника, так что мы получаем `Optional<Stream<Employee>>`. После этого метод `orElseGet` извлекает находящееся внутри значение, которое имеет тип `Stream<Employee>`.

Если метод `findEmployeeById` вернул пустой `Optional`, то `optional.map(Stream::of)` также вернет пустой `Optional`, а метод `orElseGet(Stream::empty)` вернет пустой поток.

В итоге мы получаем комбинацию элементов типа `Stream<Employee>` и пустых потоков, а именно для таких ситуаций предназначен метод `flatMap` интерфейса `Stream`. Он сводит всё к `Stream<Employee>`, принимая во внимание только непустые потоки, так что метод `collect` может вернуть их в виде списка `List` работников.

Этот процесс показан на рис. 6.1.

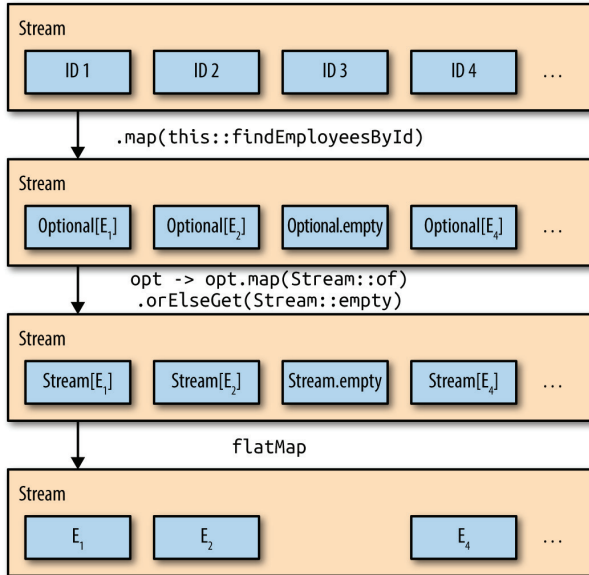


Рис. 6.1 ❖ Методы Optional.map и flatMap

Метод `Optional.map` задуман как подспорье для упрощения потокового кода. Подход на основе пары операций `filter/map`, безусловно, интуитивно более понятен, особенно для разработчиков, непривычных к операциям `flatMap`, но результат получается одинаковым.

Разумеется, вы вправе передать методу `Optional.map` любую функцию. В официальной документации демонстрируется преобразование имен файлов в потоки ввода. Еще один пример приведен в рецепте 6.4.

Кстати говоря, в Java 9 в класс `Optional` добавлен метод `stream`. Если `Optional` не пуст, то этот метод возвращает одноэлементный поток, обернутый вокруг находящегося внутри значения, в противном случае возвращается пустой поток. Детали см. в рецепте 10.6.

См. также

В рецепте 6.3 показано, как использовать класс `Optional` на уровне DAO (объектов доступа к данным). В рецепте 3.11 обсуждается метод потоков `flatMap`, а в рецепте 6.4 – метод `flatMap` класса `Optional`. В рецепте 10.6 рассматриваются новые методы, добавленные в класс `Optional` в версии Java 9.

Глава 7

Файловый ввод-вывод

Пакет неблокирующего (или «нового») ввода-вывода, известный как NIO, был добавлен в версии J2SE 1.4¹. В расширении NIO.2, добавленном в Java 7, появились новые классы для работы с файлами и каталогами. Добавления включены в пакет `java.nio.file`, которому и посвящена эта глава. Несколько новых классов в этом пакете, например `java.nio.files.File`, в Java 8 было дополнено методами, относящимися к потокам.

К сожалению, в этом месте метафора потоков из функционального программирования вступает в конфликт с одноименным термином из области ввода-вывода, что может приводить к недоразумениям. Например, интерфейс `java.nio.file.DirectoryStream` не имеет ничего общего с функциональными потоками. Он реализован классами, которые обходят дерево каталогов, применяя традиционный цикл `for-each`².

Эта глава посвящена средствам ввода-вывода, предназначенным для поддержки функциональных потоков. В табл. 7.1 перечислены методы, добавленные для этой цели в класс `java.nio.file.Files` в Java 8. Отметим, что все методы класса `Files` статические.

Таблица 7.1. Методы класса `java.nio.file.Files`, возвращающие поток

Метод	Тип возвращаемого значения
<code>lines</code>	<code>Stream<String></code>
<code>list</code>	<code>Stream<Path></code>
<code>walk</code>	<code>Stream<Path></code>
<code>find</code>	<code>Stream<Path></code>

В рецептах из этой главы рассматриваются вышеперечисленные методы.

¹ Большинство Java-разработчиков изумляется, узнав, что NIO был добавлен так давно.

² И только усугубляет неразбериху тот факт, что `DirectoryStream.Filter` – на самом деле функциональный интерфейс, хотя с функциональными потоками по-прежнему никак не связан. Он служит для выборки определенных элементов из дерева каталогов.

7.1. ОБРАБОТКА ФАЙЛОВ

Проблема

Требуется обработать содержимое текстового файла с помощью потоков.

Решение

Воспользоваться статическим методом `lines` из класса `java.io.BufferedReader` или `java.nio.file.Files` для возврата содержимого файла в виде потока.

Обсуждение

Во всех Unix-системах на базе FreeBSD (включая macOS) в каталоге `/usr/share/dict/` находится файл, содержащий второе издание словаря Вебстера. Файл `web2` включает примерно 230 000 слов, по одному слову в строке.

Пусть требуется найти 10 самых длинных слов в этом словаре. Метод `Files.lines` выводит слова в виде потока строк, к которому затем можно применить обычные средства потоковой обработки, например методы `map` и `filter` (см. пример 7.1).

Пример 7.1 ❖ Нахождение 10 самых длинных слов в словаре `web2`

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2")) {
    lines.filter(s -> s.length() > 20)
        .sorted(Comparator.comparingInt(String::length).reversed())
        .limit(10)
        .forEach(w -> System.out.printf("%s (%d)%n", w, w.length()));
} catch (IOException e) {
    e.printStackTrace();
}
```

Предикат в методе `filter` пропускает только слова длинее 20 символов. Затем метод `sorted` сортирует слова по длине в порядке убывания. Метод `limit` прекращает обработку после первых 10 слов, которые затем распечатываются. Поскольку поток открыт в блоке `try` с ресурсами, то система автоматически закрывает его, и сам файл словаря после выхода из блока `try`.

Потоки и интерфейс `AutoCloseable`

Интерфейс `Stream` расширяет `BaseStream`, который, в свою очередь, является подынтерфейсом `AutoCloseable`. Поэтому потоки можно использовать внутри блока `try` с ресурсами, появившегося в Java 7. При выходе из такого блока система автоматически вызывает метод `close`, который не только закрывает сам поток, но и вызывает обработчики закрытия во всем конвейере, чтобы освободить ресурсы.

До сих пор блок `try` с ресурсами в этой книге не встречался, потому что мы генерировали потоки из коллекций, хранящихся целиком в памяти. Но в этом рецепте поток основан на файле, поэтому `try` с ресурсами гарантирует корректное закрытие файла словаря.

В примере 7.2 показаны результаты выполнения кода из примера 7.1.

Пример 7.2 ❖ Самые длинные слова в словаре

```
formaldehydesulphoxylate (24)
pathologicopsychological (24)
scientificphilosophical (24)
tetraiodophenolphthalein (24)
thyroparathyroidectomize (24)
anthropomorphologically (23)
blepharosphincterectomy (23)
epididymodeferentectomy (23)
formaldehydesulphoxylic (23)
gastroenteroanastomosis (23)
```

В словаре есть пять слов, насчитывающих 24 символа. Они расположены в алфавитном порядке только потому, что исходный файл был отсортирован по алфавиту. Добавив вызов `thenComparing` в аргумент `Comparator` метода `sorted`, мы сможем самостоятельно задать порядок сортировки слов одинаковой длины.

Далее следуют слова длиной 23 символа, в основном относящиеся к медицинской тематике¹.

Применив подчиненный коллектор `Collectors.counting`, мы можем определить, сколько слов каждой длины встречается в словаре (пример 7.3).

Пример 7.3 ❖ Нахождение количества слов каждой длины

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2"))) {
    lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(String::length, Collectors.counting()))
        .forEach((len, num) -> System.out.println(len + ": " + num));
}
```

В этом фрагменте используется коллектор `groupingBy` для создания отображения, ключом которого служит длина слова, а значением – количество слов такой длины. Получается вот что:

```
21: 82
22: 41
23: 17
24: 5
```

Результат информативен, но не слишком. Кроме того, данные отсортированы в порядке возрастания, а хотелось бы наоборот.

Вместо этого можно воспользоваться новыми статическими методами `comparingByKey` и `comparingByValue`, добавленными в интерфейс `Map.Entry`. Каждый из них принимает факультативный компаратор, как было описано в рецепте 4.4. В данном случае сортировка с помощью компаратора `reverseOrder` дает результат в порядке, противоположном естественному (пример 7.4).

¹ К счастью, слово *blepharosphincterectomy* означает не то, что кажется. Это ослабление давления глазного века на роговицу – плохо, но могло бы быть и хуже.

Пример 7.4 ❖ Количество слов каждой длины в порядке убывания длин

```
try (Stream<String> lines = Files.lines(Paths.get("/usr/share/dict/web2"))) {
    Map<Integer, Long> map = lines.filter(s -> s.length() > 20)
        .collect(Collectors.groupingBy(String::length, Collectors.counting()));

    map.entrySet().stream()
        .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
        .forEach(e -> System.out.printf("Длина %d: %d слов%n",
            e.getKey(), e.getValue()));
}
```

Получается такой результат:

```
Длина 24: 5 слов
Длина 23: 17 слов
Длина 22: 41 слов
Длина 21: 82 слов
```

В классе `BufferedReader` также имеется метод `lines`, правда не статический, а метод экземпляра. В примере 7.5 приведен вариант примера 7.4 для `BufferedReader`.

Пример 7.5 ❖ Использование метода `BufferedReader.lines`

```
try (Stream<String> lines =
    new BufferedReader(
        new FileReader("/usr/share/dict/words")).lines()) {
    // ...то же, что в предыдущем примере ...
}
```

И снова, поскольку `Stream` наследует `AutoCloseable`, при выходе из блока `try` с ресурсами поток автоматически закрывается, а вместе с ним и `BufferedReader`.

См. также

Сортировка отображений рассматривается в рецепте 4.4.

7.2. ПОЛУЧЕНИЕ ФАЙЛОВ В ВИДЕ ПОТОКА

Проблема

Требуется обработать все файлы в каталоге как поток.

Решение

Использовать статический метод `Files.list`.

Обсуждение

Статический метод `list` класса `java.nio.file.Files` принимает аргумент типа `Path` и возвращает поток, обернутый `DirectoryStream`¹. Интерфейс `Directo-`

¹ Это поток ввода-вывода, а не функциональный поток.

`guStream` расширяет `AutoCloseable`, поэтому при использовании метода `list` рекомендуется применять блок `try` с ресурсами, как в примере 7.6.

Пример 7.6 ❖ Использование метода `Files.list(path)`

```
try (Stream<Path> list = Files.list(Paths.get("src/main/java"))) {
    list.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

В предположении, что эта программа запускается в корне проекта со стандартной структурой Maven или Gradle, будут напечатаны имена всех файлов и папок в каталоге `src/main/java`. По выходе из блока `try` с ресурсами система вызовет метод `close` потока, который, в свою очередь, вызовет метод `close` интерфейса `DirectoryStream`. Рекурсивный обход каталога не производится.

Вот список каталогов и файлов, полученный при запуске этой программы для исходного кода настоящей книги:

```
src/main/java/collectors
src/main/java/concurrency
src/main/java/datetime
...
src/main/java/Summarizing.java
src/main/java/tasks
src/main/java/UseFilenameFilter.java
```

Из сигнатуры метода `list` видно, что он принимает каталог и возвращает значение типа `Stream<Path>`:

```
public static Stream<Path> list(Path dir) throws IOException
```

Если попытаться передать методу не каталог, то будет возбуждено исключение `NotDirectoryException`.

В документации подчеркивается, что результирующий поток *слабо состоятелен*, т. е. «он потокобезопасен, но во время обхода каталог не замораживается, поэтому изменения, произошедшие в каталоге после возврата из этого метода, необязательно будут отражены».

См. также

Обход файловой системы с помощью поиска в глубину описан в рецепте 7.3.

7.3. ОБХОД ФАЙЛОВОЙ СИСТЕМЫ

Проблема

Требуется обойти файловую систему в глубину.

Решение

Использовать статический метод `Files.walk`.

Обсуждение

Статический метод `Files.walk` из пакета `java.nio.file` имеет такую сигнатуру:

```
public static Stream<Path> walk(Path start,
                               FileVisitOption... options)
    throws IOException
```

Методу передаются начальный путь типа `Path` и список значений типа `FileVisitOption` переменной длины. Возвращаемый поток значений типа `Path` лениво заполняется в процессе обхода файловой системы в глубину, отправляясь от начального пути.

Возвращаемый поток инкапсулирует `DirectoryStream`, поэтому настоятельно рекомендуется вызывать этот метод из блока `try` с ресурсами, как показано в примере 7.7.

Пример 7.7 ❖ Обход дерева

```
try (Stream<Path> paths = Files.walk(Paths.get("src/main/java"))) {
    paths.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Метод `walk` принимает 0 или более значений типа `FileVisitOption` во втором и последующих аргументах. В этом примере не задано ни одно. `FileVisitOption` – перечисление, добавленное в версии Java 1.7, в котором определен лишь один элемент, `FileVisitOption.FOLLOW_LINKS`. Следование по ссылкам означает, что, по крайней мере в принципе, дерево может содержать цикл, поэтому поток запоминает посещенные файлы. Если обнаружен цикл, то возбуждается исключение `FileSystemLoopException`.

При запуске для исходного кода к этой книге получаются такие результаты:

```
src/main/java
src/main/java/collectors
src/main/java/collectors/Actor.java
src/main/java/collectors/AddCollectionToMap.java
src/main/java/collectors/Book.java
src/main/java/collectors/CollectorsDemo.java
src/main/java/collectors/ImmutableCollections.java
src/main/java/collectors/Movie.java
src/main/java/collectors/MysteryMen.java
src/main/java/concurrency
src/main/java/concurrency/CommonPoolSize.java
src/main/java/concurrency/CompletableFutureDemos.java
src/main/java/concurrency/FutureDemo.java
src/main/java/concurrency/ParallelDemo.java
src/main/java/concurrency/SequentialToParallel.java
src/main/java/concurrency/Timer.java
src/main/java/datetime
...
```

Пути обходятся лениво. Гарантируется, что результирующий поток содержит хотя бы один элемент – начальный путь, заданный в аргументе. Встречая очередной путь, система смотрит, является ли он каталогом, и если да, то обходит его до перехода к следующему пути на том же уровне. Таким образом, обход производится в глубину. После посещения всех элементов каждый каталог закрывается.

Существует также перегруженный вариант этого метода:

```
public static Stream<Path> walk(Path start,
                               int maxDepth,
                               FileVisitOption... options)
    throws IOException
```

Аргумент `maxDepth` – максимальное количество посещаемых уровней каталогов. Нуль означает, что нужно ограничиться только начальным уровнем. В варианте этого метода без параметра `maxDepth` предполагается значение `Integer.MAX_VALUE`, означающее, что нужно посещать все уровни.

См. также

Получение списка файлов в одном каталоге рассмотрено в рецепте 7.2. Поиск файлов обсуждается в рецепте 7.4.

7.4. ПОИСК В ФАЙЛОВОЙ СИСТЕМЕ

Проблема

Требуется найти в файловой системе файлы, обладающие заданными свойствами.

Решение

Использовать статический метод `Files.find` из пакета `java.nio.file`.

Обсуждение

Сигнатура метода `Files.find` имеет вид:

```
public static Stream<Path> find(Path start,
                               int maxDepth,
                               BiPredicate<Path, BasicFileAttributes> matcher,
                               FileVisitOption... options)
    throws IOException
```

Он похож на метод `walk`, но дополнительно принимает аргумент типа `BiPredicate` для определения того, следует ли возвращать путь. Метод `find` начинает работу с указанного пути и производит поиск в глубину не более чем на `maxDepth` уровней. Следовать ли по ссылкам, зависит от параметра `FileVisitOption`. Каждый посещенный путь проверяется на соответствие предикату `BiPredicate`.

Предикат `BiPredicate` должен вернуть булево значение, зависящее от пути и ассоциированного с ним объекта типа `BasicFileAttributes`. Так, в примере 7.8 просматривается весь исходный код к книге и возвращаются пути, соответствующие обычным файлам (не каталогам) из пакета `fileio`.

Пример 7.8 ❖ Поиск обычных файлов из пакета `fileio`

```
try (Stream<Path> paths =
    Files.find(Paths.get("src/main/java"), Integer.MAX_VALUE,
        (path, attributes) ->
            !attributes.isDirectory() && path.toString().contains("fileio"))) {
    paths.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Печатаются такие пути:

```
src/main/java/fileio/FileList.java
src/main/java/fileio/ProcessDictionary.java
src/main/java/fileio/SearchForFiles.java
src/main/java/fileio/WalkTheTree.java
```

Каждый файл, встретившийся при обходе дерева, сопоставляется с заданным предикатом. Результат такой же, как если бы мы вызвали метод `walk` с фильтром, но в документации утверждается, что этот подход может оказаться эффективнее, поскольку не производятся избыточные операции получения объектов `BasicFileAttributes`.

Как обычно, результирующий поток инкапсулирует объект `DirectoryStream`, поэтому закрытие потока закрывает и этот объект. Поэтому рекомендуется использовать блок `try` с ресурсами, как и показано в примере.

См. также

Обход файловой системы обсуждается в рецепте 7.3.

Глава 8

Пакет `java.time`

Хороший друг никогда не позволит другу использовать `java.util.Date`.

– Тим Йейтс

Начиная с самой первой версии языка в стандартную библиотеку были включены два класса для работы с датами и временем: `java.util.Date` и `java.util.Calendar`. Первый из них – классический пример того, как *не надо* проектировать класс. Практически все методы в его открытом API объявлены нерекомендуемыми – и это еще в версии Java 1.1 (приблизительно 1997 год). В комментариях рекомендуется использовать класс `Calendar`, хотя это тоже удовольствие ниже среднего.

Оба класса появились раньше, чем в язык были добавлены перечисления, поэтому для таких полей, как месяцы, используются целочисленные константы. Оба класса изменяемые и потому не являются потокобезопасными. Чтобы немного исправить положение, в библиотеку был позднее добавлен класс `java.sql.Date`, являющийся подклассом `java.util.Date`, но фундаментальных проблем он не решил.

Наконец, в версию Java SE 8 был включен абсолютно новый пакет, решивший все проблемы. Пакет `java.time` основан на библиотеке Joda-Time (<http://www.joda.org/joda-time/>), которая много лет использовалась как свободная альтернатива с открытым исходным кодом. Проектировщики Joda-Time помогли спроектировать и собрать новый пакет, и в будущих разработках рекомендуется использовать только его.

Новый пакет разработан в соответствии со спецификацией JSR-310 «Date and Time API» и поддерживает стандарт ISO 8601. Он корректно учитывает високосные годы и правила перехода на летнее время, действующие в разных странах.

В этой главе собраны рецепты, иллюстрирующие полезность пакета `java.time`. Хочется надеяться, что здесь вы найдете ответы на основные вопросы и ссылки на дополнительные источники информации.

Для справки можно использовать пособие Java Tutorial, в котором имеется великолепный раздел, посвященный библиотеке Date-Time. См. <https://docs.oracle.com/javase/tutorial/datetime/TOC.html>.

8.1. ОСНОВНЫЕ КЛАССЫ ДЛЯ РАБОТЫ С ДАТАМИ И ВРЕМЕНЕМ

Проблема

Желательно использовать новые классы для работы с датами и временем из пакета java.time.

Решение

Использовать фабричные методы классов Instant, Duration, Period, LocalDate, LocalTime, LocalDateTime, ZonedDateTime и других.

Обсуждение

Все новые классы порождают неизменяемые экземпляры, поэтому они потокобезопасны. Ни в одном из них нет открытых конструкторов, так что экземпляры создаются фабричными методами.

Особенно интересны два статических фабричных метода: now и of. Метод now служит для создания экземпляра, соответствующего текущей дате или моменту времени (пример 8.1).

Пример 8.1 ❖ Фабричный метод now

```
System.out.println("Instant.now(): " + Instant.now());
System.out.println("LocalDate.now(): " + LocalDate.now());
System.out.println("LocalTime.now(): " + LocalTime.now());
System.out.println("LocalDateTime.now(): " + LocalDateTime.now());
System.out.println("ZonedDateTime.now(): " + ZonedDateTime.now());
```

В примере 8.2 показаны результаты работы этого кода.

Пример 8.2 ❖ Результаты вызова метода now

```
Instant.now(): 2017-06-20T17:27:08.184Z
LocalDate.now(): 2017-06-20
LocalTime.now(): 13:27:08.318
LocalDateTime.now(): 2017-06-20T13:27:08.319
ZonedDateTime.now(): 2017-06-20T13:27:08.319-04:00[America/New_York]
```

Все выходные значения форматированы в соответствии со стандартом ISO 8601. Для дат основной формат уууу-мм-dd, для времени – hh:mm:ss.sss. Формат LocalDateTime является объединением того и другого с заглавной буквой T в качестве разделителя. Если требуется указать часовой пояс, то добавляется числовое смещение (здесь -04:00) от времени UTC, а также *название региона* (здесь America/New_York). Метод toString класса Instant показывает время с точностью до наносекунд по Гринвичу (время Z).

Метод now имеется также в классах Year, YearMonth и ZoneId.

Статический фабричный метод of используется для порождения новых значений. В случае LocalDate его аргументами являются год, месяц (число или элемент перечисления) и день месяца.



В качестве месяца во всех методах можно задавать как элемент перечисления `Month`, например `Month.JANUARY`, так и целое число, начиная с 1. Поскольку в классе `Calendar` нумерация месяцев начинается с нуля (т. е. `Calendar.JANUARY` равно 0), остерегайтесь ошибок со сдвигом на 1. Всюду, где возможно, используйте перечисление `Month`.

В случае `LocalTime` существует несколько перегруженных вариантов метода `of`, зависящих от того, сколько компонент времени (часы, минуты, секунды, наносекунды) задается. Метод `of` класса `LocalDateTime` принимает дату и время. См. пример 8.3.

Пример 8.3 ❖ Метод `of` классов для работы с датой и временем

```
System.out.println("Первая высадка на Луну:");
LocalDate moonLandingDate = LocalDate.of(1969, Month.JULY, 20);
LocalTime moonLandingTime = LocalTime.of(20, 18);
System.out.println("Дата: " + moonLandingDate);
System.out.println("Время: " + moonLandingTime);

System.out.println("Нил Армстронг выходит на поверхность: ");
LocalTime walkTime = LocalTime.of(20, 2, 56, 150_000_000);
LocalDateTime walk = LocalDateTime.of(moonLandingDate, walkTime);
System.out.println(walk);
```

Результат работы этой программы показан в примере 8.3.

```
Первая высадка на Луну:
Дата: 1969-07-20
Время: 20:18
Нил Армстронг выходит на поверхность:
1969-07-20T20:02:56.150
```

Последний аргумент метода `LocalTime.of` – количество наносекунд, поэтому в примере использована появившаяся в Java 7 возможность разделять группы цифр знаками подчеркивания для удобочитаемости.

Класс `Instant` моделирует один момент времени.

Класс `ZonedDateTime` объединяет дату и время с информацией о часовом поясе из класса `ZoneId`. Часовые пояса выражаются во всемирном координированном времени (UTC).

Существует два вида идентификации часового пояса:

- фиксированное смещение относительно UTC, например `-05:00`;
- географический регион, например `America/Chicago`.

Технически есть и третий тип идентификации – смещение относительно гринвичского меридиана (время «зулу»). Оно обозначается числовым значением с буквой Z.

Правила изменения смещения хранятся в классе `ZoneRules` и запрашиваются у поставщика `ZoneRulesProvider`. В классе `ZoneRules` есть такие методы, как `isDaylightSavings(Instant)`.

Получить текущее значение `ZoneId` можно от статического метода `systemDefault`. Полный список имеющихся идентификаторов регионов возвращает статический метод `getAvailableZoneIds`:


```
Set<String> regionNames = ZoneId.getAvailableZoneIds();
System.out.println("Существует " + regionNames.size() + " названий регионов");
```

В jdk1.8.0_131 имеется 600 названий регионов¹.

В Date-Time API используются стандартные префиксы для имен методов. Если вы знакомы с префиксами в табл. 8.1, то можете догадаться, что делает метод².

Таблица 8.1. Префиксы методов из Date-Time API

Метод	Тип	Назначение
of	Статический фабричный	Создает экземпляр
from	Статический фабричный	Преобразует входные параметры в целевой класс
parse	Статический фабричный	Разбирает входную строку
format	Экземпляра	Порождает форматированный вывод
get	Экземпляра	Возвращает часть объекта
is	Экземпляра	Запрашивает состояние объекта
with	Экземпляра	Создает новый объект, изменяя один элемент существующего
plus, minus	Экземпляра	Создает новый объект, прибавляя или вычитая что-то из существующего
to	Экземпляра	Преобразует объект в другой тип
at	Экземпляра	Объединяет этот объект с другим

Метод `of` был показан выше. Методы `parse` и `format` обсуждаются в рецепте 8.5. Метод `with` рассматривается в рецепте 8.2, это неизменяющий аналог метода установки. Там же рассматриваются методы `plus`, `minus` и их вариации.

В примере 8.4 демонстрируется использование метода `at` для добавления часового пояса к локальным дате и времени.

Пример 8.4 ❖ Добавление часового пояса к `LocalDateTime`

```
LocalDateTime dateTime = LocalDateTime.of(2017, Month.JULY, 4, 13, 20, 10);
ZonedDateTime nyc = dateTime.atZone(ZoneId.of("America/New_York"));
System.out.println(nyc);
```

```
ZonedDateTime london = nyc.withZoneSameInstant(ZoneId.of("Europe/London"));
System.out.println(london);
```

В результате печатается:

```
2017-07-04T13:20:10-04:00[America/New_York]
2017-07-04T18:20:10+01:00[Europe/London]
```

Как видим, метод `withZoneSameInstant` принимает время `ZonedDateTime` в одном часовом поясе и находит соответствующее ему время в другом часовом поясе.

В пакет входят два перечисления: `Month` и `DayOfWeek`. В перечислении `Month` определены константы для каждого месяца стандартного календаря (от `JANUA-`

¹ Может быть, так только мне кажется, но, по-моему, это много.

² Основана на аналогичной таблице из пособия Java Tutorial по адресу <https://docs.oracle.com/javase/tutorial/datetime/overview/naming.html>.

RY до DECEMBER). Там же есть много полезных методов, некоторые из которых показаны в примере 8.5.

Пример 8.5 ❖ Некоторые методы перечисления Month

```
System.out.println("Дней в високосном феврале: " +
    Month.FEBRUARY.length(true));
System.out.println("Порядковый номер первого дня августа в году (високосный год): " +
    Month.AUGUST.firstDayOfYear(true));
System.out.println("Month.of(1): " + Month.of(1));
System.out.println("Прибавление двух месяцев: " + Month.JANUARY.plus(2));
System.out.println("Вычитание месяца: " + Month.MARCH.minus(1));
```

❶ Аргумент – булева величина leapYear

Эта программа печатает такой результат:

```
Дней в високосном феврале: 29
Порядковый номер первого дня августа в году (високосный год): 214
Month.of(1): JANUARY
Прибавление двух месяцев: MARCH
Вычитание месяца: FEBRUARY
```

В последних двух примерах, где демонстрируются методы plus и minus, создаются новые экземпляры.

❷ Поскольку классы в пакете java.time неизменяемые, любой метод экземпляра, который, на первый взгляд, модифицирует объект, например plus, minus или with, на самом деле порождает новый объект.

Элементы перечисления DayOfWeek представляют семь дней недели, от MONDAY до SUNDAY. Значения элементов соответствуют стандарту ISO, т. е. MONDAY равно 1, а SUNDAY – 7.

См. также

Методы разбора и форматирования обсуждаются в рецепте 8.5. Преобразование существующей даты и времени в новую – тема рецепта 8.2. Классы Duration и Period рассматриваются в рецепте 8.8.

8.2. СОЗДАНИЕ ДАТЫ И ВРЕМЕНИ НА ОСНОВЕ СУЩЕСТВУЮЩИХ ЭКЗЕМПЛЯРОВ

Проблема

Требуется модифицировать экземпляр одного из классов даты/времени.

Решение

Если требуется просто сложить или вычесть, пользуйтесь методами plus или minus, в противном случае – методом with.

Обсуждение

Одна из особенностей нового Date-Time API заключается в том, что объекты всех классов неизменяемы. Один раз создав объект `LocalDate`, `LocalTime`, `LocalDateTime` или `ZonedDateTime`, вы уже не сможете его модифицировать. Это замечательно с точки зрения потокобезопасности, но что, если требуется создать новый экземпляр на основе существующего?

В классе `LocalDate` есть несколько методов для сложения и вычитания в применении к датам:

- `LocalDate plusDays(long daysToAdd);`
- `LocalDate plusWeeks(long weeksToAdd);`
- `LocalDate plusMonths(long monthsToAdd);`
- `LocalDate plusYears(long yearsToAdd);`

Каждый метод возвращает новый объект `LocalDate`, являющийся результатом прибавления указанного значения к существующей дате.

Аналогичные методы есть в классе `LocalTime`:

- `LocalTime plusNanos(long nanosToAdd);`
- `LocalTime plusSeconds(long secondsToAdd);`
- `LocalTime plusMinutes(long minutesToAdd);`
- `LocalTime plusHours(long hoursToAdd);`

Они также возвращают новый объект – результат сложения исходного с указанной величиной. В классе `LocalDateTime` имеются все методы из `LocalDate` и `LocalTime`. В примере 8.6 демонстрируется использование различных методов `plus` из классов `LocalDate` и `LocalTime`.

Пример 8.6 ❖ Использование методов `plus` для объектов `LocalDate` и `LocalTime`

```
@Test
public void localDatePlus() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");
    LocalDate start = LocalDate.of(2017, Month.FEBRUARY, 2);

    LocalDate end = start.plusDays(3);
    assertEquals("2017-02-05", end.format(formatter));

    end = start.plusWeeks(5);
    assertEquals("2017-03-09", end.format(formatter));

    end = start.plusMonths(7);
    assertEquals("2017-09-02", end.format(formatter));

    end = start.plusYears(2);
    assertEquals("2019-02-02", end.format(formatter));
}

@Test
public void localTimePlus() throws Exception {
    DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_TIME;
    LocalTime start = LocalTime.of(11, 30, 0, 0);

    LocalTime end = start.plusNanos(1_000_000);
```

```

assertEquals("11:30:00.001", end.format(formatter));

end = start.plusSeconds(20);
assertEquals("11:30:20", end.format(formatter));

end = start.plusMinutes(45);
assertEquals("12:15:00", end.format(formatter));

end = start.plusHours(5);
assertEquals("16:30:00", end.format(formatter));
}

```

В этих классах есть также два дополнительных метода, `plus` и `minus`. Вот их сигнатуры в классе `LocalDateTime`:

```

LocalDateTime plus(long amountToAdd, TemporalUnit unit)
LocalDateTime plus(TemporalAmount amountToAdd)

LocalDateTime minus(long amountToSubtract, TemporalUnit unit)
LocalDateTime minus(TemporalAmount amountToSubtract)

```

В классах `LocalDate` и `LocalTime` есть аналогичные методы, только они возвращают значения другого типа. Интересно, что методы `minus` просто вызывают соответственные методы `plus` с противоположным значением первого аргумента.

В методах, принимающих `TemporalAmount`, аргумент обычно имеет тип `Period` или `Duration`, но возможен любой тип, реализующий интерфейс `TemporalAmount`. В этом интерфейсе определены методы `addTo` и `subtractFrom`:

```

Temporal addTo(Temporal temporal)
Temporal subtractFrom(Temporal temporal)

```

Взглянув на стек вызовов, мы увидим, что `minus` вызывает `plus` с противоположным аргументом, а тот делегирует работу методу `TemporalAmount.addTo(Temporal)`, который, в свою очередь, вызывает `plus(long, TemporalUnit)`, и вот тот уже выполняет заказанную операцию¹.

В примере 8.7 демонстрируются методы `plus` и `minus`.

Пример 8.7 ❖ Методы `plus` и `minus`

```

@Test
public void plus_minus() throws Exception {
    Period period = Period.of(2, 3, 4); // 2 years, 3 months, 4 days
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);

    LocalDateTime end = start.plus(period);
    assertEquals("2019-05-06T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.plus(3, ChronoUnit.HALF_DAYS);
    assertEquals("2017-02-03T23:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}

```

¹ Слава косвенности!

