

Java SE 8

Вводный курс



Кей С. Хорстманн

Java SE 8

for the Really Impatient

Cay S. Horstmann

 Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Java SE 8

Вводный курс

Кей С. Хорстманн



Издательский дом “Вильямс”
Москва – Санкт-Петербург – Киев
2014

ББК 32.973.26-018.2.75

X82

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция И.В. Берштейна

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

Хорстмани, Кей С.

X82 Java SE 8. Вводный курс. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2014. — 208 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1900-7 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc., Copyright © 2014 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2014

Научно-популярное издание

Кей С. Хорстмани

Java SE 8. Вводный курс

Литературный редактор **И.А. Попова**

Верстка **Л.В. Чернокозинская**

Художественный редактор **Е.П. Дынник**

Корректор **Л.А. Гордиенко**

Подписано в печать 05.05.2014. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 16,77. Уч.-изд. л. 11,72.

Тираж 1500 экз. Заказ № 3146.

Первая Академическая типография “Наука”
199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1900-7 (рус.)

ISBN 978-0-321-92776-7 (англ.)

© 2014, Издательский дом “Вильямс”

© 2014, Pearson Education, Inc.

Оглавление

ОБ АВТОРЕ	11
ПРЕДИСЛОВИЕ	13
БЛАГОДАРНОСТИ	15
1 ЛЯМБДА-ВЫРАЖЕНИЯ	17
2 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ПОТОКОВ ВВОДА-ВЫВОДА	35
3 ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ ЛЯМБДА-ВЫРАЖЕНИЙ	61
4 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС JAVAFX	81
5 НОВЫЙ ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ДЛЯ ДАТЫ И ВРЕМЕНИ	111
6 УСОВЕРШЕНСТВОВАНИЯ ПАРАЛЛЕЛИЗМА	127
7 ИНТЕРПРЕТАТОР NASHORN ЯЗЫКА JAVASCRIPT	145
8 РАЗНЫЕ ПОЛЕЗНЫЕ СРЕДСТВА	163
9 НЕДОСТАТОЧНО ОСВЕЩЕННЫЕ ЯЗЫКОВЫЕ СРЕДСТВА В JAVA 7	183
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	203

Содержание

ОБ АВТОРЕ	11
ПРЕДИСЛОВИЕ	13
БЛАГОДАРНОСТИ	15
1 ЛЯМБДА-ВЫРАЖЕНИЯ	17
Назначение лямбда-выражений	18
Синтаксис лямбда-выражений	20
Функциональные интерфейсы	22
Ссылки на методы	24
Ссылки на конструкторы	25
Область действия переменных	26
Методы по умолчанию	29
Статические методы в интерфейсах	32
Упражнения	33
2 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ПОТОКОВ ВВОДА-ВЫВОДА	35
От итерации к операциям с потоками ввода-вывода	36
Создание потока ввода-вывода	38
Методы filter(), map() и flatMap()	39
Извлечение подпотоков и объединение потоков ввода-вывода	40
Преобразования с сохранением состояния	41
Простые операции сведения	42
Тип данных Optional	43
Обращение со значениями типа Optional	43
Формирование значений типа Optional	44
Составление функций дополнительных значений методом flatMap()	45
Операции сведения	46
Накопление результатов	47
Накопление данных в отображениях	49
Группирование и разделение	50
Потоки ввода-вывода примитивных типов	53
Параллельные потоки ввода-вывода	55
Функциональные интерфейсы	57
Упражнения	58
3 ПРОГРАММИРОВАНИЕ С ПОМОЩЬЮ ЛЯМБДА-ВЫРАЖЕНИЙ	61
Отложенное выполнение	62
Параметры лямбда-выражений	63
Выбор функционального интерфейса	64
Возврат функций	67
Составление операций	68
Отложенность операций	70
Распараллеливание операций	71
Обработка исключений	72

Лямбда-выражения и обобщения	74
Одноместные операции	76
Упражнения	77
4 ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС JAVAFX	81
Краткая история программирования ГПИ средствами Java	82
Применение JavaFX	84
Обработка событий	85
Свойства JavaFX	86
Привязки	88
Компоновка	92
Язык разметки FXML	98
Таблицы стилей CSS	101
Анимация и спецэффекты	103
Декоративные элементы управления	105
Упражнения	109
5 НОВЫЙ ПРИКЛАДНОЙ ПРОГРАММНЫЙ ИНТЕРФЕЙС API ДЛЯ ДАТЫ И ВРЕМЕНИ	111
Временная шкала	112
Местные даты	115
Корректоры дат	117
Местное время	118
Поясное время	119
Форматирование и синтаксический анализ даты и времени	122
Взаимодействие с устаревшим кодом	125
Упражнения	126
6 УСОВЕРШЕНСТВОВАНИЯ ПАРАЛЛЕЛИЗМА	127
Атомарные значения	128
Усовершенствования в классе ConcurrentHashMap	131
Обновление значений	132
Групповые операции	134
Представления множеств	136
Параллельные операции с массивами	137
Завершаемые будущие действия	138
Будущие действия	138
Составление будущих действий	139
Конвейер составления	139
Составление асинхронных операций	140
Упражнения	143
7 ИНТЕРПРЕТАТОР NASHORN ЯЗЫКА JAVASCRIPT	145
Выполнение интерпретатора Nashorn из командной строки	146
Выполнение интерпретатора Nashorn из кода Java	148
Вызов методов	149
Построение объектов	150
Символьные строки	151
Числа	151
Обращение с массивами	152
Списки и отображения	153
Лямбда-выражения	154
Расширение классов и реализация интерфейсов Java	154
Исключения	156

Написание сценариев командного процессора	156
Выполнение команд из командного процессора	157
Интерполяция символьных строк	158
Ввод данных в сценарий	158
Nashorn и JavaFX	160
Упражнения	161
РАЗНЫЕ ПОЛЕЗНЫЕ СРЕДСТВА	163
Символьные строки	164
Числовые классы	165
Новые математические функции	166
Коллекции	167
Методы, введенные в классы коллекций	167
Компараторы	167
Класс Collections	169
Обращение с файлами	169
Потоки ввода-вывода строк	169
Потоки ввода-вывода содержимого каталогов	171
Кодировка Base64	172
Аннотации	173
Повторяющиеся аннотации	173
Аннотации к использованию типов	175
Рефлексия параметров метода	176
Различные незначительные изменения	177
Проверки пустых значений	177
Отложенные сообщения	177
Регулярные выражения	178
Региональные настройки	178
Технология JDBC	180
Упражнения	180
9 НЕДОСТАТОЧНО ОСВЕЩЕННЫЕ ЯЗЫКОВЫЕ СРЕДСТВА В JAVA 7	183
Изменения в обработке исключений	184
Оператор try с ресурсами	185
Подавляемые исключения	186
Перехват нескольких исключений	187
Упрощение обработки исключений для рефлексивных методов	187
Обращение с файлами	188
Пути	188
Чтение и запись данных в файлы	190
Создание файлов и каталогов	191
Копирование, перемещение и удаление файлов	192
Реализация методов equals(), hashCode() и compareTo()	193
Безопасная проверка на равенство пустым значениям	193
Вычисление хеш-кодов	193
Сравнение числовых типов	194
Требования к безопасности	195
Прочие изменения	198
Преобразование символьных строк в числа	198
Глобальный регистратор	198
Проверки на пустые значения	199
Класс ProcessBuilder	199
Класс URLClassLoader	200
Класс BitSet	200
Упражнения	201
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ	203

Грегу Доенчу, моему неизменному редактору
в течение двух десятилетий, терпение, любезность
и рассудительность которого так меня восхищают.

Об авторе

Кей С. Хорстманн — автор книги *Scala for the Impatient* (издательство Addison-Wesley, 2012 г.), а также основной автор двухтомного издания *Core Java™, Volumes I and II, Ninth Edition* (издательство Prentice Hall, 2013 г.; в русском переводе это издание вышло в двух томах под общим названием *Java. Библиотека профессионала, 9-е издание* в ИД “Вильямс”, 2014 г.). Он также написал десяток других книг для профессиональных программистов и изучающих вычислительную технику. Кей служит профессором на кафедре вычислительной техники при университете штата Калифорния в г. Сан-Хосе и носит почетное звание “Чемпион по Java”.