```

end = start.minus(period);
assertEquals("2014-10-29T11:30:00",
    end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

end = start.minus(2, ChronoUnit.CENTURIES);
assertEquals("1817-02-02T11:30:00",
    end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

end = start.plus(3, ChronoUnit.MILLENNIA);
assertEquals("5017-02-02T11:30:00",
    end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}

```



Видя в API аргумент типа TemporalUnit, помните, что этот интерфейс реализован классом ChronoUnit, в котором определено много удобных констант.

Наконец, в каждом классе есть набор методов with, позволяющих изменять какое-то одно поле. Таких методов много, от withNano до withYear, сигнатуры наиболее интересных приведены ниже. Следующие методы определены в классе LocalDateTime:

```

LocalDateTime withNano(int nanoOfSecond)
LocalDateTime withSecond(int second)
LocalDateTime withMinute(int minute)
LocalDateTime withHour(int hour)
LocalDateTime withDayOfMonth(int dayOfMonth)
LocalDateTime withDayOfYear(int dayOfYear)
LocalDateTime withMonth(int month)
LocalDateTime withYear(int year)

```

В примере 8.8 демонстрируются их возможности.

Пример 8.8 ❖ Использование методов with класса LocalDateTime

```

@Test
public void with() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    LocalDateTime end = start.withMinute(45);
    assertEquals("2017-02-02T11:45:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withHour(16);
    assertEquals("2017-02-02T16:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withDayOfMonth(28);
    assertEquals("2017-02-28T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withDayOfYear(300);
    assertEquals("2017-10-27T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    end = start.withYear(2020);
    assertEquals("2020-02-02T11:30:00",

```

```

        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
    }

    @Test(expected = DateTimeException.class)
    public void withInvalidDate() throws Exception {
        LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
        start.withDayOfMonth(29);
    }

```

Поскольку 2017 год не високосный, нельзя присвоить дате 29 февраля. Такая попытка приводит к исключению `DateTimeException`, как показывает последний тест.

Существуют также методы `with`, принимающие объекты типа `TemporalAdjuster` или `TemporalField`:

```

LocalDateTime with(TemporalAdjuster adjuster)
LocalDateTime with(TemporalField field, long newValue)

```

Вариант, принимающий `TemporalField`, позволяет системе подправить дату. Так, в примере 8.9 мы взяли последний день января и попытались изменить месяц на февраль. Согласно документации, система выбирает предшествующую допустимую дату, в данном случае это будет последний день февраля.

Пример 8.9 ❖ Изменение месяца, так что получается недопустимая дата

```

@Test
public void temporalField() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.JANUARY, 31, 11, 30);
    LocalDateTime end = start.with(ChronoField.MONTH_OF_YEAR, 2);
    assertEquals("2017-02-28T11:30:00",
        end.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
}

```

Понятно, что существуют какие-то довольно сложные правила, но все они хорошо документированы.

Метод `with`, принимающий `TemporalAdjuster`, обсуждается в рецепте 8.3.

См. также

Классы `TemporalAdjuster` и `TemporalQuery` рассматриваются в рецепте 8.3.

8.3. КОРРЕКТОРЫ И ЗАПРОСЫ

Проблема

Имеется временное значение, и требуется подкорректировать его, руководствуясь собственной логикой, или получить о нем информацию.

Решение

Создать объект `TemporalAdjuster` или сформулировать запрос типа `TemporalQuery`.

Обсуждение

Классы TemporalAdjuster и TemporalQuery открывают интересные способы работы с классами из Date-Time API. Помимо ряда полезных встроенных методов, они позволяют реализовать свои собственные. В этом рецепте иллюстрируются обе возможности.

Работа с TemporalAdjuster

Интерфейс TemporalAdjuster предоставляет методы, принимающие значение типа Temporal и возвращающие результат его корректировки. Класс TemporalAdjusters содержит набор корректоров в виде удобных статических методов.

TemporalAdjuster используется путем вызова метода with от имени некоторого временного объекта, например:

```
LocalDateTime with(TemporalAdjuster adjuster)
```

В интерфейсе TemporalAdjuster определен метод adjustInto, который тоже работает, но способ, показанный выше, предпочтительнее.

Ниже перечислены методы класса TemporalAdjusters:

```
static TemporalAdjuster firstDayOfNextMonth()
static TemporalAdjuster firstDayOfNextYear()
static TemporalAdjuster firstDayOfYear()
static TemporalAdjuster firstInMonth(DayOfWeek dayOfWeek)

static TemporalAdjuster lastDayOfMonth()
static TemporalAdjuster lastDayOfYear()
static TemporalAdjuster lastInMonth(DayOfWeek dayOfWeek)

static TemporalAdjuster next(DayOfWeek dayOfWeek)
static TemporalAdjuster nextOrSame(DayOfWeek dayOfWeek)
static TemporalAdjuster previous(DayOfWeek dayOfWeek)
static TemporalAdjuster previousOrSame(DayOfWeek dayOfWeek)
```

В тестах из примера 8.10 показано несколько из этих методов в действии.

Пример 8.10 ❖ Использование статических методов класса TemporalAdjusters

```
@Test
public void adjusters() throws Exception {
    LocalDateTime start = LocalDateTime.of(2017, Month.FEBRUARY, 2, 11, 30);
    LocalDateTime end = start.with(TemporalAdjusters.firstDayOfNextMonth());
    assertEquals("2017-03-01T11:30", end.toString());

    end = start.with(TemporalAdjusters.next(DayOfWeek.THURSDAY));
    assertEquals("2017-02-09T11:30", end.toString());

    end = start.with(TemporalAdjusters.previousOrSame(DayOfWeek.THURSDAY));
    assertEquals("2017-02-02T11:30", end.toString());
}
```

Интересно становится, когда начинаешь писать собственный корректор. TemporalAdjuster – функциональный интерфейс с таким единственным абстрактным методом:

```
Temporal adjustInto(Temporal temporal)
```

В разделе пособия Java Tutorial, посвященном Date-Time API, приведен пример корректора `PaydayAdjuster`, в котором предполагается, что выплаты работникам производятся два раза в месяц. По правилам аванс выплачивается 15-го числа, а окончательный расчет – в последний день месяца. Но если дата выплаты приходится на выходной, то она производится в предшествующую пятницу.

В примере 8.11 воспроизведен код из пособия. Отметим, что метод включен в класс, реализующий интерфейс `TemporalAdjuster`.

Пример 8.11 ❖ `PaydayAdjuster` (из Java Tutorial)

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.Temporal;
import java.time.temporal.TemporalAdjuster;
import java.time.temporal.TemporalAdjusters;

public class PaydayAdjuster implements TemporalAdjuster {
    public Temporal adjustInto(Temporal input) {
        LocalDate date = LocalDate.from(input); ❶
        int day;
        if (date.getDayOfMonth() < 15) {
            day = 15;
        } else {
            day = date.with(TemporalAdjusters.lastDayOfMonth())
                .getDayOfMonth();
        }
        date = date.withDayOfMonth(day);
        if (date.getDayOfWeek() == DayOfWeek.SATURDAY ||
            date.getDayOfWeek() == DayOfWeek.SUNDAY) {
            date = date.with(TemporalAdjusters.previous(DayOfWeek.FRIDAY));
        }
        return input.with(date);
    }
}
```

❶ Полезный способ преобразовать произвольный временной объект в `LocalDate`

В июле 2017 года 15-е число пришлось на субботу, а 31-е – на понедельник. Тест в примере 8.12 показывает, что для июля 2017-го код дает правильный результат.

Пример 8.12 ❖ Тестирование корректора для июля 2017-го

```
@Test
public void payDay() throws Exception {
    TemporalAdjuster adjuster = new PaydayAdjuster();
    IntStream.rangeClosed(1, 14)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
```



```

        assertEquals(14, date.with(adjuster).getDayOfMonth());

    IntStream.rangeClosed(15, 31)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
            assertEquals(31, date.with(adjuster).getDayOfMonth()));
}

```

Это работает, но есть пара досадных мелочей. Во-первых, в Java 8 нельзя создать поток дат, не прибегая к какому-то обходному механизму, например к показанному выше подсчету дней. В Java 9 это исправлено – добавлен метод, возвращающий поток дат. Детали см. в рецепте 10.7.

Вторая мелочь состоит в том, что мы создали класс, реализующий интерфейс. Но, поскольку TemporalAdjuster – функциональный интерфейс, реализовать его можно было с помощью лямбда-выражения или ссылки на метод.

Напишем служебный класс Adjusters, содержащий статические методы для всего, что нам нужно, – как в примере 8.13.

Пример 8.13 ❖ Служебный класс, содержащий корректоры

```

public class Adjusters {
    public static Temporal adjustInto(Temporal input) { ❶
        LocalDate date = LocalDate.from(input); ❷
        // ... реализация такая же, как выше ...
        return input.with(date);
    }
}

```

- ❶ Не реализует TemporalAdjuster
- ❷ Статический метод, поэтому создавать объект не нужно

Эквивалентный тест показан в примере 8.14.

Пример 8.14 ❖ Использование ссылки на метод в качестве корректора

```

@Test
public void payDayWithMethodRef() throws Exception {
    IntStream.rangeClosed(1, 14)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
            assertEquals(14,
                date.with(Adjusters::adjustInto).getDayOfMonth())); ❶

    IntStream.rangeClosed(15, 31)
        .mapToObj(day -> LocalDate.of(2017, Month.JULY, day))
        .forEach(date ->
            assertEquals(31,
                date.with(Adjusters::adjustInto).getDayOfMonth()));
}

```

- ❶ Ссылка на метод adjustInto

Скорее всего, этот подход покажется вам более подходящим, если нужно написать несколько корректоров.

Работа с TemporalQuery

Интерфейс TemporalQuery используется как аргумент метода query, вызываемого от имени временных объектов. Например, в классе LocalDate метод query имеет такую сигнатуру:

```
<R> R query(TemporalQuery<R> query)
```

Этот метод вызывает TemporalQuery.queryFrom(TemporalAccessor), передавая this в качестве аргумента, и возвращает то, для чего предназначен запрос. Для вычислений доступны все методы интерфейса TemporalAccessor.

В составе API имеется класс TemporalQueries, в который включены константы для многих типичных запросов:

```
4static TemporalQuery<Chronology> chronology()
static TemporalQuery<LocalDate> localDate()
static TemporalQuery<LocalTime> localTime()
static TemporalQuery<ZoneOffset> offset()
static TemporalQuery<TemporalUnit> precision()
static TemporalQuery<ZoneId> zone()
static TemporalQuery<ZoneId> zoneId()
```

Простой тест в примере 8.15 показывает, как это работает.

Пример 8.15 ❖ Использование методов из класса TemporalQueries

```
@Test
public void queries() throws Exception {
    assertEquals(ChronoUnit.DAYS,
        LocalDate.now().query(TemporalQueries.precision()));
    assertEquals(ChronoUnit.NANOS,
        LocalTime.now().query(TemporalQueries.precision()));
    assertEquals(ZoneId.systemDefault(),
        ZonedDateTime.now().query(TemporalQueries.zone()));
    assertEquals(ZoneId.systemDefault(),
        ZonedDateTime.now().query(TemporalQueries.zoneId()));
}
```

Но, как и в случае с TemporalAdjuster, самое интересное – написать свой запрос. В интерфейсе TemporalQuery есть только один абстрактный метод:

```
R queryFrom(TemporalAccessor temporal)
```

Допустим, у нас имеется метод, который получает объект типа TemporalAccessor и вычисляет количество дней между датой, заданной в аргументе, и 19 сентября, международным днем «Говори, как пират»¹. Такой метод показан в примере 8.16.

Пример 8.16 ❖ Метод, вычисляющий, сколько дней осталось до пиратского праздника

```
private long daysUntilPirateDay(TemporalAccessor temporal) {
    int day = temporal.get(ChronoField.DAY_OF_MONTH);
```

¹ Эй, братишка, а я бы не прочь вкрячить тебя в свою профессиональную сетку на LinkedIn.

```

    int month = temporal.get(ChronoField.MONTH_OF_YEAR);
    int year = temporal.get(ChronoField.YEAR);
    LocalDate date = LocalDate.of(year, month, day);
    LocalDate tlapd = LocalDate.of(year, Month.SEPTEMBER, 19);
    if (date.isAfter(tlapd)) {
        tlapd = tlapd.plusYears(1);
    }
    return ChronoUnit.DAYS.between(date, tlapd);
}

```

Поскольку сигнатура этого метода совместима с единственным абстрактным методом интерфейса `TemporalQuery`, мы можем вызывать его по ссылке, как показано в примере 8.17.

Пример 8.17 ❖ Выполнение запроса `TemporalQuery` с помощью ссылки на метод

```

@Test
public void pirateDay() throws Exception {
    IntStream.range(10, 19)
        .mapToObj(n -> LocalDate.of(2017, Month.SEPTEMBER, n))
        .forEach(date ->
            assertTrue(date.query(this::daysUntilPirateDay) <= 9));
    IntStream.rangeClosed(20, 30)
        .mapToObj(n -> LocalDate.of(2017, Month.SEPTEMBER, n))
        .forEach(date -> {
            Long days = date.query(this::daysUntilPirateDay);
            assertTrue(days >= 354 && days < 365);
        });
}

```

Этот подход можно использовать и для определения собственных запросов.

8.4. ПРЕОБРАЗОВАНИЕ `JAVA.UTIL.DATE` В `JAVA.TIME.LOCALDATE`

Проблема

Требуется преобразовать объект типа `java.util.Date` или `java.util.Calendar` в один из новых классов, присутствующих в пакете `java.time`.

Решение

Использовать класс `Instant` как мост, либо воспользоваться методами классов `java.sql.Date` и `java.sql.Timestamp`, либо даже произвести преобразование через промежуточную строку или целое число.

Обсуждение

Изучая новые классы в пакете `java.time`, мы с удивлением отмечаем недостаток встроенных механизмов преобразования из стандартных классов даты и времени в пакете `java.util` в рекомендуемые новые классы.

Один из возможных подходов к преобразованию `java.util.Date` в `java.time.LocalDate` – вызвать метод `toInstant` для создания объекта `Instant`. Затем к нему можно применить часовой пояс `ZoneId` по умолчанию и извлечь из получившегося объекта `ZonedDateTime` локальную дату `LocalDate`, как показано в примере 8.18.

Пример 8.18 ❖ Преобразование `java.util.Date` в `java.time.LocalDate` через `Instant`

```
public LocalDate convertFromUtilDateUsingInstant(Date date) {
    return date.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
}
```

Поскольку в объекте класса `java.util.Date` хранится информация о дате и времени, но нет часового пояса¹, в новом API он представляет момент времени `Instant`. Применяв метод `atZone` с указанием системного часового пояса, мы восстановим часовой пояс. А уже потом из получившегося объекта `ZonedDateTime` можно извлечь `LocalDate`.

Другой подход к преобразованию всех дат из пакета `util` в даты в смысле `Date-Time API` – воспользоваться вспомогательными методами преобразования в классах `java.sql.Date` (пример 8.19) и `java.sql.Timestamp` (пример 8.20).

Пример 8.19 ❖ Методы преобразования в классе `java.sql.Date`

```
LocalDate toLocalDate()
static Date valueOf(LocalDate date)
```

Пример 8.20 ❖ Методы преобразования в классе `java.sql.Timestamp`

```
LocalDateTime toLocalDateTime()
static Timestamp valueOf(LocalDateTime dateTime)
```

Создать класс для преобразования нетрудно.

Пример 8.21 ❖ Преобразование классов из `java.util` в классы из `java.time` (это только часть)

```
package datetime;

import java.sql.Timestamp;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Date;

public class ConvertDate {
    public LocalDate convertFromSqlDatetoLD(java.sql.Date sqlDate) {
        return sqlDate.toLocalDate();
    }

    public java.sql.Date convertToSqlDateFromLD(LocalDate localDate) {
        return java.sql.Date.valueOf(localDate);
    }
}
```

¹ При печати отформатированного объекта `java.util.Date` используется часовой пояс, подразумеваемый в Java по умолчанию.

Поскольку нужные нам методы находятся в классе `java.sql.Date`, возникает вопрос: как преобразовать класс `java.util.Date` (который в основном и используют разработчики) в `java.sql.Date`? Можно, например, воспользоваться конструктором `java.sql.Date`, который принимает количество миллисекунд от начального момента в виде числа типа `long`.

Начальный момент и Java

В операционных системах, производных от Unix, *начальным моментом* называется время 00:00:00 UTC 1 января 1970 г. При отсчете от этого момента високосные секунды не учитываются. Системные часы в современных компьютерах основаны на этом значении.

Отметим, что количество секунд от начального значения перестанет помещаться в 32-разрядное число со знаком в 3:14:07 UTC 19 января 2038 г., в этот момент все 32-разрядные операционные системы внезапно станут считать, что на дворе 13 декабря 1901 г. Это так называемая «проблема 2038 года»¹. Хотя к тому времени практически все операционные системы должны стать 64-разрядными, встраиваемые системы обновляются редко, а то и вовсе никогда².

В Java астрономическое время измеряется в миллисекундах, что, на первый взгляд, усугубляет проблему, но зато для его хранения используется тип `long`, а не `int`, что дает нам фору в несколько тысяч лет.

В классе `java.util.Date` имеется метод `getTime`, который возвращает значение типа `long`, а в классе `java.sql.Date` – конструктор, принимающий `long` в качестве аргумента³.

Следовательно, у нас есть еще один метод преобразования объекта `java.util.Date` в `java.time.LocalDate` – через промежуточный объект `java.sql.Date`, как показано в примере 8.22.

Пример 8.22 ❖ Преобразование `java.util.Date` в `java.time.LocalDate`

```
public LocalDate convertUtilDateToLocalDate(java.util.Date date) {  
    return new java.sql.Date(date.getTime()).toLocalDate()  
}
```

Еще в версии Java 1.1 практически весь класс `java.util.Date` был объявлен *нерекомендуемым*, а ему на смену пришел `java.util.Calendar`. Преобразование между экземплярами последнего и новыми классами из пакета `java.time` можно выполнить с помощью метода `toInstant`, подкорректировав часовой пояс (пример 8.23).

¹ Подробности см. в статье https://en.wikipedia.org/wiki/Year_2038_problem.

² Я надеюсь, что к тому моменту мирно уйду на пенсию, но не хотел бы оказаться на искусственной вентиляции легких, когда это случится.

³ На самом деле это единственный не объявленный *нерекомендуемым* конструктор класса `java.sql.Date`, хотя для корректировки существующего значения можно также использовать метод `setTime`.

Пример 8.23 ❖ Преобразование java.util.Calendar в java.time.ZonedDateTime

```
public ZonedDateTime convertFromCalendar(Calendar cal) {
    return ZonedDateTime.ofInstant(cal.toInstant(), cal.getTimeZone().toZoneId());
}
```

В этом методе применен класс ZonedDateTime. В классе LocalDateTime также имеется метод ofInstant, но по какой-то причине он тоже принимает во втором аргументе объект типа ZoneId. Это странно, потому что в LocalDateTime информация о часовом поясе не хранится. Поэтому интуитивно кажется более естественным использовать метод из класса ZonedDateTime.

Можно также явно воспользоваться различными методами чтения класса Calendar и перейти прямо к LocalDateTime (пример 8.24), если вы хотите вообще проигнорировать часовой пояс.

Пример 8.24 ❖ Использование методов чтения из класса Calendar для преобразования в LocalDateTime

```
public LocalDateTime convertFromCalendarUsingGetters(Calendar cal) {
    return LocalDateTime.of(cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH),
        cal.get(Calendar.DAY_OF_MONTH),
        cal.get(Calendar.HOUR),
        cal.get(Calendar.MINUTE),
        cal.get(Calendar.SECOND));
}
```

Еще один способ – создать отформатированное строковое представление объекта календаря, а затем сконструировать объект нового класса, разобрав эту строку (пример 8.25).

Пример 8.25 ❖ Создание и разбор строкового представления временной метки

```
public LocalDateTime convertFromUtilDateToLDUsingString(Date date) {
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
    return LocalDateTime.parse(df.format(date),
        DateTimeFormatter.ISO_LOCAL_DATE_TIME);
}
```

Вообще-то, никакого преимущества это не дает, но полезно знать, что и такая возможность существует. Наконец, хотя в классе Calendar прямого метода преобразования нет, оказывается, что он имеется в классе GregorianCalendar.

Пример 8.26 ❖ Преобразование GregorianCalendar в ZonedDateTime

```
public ZonedDateTime convertFromGregorianCalendar(Calendar cal) {
    return ((GregorianCalendar) cal).toZonedDateTime();
}
```

Это работает, но в предположении, что используется григорианский календарь. Поскольку это единственная реализация Calendar в стандартной библиотеке, то, скорее всего, так и есть, но стопроцентной гарантии дать нельзя.

Наконец, в Java 9 в класс `LocalDate` добавлен метод `ofInstant`, так что преобразование упростилось (пример 8.27).

Пример 8.27 ❖ Преобразование `java.util.Date` в `java.time.LocalDate` (ТОЛЬКО В JAVA 9)

```
public LocalDate convertFromUtilDateJava9(Date date) {
    return LocalDate.ofInstant(date.toInstant(), ZoneId.systemDefault());
}
```

Это более прямолинейный подход, но работает он только в Java 9.

8.5. РАЗБОР И ФОРМАТИРОВАНИЕ

Проблема

Требуется разобрать или отформатировать объекты новых классов даты и времени.

Решение

Класс `DateTimeFormatter` создает форматы даты и времени, которые можно использовать для разбора и форматирования.

Обсуждение

В классе `DateTimeFormatter` имеется много вариантов форматирования – от констант вида `ISO_LOCAL_DATE` до буквенных образцов вида `uuuu-MMM-dd` и локализованных стилей с учетом заданной локали.

По счастью, сам процесс разбора и форматирования почти тривиален. Во всех основных классах даты и времени имеются методы `format` и `parse`. В примере 8.28 показаны сигнатуры методов из класса `LocalDate`:

Пример 8.28 ❖ Методы разбора и форматирования объектов `LocalDate`

```
static LocalDate parse(CharSequence text) ❶
static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
String format(DateTimeFormatter formatter)
```

❶ Используется `ISO_LOCAL_DATE`

Пример 8.29 ❖ Разбор и форматирование `LocalDate`

```
LocalDateTime now = LocalDateTime.now();
String text = now.format(DateTimeFormatter.ISO_DATE_TIME); ❶
LocalDateTime dateTime = LocalDateTime.parse(text); ❷
```

❶ Форматирование `LocalDateTime`

❷ Разбор строки с порождением `LocalDateTime`

Интересно поэкспериментировать с различными форматами даты и времени, локалями и т. д. (пример 8.30).

Пример 8.30 ❖ Форматирование дат

```

LocalDate date = LocalDate.of(2017, Month.MARCH, 13);

System.out.println("Full : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));
System.out.println("Long : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG)));
System.out.println("Medium : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)));
System.out.println("Short : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT)));

System.out.println("France : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(Locale.FRANCE)));
System.out.println("India : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(new Locale("hin", "IN"))));
System.out.println("Brazil : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(new Locale("pt", "BR"))));
System.out.println("Japan : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(Locale.JAPAN)));

Locale loc = new Locale.Builder()
    .setLanguage("sr")
    .setScript("Latn")
    .setRegion("RS")
    .build();
System.out.println("Serbian : " +
    date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
        .withLocale(loc)));

```

В результате будет напечатано¹:

```

Full : Monday, March 13, 2017
Long : March 13, 2017
Medium : Mar 13, 2017
Short : 3/13/17
France : lundi 13 mars 2017
India : Monday, March 13, 2017
Brazil : Segunda-feira, 13 de Março de 2017
Japan : 2017 年3 月13 日
Serbian: ponedeljak, 13. mart 2017.

```

Методы `parse` и `format` возбуждают соответственно исключения `DateTimeParseException` и `DateTimeException`, которые имеет смысл перехватывать.

¹ Неправда, что я специально выбрал необычные языки и форматы вывода, чтобы подвергнуть испытанию способность издательства O'Reilly правильно напечатать результаты. По крайней мере, сознательно я этого не делал.

Если вы хотите создать свой формат, воспользуйтесь методом `ofPattern`. Доступимые значения подробно описаны в документации (см. пример 8.31).

Пример 8.31 ❖ Определение своего формата

```
ZonedDateTime moonLanding = ZonedDateTime.of(
    LocalDate.of(1969, Month.JULY, 20),
    LocalTime.of(20, 18),
    ZoneId.of("UTC")
);
System.out.println(moonLanding.format(DateTimeFormatter.ISO_ZONED_DATE_TIME));

DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("uuuu/MMMM/dd hh:mm:ss a zzz GG");
System.out.println(moonLanding.format(formatter));

formatter = DateTimeFormatter.ofPattern("uuuu/MMMM/dd hh:mm:ss a VV xxxxx");
System.out.println(moonLanding.format(formatter));
```

В результате печатается:

```
1969-07-20T20:18:00Z[UTC]
1969/July/20 08:18:00 PM UTC AD
1969/July/20 08:18:00 PM UTC +00:00
```

Что именно можно сделать и что обозначают различные буквы формата, описано в документации по классу `DateTimeFormatter`.

Для демонстрации локализованного форматера даты и времени рассмотрим проблему перехода на летнее время. В Восточном часовом поясе США в 2018 году переход на летнее время производится 11 марта в 2 часа пополудни, стрелки переводятся на один час вперед. Что произойдет, если в этот день запросить поясное время в 2:30 пополудни? См. пример 8.32.

Пример 8.32 ❖ Перевод часов вперед

```
ZonedDateTime zdt = ZonedDateTime.of(2018, 3, 11, 2, 30, 0, 0,
    ZoneId.of("America/New_York"));
System.out.println(
    zdt.format(DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL)));
```

Здесь используется перегруженный вариант метода `of`, который принимает год, месяц, день месяца, часы, минуты, секунды, наносекунды и часовой пояс `ZoneId`. Отметим, что все аргументы (кроме `ZoneId`) имеют тип `int`, т. е. использовать перечисление `Month` невозможно.

Эта программа печатает:

```
Sunday, March 11, 2018 3:30:00 AM EDT
```

Таким образом, метод корректно изменил время 2:30 (несуществующее) на 3:30.

8.6. НАХОЖДЕНИЕ ЧАСОВЫХ ПОЯСОВ С НЕОБЫЧНЫМ СМЕЩЕНИЕМ

Проблема

Требуется найти все часовые пояса со смещением, равным нецелому количеству часов.

Решение

Получить смещение для каждого часового пояса в секундах и вычислить остаток от его деления на 3600.

Обсуждение

Для большинства часовых поясов смещение от UTC кратно одному часу. Так, восточное поясное время в США – это UTC-05:00, а время в метрополии Франции (CET) – UTC+01:00. Но встречаются часовые пояса со смещением, кратным получасу, например индийское стандартное время (IST) равно UTC+05:30. Есть даже смещения на 45 минут, например для островов Чатем в Новой Зеландии оно равно UTC+12:45. В этом рецепте показано, как с помощью пакета `java.time` найти все часовые пояса с нецелым смещением.

В примере 8.33 показано, как найти `ZoneOffset` для всех часовых поясов и сравнить его значение в секундах с количеством секунд в часе.

Пример 8.33 ❖ Нахождение смещения в секундах для каждого часового пояса

```
public class FunnyOffsets {
    public static void main(String[] args) {
        Instant instant = Instant.now();
        ZonedDateTime current = instant.atZone(ZoneId.systemDefault());
        System.out.printf("Текущее время %s%n", current);

        System.out.printf("%10s %20s %13s%n", "Offset", "ZoneId", "Time");
        ZoneId.getAvailableZoneIds().stream()
            .map(ZoneId::of)           ❶
            .filter(zoneId -> {
                ZoneOffset offset = instant.atZone(zoneId).getOffset();    ❷
                return offset.getTotalSeconds() % (60 * 60) != 0;           ❸
            })
            .sorted(comparingInt(zoneId ->
                instant.atZone(zoneId).getOffset().getTotalSeconds()))
            .forEach(zoneId -> {
                ZonedDateTime zdt = current.withZoneSameInstant(zoneId);
                System.out.printf("%10s %25s %10s%n",
                    zdt.getOffset(), zoneId,
```

```

        zdt.format(DateTimeFormatter.ofLocalizedTime(
            FormatStyle.SHORT)));
    });
}
}

```

- ❶ Отобразить строковый идентификатор региона на часовой пояс
- ❷ Вычислить смещение
- ❸ Только часовые пояса, для которых смещение не делится на 3600

Статический метод `ZoneId.getAvailableZoneIds` возвращает множество `Set<String>`, содержащее идентификаторы всех регионов мира. С помощью метода `ZoneId.of` поток строк преобразуется в поток объектов типа `ZoneId`.

Лямбда-выражение в фильтре сначала применяет метод `atZone` к объекту `Instant`, чтобы получить объект `ZonedDateTime`, у которого имеется метод `getOffset`. Наконец, в классе `ZoneOffset` имеется метод `getTotalSeconds`. В документации по этому методу сказано, что это «основной способ получить величину смещения. Он возвращает сумму часов, минут и секунд в виде единого смещения, которое можно прибавить к времени». Предикат фильтра возвращает `true` только для тех поясов, для которых полное число секунд нацело делится на 3600 (60 с/мин * 60 мин/час).

Перед печатью объекты `ZoneId` сортируются. Метод `sorted` принимает компаратор. В данном случае используется статический метод `Comparator.comparingInt`, который порождает компаратор для сортировки по заданному целочисленному ключу. Здесь мы используем уже встречавшееся ранее вычисление, чтобы получить смещение в секундах. В результате объекты `ZoneId` оказываются отсортированы по величине смещения.

Затем для каждого часового пояса вызывается метод `withZoneSameInstant`, который вычисляет текущее время `ZonedDateTime` в нем. В печатаемой строке указываются смещение, идентификатор часового пояса и отформатированное локализованное время в этом поясе.

Результат показан в примере 8.34.

Пример 8.34 ❖ Смещения часовых поясов, не кратные одному часу

```

Current time is 2016-08-08T23:12:44.264-04:00[America/New_York]
Offset      ZoneId      Time
-09:30 Pacific/Marquesas  5:42 PM
-04:30 America/Caracas  10:42 PM
-02:30 America/St_Johns  12:42 AM
-02:30 Canada/Newfoundland 12:42 AM
+04:30 Iran 7:42 AM
+04:30 Asia/Tehran 7:42 AM
+04:30 Asia/Kabul 7:42 AM
+05:30 Asia/Kolkata 8:42 AM
+05:30 Asia/Colombo 8:42 AM
+05:30 Asia/Calcutta 8:42 AM
+05:45 Asia/Kathmandu 8:57 AM
+05:45 Asia/Katmandu 8:57 AM
+06:30 Asia/Rangoon 9:42 AM

```

```
+06:30 Indian/Cocos 9:42 AM
+08:45 Australia/Eucla 11:57 AM
+09:30 Australia/North 12:42 PM
+09:30 Australia/Yancowinna 12:42 PM
+09:30 Australia/Adelaide 12:42 PM
+09:30 Australia/Broken_Hill 12:42 PM
+09:30 Australia/South 12:42 PM
+09:30 Australia/Darwin 12:42 PM
+10:30 Australia/Lord_Howe 1:42 PM
+10:30 Australia/LHI 1:42 PM
+11:30 Pacific/Norfolk 2:42 PM
+12:45 NZ-CHAT 3:57 PM
+12:45 Pacific/Chatham 3:57 PM
```

На этом примере видно, как с помощью комбинации нескольких классов из пакета `java.time` можно решить интересную задачу.

8.7. НАХОЖДЕНИЕ НАЗВАНИЙ РЕГИОНОВ ПО СМЕЩЕНИЮ

Проблема

Требуется по заданному смещению от UTC узнать название региона по стандарту ISO 8601.

Решение

Профильтровать все имеющиеся идентификаторы часовых поясов по смещению.

Обсуждение

Названия часовых поясов, например «Eastern Daylight Time» или «Indian Standard Time», хорошо известны, но они неофициальны, а их сокращения (EDT или IST) не всегда уникальны. В стандарте ISO 8601 часовые пояса идентифицируются двумя способами:

- по названию региона, например «America/Chicago»;
- по смещению от UTC в часах и минутах, например «+05:30».

Допустим, требуется узнать название региона по заданному смещению от UTC. Одно и то же смещение может быть у многих регионов, но легко вычислить список названий.

Класс `ZoneOffset` описывает смещение часового пояса от UTC. Если величина смещения известна, то мы можем профильтровать по нему весь список названий регионов, как показано в примере 8.35.

Пример 8.35 ❖ Получение названий регионов с заданным смещением

```
public static List<String> getRegionNamesForOffset(ZoneOffset offset) {
    LocalDateTime now = LocalDateTime.now();
    return ZoneId.getAvailableZoneIds().stream()
```

```

        .map(ZoneId::of)
        .filter(zoneId -> now.atZone(zoneId).getOffset().equals(offset))
        .map(ZoneId::toString)
        .sorted()
        .collect(Collectors.toList());
    }

```

Метод `ZoneId.getAvailableZoneIds` возвращает список строк. Каждую строку можно отобразить на `ZoneId` с помощью статического метода `ZoneId.of`. Затем мы определяем соответствующий этому `ZoneId` объект `ZonedDateTime`, применяя метод `atZone` класса `LocalDateTime`, получаем для него смещение `ZoneOffset` и оставляем только те пояса, для которых смещение равно заданному. Эти пояса отображаются на строки, которые сортируются и собираются в список.

Как получить `ZoneOffset`? Если известен идентификатор `ZoneId`, то можно действовать, как показано в примере 8.36.

Пример 8.36 ❖ Получить названия регионов с заданным смещением

```

public static List<String> getRegionNamesForZoneId(ZoneId zoneId) {
    LocalDateTime now = LocalDateTime.now();
    ZonedDateTime zdt = now.atZone(zoneId);
    ZoneOffset offset = zdt.getOffset();
    return getRegionNamesForOffset(offset);
}

```

Это работает для любого `ZoneId`.

Например, если мы хотим получить список названий регионов с таким же смещением, как у текущего местоположения, то можем воспользоваться кодом из примера 8.37.

Пример 8.37 ❖ Получение списка регионов с таким же смещением, как у текущего

```

@Test
public void getRegionNamesForSystemDefault() throws Exception {
    ZonedDateTime now = ZonedDateTime.now();
    ZoneId zoneId = now.getZone();
    List<String> names = getRegionNamesForZoneId(zoneId);

    assertTrue(names.contains(zoneId.getId()));
}

```

Если название региона неизвестно, но мы знаем, на сколько часов и минут оно смещено от GMT, то можно воспользоваться вспомогательным методом `ofHoursMinutes` класса `ZoneOffset`. В примере 8.38 показано, как это сделать.

Пример 8.38 ❖ Получение названий регионов с заданным смещением в часах и минутах

```

public static List<String> getRegionNamesForOffset(int hours, int minutes) {
    ZoneOffset offset = ZoneOffset.ofHoursMinutes(hours, minutes);
    return getRegionNamesForOffset(offset);
}

```

В примере 8.39 приведены тесты для этого метода.

Пример 8.39 ❖ Тестирование метода получения названий регионов по смещению

```
@Test
public void getRegionNamesForGMT() throws Exception {
    List<String> names = getRegionNamesForOffset(0, 0);

    assertTrue(names.contains("GMT"));
    assertTrue(names.contains("Etc/GMT"));
    assertTrue(names.contains("Etc/UTC"));
    assertTrue(names.contains("UTC"));
    assertTrue(names.contains("Etc/Zulu"));
}

@Test
public void getRegionNamesForNepal() throws Exception {
    List<String> names = getRegionNamesForOffset(5, 45);

    assertTrue(names.contains("Asia/Kathmandu"));
    assertTrue(names.contains("Asia/Katmandu"));
}

@Test
public void getRegionNamesForChicago() throws Exception {
    ZoneId chicago = ZoneId.of("America/Chicago");
    List<String> names = RegionIdsByOffset.getRegionNamesForZoneId(chicago);

    assertTrue(names.contains("America/Chicago"));
    assertTrue(names.contains("US/Central"));
    assertTrue(names.contains("Canada/Central"));
    assertTrue(names.contains("Etc/GMT+5") || names.contains("Etc/GMT+6"));
}
```

Полный список названий регионов можно найти в Википедии по адресу https://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

8.8. ВРЕМЯ МЕЖДУ СОБЫТИЯМИ

Проблема

Требуется узнать время между двумя событиями.

Решение

Если нужно время в виде, понятном человеку, то использовать метод `between` или `until` временных классов или метод `between` класса `Period`. Все они порождают объект типа `Period`. В противном случае использовать класс `Duration` для получения длительности промежутка на временной шкале в секундах и наносекундах.

Обсуждение

В составе Date-Time API имеется интерфейс `java.time.temporal.TemporalUnit`, реализованный перечислением `ChronoUnit`, находящимся в том же пакете. Ме-

тод `between` этого интерфейса принимает два объекта типа `TemporalUnit` и возвращает `long`:

```
long between(Temporal temporal1Inclusive,
             Temporal temporal2Exclusive)
```

Типы начала и конца промежутка должны быть совместимы. До начала вычислений реализация преобразует второй аргумент к типу первого. Если второй момент раньше первого, результат будет отрицательным.

Метод возвращает количество «временных единиц» между аргументами. Это особенно удобно, если использовать константы из перечисления `ChronoUnit`.

Пусть, например, нужно узнать, сколько дней осталось до определенной даты. Тогда нужно указать константу `ChronoUnit.DAYS`, как показано в примере 8.40.

Пример 8.40 ❖ Сколько дней осталось до дня голосования

```
LocalDate electionDay = LocalDate.of(2020, Month.NOVEMBER, 3);
LocalDate today = LocalDate.now();
```

```
System.out.printf("Осталось %d дней...\n",
                  ChronoUnit.DAYS.between(today, electionDay));
```

Поскольку метод `between` вызывается от имени элемента перечисления `DAYS`, будет возвращено количество дней. В `ChronoUnit` определены также константы `HOURS`, `WEEKS`, `MONTHS`, `YEARS`, `DECADES`, `CENTURIES` и другие¹.

Использование класса `Period`

Если вы хотите разбить промежуток времени на годы, месяцы и дни, пользуйтесь классом `Period`. Метод `until`, существующий во многих основных классах, имеет перегруженный вариант, который возвращает `Period`:

```
// В java.time.LocalDate
Period until(ChronoLocalDate endDateExclusive)
```

Приведенный выше код можно переписать, как в примере 8.41.

Пример 8.41 ❖ Использование класса `Period` для получения количества лет, месяцев и дней

```
LocalDate electionDay = LocalDate.of(2020, Month.NOVEMBER, 3);
LocalDate today = LocalDate.now();
```

```
Period until = today.until(electionDay);
```

```
years = until.getYears();
months = until.getMonths();
```

¹ Включая, хотите верить, хотите нет, константу `FOREVER` (вечно). Если вы найдете ей применение, дайте мне знать, мне очень любопытно, для чего бы она могла понадобиться.

```
days = until.getDays();
System.out.printf("%d year(s), %d month(s), and %d day(s)%n",
    years, months, days);
```

❶ Эквивалентно `Period.between(today, electionDay)`

Как следует из комментария, в классе `Period` имеется также статический метод `between`, работающий аналогично. Пользуйтесь тем, что вам больше по вкусу.

Класс `Period` используется, когда нужно представить промежуток времени в понятном человеку виде – количество лет, месяцев, дней и т. п.

Использование класса *Duration*

Класс `Duration` представляет промежуток времени в виде количества секунд и наносекунд и потому удобен для работы с классом `Instant`. Результат можно преобразовать в различные типы. В классе хранится число типа `long`, равное числу секунд, и число типа `int`, представляющее наносекунды. Эти числа могут быть отрицательны, если начальный момент позже конечного.

В примере 8.42 демонстрируется примитивный способ хронометража с помощью класса `Duration`.

Пример 8.42 ❖ Хронометраж работы метода

```
public static double getTiming(Instant start, Instant end) {
    return Duration.between(start, end).toMillis() / 1000.0;
}
```

```
Instant start = Instant.now();
// ... вызвать хронометрируемый метод ...
Instant end = Instant.now();
System.out.println(getTiming(start, end) + " секунд");
```

Это хронометраж «для бедных», но зато всё просто.

В классе `Duration` есть методы преобразования: `toDays`, `toHours`, `toMillis`, `toMinutes` и `toNanos`. В частности, в примере 8.42 мы воспользовались методом `toMillis`, а затем поделили результат на 1000, чтобы получить количество секунд.

Глава 9

Параллелизм и конкурентность

Эта глава посвящена вопросам параллелизма и конкурентности в Java 8. Некоторые идеи восходят к гораздо более ранним добавлениям в язык (особенно к пакету `java.util.concurrent`, добавленному в Java 5), но именно в Java 8 появился ряд средств, позволяющих работать на более высоком уровне абстракции.

Одна из проблем, связанных с параллелизмом и конкурентностью, состоит в том, что стоит вам заговорить о них, как кто-нибудь потребует – и весьма громко – правильно употреблять слова. Давайте решим этот вопрос сразу:

- *конкурентность* – это когда несколько задач работает в чередующиеся и частично перекрывающиеся промежутки времени;
- *параллелизм* – это когда несколько задач работает строго одновременно.

Программа проектируется для работы в конкурентном режиме, т. е. мы разлагаем ее на независимые операции, которые могут работать одновременно, пусть даже не строго в один и тот же момент. Конкурентное приложение состоит из независимо выполняемых процессов. Конкурентные задачи можно затем запустить параллельно. Повысится ли при этом производительность, зависит от количества процессорных блоков¹.

Почему параллелизм не всегда повышает производительность? Причин много, но в Java распараллеливание по умолчанию сводится к разделению работы на несколько частей, назначению каждой части процесса из общего пула разветвления-соединения и последующему объединению результатов. Какого повышения производительности можно при этом ожидать, зависит от того, насколько хорошо задача поддается такому алгоритму. Один из рецептов в этой главе содержит рекомендации по поводу того, как принимать решение о целесообразности распараллеливания.

¹ Прекрасное и относительно короткое обсуждение этих понятий можно найти в презентации «Concurrency Is Not Parallelism» Роба Пайка, автора языка программирования Go. Видео можно найти по адресу https://www.youtube.com/watch?v=cN_DpYBzKso.

В Java 8 попробовать параллелизм очень просто. Есть классическая презентация Рича Хикки (создателя языка программирования Clojure) под названием «Simple Made Easy»¹. Один из основных посылов его лекции заключается в том, что слова «простой» и «легкий» обозначают разные вещи. В двух словах: *простое* – это то, что концептуально ясно, а *легкое*, хоть и делается элементарно, может скрывать за собой значительные сложности. Например, есть простые алгоритмы сортировки, есть не очень простые, но вызвать метод потока `sorted` всегда легко².

Параллельная и конкурентная обработка – сложная тема, и сделать это правильно нелегко. С самого начала в Java были включены низкоуровневые механизмы для поддержки многопоточного доступа в виде методов `wait`, `notify` и `notifyAll` класса `Object` и ключевого слова `synchronized`. Правильно организовать конкурентность с такими примитивами очень трудно, поэтому позже в язык был добавлен пакет `java.util.concurrent`, позволявший работать с конкурентностью на более высоком уровне абстракции с помощью таких классов, как `ExecutorService`, `ReentrantLock` и `BlockingQueue`. И все равно управлять конкурентностью сложно, особенно при наличии ужасного монстра, именуемого «разделяемым изменяемым состоянием».

В Java 8 легко запросить параллельный поток, для этого нужен всего один вызов метода. То, что это легко, сомнению не подлежит. Беда, однако, в том, что проблему повышения производительности простой никак не назовешь. Все сложности, с которыми мы сталкивались в прошлом, никуда не делись, просто они прячутся под поверхностью.

Рецепты в этой главе не следует считать исчерпывающим обсуждением конкурентности и параллелизма. Этим темам можно посвятить целую книгу, и такие книги есть³. Наша цель – показать, какие имеются механизмы и как их рекомендуется применять. Вы сможете затем применить изложенные идеи в своих программах, произвести измерения и принять решение.

9.1. ПРЕОБРАЗОВАНИЕ ПОСЛЕДОВАТЕЛЬНОГО ПОТОКА В ПАРАЛЛЕЛЬНЫЙ

Проблема

Требуется сделать поток последовательным или параллельным, не считаясь с режимом по умолчанию.

¹ Видео: <http://www.infoq.com/presentations/Simple-Made-Easy>, текст: <http://bit.ly/hickey-simplemadeeasy>.

² Еще один отличный пример простого и легкого привел Патрик Стюарт, рассказывая о том, как он играл капитана Пикара в фильме «Звездный путь: следующее поколение». Автор сценария пытался подробно описать ему все шаги выхода на орбиту вокруг планеты. «Чепуха все это, – ответил Стюарт. – Нужно просто сказать: стандартная орбита, энсин».

³ Особо отметим книги Brian Goetz «Java Concurrency in Practice» (Addison-Wesley Professional) и Venkat Subramaniam «Programming Concurrency on the JVM» (Pragmatic Bookshelf).

Решение

Использовать методы `stream` или `parallelStream` интерфейса `Collection` либо методы `sequential` или `parallel` интерфейса `Stream`.

Обсуждение

По умолчанию поток, созданный в Java, является последовательным. В интерфейсе `BaseStream` (которому наследует `Stream`) имеется метод `isParallel`, который показывает, в каком режиме работает поток: последовательном или параллельном.

В примере 9.1 показано, что все стандартные механизмы создания потоков по умолчанию последовательные.

Пример 9.1 ❖ Создание последовательных потоков (в составе теста JUnit)

```
@Test
public void sequentialStreamOf() throws Exception {
    assertFalse(Stream.of(3, 1, 4, 1, 5, 9).isParallel());
}

@Test
public void sequentialIterateStream() throws Exception {
    assertFalse(Stream.iterate(1, n -> n + 1).isParallel());
}

@Test
public void sequentialGenerateStream() throws Exception {
    assertFalse(Stream.generate(Math::random).isParallel());
}

@Test
public void sequentialCollectionStream() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertFalse(numbers.stream().isParallel());
}
```

Если бы источником была коллекция, то можно было бы вызвать метод `parallelStream` для получения (возможно) параллельного потока, как в примере 9.2.

Пример 9.2 ❖ Использование метода `parallelStream`

```
@Test
public void parallelStreamMethodOnCollection() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertTrue(numbers.parallelStream().isParallel());
}
```

Причина оговорки «возможно» – в том, что этому методу разрешено возвращать последовательный поток, но по умолчанию поток будет параллельным. В документации говорится, что последовательный поток возможен только в случае, когда вы создаете собственный `splitterator`, что довольно необычно¹.

¹ Тема, безусловно, интересная, но выходит за рамки этой книги.

Другой способ создать параллельный поток – воспользоваться методом `parallel` существующего потока, как в примере 9.3.

Пример 9.3 ❖ Использование метода потока `parallel`

```
@Test
public void parallelMethodOnStream() throws Exception {
    assertTrue(Stream.of(3, 1, 4, 1, 5, 9)
        .parallel()
        .isParallel());
}
```

Отметим, кстати, что существует также метод `sequential`, который возвращает последовательный поток, как в примере 9.4.

Пример 9.4 ❖ Преобразование параллельного потока в последовательный

```
@Test
public void parallelStreamThenSequential() throws Exception {
    List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
    assertFalse(numbers.parallelStream()
        .sequential()
        .isParallel());
}
```

Но будьте осторожны – тут есть подводный камень. Допустим, вы планируете построить конвейер, часть которого вполне может работать параллельно, но другие части должны быть последовательными. Возникает соблазн написать код, показанный в примере 9.5.

Пример 9.5 ❖ Переключение с параллельного потока на последовательный (РАБОТАЕТ НЕ ТАК, КАК МОЖНО БЫЛО ОЖИДАТЬ)

```
List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9);
List<Integer> nums = numbers.parallelStream() ❶
    .map(n -> n * 2)
    .peek(n -> System.out.printf("%s обрабатывает %d\n",
        Thread.currentThread().getName(), n))
    .sequential() ❷
    .sorted()
    .collect(Collectors.toList());
```

- ❶ Запросить параллельный поток
- ❷ Перед сортировкой перейти в последовательный режим

Идея этого примера – в том, чтобы сначала удвоить все числа, а затем отсортировать их. Поскольку функция удвоения не имеет состояния и ассоциативна, ее вполне можно выполнить параллельно. Однако операция сортировки принципиально последовательная¹.

¹ Можно рассуждать следующим образом: сортировка с использованием параллельного потока означала бы, что диапазон делится на равные части, каждая из которых сортируется отдельно, а затем производится попытка объединить отсортированные части. Но результат при этом не будет отсортирован.

Метод `peek` в этом примере нужен для того, чтобы показать имя обрабатывающего потока¹, он вызывается после `parallelStream`, но до `sequential`. В результате печатается:

```
main обрабатывает 6
main обрабатывает 2
main обрабатывает 8
main обрабатывает 2
main обрабатывает 10
main обрабатывает 18
```

Вся обработка производится потоком (`thread`) `main`, т. е. поток (`stream`) получится последовательным, несмотря на вызов `parallelStream`. Почему? Вспомним, что потоковые операции не начинаются, пока не настанет черед терминальной операции, и именно в этот момент вычисляется состояние потока. Поскольку перед обращением к методу `collect` последним вызывался метод `sequential`, то поток считается последовательным, и элементы обрабатываются соответственно.



Во время выполнения поток может быть параллельным или последовательным. Методы `parallel` и `sequential` просто устанавливают или сбрасывают булев флажок, который проверяется в момент достижения терминальной операции.

Если вы твердо решили выполнять часть потоковых операций параллельно, а часть последовательно, заведите два потока. Это корявое решение, но лучшего не существует.

9.2. Когда РАСПАРАЛЛЕЛИВАНИЕ ПОМОГАЕТ

Проблема

Вы хотите получить выгоду от использования параллельных потоков.

Решение

Использовать параллельные потоки при подходящих условиях.

Обсуждение

Потоковый API спроектирован так, чтобы было удобно переключаться между последовательным и параллельным потоком, но сможет ли это повысить производительность – другой вопрос. Помните, что переход на параллельные потоки – это оптимизация. Сначала убедитесь, что код работает правильно, а только потом решайте, стоит ли использовать параллельные потоки. Такие решения лучше принимать на основе анализа работы с реальными данными.

¹ К сожалению, при переводе возникает неоднозначность, поскольку английские термины «`stream`» (поток данных) и «`thread`» (поток выполнения) переводятся одним словом «поток». Хотелось надеяться, что из контекста ясно, какой поток имеется в виду в каждом случае. – *Прим. перев.*

По умолчанию в Java 8 для распараллеливания потоков используется общий пул разветвления-соединения. Размер пула равен количеству процессоров, узнать которое позволяет метод `Runtime.getRuntime().availableProcessors()`¹. Управление пулом сопряжено с накладными расходами – нужно разбивать работу на части и объединять частичные результаты в общий ответ.

Чтобы оправдать эти расходы, должны выполняться следующие условия:

- наличие большого объема данных или
- значительное время обработки одного элемента и
- источник данных должен легко разбиваться на части и
- операции должны не иметь состояния и быть ассоциативными.

Первые два требования часто объединяются. Если N – количество элементов данных, а Q – время обработки одного элемента, то в общем случае необходимо, чтобы величина $N * Q$ превышала некоторый порог². Следующее требование означает, что необходима структура данных, которую легко разбить на части, например массив. И наконец, если при выполнении операций сохраняется некоторое состояние или результат зависит от порядка их выполнения, то это, очевидно, приведет к проблемам при распараллеливании.

Ниже приведен пример простейшего вычисления, где распараллеливание потока дает преимущество. Код в примере 9.6 вычисляет сумму очень небольшого количества целых чисел.

Пример 9.6 ❖ Вычисление суммы целых чисел в последовательном потоке

```
public static int doubleIt(int n) {
    try {
        Thread.sleep(100);           ❶
    } catch (InterruptedException ignore) {
    }
    return n * 2;
}

// в main...
Instant before = Instant.now();     ❷
total = IntStream.of(3, 1, 4, 1, 5, 9)
    .map(ParallelDemo::doubleIt)
    .sum();
Instant after = Instant.now();      ❸
Duration duration = Duration.between(start, end);
System.out.println("Сумма удвоенных чисел = " + total);
System.out.println("время = " + duration.toMillis() + " мс");
```

- ❶ Искусственная задержка
- ❷ Измерить затраченное время

¹ Технически размер пула на 1 меньше количества процессоров, но надо еще учесть главный поток, который тоже используется.

² Часто можно встретить условие $N * Q > 10\ 000$, но обычно при этом не указывают размерность Q , так что интерпретировать это условие трудно.

Поскольку сложение – очень быстрая операция, распараллеливание вряд ли даст какой-то эффект, если не ввести искусственную задержку. Здесь N очень мало, поэтому мы увеличим Q , добавив вызов `sleep` с задержкой 100 мс.

По умолчанию потоки последовательные. Поскольку удвоение элемента задерживается на 100 мс, а всего элементов шесть, весь процесс должен занять чуть больше 0.6 секунды, что мы и наблюдаем:

```
Сумма удвоенных чисел = 46
время = 621 мс
```

Теперь распараллелим поток. Для этой цели в интерфейсе `Stream` имеется метод `parallel`, показанный в примере 9.7.

Пример 9.7 ❖ Использование параллельного потока

```
total = IntStream.of(3, 1, 4, 1, 5, 9)
    .parallel()
    .map(ParallelDemo::doubleIt)
    .sum();
```

❶ Использование параллельного потока

На машине с 8 ядрами создается пул разветвления-соединения размера восемь¹. Это означает, что каждому элементу потока выделяется отдельное ядро (в предположении, что больше ничего не работает – к этому моменту мы еще вернемся), поэтому все операции удвоения производятся практически одновременно.

Получается такой результат:

```
Сумма удвоенных чисел = 46
время = 112 мс
```

Поскольку каждая операция удвоения задержана на 100 миллисекунд и потоков достаточно для того, чтобы каждое число обрабатывалось отдельным потоком, общее время вычисления чуть больше 100 мс.

Применение JMH для хронометража

Известно, что правильно замерить производительность весьма сложно, поскольку она зависит от таких разных факторов, как кэширование, время инициализации JVM и т. д. Приведенная выше демонстрация далеко не точна. Для более строгого тестирования часто применяют библиотеку точного эталонного тестирования JMH (Java Micro-benchmark Harness, <http://openjdk.java.net/projects/code-tools/jmh/>).

JMH позволяет использовать аннотации для задания режима хронометража, области измерений, аргументов JVM и многого другого. В примере 9.8 показан результат рефакторинга предыдущего кода с применением JMH.

¹ На самом деле в пуле будет 7 потоков, но к ним нужно добавить еще главный поток.