Предисловие

Эта книга является кратким введением во многие новые языковые средства Java 8 и некоторые языковые средства Java 7, не привлекшие должного внимания тех, кто уже имеет опыт программирования на Java. Она написана в стиле изложения “для нетерпеливых”, впервые опробованном мною в книге *Scala for the Impatient*. В этой книге мне хотелось перейти сразу к сути дела, не вдаваясь в подробное рассмотрение преимуществ одних понятий над другими. Поэтому материал в этой книге представлен небольшими фрагментами, организованными таким образом, чтобы читатель мог извлечь полезные сведения по мере надобности. Такой подход получил широкое признание в сообществе программирующих на Scala, и поэтому я прибегнул к нему и в этой книге.

В версии Java 8 произведены существенные обновления в языке программирования и библиотеке Java. Так, благодаря лямбда-выражениям стало возможным написание “фрагментов вычислений” в краткой форме, чтобы передавать их другому коду. Их получатель может выполнить заданное вычисление в подходящий момент и зачастуюенным образом. Данное нововведение имеет очень большое значение для построения библиотек.

В частности, совершенно изменился порядок обращения с коллекциями. Вместо того чтобы задавать порядок вычисления результата: обойти коллекцию от начала и до конца, вычислить значение элемента, совпадающего с определенным условием, а затем сложить это значение с суммой, теперь достаточно указать, что именно требуется: выдать сумму всех элементов, совпадающих с некоторым условием. В этом случае библиотека сможет изменить порядок вычисления, выгодно воспользовавшись, например, параллелизмом. А если требуется проверить на совпадение лишь первые сто элементов коллекции, то вычисление можно остановить, не поддерживая для этой цели счетчик вручную.

Данный принцип внедрен в совершенно новом прикладном программном интерфейсе (API) для потоков ввода-вывода в Java 8. В главе 1 книги будет представлен синтаксис лямбда-выражений, в главе 2 — полный обзор потоков ввода-вывода. В главе 3 приводятся рекомендации по эффективной разработке собственных библиотек с помощью лямбда-выражений.

С появлением версии Java 8 разработчикам клиентских приложений придется перейти на прикладной программный интерфейс JavaFX API, поскольку библиотека Swing теперь работает в “профилактическом режиме”. В главе 4 приводится краткое введение в JavaFX API для тех программистов, которым требуется организовать графическое представление данных в программе, когда изображение оказывается на гляднее тысячи строк текста.

Ожидая многие годы, программирующие на Java могут наконец-то воспользоваться тщательно разработанной библиотекой для поддержки даты и времени. С этой целью в главе 5 подробно рассматривается прикладной программный интерфейс `java.time`. Каждая новая версия Java начинается с усовершенствований в прикладном программном интерфейсе API параллелизма, и в этом отношении версия Java 8 не стала исключением. В главе 6 рассматриваются усовершенствования атомарных

счетчиков, параллельных хеш-отображений, параллельно выполняемых операций и составляемых будущих действий асинхронных операций.

В состав Java 8 включен интерпретатор Nashorn, представляющий собой высококачественную реализацию языка JavaScript. В главе 7 будет показано, как выполнять код JavaScript на виртуальной машине Java и как организовать его взаимодействие с кодом Java. В главе 8 рассматриваются более мелкие, но не менее полезные нововведения в версии Java 8, а в главе 9 — аналогичные мелкие нововведения, но уже в версии Java 7. В частности, усовершенствования в “новой системе ввода-вывода” для обращения с файлами и каталогами, а также другие расширения функциональных возможностей библиотек, которые могли быть упущены из виду в версии Java 7.