Пример 9.8 ❖ Хронометраж операции удвоения с применением JMH

```
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
@Fork(value = 2, jvmArgs = {"-Xms4G", "-Xmx4G"})
public class DoublingDemo {
    public int doubleIt(int n) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException ignored) {}
        return n * 2;
    }

    @Benchmark
    public int doubleAndSumSequential() {
        return IntStream.of(3, 1, 4, 1, 5, 9)
            .map(this::doubleIt)
            .sum();
    }

    @Benchmark
    public int doubleAndSumParallel() {
        return IntStream.of(3, 1, 4, 1, 5, 9)
            .parallel()
            .map(this::doubleIt)
            .sum();
    }
}
```

По умолчанию сначала выполняется несколько итераций для прогрева кэша, а затем 20 итераций в двух разных потоках. Вот результаты типичного прогона:

```
Benchmark Mode Cnt Score Error Units
DoublingDemo.doubleAndSumParallel avgt 40 103.523 ± 0.247 ms/op
DoublingDemo.doubleAndSumSequential avgt 40 620.242 ± 1.656 ms/op
```

Значения практически такие же, как при грубой оценке, т. е. последовательная обработка занимает в среднем 620 мс, а параллельная – около 103 мс. Параллельное выполнение в системе, в которой достаточно процессоров, чтобы назначить отдельный поток каждому из шести чисел, приблизительно в шесть раз быстрее, чем выполнение тех же вычислений последовательно.

Суммирование значений примитивных типов

В предыдущем примере мы искусственно увеличили Q при небольшом N , чтобы продемонстрировать эффективность параллельных потоков. В этом разделе

мы выберем N достаточно большим, чтобы можно было делать выводы, и сравним параллельное и последовательное выполнения для потоков общего вида и потоков значений примитивного типа, а также потоки с непосредственным итерированием:

i Пример в этом разделе очень прост, но основан на аналогичной демонстрации в прекрасной книге «Java 8 and 9 in Action»¹.

Итеративное решение показано в примере 9.9.

Пример 9.9 ❖ Итеративное суммирование чисел в цикле

```
public long iterativeSum() {
    long result = 0;
    for (long i = 1L; i <= N; i++) {
        result += i;
    }
    return result;
}
```

В примере 9.10 показаны последовательный и параллельный подходы к суммированию потока `Stream<Long>`.

Пример 9.10 ❖ Суммирование потока общего вида

```
public long sequentialStreamSum() {
    return Stream.iterate(1L, i -> i + 1)
        .limit(N)
        .reduce(0L, Long::sum);
}

public long parallelStreamSum() {
    return Stream.iterate(1L, i -> i + 1)
        .limit(N)
        .parallel()
        .reduce(0L, Long::sum);
}
```

Метод `parallelStreamSum` работает в наименее выгодных условиях в том смысле, что обрабатываются поток `Stream<Long>` вместо `LongStream` и коллекция данных, порожденная методом `iterate`. Система не знает, как эффективно разбить работу на части.

Напротив, в примере 9.11 используются класс `LongStream` (в котором имеется метод `sum`) и диапазон `rangeClosed`, который Java умеет разбивать на части.

Пример 9.11 ❖ Использование потока `LongStream`

```
public long sequentialLongStreamSum() {
    return LongStream.rangeClosed(1, N)
        .sum();
}
```

¹ Urma, Fusco, Mycroft «Java 8 and 9 in Action» (Manning Publishers, 2017).

```

}
public long parallelLongStreamSum() {
    return LongStream.rangeClosed(1, N)
        .parallel()
        .sum();
}

```

Ниже приведены результаты, полученные с помощью JMH для $N = 10\,000\,000$:

```

Benchmark Mode Cnt Score Error Units
iterativeSum avgt 40 6.441 ± 0.019 ms/op
sequentialStreamSum avgt 40 90.468 ± 0.613 ms/op
parallelStreamSum avgt 40 99.148 ± 3.065 ms/op
sequentialLongStreamSum avgt 40 6.191 ± 0.248 ms/op
parallelLongStreamSum avgt 40 6.571 ± 2.756 ms/op

```

Обратили внимание, во что обходятся упаковка и распаковка? При использовании `Stream<Long>` программа работает гораздо медленнее, чем в случае `LongStream`, особенно в сочетании с тем фактом, что поток, порожденный методом `iterate`, трудно разбить на части для обработки пулом разветвления-соединения. А `LongStream` в сочетании с методом `rangeClosed` работает настолько быстро, что между параллельным и последовательным решениями почти нет разницы в производительности.

9.3. ИЗМЕНЕНИЕ РАЗМЕРА ПУЛА

Проблема

Требуется изменить подразумеваемое по умолчанию количество потоков в общем пуле.

Решение

Изменить системный параметр или передавать задачи собственному экземпляру `ForkJoinPool`.

Обсуждение

В документации по классу `java.util.concurrent.ForkJoinPool` сказано, что при конструировании общего пула мы можем задавать следующие системные свойства:

- `java.util.concurrent.ForkJoinPool.common.parallelism`;
- `java.util.concurrent.ForkJoinPool.common.threadFactory`;
- `java.util.concurrent.ForkJoinPool.common.exceptionHandler`.

По умолчанию размер общего пула потоков равен количеству имеющихся процессоров, возвращаемому методом `Runtime.getRuntime().availableProcessors()`. Чтобы задать степень параллелизма, нужно присвоить параметру `parallelism` неотрицательное целое число.

Этот параметр можно задавать из программы или в командной строке. В примере 9.12 показано, как с помощью метода `System.setProperty` указать требуемую степень параллелизма.

Пример 9.12 ❖ Задание размера общего пула из программы

```
System.setProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism", "20");
long total = LongStream.rangeClosed(1, 3_000_000)
    .parallel()
    .sum();
int poolSize = ForkJoinPool.commonPool().getPoolSize();
System.out.println("Размер пула: " + poolSize);    ❶
```

❶ Печатается Размер пула: 20



Задавать размер пула больше числа имеющихся процессорных ядер вряд ли имеет смысл, т. к. производительность, скорее всего, не повысится.

Флаг `-D` в командной строке позволяет задать любое системное свойство. Отметим, что задание свойства из программы имеет больший приоритет, чем задание в командной строке, как показано в примере 9.13.

Пример 9.13 ❖ Задание размера общего пула с помощью системного параметра

```
$ java -cp build/classes/main concurrency.CommonPoolSize
Размер пула: 20

// ...закомментировать строку System.setProperty("...parallelism,20") ...
$ java -cp build/classes/main concurrency.CommonPoolSize
Размер пула: 7

$ java -cp build/classes/main \
    -Djava.util.concurrent.ForkJoinPool.common.parallelism=10 \
    concurrency.CommonPoolSize
Размер пула: 10
```

Этот пример был запущен на машине с 8 процессорами. Размер пула по умолчанию равен 7, так что с учетом главного потока получается восемь активных потоков.

Использование своего собственного пула `ForkJoinPool`

В классе `ForkJoinPool` имеется конструктор, принимающий целое число – степень параллелизма. Поэтому мы можем создать собственный пул, отдельный от общего, и передавать задачи своему пулу (пример 9.14).

Пример 9.14 ❖ Создание своего собственного пула `ForkJoinPool`

```
ForkJoinPool pool = new ForkJoinPool(15);    ❶
ForkJoinTask<Long> task = pool.submit(      ❷
    () -> LongStream.rangeClosed(1, 3_000_000)
        .parallel()
        .sum());
```

```

try {
    total = task.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    pool.shutdown();
}
poolSize = pool.getPoolSize();
System.out.println("Размер пула: " + poolSize);

```

- ❶ Создать пул ForkJoinPool размера 15
- ❷ Передать задачу типа Callable<Integer>
- ❸ Выполнить задачу и дождаться результата
- ❹ Печатается Размер пула: 15

Общий пул, используемый при вызове метода потока `parallel`, в большинстве случаев дает хорошие результаты. Если вы хотите изменить его размер, пользуйтесь системным свойством. Если это все равно не приносит желаемого эффекта, попробуйте создать свой экземпляр `ForkJoinPool` и передать ему работу.

В любом случае, сначала соберите данные о производительности и только потом принимайте окончательное решение.

См. также

Для параллельных вычислений с собственным пулом предназначен также класс `CompletableFuture`, рассматриваемый в рецепте 9.5.

9.4. ИНТЕРФЕЙС FUTURE

Проблема

Требуется представить результат асинхронного вычисления, проверить, завершено ли оно, при необходимости отменить и извлечь результат.

Решение

Использовать класс, реализующий интерфейс `java.util.concurrent.Future`.

Обсуждение

Эта книга посвящена новым средствам в Java 8 и 9, одно из которых – весьма полезный класс `CompletableFuture`. Среди прочего `CompletableFuture` реализует интерфейс `Future`, поэтому стоит сделать краткий обзор возможностей этого интерфейса.

Пакет `java.util.concurrent` был добавлен в Java 5, чтобы у разработчиков была возможность работать на более высоком уровне абстракции, чем тот, что могут обеспечить примитивы `wait` и `notify`. В частности, в этом пакете имеется интерфейс `ExecutorService`, а в нем метод `submit`, который принимает объект типа `Callable` и возвращает экземпляр `Future`, обертывающий нужный нам объект.

В примере 9.15 приведен код, который передает работу объекту `ExecutorService`, печатает строку, а затем извлекает значение из `Future`.

Пример 9.15 ❖ Передача `Callable` и возврат `Future`

```
ExecutorService service = Executors.newCachedThreadPool();
Future<String> future = service.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        Thread.sleep(100);
        return "Hello, World!";
    }
});

System.out.println("Идет обработка...");
getIfNotCancelled(future);
```

Метод `getIfNotCancelled` показан в примере 9.16.

Пример 9.16 ❖ Извлечение значения из `Future`

```
public void getIfNotCancelled(Future<String> future) {
    try {
        if (!future.isCancelled()) {           ❶
            System.out.println(future.get());  ❷
        } else {
            System.out.println("Отменено");
        }
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

- ❶ Проверка состояния `Future`
- ❷ Блокирующий вызов для извлечения значения

Метод `isCancelled` проверяет, был ли отменен объект `Future`. Чтобы извлечь значение, хранящееся внутри `Future`, вызывается метод `get`. Это блокирующий вызов, который возвращает значение универсального типа. Вызов метода обернут блоком `try/catch`, чтобы перехватить объявленные исключения.

В результате печатается:

```
Идет обработка...
Hello, World!
```

Поскольку переданная операция возвращает объект `Future<String>` немедленно, программа сразу же печатает сообщение «Идет обработка...». После этого вызов `get` блокирует выполнение вплоть до завершения объекта `Future`, а затем печатается результат.

Эта книга посвящена Java 8, поэтому отметим, что интерфейс `Callable` можно реализовать, не прибегая к анонимному внутреннему классу, а с помощью лямбда-выражения, как показано в примере 9.17.

Пример 9.17 ❖ Использование лямбда-выражения и проверка завершенности Future

```
future = service.submit(() -> {    ❶
    Thread.sleep(10);
    return "Hello, World!";
});

System.out.println("Обработка продолжается...");

while (!future.isDone()) {        ❷
    System.out.println("Ожидание...");
}

getIfNotCancelled(future);
```

- ❶ Реализация Callable в виде лямбда-выражения
- ❷ Ждать завершения Future

Здесь не только вызывается лямбда-выражение, но и добавлен вызов метода `isDone` в цикле `while`, чтобы опрашивать Future в ожидании завершения.



Вызов `isDone` в цикле называется *активным ожиданием* и в общем случае не рекомендуется, поскольку потенциально метод может вызываться миллионы раз. Класс `CompletableFuture`, обсуждаемый далее в этой главе, предлагает более правильный способ реакции на завершение Future.

На этот раз вывод выглядит так:

```
Обработка продолжается...
Ожидание...
Ожидание...
Ожидание...
// ... еще много таких же сообщений ...
Ожидание...
Ожидание...
Hello, World!
```

Очевидно, необходим более элегантный способ уведомления о том, что объект Future завершился, особенно если планируется использовать будущий результат в другом вычислении. Эту проблему, в числе прочих, решает класс `CompletableFuture`.

Наконец, в интерфейсе Future имеется метод `cancel` на случай, если мы передумаем выполнять операцию.

Пример 9.18 ❖ Отмена объекта Future

```
future = service.submit(() -> {
    Thread.sleep(10);
    return "Hello, World!";
});

future.cancel(true);

System.out.println("Обработка продолжается дальше...");

getIfNotCancelled(future);
```

В результате печатается:

```
Обработка продолжается дальше...
Отменено
```

Поскольку класс `CompletableFuture` расширяет `Future`, все рассмотренные в этом рецепте методы по-прежнему доступны.

См. также

Завершаемые будущие объекты рассматриваются в рецептах 9.5, 9.6 и 9.7.

9.5. ЗАВЕРШЕНИЕ COMPLETABLEFUTURE

Проблема

Требуется явно завершить объект `CompletableFuture`, записав в него значение или заставив его возбуждать исключение при вызове метода `get`.

Решение

Использовать методы `completedFuture`, `complete` или `completeExceptionally`.

Обсуждение

Класс `CompletableFuture` реализует интерфейс `Future`, а также интерфейс `CompletionStage`, содержащий десятки методов на разные случаи жизни.

Но главная прелесть `CompletableFuture` заключается в том, что он позволяет координировать действия без написания вложенных обратных вызовов. Этой теме посвящены следующие два рецепта. А сейчас обсудим вопрос о том, как завершить объект `CompletableFuture`, если известно значение, которое мы хотим вернуть.

Пусть нашему приложению нужно получить изделие по его идентификатору, причем процесс получения может быть дорогостоящим, т. к. включает удаленный доступ: вызов REST-совместимой веб-службы по сети, обращение к базе данных или еще что-то в этом роде.

Поэтому мы решили создать локальный кэш изделий в виде `Map`. Тогда при запросе изделия система сначала проверит, есть ли оно уже в кэше, и, только если нет, начнет более дорогую операцию. В примере 9.19 показаны локальный и удаленный доступы к изделию.

Пример 9.19 ❖ Получение изделия

```
private Map<Integer, Product> cache = new HashMap<>();
private Logger logger = Logger.getLogger(this.getClass().getName());

private Product getLocal(int id) {
    return cache.get(id);
}
```

❶

```
private Product getRemote(int id) {
    try {
        Thread.sleep(100);
        if (id == 666) {
            throw new RuntimeException("Дьявольский запрос");
        }
    } catch (InterruptedException ignored) {
    }
    return new Product(id, "name");
}
```

- ❶ Возвращает управление сразу, но может вернуть null
- ❷ Имитировать задержку получения
- ❸ Имитировать ошибку сети, базы данных или еще чего-то

Теперь наша задача – написать метод `getProduct`, который принимает идентификатор и возвращает изделие. Однако если возвращать значение типа `CompletableFuture<Product>`, то метод сможет вернуть управление немедленно, и мы получим возможность заняться другими делами, пока не будет получено значение.

Чтобы воплотить эту идею в жизнь, мы должны каким-то образом завершить объект `CompletableFuture`. Сделать это можно тремя способами:

```
boolean complete(T value)
static <U> CompletableFuture<U> completedFuture(U value)
boolean completeExceptionally(Throwable ex)
```

Метод `complete` используется, когда уже имеется экземпляр `CompletableFuture` и мы хотим записать в него конкретное значение. `completedFuture` – фабричный метод, который создает объект `CompletableFuture`, содержащий уже вычисленное значение. Метод `completeExceptionally` завершает объект `Future`, записывая в него указанное исключение.

Объединяя все вместе, получаем код в примере 9.20. Здесь предполагается, что у нас уже есть унаследованный механизм получения изделия из удаленной системы, который желательно использовать для заполнения `Future`.

Пример 9.20 ❖ Завершение CompletableFuture

```
public CompletableFuture<Product> getProduct(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            return CompletableFuture.completedFuture(product);
        } else {
            CompletableFuture<Product> future = new CompletableFuture<>();
            Product p = getRemote(id);
            cache.put(id, p);
            future.complete(p);
            return future;
        }
    } catch (Exception e) {
```



```

    CompletableFuture<Product> future = new CompletableFuture<>();
    future.completeExceptionally(e);
    return future;
}
}

```

- ❶ Завершить, записав изделие из кэша, если оно там есть
- ❷ Унаследованный механизм получения
- ❸ Завершить после получения (пример асинхронного выполнения см. ниже)
- ❹ Завершить с исключением, если что-то пошло не так

Метод сначала пытается найти изделие в кэше. Если возвращается значение, не равное null, то вызывается фабричный метод `CompletableFuture.completeFuture`, чтобы вернуть его в виде будущего объекта.

Если же изделия нет в кэше, то необходим удаленный доступ. В коде моделируется синхронный подход (подробнее об этом ниже), который, скорее всего, был реализован в унаследованном коде. Создается объект `CompletableFuture`, и, чтобы поместить в него полученное значение, вызывается метод `complete`.

Наконец, если происходит что-то ужасное (здесь это моделируется изделием с идентификатором 666), то возбуждается исключение `RuntimeException`. Метод `completeExceptionally` принимает это исключение в качестве аргумента и записывает его в объект `Future`.

В тестах из примера 9.21 иллюстрируется обработка исключения.

Пример 9.21 ❖ Использование метода `completeExceptionally` класса `CompletableFuture`

```

@Test(expected = ExecutionException.class)
public void testException() throws Exception {
    demo.getProduct(666).get();
}

@Test
public void testExceptionWithCause() throws Exception {
    try {
        demo.getProduct(666).get();
        fail("Хьюстон, у нас проблема...");
    } catch (ExecutionException e) {
        assertEquals(ExecutionException.class, e.getClass());
        assertEquals(RuntimeException.class, e.getCause().getClass());
    }
}
}

```

Оба теста проходят. Если для объекта `CompletableFuture` вызывался метод `completeExceptionally`, то метод `get` возбуждает исключение `ExecutionException`, причиной которого является оригинальное исключение, в данном случае `RuntimeException`.

- ❶ В сигнатуре метода `get` объявлено контролируемое исключение `ExecutionException`. Метод `join` отличается от `get` только тем, что при исключительном завершении возбуждает неконтролируемое исключение `CompletionException`, причиной которого также является оригинальное исключение.

В этом примере, скорее всего, нужно будет заменить часть, связанную с синхронным получением изделия. Для этого можно использовать `supplyAsync` – один из статических фабричных методов класса `CompletableFuture`. Ниже приведен полный перечень таких методов:

```
static CompletableFuture<Void> runAsync(Runnable runnable)
static CompletableFuture<Void> runAsync(Runnable runnable,
                                       Executor executor)

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,
                                       Executor executor)
```

Методы `runAsync` полезны, если возвращать ничего не нужно. Методы `supplyAsync` возвращают объект, получая его от заданного поставщика `Supplier`. В методах с одним аргументом используется встроенный общий пул разветвления-соединения, а в методах с двумя аргументами исполнитель передается явно во втором аргументе.

В примере 9.22 показана асинхронная версия этого кода.

Пример 9.22 ❖ Использование `supplyAsync` для получения изделия

```
public CompletableFuture<Product> getProductAsync(int id) {
    try {
        Product product = getLocal(id);
        if (product != null) {
            logger.info("getLocal c id=" + id);
            return CompletableFuture.completedFuture(product);
        } else {
            logger.info("getRemote c id=" + id);
            return CompletableFuture.supplyAsync(() -> { ❶
                Product p = getRemote(id);
                cache.put(id, p);
                return p;
            });
        }
    } catch (Exception e) {
        logger.info("возбуждено исключение");
        CompletableFuture<Product> future = new CompletableFuture<>();
        future.completeExceptionally(e);
        return future;
    }
}
```

❶ Та же операция, что и раньше, но изделие возвращается асинхронно

В этом случае изделие возвращается с помощью лямбда-выражения, реализующего интерфейс `Supplier<Product>`. Мы всегда можем вынести его в отдельный метод и свести код к ссылке на этот метод.

Проблема в том, как вызвать другую операцию после завершения `CompletableFuture`. Координация работы нескольких объектов `CompletableFuture` – тема следующего рецепта.

См. также

Приведенный в этом рецепте пример основан на примере из статьи в блоге Кеннета Йоргенсена по адресу <http://kennethjorgensen.com/blog/2016/introduction-to-completablefutures>.

9.6. КООРДИНАЦИЯ НЕСКОЛЬКИХ COMPLETABLEFUTURE, ЧАСТЬ 1

Проблема

Требуется, чтобы завершение одного Future стало сигналом для запуска следующего действия.

Решение

Воспользоваться различными методами экземпляра из класса `CompletableFuture`, предназначенными для координации действия: `thenApply`, `thenCompose`, `thenRun` и другими.

Обсуждение

Самое замечательное в классе `CompletableFuture` – то, как просто с его помощью объекты `Future` связываются вместе. Мы можем создать несколько будущих объектов, представляющих различные задачи, а затем организовать их совместную работу, так что завершение одного объекта станет сигналом для выполнения следующего.

В качестве тривиального примера рассмотрим следующий процесс:

- запросить у `Supplier` строку, содержащую число;
- разобрать строку и выделить целое число;
- умножить число на 2;
- напечатать результат.

В примере 9.23 показано, как просто это сделать.

Пример 9.23 ❖ Координация задач с помощью класса `CompletableFuture`

```
private String sleepThenReturnString() {
    try {
        Thread.sleep(100);           ❶
    } catch (InterruptedException ignored) {
    }
    return "42";
}

CompletableFuture.supplyAsync(() -> this::sleepThenReturnString)
    .thenApply(Integer::parseInt)    ❷
    .thenApply(x -> 2 * x)           ❸
    .thenAccept(System.out::println) ❹
```

```

.join();
System.out.println("Выполняется...");

```

④

- ❶ Ввести искусственную задержку
- ❷ Вызвать функцию, когда закончится предыдущий этап
- ❸ Вызвать потребителя, когда закончится предыдущий этап
- ❹ Извлечь окончательный результат

Печатается строка «Выполняется...», а вслед за ней число 84. Метод `supplyAsync` принимает поставщика `Supplier` (в данном случае он предоставляет строки). Метод `thenApply` принимает функцию, аргументом которой является результат предыдущей операции `CompletionStage`. Функция, переданная в первом вызове `thenApply`, преобразует строку в целое число, а переданная во втором вызове – умножает это число на 2. Наконец, метод `thenAccept` принимает потребителя `Consumer`, который будет выполнен после того, как закончится предыдущий этап.

В классе `CompletableFuture` есть еще много методов координации. В табл. 9.1 приведен их полный перечень (без учета перегруженных вариантов, которые обсуждаются ниже).

Таблица 9.1. Методы координации объектов `CompletableFuture`

Модификатор(ы)	Тип возвращаемого значения	Имя метода	Аргументы
	<code>CompletableFuture<Void></code>	<code>acceptEither</code>	<code>CompletionStage<? extends T> other, Consumer<? super T> action</code>
<code>static</code>	<code>CompletableFuture<Void></code>	<code>allOf</code>	<code>CompletableFuture<?>... cfs</code>
<code>static</code>	<code>CompletableFuture<Void></code>	<code>anyOf</code>	<code>CompletableFuture<?>... cfs</code>
<code><U></code>	<code>CompletableFuture<U></code>	<code>applyToEither</code>	<code>CompletionStage<? extends T> other, Function<? super T, U> fn</code>
	<code>CompletableFuture<Void></code>	<code>runAfterBoth</code>	<code>CompletionStage<?> other, Runnable action</code>
	<code>CompletableFuture<Void></code>	<code>runAfterEither</code>	<code>CompletionStage<?> other, Runnable action</code>
	<code>CompletableFuture<Void></code>	<code>thenAccept</code>	<code>Consumer<? super T> action</code>
<code><U></code>	<code>CompletableFuture<U></code>	<code>thenApply</code>	<code>Function<? super T> action, ? extends U> fn</code>
<code><U, V></code>	<code>CompletableFuture<V></code>	<code>thenCombine</code>	<code>CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn</code>
<code><U></code>	<code>CompletableFuture<U></code>	<code>thenCompose</code>	<code>Function<? super T, ? extends CompletionStage<U>> fn</code>
	<code>CompletableFuture<Void></code>	<code>thenRun</code>	Выполнимое действие
	<code>CompletableFuture<T></code>	<code>whenComplete</code>	<code>BiConsumer<? super T, ? super Throwable> action</code>

Во всех перечисленных в таблице методах используется общий пул `ForkJoinPool` рабочих потоков, размер которого равен количеству имеющихся процессоров. Мы уже обсуждали методы `runAsync` и `supplyAsync`. Это фабричные методы, которые принимают объект `Runnable` или `Supplier` и возвращают `CompletableFuture`. Как видно из таблицы, мы затем можем сцепить их с такими методами, как `thenApply` или `thenCompose`, и тем самым указать задачи, которые должны начаться, после того как предыдущая завершится.

В таблице отсутствуют дополнительные методы – по два для каждого с именами, заканчивающимися словом `Async`: один принимает `Executor`, другой – нет. Например, для метода `thenAccept` есть такие варианты:

```
CompletableFuture<Void> thenAccept(Consumer<? super T> action)
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)
CompletableFuture<Void> thenAcceptAsync(
    Consumer<? super T> action, Executor executor)
```

Метод `thenAccept` выполняет операцию `Consumer` в том же потоке, что и исходную задачу, тогда как второй вариант снова передает ее пулу. В третьем варианте имеется еще аргумент `Executor`, который исполняет задачу вместо общего пула разветвления-соединения.



Использовать ли асинхронные варианты методов – предмет компромисса. Отдельная задача, будучи запущенной асинхронно, возможно, завершится быстрее, но из-за накладных расходов суммарное время работы может и не уменьшиться.

Если вы решите использовать собственный исполнитель `Executor` вместо общего пула, имейте в виду, что класс `ExecutorService` реализует интерфейс `Executor`. В примере 9.24 показан вариант кода с отдельным пулом.

Пример 9.24 ❖ Выполнение задач `CompletableFuture` в отдельном пуле потоков

```
ExecutorService service = Executors.newFixedThreadPool(4);
CompletableFuture.supplyAsync(() -> this::sleepThenReturnString, service) ❶
    .thenApply(Integer::parseInt)
    .thenApply(x -> 2 * x)
    .thenAccept(System.out::println)
    .join();
System.out.println("Выполняется...");
```

❶ Задать отдельный пул в аргументе

Последующие методы `thenApply` и `thenAccept` работают в том же потоке, что `supplyAsync`. При использовании метода `thenApplyAsync` задача будет передана общему пулу, если только не задан другой пул в дополнительном аргументе.

Ожидание завершения в общем пуле `ForkJoinPool`

По умолчанию класс `CompletableFuture` пользуется так называемым «общим» пулом разветвления-соединения – оптимизированным пулом потоков с применением *перехвата работы*, когда все потоки пула «стремятся найти и выполнить

задачи, переданные пулу или созданные другими активными задачами». Важно отметить, что все рабочие потоки – это потоки-демоны, т. е. если программа завершается до завершения потоков, то все они будут сняты.

Это означает, что если в примере 9.23 опустить вызов `join()`, то мы увидим только сообщение «Выполняется...», а результат Future не появится. Система снимает поток до того, как задача доходит до конца.

Исправить это можно двумя способами. Первый – вызывать метод `get` или `join`, как показано в примере; этот метод блокирует выполнение до получения результата. Второй – сказать программе, чтобы она ждала завершения всех потоков, указав тайм-аут для общего пула:

```
ForkJoinPool.commonPool().awaitQuiescence(long timeout, TimeUnit unit)
```

Если тайм-аут достаточно велик, то все объекты Future завершатся. Метод `awaitQuiescence` говорит системе, что нужно подождать, пока все рабочие потоки не перейдут в состояние бездействия или пока не истечет величина тайм-аута – в зависимости от того, что случится раньше.

Если объект `CompletableFuture` возвращает значение, то получить его можно методом `get` или `join`. Оба блокируют выполнение до тех пор, пока Future завершится или возбудит исключение. Разница между ними в том, что `get` возбуждает контролируемое исключение `ExecutionException`, а `join` – неконтролируемое исключение `CompletionException`. Это значит, что `join` проще использовать в лямбда-выражениях.

Объект `CompletableFuture` можно также отменить методом `cancel`, принимающим параметр типа `boolean`:

```
boolean cancel(boolean mayInterruptIfRunning)
```

Если Future еще не завершен, то этот метод завершит его с исключением `CancellationException`. Все зависимые объекты Future также будут завершены с исключением `CompletionException`, а в качестве причины будет указано `CancellationException`. В текущей версии аргумент просто игнорируется¹.

В примере 9.23 были продемонстрированы методы `thenApply` и `thenAccept`. Метод `thenCompose` – это метод экземпляра, который позволяет сцепить другой объект Future с оригинальным; при этом результат первого объекта будет доступен второму. Код в примере 9.25 – наверное, самый сложный способ сложить два числа.

Пример 9.25 ❖ Композиция двух объектов Future

```
@Test
public void compose() throws Exception {
    int x = 2;
```

¹ Интересно, что согласно документации булев параметр «игнорируется, потому что прерывания не используются для управления обработкой».

```

int y = 3;
CompletableFuture<Integer> completableFuture =
    CompletableFuture.supplyAsync(() -> x)
        .thenCompose(n -> CompletableFuture.supplyAsync(() -> n + y));

assertTrue(5 == completableFuture.get());
}

```

Аргументом `thenCompose` является функция, которая принимает результат первого объекта `Future` и преобразует его в выход второго. Если вы хотите, чтобы оба объекта `Future` были независимы, то пользуйтесь методом `thenCombine`, как показано в примере 9.26¹.

Пример 9.26 ❖ Комбинирование двух объектов `Future`

```

@Test
public void combine() throws Exception {
    int x = 2;
    int y = 3;
    CompletableFuture<Integer> completableFuture =
        CompletableFuture.supplyAsync(() -> x)
            .thenCombine(CompletableFuture.supplyAsync(() -> y),
                (n1, n2) -> n1 + n2);

    assertTrue(5 == completableFuture.get());
}

```

Метод `thenCombine` принимает объекты `Future` и `BiFunction`, причем результаты обоих `Future` доступны функции, вычисляющей результат.

Сделаем одно важное замечание. Сигнатура метода `handle` имеет вид:

```
<U> CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn)
```

Оба аргумента функции `BiFunction` – результаты объектов `Future`, если они завершились нормально, и возбужденные исключения в противном случае. Что возвращать, решает ваш код. Существуют также методы `handleAsync`, принимающие либо `BiFunction`, либо `BiFunction` и `Executor`. См. пример 9.27.

Пример 9.27 ❖ Использование метода `handle`

```

private CompletableFuture<Integer> getIntegerCompletableFuture(String num) {
    return CompletableFuture.supplyAsync(() -> Integer.parseInt(num))
        .handle((val, exc) -> val != null ? val : 0);
}

@Test
public void handleWithException() throws Exception {
    String num = "abc";
    CompletableFuture<Integer> value = getIntegerCompletableFuture(num);
    assertTrue(value.get() == 0);
}

@Test

```

¹ Пожалуй, все-таки вот *это* – самый сложный способ сложения двух чисел.

```
public void handleWithoutException() throws Exception {
    String num = "42";
    CompletableFuture<Integer> value = getIntegerCompletableFuture(num);
    assertTrue(value.get() == 42);
}
```

Здесь мы просто разбираем строку, пытаюсь выделить из нее целое число. Если все хорошо, то это число возвращается. В противном случае возбуждается исключение `ParseException`, и метод `handle` возвращает 0. Тесты показывают, как операция работает в обоих случаях.

Как видим, комбинировать задачи можно разными способами – синхронно или асинхронно, с использованием общего пула или своего исполнителя. В следующем рецепте приведен развернутый пример использования.

См. также

Более полный пример приведен в рецепте 9.7.

9.7. КООРДИНАЦИЯ НЕСКОЛЬКИХ COMPLETABLEFUTURE, ЧАСТЬ 2

Проблема

Вы хотите увидеть развернутый пример координации объектов `CompletableFuture`.

Решение

Мы будем рассматривать доступ к веб-страницам для каждой даты бейсбольного сезона. На странице находятся ссылки на матчи, сыгранные в этот день. Мы загрузим информацию о статистике каждого матча и преобразуем ее в Java-класс. Затем асинхронно сохраним данные, вычислим результаты в каждой игре, найдем игру с максимальным итоговым счетом и напечатаем этот счет и игру, в которой он имел место.

Обсуждение

Здесь демонстрируется более сложный пример, чем обычно в этой книге. Хочется надеяться, что это поможет вам составить впечатление о доступных возможностях и о том, как комбинировать задачи `CompletableFuture` для достижения собственных целей.

В приложении используется тот факт, что главная лига бейсбола публикует набор веб-страниц, содержащих счет каждого матча, сыгранного в указанный день¹. На рис. 9.1 показана страница для всех матчей, сыгранных 14 июня 2017 г.

¹ Чтобы понять этот пример, о бейсболе нужно знать только то, что играют две команды и ведется учет перебежек каждой до тех пор, пока кто-то не выиграет. Собранные данные и называются статистикой игры (box score).

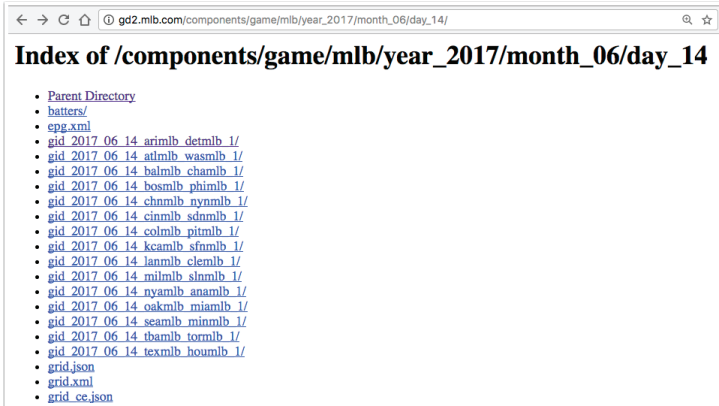


Рис. 9.1 ❖ Матчи, сыгранные 14 июня 2017 г.

Каждая ссылка на этой странице ведет на страницу матча, адрес которой начинается словом «gid», за которой следуют год, месяц, день, код команды гостей и код команды хозяев. Перейдя по этой ссылке, мы попадем на страницу, содержащую список файлов, один из которых называется *boxscore.json*.

Вот что должно делать приложение:

1. Обратиться к сайту, содержащему список матчей в диапазоне дат.
2. Найти на каждой странице ссылки на матчи.
3. Скачать файл *boxscore.json* для каждого матча.
4. Преобразовать данные из формата JSON в Java-объект.
5. Сохранить скачанные данные в локальных файлах.
6. Найти счет каждого матча.
7. Определить игру с максимальным суммарным счетом.
8. Напечатать счет каждой игры, а также матч с максимальным счетом и его результат.

Приложение можно построить так, что многие из перечисленных задач будут выполняться конкурентно, и запустить его в параллельном режиме.

Полный код примера слишком велик для включения в книгу, но он есть на сопроводительном сайте (<https://github.com/kousen/cfboxscores>). В этом рецепте демонстрируется использование параллельных потоков и завершаемых объектов Future.

Наша первая задача – найти ссылки на матчи для каждой даты в указанном диапазоне. Класс `GamePageLinksSupplier` в примере 9.28 реализует интерфейс `Supplier` и порождает список строк, представляющих ссылки на матчи.

Пример 9.28 ❖ Получить ссылки на матчи в диапазоне дат

```
public class GamePageLinksSupplier implements Supplier<List<String>> {
    private static final String BASE =
        "http://gd2.mlb.com/components/game/mlb/";
    private LocalDate startDate;
```

```

private int days;

public GamePageLinksSupplier(LocalDate startDate, int days) {
    this.startDate = startDate;
    this.days = days;
}

public List<String> getGamePageLinks(LocalDate localDate) {
    // Для разбора HTML-страницы и извлечения ссылок, начинающихся
    // словом "gid", используем библиотеку JSoup.
}

@Override
public List<String> get() { ❶
    return Stream.iterate(startDate, d -> d.plusDays(1))
        .limit(days)
        .map(this::getGamePageLinks)
        .flatMap(list -> list.isEmpty() ? Stream.empty() : list.stream())
        .collect(Collectors.toList());
}
}

```

❶ Метод, который должен реализовать поставщик Supplier<List<String>>

Метод `get` обходит диапазон дат, вызывая метод `iterate` интерфейса `Stream`. Он начинает обход с указанной даты и прибавляет к ней по одному дню, пока не дойдет до конечной даты.

❷ Метод `datesUntil`, добавленный в класс `LocalDate` в Java 9, порождает поток `Stream<LocalDate>`. Подробнее см. рецепт 10.7.

Каждая дата `LocalDate` передается методу `getGamePageLinks`, который разбирает HTML-страницу с помощью библиотеки `JSoup` (<https://jsoup.org/>), находит все ссылки, начинающиеся словом «gid», и возвращает их в виде строк.

Следующий шаг – скачать и разобрать файл `boxscore.json` с каждой страницы матча. Это делает класс `BoxscoreRetriever`, который реализует интерфейс `Function<List<String>, List<Result>>`, как показано в примере 9.29.

Пример 9.29 ❖ Получить список результатов матчей, зная список ссылок на матчи

```

public class BoxscoreRetriever implements Function<List<String>, List<Result>> {
    private static final String BASE =
        "http://gd2.mlb.com/components/game/mlb/";

    private OkHttpClient client = new OkHttpClient();
    private Gson gson = new Gson();

    @SuppressWarnings("ConstantConditions")
    public Optional<Result> gamePattern2Result(String pattern) {
        // ... код опущен ...
        String boxscoreUrl = BASE + dateUrl + pattern + "boxscore.json";

        // .. настроить OkHttpClient для сетевого вызова ...
        try {

```

```

// ... получить ответ ...
if (!response.isSuccessful()) {
    System.out.println("Не найдены данные для " + boxscoreUrl);
    return Optional.empty(); ❶
}

return Optional.ofNullable(
    gson.fromJson(response.body().charStream(), Result.class)); ❷
} catch (IOException e) {
    e.printStackTrace();
    return Optional.empty(); ❶
}
}

@Override
public List<Result> apply(List<String> strings) {
    return strings.parallelStream()
        .map(this::gamePattern2Result)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .collect(Collectors.toList());
}
}

```

- ❶ Если данные о матче не найдены (из-за дождя или по какой-то другой причине), вернуть пустой `Optional`
- ❷ С помощью класса `Gson` преобразовать JSON в `Result`

Опираясь на библиотеку `OkHttp` (<http://square.github.io/okhttp/>) и библиотеку `Gson` для разбора JSON, этот класс скачивает статистику матча в формате JSON и преобразует ее в объект типа `Result`. Класс реализует интерфейс `Function`, поэтому предоставляет метод `apply` для преобразования списка строк в список результатов. Данные о матче могут отсутствовать, если игра не состоялась из-за дождя или произошла какая-то сетевая ошибка, поэтому метод `gamePattern2Result` возвращает объект типа `Optional<Result>`, который в таких случаях будет пуст.

Метод `apply` обрабатывает поток ссылок на матчи и преобразует каждую в объект `Optional<Result>`. Затем к потоку применяется фильтр, который пропускает только непустые экземпляры `Optional`, после чего для каждого экземпляра вызывается метод `get`. И наконец все результаты собираются в список.

- i** В Java 9 в класс `Optional` добавлен метод `stream`, который упрощает цепочку `filter(Optional::isPresent)` плюс `map(Optional::get)`. Подробнее см. рецепт 10.6.

Скачав результаты матчей, мы можем сохранить их локально. Это делают методы, показанные в примере 9.30.

Пример 9.30 ❖ Сохранить результаты каждого матча в отдельном файле

```

private void saveResultList(List<Result> results) {
    results.parallelStream().forEach(this::saveResultToFile);
}

```

```

public void saveResultToFile(Result result) {
    // ... построить имя файла на основе даты и названий команд ...
    try {
        File file = new File(dir + "/" + fileName);
        Files.write(file.toPath().toAbsolutePath(),
                    gson.toJson(result).getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

❶ Создать новый или перезаписать существующий файл, затем закрыть его

Метод `Files.write` с аргументами по умолчанию создает файл, если он еще не существует, либо перезаписывает существующий файл, а после записи закрывает его.

Есть еще два метода постобработки. Метод `getMaxScore` находит максимальный счет, а метод `getMaxGame` возвращает матч с максимальным счетом. Оба показаны в примере 9.31.

Пример 9.31 ❖ Получение максимального суммарного счета и соответствующего матча

```

private int getTotalScore(Result result) {
    // ... просуммировать очки обеих команд ...
}

public OptionalInt getMaxScore(List<Result> results) {
    return results.stream()
        .mapToInt(this::getTotalScore)
        .max();
}

public Optional<Result> getMaxGame(List<Result> results) {
    return results.stream()
        .max(Comparator.comparingInt(this::getTotalScore));
}

```

Теперь, наконец, все показанные выше методы и классы можно объединить с завершаемыми объектами `Future`. В примере 9.32 приведен основной код приложения.

Пример 9.32 ❖ Основной код приложения

```

public void printGames(LocalDate startDate, int days) {
    CompletableFuture<List<Result>> future =
        CompletableFuture.supplyAsync(
            new GamePageLinksSupplier(startDate, days))
            .thenApply(new BoxscoreRetriever());
    CompletableFuture<Void> futureWrite =
        future.thenAcceptAsync(this::saveResultList)
            .exceptionally(ex -> {
                System.err.println(ex.getMessage());
                return null;
            });
}

```

```

    });

    CompletableFuture<OptionalInt> futureMaxScore =
        future.thenApplyAsync(this::getMaxScore);
    CompletableFuture<Optional<Result>> futureMaxGame =
        future.thenApplyAsync(this::getMaxGame);
    CompletableFuture<String> futureMax =
        futureMaxScore.thenCombineAsync(futureMaxGame,
            (score, result) ->
                String.format("Максимальный счет: %d, соответствующий матч: %s",
                    score.orElse(0), result.orElse(null)));

    CompletableFuture.allOf(futureWrite, futureMax).join();

    future.join().forEach(System.out::println);
    System.out.println(futureMax.join());
}

```

- ❶ Координированные задачи получения результатов матчей
- ❷ Сохранить в файле, завершить с исключением в случае ошибки
- ❸ Скомбинировать обе задачи нахождения максимума
- ❹ Дождаться завершения всех задач

Создается несколько объектов `CompletableFuture`. В первом класс `GamePageLinksSupplier` используется для получения всех ссылок на страницы матчей в указанном диапазоне дат, а затем объект типа `BoxscoreRetriever` применяется для преобразования их в результаты матчей. Второй занимается записью результатов на диск и завершается с исключением в случае ошибки. Затем выполняются шаги постобработки: нахождение максимального суммарного счета и игры, в которой он имел место¹. Для завершения всех задач вызывается метод `allOf`, после чего результаты выводятся на печать.

Обратите внимание на метод `thenApplyAsync`, который, строго говоря, необязателен, но позволяет выполнить задачи асинхронно.

Запустим эту программу для трех дней, начиная с 5 мая 2017 г.:

```

GamePageParser parser = new GamePageParser();
parser.printGames(LocalDate.of(2017, Month.MAY, 5), 3);

```

Вот что она напечатает:

```

Не найдены данные для Los Angeles at San Diego on May 5, 2017
May 5, 2017: Arizona Diamondbacks 6, Colorado Rockies 3
May 5, 2017: Boston Red Sox 3, Minnesota Twins 4
May 5, 2017: Chicago White Sox 2, Baltimore Orioles 4
// ... счет других матчей ...
May 7, 2017: Toronto Blue Jays 2, Tampa Bay Rays 1
May 7, 2017: Washington Nationals 5, Philadelphia Phillies 6
Максимальный счет: 23, соответствующая игра: May 7, 2017: Boston Red Sox 17, Minnesota Twins 6

```

¹ Очевидно, что это можно было бы сделать в одном шаге, но так мы получаем изящный пример применения `thenCombine`.

Надеюсь, вы теперь представляете, как сочетаются многие возможности, рассмотренные в книге: будущие объекты `CompletableFuture`, функциональные интерфейсы типа `Supplier` и `Function`, классы `Optional`, `Stream` и `LocalDate`, а также методы `map`, `filter` и `flatMap`. Все вместе они позволяют решить интересную задачу.

См. также

Методы координации объектов `CompletableFuture` обсуждаются в рецепте 9.6.

Глава 10

Нововведения в Java 9

На момент написания этой книги считалось, что функциональность Java SDK 9 уже определена и не подлежит изменению, но официальная версия еще не была выпущена. Наибольшее внимание было приковано к проекту Jigsaw, добавляющему в язык новый механизм модульности.

В эту главу включены рецепты, относящиеся к таким нововведениям, как закрытые методы в интерфейсах, фабричные методы для создания неизменяемых коллекций, а также новые методы потоков и классов `Optional` и `Collectors`. Все рецепты тестировались для версии Java SE 9 Early Access build 174.

Для сведения перечислим нововведения в Java 9, не нашедшие отражения в этой главе:

- интерактивная консоль `jshell`;
- модифицированный блок `try` с ресурсами;
- ослабленные ограничения на ромбовидный оператор;
- новые предупреждения о нереконмендованности;
- классы реактивных потоков;
- API обхода стека;
- переработанный класс `Process`.

Некоторые из этих новшеств относительно мелкие (изменения в ромбовидном операторе, требования к блоку `try` с ресурсами, предупреждения о нереконмендованности). Другие слишком специализированы (API обхода стека и изменения в API процессов). А новая оболочка прекрасно документирована, и есть даже отдельное пособие по ней.

Наконец, реактивные потоки завораживают, но сообщество разработчиков открытого исходного кода уже предлагает подобные API, в т. ч. `Reactive Streams` (<http://www.reactive-streams.org/>), `RxJava` и другие, так что надо еще посмотреть, как сообщество отнесется к поддержке нового API в Java 9.

Я надеюсь, что приведенные в этой главе рецепты охватывают большинство типичных ситуаций. Если окажется, что это не так, то в следующее издание книги будут добавлены новые рецепты¹.

¹ Возможно, к тому времени даже наступит определенность с Jigsaw. Надежда умирает последней. :)

Рецепты, собранные в этой главе, стилистически отличаются от всех остальных. Эта книга ориентирована на конкретные ситуации, каждый рецепт предназначен для решения задач определенного вида. Но в некоторых рецептах из этой главы просто обсуждаются новые возможности API.

10.1. Модули в ПРОЕКТЕ JIGSAW

Проблема

Требуется получить доступ к модулям Java из стандартной библиотеки и инкапсулировать свой собственный код в модули.

Решение

Изучить основы модулей (проект Jigsaw) и узнать, как используется разбитый на модули JDK. Затем дождаться выхода окончательной версии Java 9 и принять решение о переходе на нее.

Обсуждение

Спецификация JSR 376 «Java Platform Module System», в которой описывается система модулей на платформе Java, – одно из самых крупных и самых противоречивых изменений, которые ждут нас в Java 9. Попытки ввести модульность в Java предпринимались на протяжении почти десяти лет¹ с разной степенью успеха и признания со стороны пользователей. Эти усилия и подытоживает JPMS.

Конечно, цель – «строгая» инкапсуляция, обеспечиваемая системой модулей, – имеет положительные последствия с точки зрения сопровождения программ, но ничто не дается даром. Внесение столь фундаментального изменения в язык с почти двадцатилетней историей обратной совместимости просто обречено быть трудным.

Так, само понятие *модуля* изменяет интерпретацию ключевых слов `public` и `private`. Если модуль не экспортирует некоторый пакет, то к находящимся в нем классам невозможно получить доступ, даже если они объявлены открытыми (`public`). И воспользоваться рефлексией для доступа к неоткрытым членам класса, не входящего в экспортируемый пакет, тоже не получится. Это совсем безразлично библиотекам и каркасам, опирающимся на рефлексию (в т. ч. таким популярным, как Spring и Hibernate), а также практически всем языкам на платформе JVM, отличным от Java. В качестве уступки разработчики предложили флаг в командной строке `--illegal-access=permit`, который станет режимом по умолчанию в Java 9, а в будущей версии будет запрещен (<http://mail.openjdk.java.net/pipermail/jigsaw-dev/2017-May/012673.html>).

Во время написания этой книги (конец июня 2017 г.) от включения спецификации JPMS в Java 9 один раз уже отказались, и сейчас она пересматривается

¹ Сам проект Jigsaw был начат в 2008 году.

в преддверии нового голосования¹. Кроме того, выпуск Java 9 был перенесен на конец сентября 2017-го.

Так или иначе, весьма вероятно, что в каком-то виде проект Jigsaw будет включен в Java 9, и его основные функциональные возможности уже определены. Цель этого рецепта – дать необходимую информацию по этому предмету, чтобы в тот момент, когда система JPMS будет, наконец, одобрена, вы были готовы воспользоваться теми преимуществами, которые она дает.

Прежде всего нужно понимать, что переводить свой код на модули вы не обязаны. Библиотеки Java приведены к модульному виду, другие библиотеки тоже находятся в процессе перехода, но со своим кодом вы можете не торопиться и подождать, пока система стабилизируется.

Модули

В новой системе определяются модули, которые имеют имя (за исключением так называемых *безымянных*) и описывают свои зависимости и экспортируемые пакеты в файле *module-info.java*. В JAR-файл модуля включен откомпилированный файл *module-info.class*. Файл *module-info.java* называется *дескриптором модуля*.

Файл *module-info.java* начинается ключевым словом `module` и содержит комбинацию ключевых слов `requires` и `exports`, описывающих функциональность модуля. Для демонстрации ниже приведен пример тривиальной программы «Hello, World!», в котором используются два модуля: `com.oreilly.suppliers` и `com.kousenit.clients`.



В настоящее время при именовании модулей рекомендуется придерживаться соглашения «инвертированный URL».

Первый модуль предоставляет поток `Stream` строк, представляющих имена, а второй печатает имя на консоли, сопровождая его приветственным сообщением.

Исходный код класса `NamesSupplier`, входящего в модуль `Supplier`, приведен в примере 10.1.

Пример 10.1 ❖ Поставщик потока имен

```
package com.oreilly.suppliers;

// предложения импорта ...

public class NamesSupplier implements Supplier<Stream<String>> {
    private Path namesPath = Paths.get("server/src/main/resources/names.txt");

    @Override
    public Stream<String> get() {
        try {
```

¹ При втором голосовании, которое закончилось 26 июня, спецификация JPMS была единогласно одобрена (с одним воздержавшимся). Подробности см. по адресу <https://jcp.org/en/jsr/results?id=6016>.

```

        return Files.lines(namesPath);
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}
}

```

Модуль хранится в модуле IntelliJ (к несчастью, в интегрированной среде IntelliJ IDEA тоже используется слово «модуль», но совсем в другом смысле), который называется *server*, потому-то это слово и является частью пути к текстовому файлу.

Ниже приведено содержимое файла *names.txt*¹:

```

Londo
Vir
G'Kar
Na'Toth
DeLenn
Lennier
Kosh

```

В примере 10.2 показан исходный код класса *Main* из клиентского модуля.

Пример 10.2 ❖ Печать имен

```

package com.kousenit.clients;

// предложения импорта ...

public class Main {
    public static void main(String[] args) throws IOException {
        NamesSupplier supplier = new NamesSupplier();

        try (Stream<String> lines = supplier.get()) { ❶
            lines.forEach(line -> System.out.printf("Hello, %s!\n", line));
        }
    }
}

```

❶ Блок `try` с ресурсами автоматически закрывает поток

В примере 10.3 показан файл *module-info.java* для модуля *Supplier*.

Пример 10.3 ❖ Определение модуля *Supplier*

```

module com.oreilly.suppliers { ❶
    exports com.oreilly.suppliers; ❷
}

```

❶ Имя модуля

❷ Сделать пакет доступным другим модулям

¹ Вот и настало время упомянуть «Вавилон 5» в этой книге. Есть подозрение, что космическая станция тоже была построена из модулей.

В примере 10.4 показан файл *module-info.java* для клиентского модуля.

Пример 10.4 ❖ Определение клиентского модуля

```
module com.kousenit.clients {           ❶
    requires com.oreilly.suppliers;     ❷
}
```

- ❶ Имя модуля
- ❷ Требуется модуль Supplier

Эта программа печатает такие строки:

```
Hello, Vir!
Hello, G'Kar!
Hello, Na'Toth!
Hello, De'lenn!
Hello, Lennier!
Hello, Kosh!
```

Предложение `exports` в модуле `Supplier` необходимо для того, чтобы класс `NamesSupplier` был виден клиентам. Предложение `requires` в клиентском модуле сообщает системе о том, что этому модулю необходимы классы из модуля `Supplier`.

Если бы мы захотели протоколировать каждый доступ к серверу из этого модуля, то могли бы воспользоваться классом `Logger` из пакета `java.util.logging`, как в примере 10.5.

Пример 10.5 ❖ Добавление протоколирования в модуль Supplier

```
public class NamesSupplier implements Supplier<Stream<String>> {
    private Path namesPath = Paths.get("server/src/main/resources/names.txt");
    private Logger logger = Logger.getLogger(this.getClass().getName()); ❶

    @Override
    public Stream<String> get() {
        logger.info("Запрос имен в " + Instant.now()); ❷
        try {
            return Files.lines(namesPath);
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

- ❶ Создать регистратор
- ❷ Протоколировать каждый доступ, добавив временную метку

Этот код не компилируется. В версии Java 9 JDK представляет собой собрание модулей, а пакет `java.util.logging` не является частью модуля `java.base`, единственного, который JVM предоставляет по умолчанию. Чтобы воспользоваться классом `Logger`, необходимо изменить файл *module-info.java*, как показано в примере 10.6.

Пример 10.6 ❖ Модифицированный файл `module-info.java`

```
module com.oreilly.suppliers {
    requires java.logging;      ❶
    exports com.oreilly.suppliers;
}
```

❶ Затребовать модуль JDK, отличный от `java.base`

Модули JDK документируются в их собственных файлах `module-info.java`. Так, в примере 10.7 показан файл `module-info.java` из модуля `java.logging`.

Пример 10.7 ❖ Файл `module-info.java` для API протоколирования

```
module java.logging {
    exports java.util.logging;
    provides jdk.internal.logger.DefaultLoggerFinder with
        sun.util.logging.internal.LoggingProviderImpl;
}
```

Этот модуль не только экспортирует пакет, но еще и предоставляет внутреннюю реализацию интерфейса поставщика службы `DefaultLoggerFinder` в виде класса `LoggingProviderImpl`, который используется, когда клиент запрашивает регистратор.

i В проекте Jigsaw также определены механизмы для работы с локаторами и поставщиками служб. Детали см. в документации.

Надеюсь, из этого краткого описания вы смогли составить представление о том, как определяются и работают модули. В ближайшие месяцы я рассчитываю услышать много нового на эту тему.

Относительно модулей существует еще очень много вопросов, которые должны быть разрешены до того, как спецификация будет одобрена. Многие из них касаются переноса унаследованного кода. Такие термины, как *безымянный* и *автоматический* модули, относятся к коду, который не входит ни в один модуль, но находится на «пути к модулям». Подобные модули образуются из существующих унаследованных JAR-файлов. Дебаты вокруг JPMS в немалой степени связаны с тем, как обрабатывать такие случаи.

См. также

Разработка Jigsaw ведется в рамках проекта Open JDK. См. краткое руководство по адресу <http://openjdk.java.net/projects/jigsaw/quick-start>. Текущая документация находится по адресу <http://openjdk.java.net/projects/jigsaw/spec/sotms/> (озаглавлена «State of the Module System»).

10.2. ЗАКРЫТЫЕ МЕТОДЫ В ИНТЕРФЕЙСАХ

Проблема

Требуется добавить закрытые методы в интерфейсы, которые можно было бы вызывать из других методов интерфейса.

Решение

Java SE 9 поддерживает ключевое слово `private` для методов интерфейса.

Обсуждение

В Java SE 8 разработчики впервые получили возможность добавлять реализации методов интерфейса, помечая их ключевым словом `default` или `static`. Следующий логический шаг – разрешить добавление закрытых методов.

Закрытые методы помечаются ключевым словом `private` и обязаны иметь реализацию. Как и закрытые методы классов, их нельзя переопределить. Более того, вызывать их можно только из того же исходного файла, в котором они определены.

Пример 10.8 несколько искусственный, но идею все же иллюстрирует.

Пример 10.8 ❖ Закрытый метод интерфейса

```
import java.util.function.IntPredicate;
import java.util.stream.IntStream;

public interface SumNumbers {
    default int addEvens(int... nums) {
        return add(n -> n % 2 == 0, nums);
    }

    default int addOdds(int... nums) {
        return add(n -> n % 2 != 0, nums);
    }

    private int add(IntPredicate predicate, int... nums) { ❶
        return IntStream.of(nums)
            .filter(predicate)
            .sum();
    }
}
```

❶ Закрытый метод

Оба метода, `addEvens` и `addOdds`, открытые (поскольку ключевое слово `default` подразумевает открытость) и принимают список целых чисел переменной длины. Предоставленная реализация методов по умолчанию делегирует работу методу `add`, который принимает в качестве аргумента предикат `IntPredicate`. Поскольку метод `add` закрытый, клиентам он недоступен, даже из класса, реализующего этот интерфейс.

В примере 10.9 демонстрируется использование этого метода.

Пример 10.9 ❖ Тестирование закрытого метода интерфейса

```
class PrivateDemo implements SumNumbers {} ❶

import org.junit.Test;
import static org.junit.Assert.*;

public class SumNumbersTest {
```

```

private SumNumbers demo = new PrivateDemo();

@Test
public void addEvens() throws Exception {
    assertEquals(2 + 4 + 6, demo.addEvens(1, 2, 3, 4, 5, 6)); ❷
}

@Test
public void addOdds() throws Exception {
    assertEquals(1 + 3 + 5, demo.addOdds(1, 2, 3, 4, 5, 6)); ❷
}
}

```

- ❶ Класс, реализующий интерфейс
- ❷ Вызов открытых методов, делегирующих работу закрытому методу

Чтобы создать экземпляр, нужен класс, поэтому мы написали пустой класс `PrivateDemo`, реализующий интерфейс `SumNumbers`. Далее мы создаем экземпляр этого класса и вызываем его открытые методы, унаследованные от интерфейса.

10.3. СОЗДАНИЕ НЕИЗМЕНЯЕМЫХ КОЛЛЕКЦИЙ

Проблема

Требуется создать неизменяемый список, множество или отображение.

Решение

Воспользоваться статическими фабричными методами `List.of`, `Set.of` и `Map.of`, добавленными в Java 9.

Обсуждение

В документации сказано, что варианты статического фабричного метода `List.of()` дают удобный способ создания неизменяемых списков. Списки, создаваемые этим методом, обладают следующими характеристиками:

- они структурно неизменяемы. Элементы нельзя добавлять, удалять или заменять. Вызов любого изменяющего метода приводит к исключению `UnsupportedOperationException`. Однако если сами содержащиеся в списке элементы изменяемы, то содержимое списка можно изменить;
- недопустимы элементы `null`. Попытка создать список с элементом `null` приводит к исключению `NullPointerException`;
- список является сериализуемым, если все его элементы сериализуемые;
- порядок элементов в списке совпадает с порядком аргументов или элементов переданного методу массива;
- они сериализуются, как описано на странице «Serialized Form».

В примере 10.10 показаны перегруженные варианты метода `of` интерфейса `List`.

Пример 10.10 ❖ Статические фабричные методы для создания неизменяемых списков