Благодарности

Как всегда, выражаю искреннюю благодарность своему редактору Грегу Доенчу (Greg Doench), которому принадлежит замысел издать краткое пособие для опытных программистов, чтобы помочь им поскорее освоить версию Java 8. Как и прежде, Дмитрий и Алина Кирсановы превратили рукопись формата XHTML в привлекательную книгу, сделав это поразительно быстро и в то же время уделив должное внимание деталям. Благодарю также рецензентов книги, обнаруживших в ней немало досадных ошибок и внесших ряд дальних предложений по ее улучшению. Среди них хотелось бы отметить Гэйла Андерсона (Gail Anderson), Пола Андерсона (Paul Anderson), Джеймса Денвира (James Denvir), Тришу Джи (Trisha Gee), Брайна Гоетца (Brian Goetz — за особенно тщательное рецензирование), Марти Холла (Marty Hall), Анджелику Лангер (Angelika Langer), Марка Лоуренса (Mark Lawrence), Стюарта Маркса (Stuart Marks), Аттилу Сегеди (Attila Szegedi), а также Джима Уивера (Jim Weaver).

Надеюсь, что чтение этого краткого введения в новые языковые средства Java 8 окажется для вас занимательным и что оно позволит вам успешнее программировать на Java. Если вы обнаружите ошибки в тексте книги или пожелаете внести предложения по ее улучшению, посетите веб-страницу по адресу <http://horstmann.com/java8> и оставьте свой комментарий. Там же вы найдете ссылку на архивный файл всего исходного кода, приведенного на страницах этой книги.

*Кей Хорстманн
Сан-Франциско, 2013 г.*

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д.43, стр. 1
в Украине: 03150, Киев, а/я 152

Глава

1

Лямбда-выражения

В этой главе...

- Назначение лямбда-выражений
- Синтаксис лямбда-выражений
- Функциональные интерфейсы
- Ссылки на методы
- Ссылки на конструкторы
- Область действия переменных
- Методы по умолчанию
- Статические методы в интерфейсах
- Упражнения

Язык программирования Java был разработан как объектно-ориентированный в 1990-е годы, когда методика объектно-ориентированного программирования стала господствующей в разработке программного обеспечения. А задолго до объектно-ориентированного программирования появились языки функционального программирования вроде Lisp и Scheme, но их преимущества не были должным образом оценены за пределами академических кругов. И лишь недавно значение функционального программирования возросло потому, что оно оказывается вполне пригодным для параллельного и событийно-ориентированного (так называемого “реактивного”) программирования. Это совсем не означает, что объекты сами по себе плохи. Напротив, выигрышная стратегия состоит в том, чтобы сочетать объектно-ориентированное программирование с функциональным. И в этом есть свой смысл, даже если вас мало интересует параллелизм. Например, библиотеки коллекций могут быть оснащены эффективными прикладными программными интерфейсами (API), если язык программирования обладает удобным синтаксисом для функциональных выражений.

К числу основных усовершенствований в Java 8 относится внедрение конструкций функционального программирования в качестве прививки на его объектно-ориентированных корнях. В этой главе будет рассмотрен основной синтаксис подобных конструкций, в следующей главе будет показано, как применять этот синтаксис на практике при обращении с коллекциями в Java, а в главе 3 — как разрабатывать собственные функциональные библиотеки.

В этой главе рассматриваются следующие основные вопросы.

- Назначение лямбда-выражения в качестве кодового блока с параметрами.
- Применение лямбда-выражения всякий раз, когда требуется выполнение кодового блока в последующие моменты времени.
- Преобразование лямбда-выражений в функциональные интерфейсы.
- Организация эффективного доступа к конечным переменным из объемлющей области действия с помощью лямбда-выражений.
- Организация ссылок на методы или конструкторы без их вызова.
- Ввод методов по умолчанию и статических методов в интерфейсы, предоставляющие конкретные их реализации.
- Разрешение любых конфликтов между методами по умолчанию из нескольких интерфейсов.

Назначение лямбда-выражений

Лямбда-выражение представляет собой кодовый блок, который может быть передан для последующего выполнения один или несколько раз. Прежде чем переходить непосредственно к рассмотрению синтаксиса лямбда-выражений (или даже необычному их названию), сделаем небольшое отступление, чтобы выяснить, где именно аналогичные кодовые блоки уже давно применяются в Java.

Когда требуется выполнить определенное задание в отдельном потоке исполнения, это задание размещается в методе `run()` из интерфейса `Runnable` следующим образом:

```
class Worker implements Runnable {  
    public void run() {
```

```

for (int i = 0; i < 1000; i++)
    doWork();
}
...
}

```

А когда требуется выполнить это задание в коде, конструируется экземпляр класса `Worker`. Затем этот экземпляр можно предъявить пулу потоков исполнения или просто начать новый поток, как показано ниже. Но самое главное, что метод `run()` содержит код, который требуется выполнить в отдельном потоке.

```

Worker w = new Worker();
new Thread(w).start();

```

В качестве другого примера рассмотрим сортировку с помощью специального компаратора. Так, если требуется отсортировать символьные строки по длине, а не в словарном порядке по умолчанию, то для этой цели можно передать объект типа `Comparator` методу `sort()` следующим образом:

```

class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return Integer.compare(first.length(), second.length());
    }
}

```

```
Arrays.sort(strings, new LengthComparator());
```

Метод `compare()` вызывается в методе `sort()` для реорганизации элементов массива, если они неупорядочены, до тех пор, пока массив не будет отсортирован. При этом методу `sort()` передается фрагмент кода, требующийся для сравнения элементов. Этот код внедряется в остальную часть логики сортировки, которую было бы нежелательно реализовывать снова.



НА ЗАМЕТКУ. В результате вызова метода `Integer.compare(x, y)` возвращается нулевое значение, если значения параметров `x` и `y` равны; отрицательное значение, если `x < y`; или же положительное значение, если `x > y`. Этот статический метод был внедрен в версии Java 7 (см. главу 9). Следует иметь в виду, что вычислять разность `x - y` для сравнения значений параметров `x` и `y` не нужно, поскольку такое вычисление может привести к переполнению при наличии крупных операндов противоположного знака.

В качестве еще одного примера отложенного выполнения рассмотрим обратный вызов функции кнопки. Действие обратного вызова можно разместить в методе класса, реализующего интерфейс приемника событий от кнопки, сконструировать экземпляр и зарегистрировать его в кнопке. И это происходит настолько часто, что многие программисты пользуются следующим синтаксисом анонимного экземпляра анонимного класса:

```

button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Thanks for clicking!");
    }
});

```

Основное значение имеет код в теле метода `handle()`. Этот код выполняется всякий раз, когда производится щелчок на кнопке.



НА ЗАМЕТКУ. В версии Java 8 прикладной программный интерфейс JavaFX рассматривается как преемник набора инструментальных средств Swing для построения графического пользовательского интерфейса (ГПИ) приложений, и поэтому именно он используется в представленных здесь примерах (подробнее о JavaFX см. в главе 4). Не вдаваясь в детали, следует заметить, что в каждом наборе инструментальных средств для построения ГПИ, будь то Swing, JavaFX или Android, для отдельной кнопки назначается некоторый код, который требуется выполнить, когда она выбирается щелчком на ней кнопкой мыши.

Во всех трех рассмотренных выше примерах кода применяется один и тот же подход. В частности, кодовый блок передается какому-нибудь другому элементу программы, будь то пул потоков исполнения, метод `soft()` или кнопка. И этот кодовый блок вызывается в какой-то последующий момент времени.

В прежних версиях Java передать кодовый блок какому-нибудь другому элементу программы было нелегко. Ведь Java — это язык объектно-ориентированного программирования, а следовательно, для передачи кодового блока приходилось конструировать объект, относящийся к классу, в котором имеется метод с требуемым кодом.

Обращаться непосредственно с кодовыми блоками можно и в других языках программирования. Но разработчики Java долго противились внедрению этого языкового средства. Ведь главная сила языка Java — в его простоте и согласованности. И если внедрить в язык программирования всякое средство, позволяющее получать более или менее краткий код, то такой язык в конечном итоге может прийти в полный беспорядок. Но и в других языках программирования не так-то просто породить поток исполнения или зарегистрировать обработчик событий от щелчка на кнопке, хотя подавляющее большинство прикладных программных интерфейсов API в них реализовано проще, согласованнее и эффективнее. Аналогичные прикладные программные интерфейсы API, принимающие объекты классов, реализующих отдельные функции, можно было бы написать и на Java, но пользоваться ими было бы неудобно.