```
static <E> List<E> of()
static <E> List<E> of(E e1)
static <E> List<E> of(E e1, E e2)
static <E> List<E> of(E e1, E e2, E e3)
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)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
static <E> List<E> of(E... elements)
```

Получающиеся в результате списки, как написано в документации, являются *структурно* неизменяемыми, т. е. для них нельзя вызывать ни одного из обычных изменяющих методов List: add, addAll, clear, remove, removeAll, replaceAll и set; попытка сделать это заканчивается исключением UnsupportedOperationException. В примере 10.11 приведено несколько тестов¹.

Пример 10.11 ❖ Демонстрация неизменяемости

```
@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityAdd() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.add(99);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityClear() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.clear();
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityRemove() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.remove(0);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilityReplace() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.replaceAll(n -> -n);
}

@Test(expected = UnsupportedOperationException.class)
public void showImmutabilitySet() throws Exception {
    List<Integer> intList = List.of(1, 2, 3);
    intList.set(0, 99);
}
```

¹ Полный комплект тестов имеется в исходном коде, прилагаемом к книге.

Но если сами находящиеся в списке объекты изменяемы, то содержимое списка можно изменить. Пусть имеется простой класс, содержащий изменяемое значение, как в примере 10.12.

Пример 10.12 ❖ Тривиальный класс, содержащий изменяемое значение

```
public class Holder {
    private int x;

    public Holder(int x) { this.x = x; }

    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }
}
```

Если создать неизменяемый список объектов такого класса, то значения, хранящиеся в них, можно будет изменить, так что создается впечатление, будто список изменился.

Пример 10.13 ❖ Изменение целого числа, обернутого классом

```
@Test
public void areWeImmutableOrArentWe() throws Exception {
    List<Holder> holders = List.of(new Holder(1), new Holder(2)); ❶
    assertEquals(1, holders.get(0).getX());

    holders.get(0).setX(4); ❷
    assertEquals(4, holders.get(0).getX());
}
```

- ❶ Неизменяемый список объектов `Holder`
- ❷ Изменяем значение, хранящееся внутри `Holder`

Так можно делать, но это нарушает дух закона, хотя и не нарушает букву. Иными словами, если вы собираетесь создать неизменяемый список, то постарайтесь поместить в него неизменяемые объекты.

Что касается множеств (опять цитируем документацию):

- дубликаты отвергаются на этапе создания. Попытка передать повторяющиеся элементы статическому фабричному методу приводит к исключению `IllegalArgumentException`;
- порядок обхода элементов не определен и в будущих версиях может измениться.

Сигнатуры методов `of` такие же, как у соответствующих методов `List`, только возвращают они `Set<E>`.

Все сказанное относится и к отображениям, только методы `of` принимают в качестве аргумента чередующиеся ключи и значения, как показано в примере 10.14.

Пример 10.14 ❖ Статические фабричные методы для создания неизменяемых отображений

```

static <K,V> Map<K,V> of()
static <K,V> Map<K,V> of(K k1, V v1)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
    K k9, V v9)
static <K,V> Map<K,V> of(K k1, V v1, K k2, V v2, K k3, V v3,
    K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
    K k9, V v9, K k10, V v10)
static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)

```

Для создания отображений, содержащих не более 10 записей, служат методы `of` с чередующимися ключами и значениями. Это может оказаться слишком громоздко, поэтому предоставляются также метод `ofEntries` и статический метод `entry` для создания записи:

```

static <K,V> Map<K,V> ofEntries(Map.Entry<? extends K,? extends V>... entries)
static <K,V> Map.Entry<K,V> entry(K k, V v)

```

В примере 10.15 показано, как с помощью этих методов создать неизменяемое отображение.

Пример 10.15 ❖ Неизменяемое отображение, построенное из записей

```

@Test
public void immutableMapFromEntries() throws Exception {
    Map<String, String> jvmLanguages = Map.ofEntries(
        Map.entry("Java", "http://www.oracle.com/technetwork/java/index.html"),
        Map.entry("Groovy", "http://groovy-lang.org/"),
        Map.entry("Scala", "http://www.scala-lang.org/"),
        Map.entry("Clojure", "https://clojure.org/"),
        Map.entry("Kotlin", "http://kotlinlang.org/"));

    Set<String> names = Set.of("Java", "Scala", "Groovy", "Clojure", "Kotlin");
    List<String> urls = List.of("http://www.oracle.com/technetwork/java/index.html",
        "http://groovy-lang.org/",
        "http://www.scala-lang.org/",
        "https://clojure.org/",
        "http://kotlinlang.org/");

    Set<String> keys = jvmLanguages.keySet();
}

```

```

Collection<String> values = jvmLanguages.values();
names.forEach(name -> assertTrue(keys.contains(name)));
urls.forEach(url -> assertTrue(values.contains(url)));

Map<String, String> javaMap = Map.of("Java",           ❷
    "http://www.oracle.com/technetwork/java/index.html",
    "Groovy",
    "http://groovy-lang.org/",
    "Scala",
    "http://www.scala-lang.org/",
    "Clojure",
    "https://clojure.org/",
    "Kotlin",
    "http://kotlinlang.org/");
javaMap.forEach((name, url) -> assertTrue(
    jvmLanguages.keySet().contains(name) && \
    jvmLanguages.values().contains(url)));
}

```

❶ Использование Map.ofEntries

❷ Использование Map.of

Комбинация методов ofEntries и entry приводит к более простому и изящному коду.

См. также

В рецепте 4.8 обсуждается создание неизменяемых коллекций в Java 8 и более ранних версиях.

10.4. ИНТЕРФЕЙС STREAM: OFNULLABLE, ITERATE, TAKEWHILE И DROPWHILE

Проблема

Использовать новую функциональность потоков, добавленную в Java 9.

Решение

Воспользоваться новыми методами интерфейса Stream: ofNullable, iterate, takeWhile и dropWhile.

Обсуждение

В версии Java 9 в интерфейс Stream добавлено несколько новых методов. В этом рецепте демонстрируются методы ofNullable, iterate, takeWhile и dropWhile.

Метод ofNullable

В Java 8 в интерфейсе Stream был перегруженный статический фабричный метод of, принимающий либо одно значение, либо список переменной длины. В любом случае, передавать аргумент, равный null, запрещено.

В Java 9 метод `ofNullable` позволяет создать поток из одного элемента, который обортывает значение, если оно не равно `null`, и является пустым в противном случае (пример 10.16).

Пример 10.16 ❖ Использование метода `Stream.ofNullable(arg)`

```
@Test
public void ofNullable() throws Exception {
    Stream<String> stream = Stream.ofNullable("abc");    ❶
    assertEquals(1, stream.count());

    stream = Stream.ofNullable(null);                ❷
    assertEquals(0, stream.count());
}
```

- ❶ Поток из одного элемента
- ❷ Возвращается `Stream.empty()`

Метод `count` возвращает количество элементов в потоке. Теперь метод `ofNullable` можно использовать с любым аргументом, не проверяя предварительно, равен ли он `null`.

Использование метода `iterate` с предикатом

Далее рассмотрим новый перегруженный вариант метода `iterate`. В Java 8 этот метод имел такую сигнатуру:

```
static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
```

Таким образом, создание потока начинается с начального элемента (`seed`), а последующие элементы порождаются применением унарного оператора. Результатом является бесконечный поток, поэтому обычно требуется операция `limit` или какая-то укорачивающая функция, например `findFirst` или `findAny`.

Новый перегруженный вариант `iterate` принимает предикат в качестве второго аргумента:

```
static<T> Stream<T> iterate(T seed, Predicate<? super T> hasNext,
    UnaryOperator<T> next)
```

Первым значением в потоке, как и раньше, является `seed`, а последующие получаются применением унарного оператора при условии, что удовлетворяется предикат `hasNext`. См. пример 10.17.

Пример 10.17 ❖ Метод `iterate` с предикатом

```
@Test
public void iterate() throws Exception {
    List<BigDecimal> bigDecimals =
        Stream.iterate(BigDecimal.ZERO, bd -> bd.add(BigDecimal.ONE))
            .limit(10)
            .collect(Collectors.toList());

    assertEquals(10, bigDecimals.size());

    bigDecimals = Stream.iterate(BigDecimal.ZERO,    ❷
```

```

        bd -> bd.longValue() < 10L,
        bd -> bd.add(BigDecimal.ONE))
        .collect(Collectors.toList());
    assertEquals(10, bigDecimals.size());
}

```

- ❶ Создание потока чисел типа `BigDecimal` в Java 8
- ❷ И в Java 9

Первый поток создается, как принято в Java 8, – путем вызова `iterate` с последующим `limit`. Во втором случае используется метод `iterate` с предикатом, и это больше походит на традиционный цикл `for`.

takeWhile и *dropWhile*

Новые методы `takeWhile` и `dropWhile` позволяют выбирать части потока на основе предиката. Для упорядоченного потока `takeWhile` «отбирает самый длинный начальный участок потока, все элементы которого удовлетворяют заданному предикату».

Метод `dropWhile` прямо противоположен – он отбирает элементы, которые остаются в потоке после отбрасывания самого длинного начального участка потока, удовлетворяющего заданному предикату.

В примере 10.18 показано, как эти методы применяются к упорядоченному потоку.

Пример 10.18 ❖ Обработка и отбрасывание начальных элементов потока

```

@Test
public void takeWhile() throws Exception {
    List<String> strings = Stream.of("this is a list of strings".split(" "))
        .takeWhile(s -> !s.equals("of")) ❶
        .collect(Collectors.toList());
    List<String> correct = Arrays.asList("this", "is", "a", "list");
    assertEquals(correct, strings);
}

```

```

@Test
public void dropWhile() throws Exception {
    List<String> strings = Stream.of("this is a list of strings".split(" "))
        .dropWhile(s -> !s.equals("of")) ❷
        .collect(Collectors.toList());
    List<String> correct = Arrays.asList("of", "strings");
    assertEquals(correct, strings);
}

```

- ❶ Вернуть поток до места первого невыполнения предиката
- ❷ Вернуть поток, начиная с места первого невыполнения предиката

Оба метода разделяют поток в одном и том же месте, только `takeWhile` возвращает элементы до точки разделения, а `dropWhile` – после нее.

Истинная ценность `takeWhile` в том, что это укорачивающая операция. При обработке очень большой коллекции отсортированных элементов мы можем остановиться, как только окажется выполнено интересующее нас условие.

Пусть, например, имеется коллекция заказов, отсортированная по стоимости заказа в порядке убывания. С помощью `takeWhile` мы можем ограничиться только заказами на сумму, большую некоторого порога, не применяя фильтра к каждому элементу.

В примере 10.19 смоделирована эта ситуация: генерируются 50 случайных чисел от 0 до 100, они сортируются в порядке убывания, а затем возвращаются только те, что больше 70.

Пример 10.19 ❖ Отбор целых чисел, больших 70

```
Random random = new Random();
List<Integer> nums = random.ints(50, 0, 100) ❶
    .boxed() ❷
    .sorted(Comparator.reverseOrder())
    .takeWhile(n -> n > 70) ❸
    .collect(Collectors.toList());
```

- ❶ Сгенерировать 50 случайных целых чисел от 0 до 100
- ❷ Упаковать их, чтобы можно было отсортировать компаратором и собрать
- ❸ Разбить поток на две части и вернуть только числа, большие 70

Этот код, пожалуй, интуитивно понятнее (хотя и необязательно эффективнее), чем код с использованием `dropWhile` в примере 10.20.

Пример 10.20 ❖ Применение метода `dropWhile` к потоку целых чисел

```
Random random = new Random();
List<Integer> nums = random.ints(50, 0, 100)
    .sorted() ❶
    .dropWhile(n -> n < 90) ❷
    .boxed()
    .collect(Collectors.toList());
```

- ❶ Отсортировать в порядке возрастания
- ❷ Разбить на две части и отбросить числа, меньшие 90

Методы, аналогичные `takeWhile` и `dropWhile`, уже много лет существуют в других языках. Теперь они появились и в Java 9.

10.5. ПОДЧИНЕННЫЕ КОЛЛЕКТОРЫ: FILTERING И FLATMAPPING

Проблема

Требуется отфильтровать элементы в подчиненном коллекторе или разгладить сгенерированную коллекцию коллекций.

Решение

В Java 9 в класс `Collectors` добавлены методы `filtering` и `flatMaping` специально для этих целей.

Обсуждение

В Java 8 в класс `Collectors` была включена операция `groupingBy`, чтобы можно было группировать объекты по некоторому свойству. Операция группировки порождает отображение ключей на списки значений. В Java 8 имеются также *подчиненные* (downstream) коллекторы, так что вместо порождения списков можно выполнить их постобработку для получения размеров, отобразить на что-то другое и т. д.

В Java 9 добавлены два новых подчиненных коллектора: `filtering` и `flatMap`.

Метод `filtering`

Пусть имеется класс `Task`, в котором хранятся бюджет задачи и список занятых в ней разработчиков, представленных экземплярами класса `Developer`. Оба класса показаны в примере 10.21.

Пример 10.21 ❖ Классы `Task` и `Developer`

```
public class Task {
    private String name;
    private long budget;
    private List<Developer> developers = new ArrayList<>();

    // ... конструкторы, методы чтения и установки и т. д. ...
}

public class Developer {
    private String name;

    // ... конструкторы, методы чтения и установки и т. д. ...
}
```

Пусть требуется сгруппировать задачи по бюджету. Простая операция `Collectors.groupingBy` показана в примере 10.22.

Пример 10.22 ❖ Группировка задач по бюджету

```
Developer venkat = new Developer("Venkat");
Developer daniel = new Developer("Daniel");
Developer brian = new Developer("Brian");
Developer matt = new Developer("Matt");
Developer nate = new Developer("Nate");
Developer craig = new Developer("Craig");
Developer ken = new Developer("Ken");

Task java = new Task("Java stuff", 100);
Task altJm = new Task("Groovy/Kotlin/Scala/Clojure", 50);
Task javascript = new Task("JavaScript (sorry)", 100);
Task spring = new Task("Spring", 50);
Task jpa = new Task("JPA/Hibernate", 20);

java.addDevelopers(venkat, daniel, brian, ken);
javascript.addDevelopers(venkat, nate);
```

```
spring.addDevelopers(craig, matt, nate, ken);
altJvm.addDevelopers(venkat, daniel, ken);

List<Task> tasks = Arrays.asList(java, altJvm, javaScript, spring, jpa);
Map<Long, List<Task>> taskMap = tasks.stream()
    .collect(groupingBy(Task::getBudget));
```

В результате получается отображение величины бюджета на список задач с таким бюджетом:

```
50: [Groovy/Kotlin/Scala/Clojure, Spring]
20: [JPA/Hibernate]
100: [Java stuff, JavaScript (sorry)]
```

Теперь, чтобы отобрать только задачи с бюджетом, превышающим некоторый порог, нужно добавить операцию `filter`, как показано в примере 10.23.

Пример 10.23 ❖ Группировка с фильтрацией

```
taskMap = tasks.stream()
    .filter(task -> task.getBudget() >= THRESHOLD)
    .collect(groupingBy(Task::getBudget));
```

Если пороговая величина равна 50, то получаем:

```
50: [Groovy/Kotlin/Scala/Clojure, Spring]
100: [Java stuff, JavaScript (sorry)]
```

Бюджеты ниже порогового вообще отсутствуют в выходном отображении. Если мы все-таки хотим увидеть их, то теперь имеется альтернатива. В Java 9 в класс `Collectors` добавлен статический метод `filtering`, аналогичный `filter`, но применяемый к подчиненному списку задач. В примере 10.24 показано, как это делается.

Пример 10.24 ❖ Группировка с фильтрацией подчиненного списка

```
taskMap = tasks.stream()
    .collect(groupingBy(Task::getBudget,
        filtering(task -> task.getBudget() >= 50, toList())));
```

Теперь в число ключей входят все значения бюджета, но задачи, бюджет которых ниже порога, отсутствуют в списках значений:

```
50: [Groovy/Kotlin/Scala/Clojure, Spring]
20: []
100: [Java stuff, JavaScript (sorry)]
```

Таким образом, операция `filtering` является подчиненным коллектором, который применяется к списку, созданному операцией `grouping`.

Метод `flatMapping`

Теперь предположим, что требуется получить список разработчиков, занятых в каждой задаче. Базовая операция группировки порождает отображение названий задач на списки задач, как в примере 10.25.

Пример 10.25 ❖ Группировка задач по названиям

```
Map<String, List<Task>> tasksByName = tasks.stream()
    .collect(groupingBy(Task::getName));
```

На выходе получается (после форматирования):

```
Java stuff: [Java stuff]
Groovy/Kotlin/Scala/Clojure: [Groovy/Kotlin/Scala/Clojure]
JavaScript (sorry): [JavaScript (sorry)]
Spring: [Spring]
JPA/Hibernate: [JPA/Hibernate]
```

Чтобы получить списки разработчиков, нужен подчиненный коллектор `mappingBy` (см. пример 10.26).

Пример 10.26 ❖ Списки разработчиков для каждой задачи

```
Map<String, Set<List<Developer>>> map = tasks.stream()
    .collect(groupingBy(Task::getName,
        Collectors.mapping(Task::getDevelopers, toSet())));
```

Как показывает тип возвращаемого значения, проблема в том, что коллектор возвращает множество `Set<List<Developer>>`. Нам необходима подчиненная операция `flatMap`, которая разгладит бы коллекцию коллекций. Теперь это возможно благодаря методу `flatMapMapping` класса `Collectors`, как показано в примере 10.27.

Пример 10.27 ❖ Использование flatMapping для получения множества разработчиков

```
Map<String, Set<Developer>> task2setdevs = tasks.stream()
    .collect(groupingBy(Task::getName,
        Collectors.flatMapMapping(task -> task.getDevelopers().stream(),
            toSet())));
```

Вот теперь получается то, что нужно:

```
Java stuff: [Daniel, Brian, Ken, Venkat]
Groovy/Kotlin/Scala/Clojure: [Daniel, Ken, Venkat]
JavaScript (sorry): [Nate, Venkat]
Spring: [Craig, Ken, Matt, Nate]
JPA/Hibernate: []
```

Метод `flatMapMapping` работает в точности как метод `flatMap` интерфейса `Stream`. Отметим, что первый аргумент `flatMapMapping` должен быть потоком, который может быть пустым или непустым в зависимости от источника.

См. также

Подчиненные коллекторы обсуждаются в рецепте 4.6, операция `flatMap` – в рецепте 3.11.

10.6. КЛАСС `OPTIONAL`: МЕТОДЫ `STREAM`, `OR`, `IFPRESENTORELSE`

Проблема

Требуется преобразовать поток `Optional` в поток содержащихся внутри элементов, или выбрать из нескольких возможностей, или сделать что-то, если элемент присутствует, а в противном случае вернуть значение по умолчанию.

Решение

Использовать новые методы `stream`, `or` и `ifPresentOrElse` класса `Optional`.

Обсуждение

Класс `Optional`, введенный в Java 8, дает понять клиенту, что возвращенное значение на законных основаниях может быть равно `null`. Но возвращается не `null`, а пустой объект `Optional`. Поэтому `Optional` служит оберткой для методов, которые иногда возвращают значение, а иногда нет.

Метод `stream`

Рассмотрим метод поиска заказчиков по идентификатору, показанный в примере 10.28.

Пример 10.28 ❖ Поиск заказчика по идентификатору

```
public Optional<Customer> findById(int id) {
    return Optional.ofNullable(map.get(id));
}
```

Здесь предполагается, что заказчики хранятся в памяти в отображении `Map`. Метод `get` интерфейса `Map` возвращает значение, если ключ присутствует, и `null` в противном случае, поэтому использование его в качестве аргумента `Optional.ofNullable` приводит либо к обертыванию значения объектом `Optional`, либо к возврату пустого `Optional`.



Напомним, что метод `Optional.of` возбуждает исключение, если аргумент равен `null`, поэтому `Optional.ofNullable(arg)` – удобное сокращение записи. Реализован он очень просто: `arg != null ? Optional.of(arg) : Optional.empty()`.

Так как `findById` возвращает `Optional<Customer>`, возврат коллекции заказчиков несколько усложняется. В Java 8 для этого можно поступить, как показано в примере 10.29.

Пример 10.29 ❖ Использование методов `filter` и `map` для объектов `Optional`

```
public Collection<Customer> findAllById(Integer... ids) {
    return Arrays.stream(ids)
        .map(this::findById)
        .filter(Optional::isPresent)
```

❶

❷

```

        .map(Optional::get)           ❸
        .collect(Collectors.toList());
    }

```

- ❶ Отображает на Stream<Optional<Customer>>
- ❷ Отфильтровывает пустые Optional
- ❸ Благодаря вызову get отображает на Stream<Customer>

Не то чтобы это было очень сложно, но в Java 9 и этот процесс упростился благодаря добавлению в класс Optional метода stream с сигнатурой

```
Stream<T> stream()
```

Если значение присутствует, то метод возвращает последовательный одноэлементный поток, содержащий только это значение. В противном случае возвращается пустой поток. Таким образом, поток Optional<Customer> можно преобразовать в поток Customer непосредственно, как показано в примере 10.30.

Пример 10.30 ❖ Применение метода flatMap совместно с Optional.stream

```

public Collection<Customer> findAllById(Integer... ids) {
    return Arrays.stream(ids)
        .map(this::findById)           ❶
        .flatMap(Optional::stream)    ❷
        .collect(Collectors.toList());
}

```

- ❶ Отображает на Stream<Optional<Customer>>
- ❷ Преобразует в Stream<Customer>

Метод носит чисто вспомогательный характер, но полезен.

Метод or

Метод orElse служит для извлечения значения, хранящегося внутри Optional. В качестве аргумента ему передается значение по умолчанию:

```
Customer customer = findById(id).orElse(Customer.DEFAULT)
```

Существует также метод orElseGet, в котором для создания значения по умолчанию вызывается поставщик Supplier – на случай, если эта операция обходится дорого:

```
Customer customer = findById(id).orElseGet(() -> createDefaultCustomer())
```

В обоих случаях возвращается объект Customer. Метод or, добавленный в класс Optional в Java 9, позволяет вместо этого вернуть объект Optional<Customer>, получив в качестве аргумента соответствующий поставщик. Таким образом, альтернативные методы поиска заказчиков можно сцеплять.

Метод or имеет такую сигнатуру:

```
Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)
```

Если значение присутствует, то метод возвращает содержащий его объект `Optional`. В противном случае вызывается поставщик `Supplier` и возвращается полученный от него объект `Optional`.

Если есть несколько способов поиска заказчика, то мы теперь можем написать код, показанный в примере 10.31.

Пример 10.31 ❖ Использование метода `or` для перебора альтернативных вариантов

```
public Optional<Customer> findById(int id) {
    return findByIdLocal(id)
        .or(() -> findByIdRemote(id))
        .or(() -> Optional.of(Customer.DEFAULT));
}
```

Этот метод ищет заказчика сначала в локальном кэше, а не найдя, обращается к удаленному серверу. Если заказчик не найден и там, то в последней ветви создается значение по умолчанию, обернувшись объектом `Optional` и возвращается.

Метод `ifPresentOrElse`

Метод `ifPresent` класса `Optional` вызывает потребителя `Consumer`, если объект `Optional` не пуст (пример 10.32).

Пример 10.32 ❖ Использование метода `ifPresent` для печати только непустых заказчиков

```
public void printCustomer(Integer id) {
    findByIdLocal(id).ifPresent(System.out::println); ❶
}

public void printCustomers(Integer... ids) {
    Arrays.asList(ids)
        .forEach(this::printCustomer);
}
```

❶ Печатаются только непустые объекты `Optional`

Это работает, но иногда требуется сделать что-то, если возвращенный объект `Optional` пуст. Новый метод `ifPresentOrElse` принимает второй аргумент типа `Runnable`, который выполняется, если `Optional` пуст. Вот его сигнатура:

```
void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)
```

Чтобы воспользоваться этим методом, просто передайте ему лямбда-выражение, которое не принимает аргументов и возвращает `void`, как в примере 10.33.

Пример 10.33 ❖ Печать заказчика или сообщения по умолчанию

```
public void printCustomer(Integer id) {
    findByIdLocal(id).ifPresentOrElse(System.out::println,
        () -> System.out.println("Заказчик с ид=" + id + " не найден"));
}
```

Этот вариант печатает заказчика, если он найден, и сообщение по умолчанию в противном случае.

Ни одно из описанных добавлений в класс `Optional` принципиально не меняет его поведения, но все они создают дополнительные удобства.

См. также

В рецептах из главы 6 рассматривается класс `Optional` в Java 8.

10.7. ДИАПАЗОНЫ ДАТ

Проблема

Требуется поток дат между двумя конечными точками.

Решение

Воспользоваться новым методом `datesUntil` класса `LocalDate`, добавленным в Java 9.

Обсуждение

Новый Date-Time API, добавленный в Java 8, стал гигантским шагом вперед по сравнению с классами `Date`, `Calendar` и `TimeStamp` из пакета `java.util`, но одно из нововведений в Java 9 закрыло зияющую дыру в этом API: отсутствие простого способа создать поток дат.

В Java 8, чтобы создать поток дат, проще всего было задать начальную дату и прибавить смещение. Например, чтобы получить все дни между двумя датами, можно написать код, показанный в примере 10.34.

Пример 10.34 ❖ Дни между двумя датами (ПРЕДУПРЕЖДЕНИЕ: ОШИБКА!)

```
public List<LocalDate> getDays_java8(LocalDate start, LocalDate end) {
    Period period = start.until(end);
    return IntStream.range(0, period.getDays())
        .mapToObj(start.plusDays)
        .collect(Collectors.toList());
}
```

❶ Подвох! См. следующий пример

Идея в том, чтобы найти промежуток между двумя датами, представленный объектом `Period`, а затем создать поток `IntStream` дней между ними. Если даты отстоят друг от друга на неделю, то получаем:

```
LocalDate start = LocalDate.of(2017, Month.JUNE, 10);
LocalDate end = LocalDate.of(2017, Month.JUNE, 17);
System.out.println(dateRange.getDays_java8(start, end));

// [2017-06-10, 2017-06-11, 2017-06-12, 2017-06-13,
// 2017-06-14, 2017-06-15, 2017-06-16]
```

На первый взгляд, все нормально, но в действительности здесь таится ошибка. Если заменить неделю месяцем, то она станет очевидна:

```

LocalDate start = LocalDate.of(2017, Month.JUNE, 10);
LocalDate end = LocalDate.of(2017, Month.JULY, 17);
System.out.println(dateRange.getDays_java8(start, end));

// []

```

Не возвращено ни одного значения. Проблема в том, что метод `getDays` класса `Period` возвращает значение поля `days` в периоде, а не общее число дней. (То же относится к методам `getMonths`, `getYears` и т. д.) Поэтому если поле `days` одинаково, то пусть даже месяцы различны, все равно получится диапазон нулевого размера.

Чтобы решить задачу правильно, нужно использовать перечисление `ChronoUnit`, реализующее интерфейс `TemporalUnit`, в котором определены константы `DAYS`, `MONTHS` и т. д. Корректная реализация в Java 8 приведена в примере 10.35.

Пример 10.35 ❖ Дни между двумя датами (РАБОТАЕТ)

```

public List<LocalDate> getDays_java8(LocalDate start, LocalDate end) {
    Period period = start.until(end);
    return LongStream.range(0, ChronoUnit.DAYS.between(start, end))
        .mapToObj(start::plusDays)
        .collect(Collectors.toList());
}

```

❶ Работает

Можно также воспользоваться методом `iterate`, но для этого нужно знать число дней (пример 10.36).

Пример 10.36 ❖ Итерирование по диапазону `LocalDate`

```

public List<LocalDate> getDaysByIterate(LocalDate start, int days) {
    return Stream.iterate(start, date -> date.plusDays(1))
        .limit(days)
        .collect(Collectors.toList());
}

```

По счастью, в Java 9 все стало гораздо проще. Теперь в классе `LocalDate` имеется метод `datesUntil`, один из перегруженных вариантов которого принимает `Period`. Ниже показаны сигнатуры обоих вариантов:

```

Stream<LocalDate> datesUntil(LocalDate endExclusive)
Stream<LocalDate> datesUntil(LocalDate endExclusive, Period step)

```

Первый вариант просто вызывает второй с аргументом `step`, равным одному дню.

Решение поставленной задачи в Java 9 показано в примере 10.37.

Пример 10.37 ❖ Диапазоны дат в Java 9

```

public List<LocalDate> getDays_java9(LocalDate start, LocalDate end) {
    return start.datesUntil(end)
        .collect(Collectors.toList());
}

```

```
public List<LocalDate> getMonths_java9(LocalDate start, LocalDate end) {  
    return start.datesUntil(end, Period.ofMonths(1)) ❷  
        .collect(Collectors.toList());  
}
```

- ❶ Подразумевается `Period.ofDays(1)`
- ❷ Период задан в месяцах

Метод `datesUntil` порождает поток, к которому применимы все обычные приемы работы с потоками.

См. также

Вычисление количества дней между двумя датами в Java 8 рассматривается в рецепте 8.8.

Приложение А

Универсальные типы и Java 8

ОБЩИЕ СВЕДЕНИЯ

Механизмы универсальности были добавлены в Java еще в версии J2SE 1.5, но разработчики в большинстве своем освоили лишь тот минимум, который был им необходим для решения конкретных задач. Однако с появлением Java 8 документация внезапно запрестрела такими сигнатурами методов, как следующая, взятая из интерфейса `java.util.Map.Entry`:

```
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>>  
    comparingByKey()
```

или такая (из интерфейса `java.util.Comparator`):

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(  
    Function<? super T,? extends U> keyExtractor)
```

или даже такой монстр из `java.util.stream.Collectors`:

```
static <T,K,D,A,M extends Map<K, D>> Collector<T,?,M> groupingBy(  
    Function<? super T,? extends K> classifier, Supplier<M> mapFactory,  
    Collector<? super T,A,D> downstream)
```

Теперь минимумом уже не ограничишься. Цель этого приложения – помочь вам разобраться с подобными сигнатурами, чтобы продуктивно использовать API.