До недавнего времени главный вопрос состоял не в том, следует ли расширять Java средствами функционального программирования, а в том, как это сделать. Несколько лет ушло на экспериментирование, прежде чем был выработан замысел, вполне пригодный для Java. В следующем разделе будет показано, как обращаться с кодовыми блоками в Java 8.

Синтаксис лямбда-выражений

Вернемся к примеру сортировки из предыдущего раздела. В приведенной ниже строке кода для сравнения методу `compare()` передаются две символьные строки, одна из которых короче другой.

```
Integer.compare(first.length(), second.length())
```

А что обозначают ссылки `first` и `second`? Они обозначают символьные строки. В связи с тем что Java — строго типизированный язык программирования, необходимо также указать тип сравниваемых строк, как показано ниже.

```
(String first, String second)  
-> Integer.compare(first.length(), second.length())
```

Итак, выше было составлено первое лямбда-выражение. Такое выражение представляет собой кодовый блок наряду с указанием любых переменных, которые должны быть переданы коду.

А откуда происходит название лямбда-выражение? Много лет назад, когда никаких компьютеров не было и в помине, специалисту по логике Алонсо Черчу (Alonzo Church) потребовалось каким-то образом формализовать средства, предназначавшиеся для эффективного вычисления некоторой математической функции. (Любопытно, что имеются функции, о которых известно, что они существуют, но никто не знает, как вычислять их значения.) Для обозначения параметров данной функции Алонсо Черч воспользовался буквой "лямбда" (λ) греческого алфавита. Если бы ему был известен прикладной программный интерфейс Java API, он написал бы определение данной функции следующим образом:

```
λfirst.λsecond.Integer.compare(first.length(), second.length())
```



НА ЗАМЕТКУ. Почему же была выбрана именно буква λ ? Неужели для этого не нашлось других букв в алфавите? Для обозначения свободных переменных в почтенном труде *Principia Mathematica* [Начала математики] употреблялся знак ударения (^), что вдохновило Черча на обозначение параметров функции прописной буквой "лямбда" (Λ). Но в конечном итоге он остановил свой выбор на строчной букве "лямбда" (λ). С тех пор выражения с переменными параметрами стали называться лямбда-выражениями.

Выше была приведена одна форма лямбда-выражения на Java, где сначала указываются параметры, затем стрелка \rightarrow и, наконец, выражение. Если в коде выполняется вычисление, не вписывающееся в одно выражение, напишите его так, как обычно пишется метод, тело которого заключается в фигурные скобки {} и явно указываются операторы возврата. Ниже приведен характерный тому пример.

```
(String first, String second) -> {
    if (first.length() < second.length()) return -1;
    else if (first.length() > second.length()) return 1;
    else return 0;
}
```

Если у лямбда-выражения отсутствуют параметры, то круглые скобки все равно нужно указывать, как и в методе без параметров:

```
() -> { for (int i = 0; i < 1000; i++) doWork(); }
```

Если же типы параметров могут быть выведены автоматически, их можно опустить, как показано в приведенном ниже примере.

```
Comparator<String> comp
= (first, second) // аналогично выражению (String first, String second)
-> Integer.compare(first.length(), second.length());
```

В данном примере компилятор может вывести строковый тип первого и второго параметров, поскольку лямбда-выражение присваивается строковому компаратору. (Более подробно эта операция присваивания будет рассмотрена в следующем разделе.) Если у метода имеется единственный параметр выводимого типа, в таком случае можно даже опустить круглые скобки:

```
EventHandler<ActionEvent> listener = event ->
    System.out.println("Thanks for clicking!");
    // вместо выражения (event) -> или (ActionEvent event) ->
```