ЧТО ЗНАЕТ КАЖДЫЙ

Желая получить коллекцию типа `List` или `Set`, мы объявляем тип ее элементов, заключая имя класса в угловые скобки:

```
List<String> strings = new ArrayList<String>();  
Set<Employee> employees = new HashSet<Employee>();
```

i В Java 7 синтаксис был немного упрощен благодаря введению *ромбовидного* оператора, используемого в примерах ниже. Поскольку ссылка в левой части объявляет саму коллекцию и тип ее элементов, например `List<String>` или `List<Integer>`, то повторять объявление при создании экземпляра в той же строке необязательно. Можно написать просто `new ArrayList<>()`, не указывая в угловых скобках типа.

Объявление типа данных коллекции преследует две цели:

- в коллекцию нельзя по ошибке поместить значение не того типа;
- больше не требуется приводить извлеченное из коллекции значение к нужному типу.

Например, если переменная `strings` объявлена, как показано выше, то в коллекцию можно будет добавлять только объекты `String`, и при извлечении элемента мы будем автоматически получать `String`.

Пример А.1 ❖ Простая демонстрация универсальных типов

```
List<String> strings = new ArrayList<>();
strings.add("Hello");
strings.add("World");
// strings.add(new Date());      ❶
// Integer i = strings.get(0);   ❶

for (String s : strings) {      ❷
    System.out.printf("Длина %s равна %d\n", s, s.length());
}
```

- ❶ Не откомпилируется
- ❷ Цикл `for-each` знает, что тип элемента коллекции – `String`

Типобезопасность вставки – вещь удобная, но разработчики редко допускают такие ошибки. А вот получение значения правильного типа при извлечении без необходимости приведения типов действительно уменьшает объем кода¹.

И еще одну вещь знает любой Java-разработчик – в универсальные коллекции нельзя помещать значения примитивных типов. Следовательно, определить типы `List<int>` или `List<double>` невозможно². К счастью, в той же версии Java, где появились универсальные типы, был добавлен еще и механизм автоматической упаковки и распаковки. Поэтому, желая поместить в коллекцию значение примитивного типа, мы указываем в ее объявлении обертывающий класс, как в примере А.2.

Пример А.2 ❖ Использование примитивных типов в универсальных коллекциях

```
List<Integer> ints = new ArrayList<>();
ints.add(3); ints.add(1); ints.add(4);
```

¹ На протяжении всей своей карьеры я ни разу не вставил в список значение не того типа. Но отказ от необходимости приводить типы оправдывает даже такой уродливый синтаксис.

² В версии Java 10, известной как Project Valhalla, предложено разрешить коллекции примитивных типов.


```
ints.add(1); ints.add(9); ints.add(2);
System.out.println(ints);

for (int i : ints) {
    System.out.println(i);
}
```

Java берет на себя обертывание значений типа `int` объектами `Integer` в момент вставки и извлечение их из объектов `Integer` при чтении. Конечно, с такой упаковкой и распаковкой связаны определенные накладные расходы, но писать код легко.

Все Java-разработчики знают и еще об одном аспекте универсальности. В документации по классу, в котором используются универсальные типы, сами эти типы обозначаются заглавными буквами в угловых скобках. Вот, например, выдержка из документации по интерфейсу `List`:

```
public interface List<E> extends Collection<E>
```

Здесь `E` – параметрический тип. В методах интерфейса используется тот же самый параметр. В примере А.3 показаны сигнатуры нескольких таких методов.

Пример А.3 ❖ Методы, объявленные в интерфейсе `List`

```
boolean add(E e)                ❶
boolean addAll(Collection<? extends E> c)  ❷
void clear()                    ❸
boolean contains(Object o)       ❸
boolean containsAll(Collection<?> c)  ❹
E get(int index)                 ❶
```

- ❶ Использование параметрического типа `E` в качестве типа аргумента или возвращаемого значения
- ❷ Ограниченный метатип
- ❸ Методы, в которых тип не упоминается
- ❹ Неизвестный тип

В некоторых из этих методов универсальный тип `E` используется в качестве типа аргумента или возвращаемого значения. В других (`clear` и `contains`) тип не упоминается вовсе. А в третьих допускается некоторое множество типов, обозначаемое вопросительным знаком.

Одно замечание по поводу синтаксиса: объявлять универсальные методы можно даже в классах, не являющихся универсальными. В таком случае один или несколько параметров универсальных типов объявляются в сигнатуре метода. Вот, например, некоторые статические методы из служебного класса `java.util.Collections`:

```
static <T> List<T> emptyList()
static <K,V> Map<K,V> emptyMap()
static <T> boolean addAll(Collection<? super T> c, T... elements)
static <T extends Object & Comparable<? super T>>
    T min(Collection<? extends T> coll)
```

В трех из них объявлен универсальный параметр `T`. В методе `emptyList` он нужен, чтобы задать тип элементов списка. В методе `emptyMap` упоминается универсальное отображение, в котором ключи имеют тип `K`, а значения – тип `V`.

В методе `addAll` объявлен универсальный тип `T`, но в качестве первого аргумента передается значение типа `Collection<? Super T>`, а последующие аргументы имеют тип `T`. Конструкция `? super T`, называемая ограниченным метатипом (`bounded wildcard`), рассматривается в следующем разделе.

На примере метода `min` видно, как универсальные типы обеспечивают безопасность, но читать такую документацию нелегко. Эта конкретная сигнатура будет подробно рассмотрена ниже, а пока скажем лишь, что на тип `T` наложены два ограничения: он должен быть подклассом `Object` и реализовывать интерфейс `Comparable`, параметризованный типом `T` или любым его предком. Аргументом метода должна быть коллекция значений типа `T` или любого его потомка.

Метатипы – это тот рубеж, который отделяет всем известное от не вполне понятного. Будьте готовы к тому, что нечто, выглядящее как наследование, на самом деле к наследованию не имеет никакого отношения.

ЧЕГО НЕКОТОРЫЕ РАЗРАБОТЧИКИ НЕ ОСОЗНАЮТ

Многие разработчики с удивлением узнают, что тип `ArrayList<String>` никаким практически полезным образом не связан с типом `ArrayList<Object>`. В коллекцию элементов типа `Object` можно добавлять значения типа, производного от `Object`, как в примере А.4.

Пример А.4 ❖ Использование `List<Object>`

```
List<Object> objects = new ArrayList<Object>();
objects.add("Hello");
objects.add(LocalDate.now());
objects.add(3);
System.out.println(objects);
```

Пока все хорошо. `String` – подкласс `Object`, поэтому ссылке на `Object` можно присвоить значение типа `String`. Но если вы думаете, что в список строк можно добавить произвольный объект, то сильно ошибаетесь. См. пример А.5.

Пример А.5 ❖ Список строк и объекты

```
List<String> strings = new ArrayList<>();
String s = "abc";
Object o = s; ❶
// strings.add(o); ❷

// List<Object> moreObjects = strings; ❸
// moreObjects.add(new Date());
// String s = moreObjects.get(0); ❹
```

- ❶ Разрешено
- ❷ Не разрешено

- ❸ Также не разрешено, но допустимо
- ❹ Поврежденная коллекция

Поскольку `String` – подкласс `Object`, мы можем присвоить значение типа `String` ссылке на `Object`. Но присвоить ссылке на `List<Object>` значение типа `List<String>` нельзя, как бы странно это ни казалось. Проблема в том, что `List<String>` *не является* подклассом `List<Object>`. Если мы указываем в объявлении списка некоторый тип, то добавить в список можно *только* объекты этого типа. Точка. Объекты подклассов и суперклассов не разрешены. Говорят, что параметрический тип – *инвариант*.

В закомментированной части показано, почему `List<String>` не является подклассом `List<Object>`. Допустим, что можно было бы присвоить `List<String>` ссылке на `List<Object>`. Тогда, используя список ссылок на объекты, мы могли бы добавить в него что-то, не являющееся строкой, а это привело бы к исключению приведения типа при попытке получить этот элемент из исходного списка строк. Компилятор не должен позволять такого.

И тем не менее хотелось бы, чтобы, определив список чисел, мы могли добавлять в него и целые, и числа с плавающей точкой одинарной и двойной точности. Чтобы так и было, требуется ввести ограничения на тип, для чего и служат метатипы.

Метатипы и PECS

Метатипом называется параметрический тип, в котором встречается вопросительный знак `?`, возможно, с ограничением сверху или снизу.

Неограниченные метатипы

Параметрические типы без ограничений полезны, но не во всех случаях. Объявив список `List` неограниченного типа, как в примере А.6, мы сможем читать из него, но не сможем ничего записать.

Пример А.6 ❖ Список с неограниченным метатипом

```
List<?> stuff = new ArrayList<>();
// stuff.add("abc");           ❶
// stuff.add(new Object());
// stuff.add(3);
int numElements = stuff.size(); ❷
```

- ❶ Добавление запрещено
- ❷ `numElements` равно нулю

Выглядит бесполезно, поскольку не видно никакого способа что-то поместить в список. Но одно из возможных применений состоит в том, что любой метод, в объявлении которого указан `List<?>` в качестве аргумента, примет вообще любой список (пример А.7).

Пример А.7 ❖ Неограниченный список передается методу в качестве аргумента

```
private static void printList(List<?> list) {
    System.out.println(list);
}

public static void main(String[] args) {
    // ... создать список ints, strings, stuff ...
    printList(ints);
    printList(strings);
    printList(stuff);
}
```

Вспомните пример выше, где был представлен метод `containsAll` интерфейса `List<E>`:

```
boolean containsAll(Collection<?> c)
```

Этот метод возвращает `true`, только если все элементы переданной коллекции встречаются в списке. Поскольку тип аргумента – неограниченный метатип, в реализации можно использовать только:

- методы самого интерфейса `Collection`, которым не нужно знать типа элемента;
- методы класса `Object`.

В случае `containsAll` это вполне приемлемо. Реализация по умолчанию (в классе `AbstractCollection`) обходит аргумент, используя `iterator`, и для каждого элемента вызывает метод `contains`, чтобы проверить, встречается ли он в списке. Оба метода, `iterator` и `contains`, определены в `Collection`, а `equals` берется из `Object`, и реализация `contains` делегирует работу методам `equals` и `hashCode` класса `Object`, которые можно переопределить в типе элементов. С точки зрения метода `containsAll`, все нужные ему методы доступны. Запреты, действующие в отношении неограниченных метатипов, к этому случаю не относятся.

Вопросительный знак лежит в основе ограничений на типы. Здесь-то и начинается самое интересное.

Метатипы, ограниченные сверху

Чтобы ограничить метатип сверху, используется ключевое слово `extends`, задающее суперкласс. В примере А.8 показано, как определить список чисел, в который можно добавлять числа типа `int`, `long`, `double` и даже `BigDecimal`.

i Ключевое слово `extends` используется даже тогда, когда ограничением сверху является интерфейс, а не класс, например `List<? extends Comparable>`.

Пример А.8 ❖ Список элементов ограниченного сверху метатипа

```
List<? extends Number> numbers = new ArrayList<>();
// numbers.add(3);           ❶
// numbers.add(3.14159);
// numbers.add(new BigDecimal("3"));
```

❶ Добавлять значения по-прежнему нельзя

Ну что ж, идея была неплохой. Но, увы, определить список с ограниченным сверху метатипом можно, но добавлять в него все равно ничего нельзя. Проблема в том, что в момент чтения значения из списка компилятору неизвестен его точный тип, он знает лишь, что тип расширяет `Number`.

Но никто не мешает сказать, что метод принимает аргумент типа `List<? extends Number>`, а затем вызывать метод со списками разных типов. См. пример А.9.

Пример А.9 ❖ Использование метатипа с ограничением сверху

```
private static double sumList(List<? extends Number> list) {
    return list.stream()
        .mapToDouble(Number::doubleValue)
        .sum();
}

public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
    List<Double> doubles = Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0);
    List<BigDecimal> bigDecimals = Arrays.asList(
        new BigDecimal("1.0"),
        new BigDecimal("2.0"),
        new BigDecimal("3.0"),
        new BigDecimal("4.0"),
        new BigDecimal("5.0")
    );

    System.out.printf("сумма ints равна %s\n", sumList(ints));
    System.out.printf("сумма doubles равна %s\n", sumList(doubles));
    System.out.printf("сумма bigDecimals равна %s\n", sumList(bigDecimals));
}
```

Отметим, что суммирование объектов `BigDecimal` с использованием соответствующих им чисел типа `double` подрывает саму идею типа `BigDecimal`, но только в примитивных потоках `IntStream`, `LongStream` и `DoubleStream` есть метод `sum`. Тем не менее этот пример иллюстрирует главное – что метод можно вызвать, передав список элементов любого подтипа `Number`. Поскольку в `Number` определен метод `doubleValue`, этот код компилируется и работает.

Прочитав из списка с ограничением сверху любой элемент, мы сможем присвоить его ссылке на ограничивающий тип, как показано в примере А.10.

Пример А.10 ❖ Чтение значения из списка с ограничением сверху

```
private static double sumList(List<? extends Number> list) {
    Number num = list.get(0);
    // ... то же, что и раньше ...
}
```

При вызове этого метода элементами списка могут быть значения типа `Number` или одного из его подтипов, поэтому ссылка на `Number` всегда будет корректной.

Метатипы, ограниченные снизу

Говоря, что метатип ограничен снизу, мы имеем в виду, что годится любой предок класса. Для задания ограничения снизу служит ключевое слово `super`, сопровождающее вопросительный знак. Следовательно, ссылка типа `List<? super Number>` может представлять `List<Number>` или `List<Object>`.

В случае ограничения сверху мы задаем тип, с которым должны быть согласованы переменные, чтобы реализация метода работала. Для сложения чисел нам был необходим метод `doubleValue`, который определен в классе `Number`. Этот метод есть и во всех подклассах `Number`, унаследованный или переопределенный. Потому-то мы и задавали в качестве входного типа `List<? extends Number>`.

Однако здесь мы берем элементы из списка и добавляем в другую коллекцию. Эта конечная коллекция могла бы иметь как тип `List<Number>`, так и `List<Object>`, поскольку индивидуальным ссылкам на `Object` можно присвоить значение типа `Number`.

Мы имеем классическую демонстрацию концепции, хотя реализацию и нельзя назвать идиоматичной в Java 8 по причинам, которые мы обсудим позже.

Рассмотрим метод `numsUpTo`, принимающий два аргумента: целое число и список, в который требуется поместить целые числа от 1 до значения первого аргумента.

Пример A.11 ❖ Метод, заполняющий переданный список

```
public void numsUpTo(Integer num, List<? super Integer> output) {
    IntStream.rangeClosed(1, num)
        .forEach(output::add);
}
```

Эта реализация не идиоматичная из-за использования списка в качестве выходного аргумента. По сути дела, это побочный эффект, и относиться к нему следует с подозрением. Тем не менее, поскольку второй аргумент имеет тип `List<? super Integer>`, мы можем передать список типа `List<Integer>`, `List<Number>` или даже `List<Object>`, как в примере A.12.

Пример A.12 ❖ Использование метода `numsUpTo`

```
ArrayList<Integer> integerList = new ArrayList<>();
ArrayList<Number> numberList = new ArrayList<>();
ArrayList<Object> objectList = new ArrayList<>();

numsUpTo(5, integerList);
numsUpTo(5, numberList);
numsUpTo(5, objectList);
```

Все возвращенные списки содержат числа от 1 до 5. Поскольку указан метатип, ограниченный снизу, мы знаем, что список будет содержать целые числа, но использовать внутри списка можем ссылки на значение любого супертипа.

В случае списка с ограничением сверху мы читаем и используем значения. В случае же списка с ограничением снизу мы поставляем значения. Эта комбинация традиционно обозначается акронимом PECS.

PECS

Акроним *PECS* означает «Producer Extends, Consumer Super» (поставщик – extends, потребитель – super). Его предложил Джошуа Блок в книге «Effective Java» в качестве мнемоники. Он означает, что если параметрический тип представляет производителя, то следует использовать extends, а если потребителя, то super. Если параметрический тип будет выступать в обеих ролях, то не используйте метатипы вовсе – обоим требованиям удовлетворяет только тип, заданный явно.

Итак, рекомендация сводится к следующим положениям:

- используйте extends, если требуется только *получать* данные из структуры;
- используйте super, если требуется только помещать данные в структуру;
- используйте явный тип, если требуется и то, и другое.

Раз уж зашла речь о терминологии, упомянем два формальных термина, которые часто употребляются в других языках, например Scala.

Ковариантность называется сохранение порядка типов от частного к общему. В Java массивы ковариантны, поскольку `String[]` – подтип `Object[]`. Как мы только что видели, коллекции в Java являются ковариантными, только если присутствует ключевое слово extends с указанием метатипа.

Противоположным понятием является *контравариантность*. В Java для обеспечения контравариантности служит ключевое слово super с метатипом. Под *инвариантностью* понимается, что тип должен быть в точности таким, как указано. Все параметрические типы в Java инвариантны, если только не используется слово extends или super. Это значит, что если метод ожидает получить список `List<Employee>`, то такой и только такой список и следует ему передать. Ни `List<Object>`, ни `List<Salaried>` не подойдут.

Правило PECS – это по-другому сформулированное формальное правило, согласно которому конструктор входного типа является контравариантным, а конструктор выходного типа – ковариантным. Эту мысль иногда формулируют так: «будь снисходителен к тому, что принимаешь, и требователен к тому, что производишь».

Несколько ограничений

И еще одно замечание, перед тем как мы перейдем к примерам из Java 8 API. На параметрический тип можно наложить несколько ограничений. В этом случае они разделяются знаком &:

```
T extends Runnable & AutoCloseable
```

Количество ограничений произвольно, но лишь одно из них может быть классом. Ограничение, в котором участвует класс, должно быть первым в списке.

ПРИМЕРЫ ИЗ JAVA 8 API

Вооружившись этими знаниями, рассмотрим несколько примеров из документации по Java 8.

Stream.max

В интерфейсе `java.util.stream.Stream` имеется метод `max`:

```
Optional<T> max(Comparator<? super T> comparator)
```

Обратите внимание на ограниченный снизу метатип в `Comparator`. Метод `max` возвращает максимальный элемент потока, применяя к элементам указанный компаратор. Тип возвращаемого значения `Optional<T>`, потому что в пустом потоке нет максимального элемента. Таким образом, метод обортывает максимальное значение объектом `Optional`, если оно существует, и возвращает пустой `Optional` в противном случае.

В примере А.13 показан простой класс `Employee`.

Пример А.13 ❖ Тривиальный класс `Employee`

```
public class Employee {
    private int id;
    private String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // ... прочие методы ...
}
```

В примере А.14 мы создаем коллекцию работников, преобразуем ее в поток, а затем применяем метод `max`, чтобы найти работника с максимальным идентификатором `id` и с максимальным именем `name` (в алфавитном порядке¹). В реализации используются анонимные внутренние классы, чтобы подчеркнуть, что параметрическим типом компаратора может быть как `Employee`, так и `Object`.

Пример А.14 ❖ Нахождение максимального объекта `Employee`

```
List<Employee> employees = Arrays.asList(
    new Employee(1, "Seth Curry"),
    new Employee(2, "Kevin Durant"),
    new Employee(3, "Draymond Green"),
    new Employee(4, "Klay Thompson"));

Employee maxId = employees.stream()
    .max(new Comparator<Employee>() {
```

¹ Строго говоря, в лексикографическом порядке, т. е. заглавные буквы предшествуют строчным.


```

@Override
public int compare(Employee e1, Employee e2) {
    return e1.getId() - e2.getId();
}
}).orElse(Employee.DEFAULT_EMPLOYEE);

Employee maxName = employees.stream()
    .max(new Comparator<Object>() {
        @Override
        public int compare(Object o1, Object o2) {
            return o1.toString().compareTo(o2.toString());
        }
    }).orElse(Employee.DEFAULT_EMPLOYEE);

System.out.println(maxId);
System.out.println(maxName);

```

- ❶ Реализация `Comparator<Employee>` с помощью анонимного внутреннего класса
- ❷ Реализация `Comparator<Object>` с помощью анонимного внутреннего класса
- ❸ Klay Thompson (максимальный идентификатор равен 4)
- ❹ Seth Curry (максимальное имя начинается буквой S)

Идея в том, что `Comparator` можно написать, пользуясь методами класса `Employee`, но допустимо также использовать методы `Object`, например `toString`. Поскольку в определении метода присутствует метатип с ключевым словом `super`, `Comparator<? super T> comparator`, разрешено использовать любой компаратор.

Заодно отметим, что так писать этот код никто уже не станет. В примере A.15 показан более идиоматичный подход.

Пример A.15 ❖ Идиоматичный подход к нахождению максимального объекта `Employee`

```

import static java.util.Comparator.comparing;
import static java.util.Comparator.comparingInt;

// ... создать работников ...

Employee maxId = employees.stream()
    .max(comparingInt(Employee::getId))
    .orElse(Employee.DEFAULT_EMPLOYEE);

Employee maxName = employees.stream()
    .max(comparing(Object::toString))
    .orElse(Employee.DEFAULT_EMPLOYEE);

System.out.println(maxId);
System.out.println(maxName);

```

Этот код, безусловно, чище, но не так хорошо подчеркивает ограниченные метатипы, как анонимные внутренние классы.

Stream.map

В качестве еще одного простого примера из того же класса рассмотрим метод `map`. Он принимает функцию с двумя аргументами, и в обоих используются метатипы:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Метод применяет к каждому элементу потока (типа `T`) функцию `mapper`, которая преобразует его в экземпляр типа `R`¹. Поэтому `map` возвращает значение типа `Stream<R>`.

Поскольку `Stream` определен как универсальный интерфейс, параметризованный типом `T`, в сигнатуре метода этот параметр можно не определять. Однако методу нужен дополнительный параметрический тип `R`, который и указывается в сигнатуре перед типом возвращаемого значения. Если бы сам интерфейс не был универсальным, то в сигнатуре метода следовало бы объявить оба параметра.

В сигнатуре интерфейса `java.util.function.Function` присутствуют два параметрических типа: первый (входной аргумент) – тип объекта, *потребляемого* из потока, а второй (выходной аргумент) – тип объекта, *порождаемого* функцией. Метатипы в объявлениях параметров означают, что входной параметр должен иметь такой же тип, что в `Stream`, или более высокий. Выходной тип может быть произвольным потомком параметрического типа возвращаемого потока.

i Пример `Function` несколько сбивает с толку, потому что с точки зрения PECS вариантность метатипов вроде бы должна быть противоположной. Но если вспомнить, что `Function<T,R>` потребляет `T` и порождает `R`, то становится понятнее, почему для `T` стоит `super`, а для `R` – `extends`.

В примере A.16 показано, как этот метод используется.

Пример A.16 ❖ Отображение `List<Employee>` на `List<String>`

```
List<String> names = employees.stream()
    .map(Employee::getName)
    .collect(toList());
```

```
List<String> strings = employees.stream()
    .map(Object::toString)
    .collect(toList());
```

В интерфейсе `Function` объявлены два универсальных параметра: для входа и для выхода. В первом примере метод `Employee::getName` получает на входе объект `Employee` из потока и возвращает значение типа `String`.

Во втором примере показано, что входной аргумент можно было бы рассматривать как `Object`, а не как `Employee`, благодаря ключевому слову `super`. А вернуть, в принципе, можно было бы список `List`, содержащий объекты подкласса `String`, но, поскольку класс `String` финальный, подклассов у него быть не может.

Теперь рассмотрим одну из сигнатур, упомянутых в начале этого приложения.

¹ В Java API принято использовать букву `T` для обозначения типа одной входной переменной, `T` и `U` для двух входных переменных и т. д. Тип возвращаемой переменной обычно обозначается `R`. В случае отображений буквой `K` обозначается тип ключа, а буквой `V` – тип значения.

Comparator.comparing

В примере A.15 используется статическая функция `comparing` интерфейса `Comparator`. Этот интерфейс существует со времен Java 1.0, поэтому появление в нем кучи дополнительных методов может стать для разработчиков сюрпризом. В Java 8 *функциональный интерфейс* определен как интерфейс, содержащий только один абстрактный метод. В случае `Comparator` это метод `compare`, который принимает два аргумента универсального типа `T` и возвращает целое число, которое меньше, равно или больше нуля, если первый аргумент соответственно меньше, равен или больше второго¹.