НА ЗАМЕТКУ. Аннотации или модификатор доступа final можно вводить в параметры лямбда-выражения таким же образом, как и в параметры методов. Ниже приведены характерные тому примеры.

```
(final String name) -> ...
(@NonNull String name) -> ...
```

Тип результата вычисления лямбда-выражения указывать вообще не нужно. Ведь он всегда выводится из контекста. Например, следующее выражение может быть использовано в контексте, где предполагается тип результата int:

```
(String first, String second) -> Integer.compare(first.length(), second.length())
```



НА ЗАМЕТКУ. Из лямбда-выражения допускается возвращать значение в одних ветвях, но не в других. Например, выражение `(int x) -> { if (x >= 0) return 1; }` недействительно.

Функциональные интерфейсы

Как обсуждалось ранее, в Java имеется немало интерфейсов, инкапсулирующих кодовые блоки, например, интерфейс Runnable или Comparator. Лямбда-выражения обратно совместимы с этими интерфейсами. В частности, лямбда-выражение можно предоставлять всякий раз, когда предполагается объект интерфейса с единственным абстрактным методом. И такой интерфейс называется функциональным.



НА ЗАМЕТКУ. В связи с изложенным выше могут возникнуть следующие вопросы: почему у функционального интерфейса должен быть единственный абстрактный метод и не являются ли все методы в интерфейсе абстрактными? По существу, в интерфейсе можно всегда переопределить методы из класса Object, например метод `toString()` или `clone()`, причем эти объявления не делают методы абстрактными. (В некоторых интерфейсах из прикладного программного интерфейса Java API переопределяются методы из класса Object, чтобы присоединить комментарии из утилиты javadoc. Соответствующий пример можно найти в прикладном программном интерфейсе API для интерфейса Comparator.) Но еще важнее, что в интерфейсах версии Java 8 могут быть определены и неабстрактные методы, как поясняется далее в разделе "Методы по умолчанию".

Для того чтобы продемонстрировать преобразование в функциональный интерфейс, рассмотрим метод `Arrays.sort()`. В качестве второго параметра этого метода требуется указывать экземпляр Comparator — интерфейса с единственным методом. Для этого достаточно предоставить лямбда-выражение следующим образом:

```
Arrays.sort(words,
    (first, second) -> Integer.compare(first.length(), second.length()));
```

Подспудно метод `Arrays.sort()` получает объект некоторого класса, реализующего интерфейс `Comparator<String>`. При вызове метода `compare()` для этого объекта выполняется тело лямбда-выражения. Управление этими объектами и классами полностью зависит от реализаций и может быть намного более эффективным, чем

применение традиционных внутренних классов. Лямбда-выражение лучше всего рассматривать в качестве функции, а не объекта. И для принятия лямбда-выражения его достаточно передать функциональному интерфейсу.

Именно такое преобразование в функциональные интерфейсы делает столь привлекательными лямбда-выражения. Их синтаксис краток и прост. Ниже приведен еще один тому пример. Такой код более удобочитаем, чем альтернативный код внутренних классов.

```
button.setOnAction(event ->
    System.out.println("Thanks for clicking!"));
```

В действительности преобразование в функциональный интерфейс — это *единственное*, что можно делать с лямбда-выражением в Java. В других языках программирования, поддерживающих функциональные литералы, можно объявлять типы функций, например `(String, String) -> int`, а также объявлять и применять переменные этих типов для сохранения функциональных выражений. Но разработчики Java решили придерживаться уже известного понятия интерфейсов вместо внедрения типов функций в этот язык программирования.



НА ЗАМЕТКУ. Лямбда-выражение нельзя даже присвоить переменной типа `Object`. Ведь тип `Object` не относится к функциональному интерфейсу.

В пакете `java.util.function` прикладного программного интерфейса Java API определяется целый ряд весьма обобщенных функциональных интерфейсов. (Подробнее о них речь пойдет в главах 2 и 3.) К их числу относится интерфейс `BiFunction<T, U, R>`, в котором описываются функции с параметрами обобщенных типов `T` и `U` и возвращаемым типом `R`. Лямбда-выражение, в котором сравниваются символьные строки, можно сохранить в