Метод `comparing` имеет такую сигнатуру:

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(
    Function<? super T,? extends U> keyExtractor)
```

Начнем разбираться в ней с аргумента `keyExtractor` типа `Function`. Как и раньше, в определении `Function` встречаются два универсальных типа: для входного и выходного значений. В данном случае входной тип ограничен снизу типом `T`, а выходной тип ограничен сверху типом `U`. Имя аргумента здесь говорит само за себя: эта функция выделяет свойство, по которому производится сортировка, а метод `compare` возвращает компаратор, который сравнивает значения этого свойства.

Поскольку цель состоит в том, чтобы использовать упорядочение потока по заданному свойству `U`, это свойство должно реализовывать интерфейс `Comparable`. Именно поэтому в объявлении типа `U` говорится, что он должен расширять `Comparable`. Понятное дело, сам `Comparable` – тоже универсальный интерфейс, который обычно параметризован типом `U`, но допускается и любой суперкласс `U`.

В итоге метод возвращает значение типа `Comparator<T>`, которое затем используется другими методами `Stream` для сортировки потока и получения нового потока того же типа.

Как этот метод используется, показано в примере A.15.

Map.Entry.comparingByKey и Map.Entry.comparingByValue

И напоследок рассмотрим добавление объектов `Employee` в отображение `Map`, ключом которого является идентификатор работника, а значением – сам объект `Employee`. Затем программа отсортирует работников по идентификатору или по имени и напечатает результаты.

Первый шаг, добавление записей в `Map`, занимает всего одну строчку – нужно только воспользоваться методом `toMap` класса `Collectors`:

```
// Добавление работников в отображение с идентификатором в качестве ключа
Map<Integer, Employee> employeeMap = employees.stream()
    .collect(Collectors.toMap(Employee::getId, Function.identity()));
```

¹ Компараторы рассматриваются в рецепте 4.1.

Метод `Collectors.toMap` имеет такую сигнатуру:

```
static <T, K, U> Collector<T, ?, Map<K, U>> toMap(
    Function<? super T,? extends K> keyMapper,
    Function<? super T,? extends U> valueMapper)
```

`Collectors` – служебный класс (т. е. содержит только статические методы), который служит для порождения реализаций интерфейса `Collector`.

В нашем примере метод `toMap` принимает два аргумента: функции, порождающие ключи, и значения выходного отображения. А возвращает он объект интерфейса `Collector`, параметризованного тремя типами.

Сигнатура интерфейса `Collector` имеет вид:

```
public interface Collector<T,A,R>
```

Здесь параметрические типы определены следующим образом:

- `T` – тип элементов, подаваемых на вход операции редукции;
- `A` – изменяемый тип аккумулятора для операции редукции (часто остается скрытой деталью реализации);
- `R` – результат операции редукции.

В данном случае в качестве функции `keyMapper` будет использован метод `getId` класса `Employee`. Это означает, что `T` – это тип `Integer`. Тип результата `R` – реализация интерфейса `Map`, в которой `K` – это `Integer`, а `U` – `Employee`.

Дальше начинается самое интересное. Тип `A` в определении `Collector` – это фактическая реализация интерфейса `Map`. Скорее всего, это `HashMap`¹, но точно мы не знаем, поскольку результат используется как аргумент метода `toMap`, и мы его не видим. Но в объявлении `Collector` указан неограниченный метатип (`?`), а это значит, что под капотом используются только методы `Object` или методы `Map`, безразличные к типу. На самом деле применяется лишь новый метод по умолчанию `merge` интерфейса `Map`, после вызова функций `keyMapper` и `valueMapper`.

Для выполнения сортировки в Java 8 в интерфейс `Map.Entry` добавлены статические методы `comparingByKey` и `comparingByValue`. В примере А.17 показана печать элементов, отсортированных по ключу.

Пример А.17 ❖ Сортировка элементов `Map` по ключу и печать

```
Map<Integer, Employee> employeeMap = employees.stream()
    .collect(Collectors.toMap(Employee::getId, Function.identity())); ❶

System.out.println("Сортировка по ключу:");
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByKey())
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue()); ❷
    });
```

❶ Добавить работников в `Map`, используя идентификатор в качестве ключа

❷ Отсортировать работников по ключу и напечатать

¹ В эталонной реализации это действительно `HashMap`.

Сигнатура метода `comparingByKey` имеет вид:

```
static <K extends Comparable<? super K>,V>
    Comparator<Map.Entry<K,V>> comparingByKey()
```

Метод `comparingByKey` не принимает аргументов и возвращает компаратор, который сравнивает объекты типа `Map.Entry`. Поскольку мы сравниваем ключи, объявленный параметрический тип ключа `K` должен быть подтипом `Comparable`. Конечно, сам `Comparable` ограничен снизу типом `K` или одним из его предков, т. е. метод `compareTo` может пользоваться свойствами класса `K` или его суперкласса.

Результат сортировки выглядит так:

Сортировка по ключу:

```
1: Seth Curry
2: Kevin Durant
3: Draymond Green
4: Klay Thompson
```

Если сортировать по значению, то возникнет интересное осложнение, причем понять, в чем ошибка, трудно, не зная, какие параметрические типы участвуют. Прежде всего приведем сигнатуру метода `comparingByValue`:

```
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>>
    comparingByValue()
```

На этот раз `V` должен быть подтипом `Comparable`.

Наивная реализация сортировки могла бы выглядеть так:

```
// Отсортировать работников по имени и напечатать (НЕ КОМПИЛИРУЕТСЯ)
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue());
    });
```

Этот код не компилируется, и выдается такая ошибка:

```
Java: incompatible types: inference variable V has incompatible bounds
equality constraints: generics.Employee
upper bounds: java.lang.Comparable<? super V>
```

Проблема в том, что значениями отображения являются экземпляры класса `Employee`, а этот класс не реализует интерфейс `Comparable`. По счастью, в API определен перегруженный вариант метода `comparingByValue`:

```
static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(
    Comparator<? super V> cmp)
```

Этот метод принимает на входе `Comparator` и возвращает новый `Comparator`, который сравнивает элементы `Map.Entry` по свойству, указанному в аргументе. Правильный способ отсортировать отображение по значению показан в примере А.18.

Пример А.18 ❖ Сортировка элементов отображения по значению и печать

```
// Сортировка работников по значению и печать
System.out.println("Сортировка по имени:");
employeeMap.entrySet().stream()
    .sorted(Map.Entry.comparingByValue(Comparator.comparing(Employee::getName)))
    .forEach(entry -> {
        System.out.println(entry.getKey() + ": " + entry.getValue());
    });
```

Передав методу `comparing` ссылку на метод `Employee::getName`, мы обеспечили сортировку работников по имени в естественном порядке:

Сортировка по имени:

```
3: Draymond Green
2: Kevin Durant
4: Klay Thompson
1: Seth Curry
```

Надеюсь, вы теперь понимаете, как читать и использовать API, и не будете путаться в хитросплетениях универсальных типов.

Замечание о стирании типа

Одна из трудностей при работе с таким языком, как Java, – необходимость поддерживать обратную совместимость с кодом, создававшимся в течение многих лет. Когда в язык добавили универсальные типы, было принято решение удалять информацию о них в процессе компиляции. В результате для параметризованных типов не создается никаких новых классов, а значит, не приходится платить за их использование на этапе выполнения.

Но все это делается глубоко внутри, а нам нужно лишь знать, что на этапе компиляции выполняются следующие действия:

- ограниченные параметрические типы заменяются границами;
- неограниченные параметрические типы заменяются типом `Object`;
- там, где необходимо, вставляются операции приведения типов;
- для сохранения полиморфизма генерируются методы-мостики.

Для типов все оказывается довольно просто. В определении универсального интерфейса `Map` участвуют два параметрических типа: `K` для ключей и `V` для значений. При создании объекта типа `Map<Integer, Employee>` компилятор заменяет `K` на `Integer`, в `V` – на `Employee`.

В примере метода `Map.Entry.comparingByKey` ключи объявлены так, что `K` extends `Comparable`. Следовательно, во всех местах класса, где встречается `K`, вместо него будет подставлено `Comparable`.

В определении интерфейса `Function` участвуют два параметрических типа, `T` и `R`, и имеется единственный абстрактный метод:

```
R apply(T t)
```

В интерфейсе `Stream` метод `map` добавляет ограничения `Function<? super T, ? extends R>`. Поэтому при использовании этого метода

```
List<String> names = employees.stream()
    .map(Employee::getName)
    .collect(Collectors.toList());
```

тип `T` в `Function` был заменен на `Employee` (поскольку это поток объектов `Employee`), а тип `R` – на `String` (поскольку метод `getName` возвращает значение типа `String`).

Вот, собственно, и всё, если не считать некоторых граничных случаев. Интересующиеся могут найти дополнительную информацию в пособии *Java Tutorial*, но вообще-то стирание типа – пожалуй, наименее сложная часть технологии.

РЕЗЮМЕ

Механизмы универсальности, впервые появившиеся в версии J2SE 1.5, по-прежнему остаются с нами, но в Java 8 сигнатуры универсальных методов стали гораздо сложнее. В большинстве функциональных интерфейсов, добавленных в язык, используются как универсальные типы, так и ограниченные метатипы – во имя типобезопасности. Надеюсь, что это приложение поможет вам разобраться в смысле различных API, а значит, и правильно их использовать.

Предметный указатель

Символы

@FunctionalInterface аннотация, 30

@SafeVarargs аннотация, 52

_ (знак подчеркивания) в числовых литералах, 74

?, параметрические метатипы, 250

<> (ромбовидный оператор), 247

А

andThen метод (Consumer), 134

andThen метод (Function), 133

and метод (Predicate), 134

anyMatch, allMatch и noneMatch методы (Stream), 81

для пустых потоков, 82

проверка числа на простоту, 81

at метод, 168

AutoCloseable интерфейс, и потоки, 158

В

BaseStream интерфейс, метод
unordered, 77

BasicFileAttributes объект, 164

between метод, 191

BiConsumer интерфейс, 41, 70, 112

BiFunction интерфейс, 49, 59, 214

BinaryOperator интерфейс, 59, 112

maxBy и minBy методы, 106

использование в методе Map.
merge, 125

BiPredicate, класс, 163

С

Callable интерфейс, реализация
лямбда-выражением, 204

cancel метод (CompletableFuture), 213

characteristics функция (Collectors), 112

chars и codePoints методы
(CharSequence), 69

ChronoUnit класс, 172, 189, 244

Collections класс

sort метод, 92

статические методы, 248

Collection интерфейс, 34

removeIf метод, с предикатом, 47

методы по умолчанию, 35

Collectors класс, 91

collectingAndThen метод, 109

counting метод, 72, 159

groupingBy метод, 101, 159

maxBy метод, 107

partitioningBy и groupingBy
методы, 102

подчиненные коллекторы, 104

summarizingDouble метод, 76

toMap метод, 258

подчиненные коллекторы filtering
и flatMapping, 236

преобразование потока

в коллекцию, 55

Collector интерфейс, 259

реализация, 111

collect метод (Stream)

перегруженные варианты, 95

преобразование потока символов
в строку, 69

combiner функция (Collectors), 112

Comparable интерфейс, 91, 258

Comparator интерфейс, 32, 159

comparing метод, 258

reverseOrder метод, 159

дополнения в Java 8, 91

сортировка потока, 91

статические методы, 37

CompletableFuture класс, 203, 206

completeExceptionally метод, 208

runAsync и supplyAsync методы, 209

выполнение в отдельном пуле
потоков, 212

комбинирование двух объектов
 Future, 214
 композиция двух объектов
 Future, 213
 координация нескольких, 210
 методы, 211
 пример завершения, 207
 compose метод (Function), 133
 computeIfPresent метод, 123
 ConcurrentMap создание методом
 Collectors.toConcurrentMap, 97
 Consumer интерфейс, 128
 accept метод, 129
 counting метод (Collectors), 72, 105
 count метод (Stream), 72

D

datesUntil метод (LocalDate), 244
 DateTimeFormatter класс, 182
 ofPattern метод, 184
 DayOfWeek перечисление, 168
 DirectoryStream интерфейс, 161, 162, 164
 DoubleConsumer интерфейс, 41
 DoubleStream интерфейс, 53
 метод max, 108
 метод summaryStatistics, 73
 DoubleSummaryStatistics класс, 74
 doubles метод (Random), 120
 dropWhile метод (Stream), 235
 Duration класс, 171, 191

E

empty метод (Optional), 145
 ExecutionException, 208
 ExecutorService интерфейс, 203, 212
 Executor интерфейс, 212
 метод handleAsync, 214
 extends ключевое слово, 251
 в PECS, 254

F

FilenameFilter интерфейс, 21
 Files класс
 find метод, 163
 lines метод, 158
 list метод, 160

walk метод, 161
 write метод, 219
 методы, возвращающие поток, 157
 FileVisitOption перечисление, 162
 filtering метод (Collectors), 236
 findAny метод (Stream), 76
 использование для параллельного
 потока после случайной задержки, 78
 применение к последовательному
 и к параллельному потокам, 79
 findFirst метод (Stream), 76
 flatMap метод (Optional), 150
 flatMap метод (Stream), 83
 применение для конкатенации
 потоков, 87
 forEach метод (Iterable and Map), 128
 ForkJoinPool класс, 212
 ожидание завершения в общем
 пуле, 212
 системные свойства, управляющие
 размером общего пула, 201
 собственный пул, 202
 format метод (LocalDate), 182
 Future интерфейс, 203
 извлечение значения, 204
 использование лямбда-выражения
 и проверка завершенности Future, 205
 метод cancel, 205
 передача Callable и возврат Future, 204
 сцепление с помощью объекта
 CompletableFuture, 210

G

getAvailableZoneIds метод, 167
 getOrDefault метод, 125
 GregorianCalendar, преобразование
 в ZonedDateTime, 181
 groupingBy метод (Collectors), 103, 159, 237
 Gson библиотека для разбора JSON, 218

H

handle метод (CompletableFuture), 214

I

identity метод (Function), 98
 identity метод (UnaryOperator), 99

ifPresentOrElse метод (Optional), 242
 ifPresent метод (Optional), 148, 242
 Instant класс
 atZone метод, 186
 now метод, 166
 как мост для преобразования
 java.util.Date, 178
 работа совместно с классом
 Duration, 191
 IntBinaryOperator класс, 59
 IntConsumer интерфейс, 41
 IntStream интерфейс, 54
 базовые реализации редукции, 59
 метод max, 108
 метод sum, 119
 метод summaryStatistics, 73
 преобразование в массив int, 56
 ints метод (Random), 120
 isDone метод (Future), 205
 isNull метод (Objects), 116
 ISO 8601 стандарт, 187
 iterate метод (Stream), 52, 244
 в Java 9, 234

J

java.base модуль, 226
 java.lang.CharSequence интерфейс,
 методы chars и codePoints, 69
 java.math.BigInteger класс, 81
 java.nio.file пакет, 157
 java.sql.Date класс, 165
 методы преобразования, 179
 java.sql.Timestamp класс, методы
 преобразования, 179
 java.util.Calendar класс, 165
 преобразование в java.time.
 ZonedDateTime, 181
 java.util.concurrent.ForkJoinPool
 класс, 201
 java.util.concurrent пакет, 192, 203
 java.util.Date класс, 165
 преобразование в java.sql.Date, 180
 преобразование в LocalDate, 182
 преобразование в новые классы
 из пакета java.time, 178
 java.util.function пакет, 39

Consumer интерфейс, реализация, 40
 поставщики, 42
 предикаты, 44
 функции, 48
 java.util.logging.Logger класс, 43
 Jigsaw проект, модули в Java, 223
 JMH (Java Micro-benchmark
 Harness), 198
 Joda-Time библиотека, 165

L

lines метод (BufferedReader), 160
 LocalDateTime класс
 atZone метод, 188
 now метод, 166
 ofInstant метод, 181
 of метод, 166
 plus и minus методы, 171
 with методы, 172
 добавление часового пояса методом
 at, 168
 преобразование java.util.Calendar
 в, 181
 LocalDate класс
 datesUntil метод, 244
 now метод, 166
 ofInstant метод, 182
 of метод, 166
 parse и format методы, 182
 итерирования, 244
 методы сложения и вычитания
 дат, 169
 преобразование java.util.Date в, 178
 LocalTime класс
 now метод, 166
 of метод, 166
 plus метод, 170
 методы для сложения и вычитания
 дат, 171
 Logger класс, перегруженные варианты
 методов, принимающие Supplier, 44
 LongConsumer интерфейс, 41
 LongStream интерфейс, 53, 200
 метод max, 108
 метод summaryStatistics, 73
 longs метод (Random), 120

М

Map.Entry интерфейс, 100, 129
 методы comparingByKey
 и comparingByValue, 159, 258
 создание неизменяемого
 отображения из записей, 232
 mapToObj метод, 55
 map метод (Optional), 152
 map метод (Stream), 256
 сравнение с flatMap, 83
 Math.random метод, использование
 в роли поставщика, 42
 max метод (Stream), 255
 merge метод (Map), 125
 module-info.java файл, 224
 Month перечисление, 168

N

negate метод (Predicate), 134
 new ключевое слово, в ссылках
 на методы, 26
 noneMatch метод (Stream), 81
 nonNull метод (Objects), 116
 now метод, 166
 numsUpTo метод, 253

O

Objects класс, 115
 deepEquals метод, 117
 isNull и notNull методы, 116
 requireNonNull метод, 44
 перегруженные варианты, 116
 ofInstant метод, 181
 ofNullable метод (Optional), 145, 240
 ofPattern метод, 184
 of метод, 95
 в классах List, Set и Map, 229
 в классах даты и времени, 167, 184
 в классе Optional, 145, 240
 перегруженные варианты, 113
 OptionalDouble класс, 145
 OptionalInt класс, 145
 OptionalLong класс, 145
 Optional тип, 143, 218
 filter метод, 47
 orElseGet метод, Supplier в качестве
 аргумента, 43

добавления в Java 9, 240
 метод ifPresentOrElse, 242
 метод or, 241
 метод stream, 240
 извлечение значений, 146
 использование в методах чтения
 и установки, 149
 методы flatMap и map, 150
 создание объектов, 143
 orElseGet метод (Optional), 147
 orElseThrow метод (Optional), 147
 orElse метод (Optional), 147
 or метод (Optional), 241
 or метод (Predicate), 134

P

parallelStream метод, 194
 parallel метод, 195
 parse, метод, 182
 partitioningBy метод (Collectors), 73
 PaydayAdjuster класс (пример), 175
 PECS (Producer Extends, Consumer
 Super), 254
 peek метод (Stream), отладка
 потоков, 68
 Period класс, 171, 190
 вычисление числа дней между двумя
 датами, 243

R

Random класс, 120
 range и rangeClosed методы, 54
 reduce метод
 конкатенация потоков, 87
 проверка правильности
 сортировки, 65
 replace метод (Map), 124
 requireNonNull метод (Objects), 116
 reverseOrder метод (Comparator), 159
 runAsync метод
 (CompletableFuture), 209, 212
 Runnable интерфейс, реализация
 с помощью анонимного внутреннего
 класса, 19
 Runtime.getRuntime().
 availableProcessors(), 197

S

SecureRandom класс, 121
 sequential метод, 195
 sorted метод, 159
 stream метод (Optional), 156, 240
 summarizingDouble метод (Collectors), 76
 sum метод, 252
 super и методы по умолчанию, 127
 super ключевое слово
 в PECS, 254
 и метатипы, ограниченные снизу, 253
 и ссылки на методы, 26
 supplyAsync метод (CompletableFuture), 209, 211

T

takeWhile метод (Stream), 235
 TemporalAccessor интерфейс, 177
 TemporalAdjuster класс, 174
 TemporalAmount интерфейс, 171
 TemporalField класс, 173
 TemporalQueries класс, 177
 TemporalQuery класс, 174, 177
 queryFrom метод, 177
 использование с помощью ссылки на метод, 178
 TemporalUnit интерфейс, 172, 189, 244
 метод between, 190
 thenAccept метод (CompletableFuture), 212
 thenApply метод (CompletableFuture), 211
 thenComparing метод (Comparator), 93
 thenCompose метод (CompletableFuture), 213
 this использование в ссылках на метод, 26
 toArray метод (Collectors), 96

W

walk метод (Files), 161
 with методы
 в классе LocalDate, 172
 и TemporalAdjuster, 174

Z

ZonedDateTime класс, 167, 188
 now метод, 166
 withZoneSameInstant метод, 186
 локализованный форматер даты и времени, 184
 преобразование GregorianCalendar в, 181
 ZoneId класс, 188
 getAvailableZoneIds метод, 167, 186
 now метод, 166
 of метод, преобразование строкового названия региона в идентификатор часового пояса, 186
 получение названий регионов по ZoneId, 188
 ZoneOffset класс, 185, 187
 getTotalSeconds метод, 186
 ofHoursMinutes метод, 188
 ZoneRules класс, 167

A

абстрактные методы
 в функциональных интерфейсах, 30
 активное ожидание, 205
 акцессоры (методы чтения), обертывание результата объекта Optional, 149
 анонимные внутренние классы
 доступ к атрибутам внешнего класса и локальным переменным, 118
 замена лямбда-выражением, 119
 реализация интерфейса Runnable, 19

Д

дата и время, классы в пакете java.time, 166
 время между событиями, 189
 диапазоны дат, 243
 в Java 8, 243
 в Java 9, метод datesUntil, 244
 корректоры и запросы, 173
 модификация существующего экземпляра, 169
 нахождение названий регионов по смещению, 187

нахождение часовых поясов
с необычным смещением, 185
преобразование `java.util.Date`
и `java.util.Calendar` в новые
классы, 178
префиксы имен методов, 168
разбор и форматирование, 182
демоны (потoki), 213
дескриптор модуля, 224

E

единственный абстрактный метод, 19

З

завершители, 110
закрытые методы в интерфейсах, 227
замыкания, 119
 композиция замыканий в
 интерфейсе `Consumer`, 134

И

инвариантность, 254

К

ковариантность, 254
коллекции
 `Collection.stream` метод, 53
 возврат полной коллекции
 и фильтрация `null`, 117
 добавление линейной коллекции
 в отображение, 97
 значений примитивных типов, 121
 использование примитивных
 типов, 247
 неизменяемые, создание средствами
 API потоков, 109
 обход, 128
 преобразование потока в
 коллекцию, 95
 создание из потока значений
 примитивных типов, 55
 создание неизменяемых коллекций
 в Java 9, 229
 сравнение с потоковой
 обработкой, 90
 универсальные, 246
конкатенация потоков, 85

конкурентность, 192
конструктор с переменным числом
аргументов, 28
контравариантность, 254
копирующий конструктор, 27
корректоры и запросы, 173
 `TemporalAdjuster` класс, 174
 `TemporalQuery` класс, 177
создание собственного запроса, 177
создание собственного
корректора, 174

Л

ленивые потоки, 89
локализованный формater даты
и времени, 184
лямбда-выражения, 19
 вопросы, 115
 доступ к внешней локальной
 переменной, 117
использование в конструкторе
`Thread`, 20
и ссылки на методы, 22
контролируемые исключения, 138
присваивание переменной, 20
реализация метода `compare`
интерфейса `Comparator`, 92
совместимость с сигнатурой
метода, 20

М

массивы
 `Arrays.stream` метод, 52
 использование ссылки
 на конструктор, 30
 преобразование `IntStream` в массив
 `int`, 56
 создание методом
 `Collectors.toArray`, 96
 создание потока, 52
метатипы, 250
 неограниченные, 250
 ограниченные сверху, 251
 ограниченные снизу, 253
метод `flatMap(Collectors)`, 238
методы по умолчанию
в интерфейсах, 31, 33

использование, 35
 конфликты, 36, 126
 минимальное и максимальное значения, нахождение, 106
 множества
 Set.of метод, 111
 и порядок следования, 78
 метод unmodifiableSet, 109
 создание неизменяемых, 231
 множественное наследование, 33
 модули, 223
 определение, 225
 примеры, 224
 Мура закон, 18

Н
 начальный момент, 180
 неизменяемость
 создание неизменяемой коллекции, 109, 229
 экземпляры класса Optional, 144

О
 обработка исключений
 в лямбда-выражениях
 возбуждение контролируемых исключений, 138
 применение вынесенного метода, 136
 метод CompletableFuture.handle, 214
 метод completeExceptionally, 208
 универсальная обертка исключений, 141
 обход каталогов в глубину, 163
 отложенное выполнение, поддержка со стороны Supplier, 43, 132
 отображение-фильтрация-редукция процесс, 57
 отображения
 добавление линейной коллекции, 97
 метод unmodifiableMap, 109
 новые методы по умолчанию, 122
 обход методом forEach, 130
 подчиненный коллектор flatMapping, 238
 создание методом Collectors.toMap, 97

создание методом groupingBy, 159
 создание немодифицируемого отображения, 110, 231
 сортировка, 99

П

палиндром, проверка, 70
 параллелизм и конкурентность, 192
 CompletableFuture класс, 206
 интерфейс Future, 203
 когда распараллеливание помогает, 196
 координация объектов CompletableFuture, 210
 определение понятий, 192
 преобразование последовательного потока в параллельный, 193
 побочные эффекты, и интерфейс Consumer, 129
 подчиненные коллекторы, 73, 236
 метод Collectors.counting, 159
 метод filtering, 236, 238
 метод flatMapping, 236, 238
 метод groupingBy, 101
 метод partitioningBy, 103
 порядок следования, 77
 и множества, 78
 поставщики, 42
 метод supplier класса Collectors, 112
 поддержка отложенного выполнения, 43
 реализация интерфейса Supplier, 42
 потоки, 51
 интерфейс AutoCloseable, 158
 и файловый ввод-вывод, 157
 конкатенация, 85
 ленивые, 89
 методы anyMatch, allMatch и noneMatch, 81
 методы flatMap и map, 83
 нахождение первого элемента, 76
 новая функциональность, добавленная в Java 9, 233
 обработка содержимого текстового файла, 158
 операции редукции, 57

отладка с помощью реек, 66
параллельные, когда
использовать, 196
подсчет элементов, 72
получение файлов в виде потока, 160
преобразование в список
и обратно, 28
преобразование последовательного
потока в параллельный, 193
преобразование строк
в коллекцию, 95
преобразование строк в потоки
и наоборот, 69
сводные статистики, 73
случайных чисел в заданном
диапазоне, 120
создание, 51
 из коллекции, 53
 методом Arrays.stream, 52
 методом Stream.generate, 53
 методом Stream.iterate, 53
 методом Stream.of, 52
создание коллекции из потока
значений примитивных типов, 55
создание потока дат, 243
сортировка с помощью
компаратора, 91
фильтрация с помощью
предикатов, 44
потребители, 39
 интерфейс Consumer
 другие применения, 41
 методы композиции, 133
 реализация, 40
предикаты, 44
 аргумент метода Collectors.
 partitioningBy, 73
 интерфейс Predicate, методы, 45
 использование в стандартной
 библиотеке, 47
 методы композиции в интерфейсе
 Predicate, 134
 нахождение строк, удовлетворяющих
 предикату, 45
префиксы методов из Date-Time
API, 168

примитивные типы в универсальных
коллекциях, 247
промежуточные операции
конвейера, 67
прослушиватели событий, 118
простое и легкое, 193
протоколирование, добавление
в модуль Supplier, 226
пул потоков
 выполнение CompletableFuture
 в отдельном пуле, 212
 изменение размера, 201
 размер, 197
пустые потоки
 метод findFirst, 77
 методы anyMatch, allMatch
 и noneMatch, 82

Р

разбиение и группировка, 102
региона, нахождение названий
по смещению от UTC, 187
ромбовидный оператор (<>), 247

С

свойства, 150
слабая состоятельность, 161
случайные числа, создание потока, 120
списки
 unmodifiableList метод, 109
 создание неизменяемых списков
 методом List.of, 229
 фильтрация null, 117
ссылки на конструкторы, 26
 и массивы, 29
 конструктор с переменным числом
 аргументов, 28
 копирующий конструктор, 27
ссылки на методы, 22
 синтаксис, 24
 ссылки на конструкторы, 26
 эквивалентное лямбда-выражение, 25
статические методы в интерфейсах, 31, 36
ссылки на методы, 23
 требования, 37
степень параллелизма, 201

строки

- группировка по длине с помощью метода `Collectors.groupingBy`, 103
- конкатенация потока с помощью метода `reduce`, 62
- лексикографическая сортировка, 91
- преобразование в потоки и наоборот, 69
- проверка сортировки методом `Stream.reduce`, 65
- разбиение на группы с помощью метода `Collectors.partitioningBy`, 102
- сборание с помощью `StringBuilder`, 63
- сортировка методом `sorted` интерфейса `Stream`, 92

Т

- терминальная операция, 79, 89
- метод `Stream.collect`, 95

У

- укорачивающая операция, 79, 89
- универсальные типы и Java 8, 246
- `PECS`, 254
- метатипы, ограниченные сверху, 251
- метатипы, ограниченные снизу, 253
- неограниченные метатипы, 250
- несколько ограничений, 254
- объявление универсальных методов в неуниверсальных типах, 248
- примеры, 255
- `Comparator.comparing` метод, 258
- `Map.Entry.comparingByKey` и `Map.Entry.comparingByValue` методы, 258
- `Stream.map` метод, 256
- `Stream.max` метод, 255
- стирание типа, 261

Ф

- файловый ввод-вывод, 157
- обработка текстового файла с помощью потоков, 158
- обход файловой системы, 161
- поиск в файловой системе, 163
- получение файлов в виде потока, 160
- Фибоначчи числа, рекурсивное вычисление, 123
- фильтры
 - применение к `Optional`, 240
 - фильтрация данных с помощью предиката, 44
- форматирование классов из пакета `java.time`
 - использование класса `DateTimeFormatter`, 182
 - класс `LocalDate`, 182
 - определение своего формата, 184
- функции
 - завершители, 110
 - метод `Function.identity`, 99
 - методы композиции, 132
- функциональные интерфейсы
 - определение, 19
 - пакет `java.util.function`, 39
 - присваивание лямбда-выражения, 20

Х

- хронометраж с помощью библиотеки `JMH`, 198

Ч

- чистые функции, 18, 129

Э

- эффективно финальные переменные, 119

Об авторе

Кен Коузен – технический писатель, разработчик программного обеспечения и частый докладчик на конференциях. Специализируется на Java и проектах с открытым исходным кодом, в т. ч. Android, Spring, Hibernate/JPA, Groovy, Grails и Gradle. Автор книг «Gradle Recipes for Android» (издательство O’Reilly) и «Making Java Groovy» (издательство Manning). Также подготовил несколько видеокурсов для O’Reilly на темы Android, Groovy, Gradle, Grails 3, Advanced Java и Spring Framework.

Выступал на технических конференциях по всему миру, в т. ч. с основными докладами на DevNexus в Атланте и Gr8conf в Миннеаполисе, Копенгагене и Нью-Дели. В 2013-м и 2016-м был удостоен звания JavaOne Rockstar.

Получил ученые степени бакалавра по инженерной механике и математике в МТИ, степени магистра и доктора по авиакосмической технике в Принстонском университете, а также степень магистра информатики в RPI. В настоящее время является президентом компании Kousen IT, Inc. со штаб-квартирой в Коннектикуте.

Об иллюстрации на обложке

На обложке книги изображен замбар (*Rusa unicolor*), вид крупного оленя, обитающего в Южной Азии вдоль берегов рек. Взрослая особь достигает высоты 100–160 см в холке и весит от 90 до 320 кг. Самцы значительно крупнее самок. Замбар – третий по размеру из ныне существующих видов оленевых после оленя-вапити и лося.

Замбары ведут ночной или сумеречный образ жизни, обычно собираются в небольшие группы. Самцы большую часть года живут в одиночестве, а самки – в стаде, иногда насчитывающем всего три особи. Замбары лучше, чем другие олени, могут подниматься на задние ноги, что позволяет им дотягиваться до высоко растущих листьев и метить территорию, а также отпугивать хищников. Замбары отличаются от других оленей тем, как самки защищают молодняк, – они предпочитают обороняться в воде, где могут воспользоваться преимуществами своего роста и умением хорошо плавать.

В Красной книге Международного союза охраны природы замбару в 2008 году присвоен статус «находится под угрозой исчезновения». Численность популяции сокращается вследствие развития промышленности и сельского хозяйства и соответственно уменьшения среды обитания, а также чрезмерного промысла – рога самцов высоко ценятся как охотничьи трофеи и используются в традиционной медицине. Хотя популяция замбаров в Азии сокращается, в Новой Зеландии и Австралии их численность неуклонно растет с момента переселения в XIX веке, и в настоящее время они даже представляют угрозу для местных видов растений.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для нашего мира. Если хотите узнать, чем вы можете помочь, зайдите на сайт animals.oreilly.com. Изображение на обложке взято из книги Lydekker «The Royal Natural History».

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.aliants-kniga.ru.
Оптовые закупки: тел. (499) 782-38-89.
Электронный адрес: books@aliants-kniga.ru.

Кен Коузен

Современный Java: рецепты программирования

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. ***. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф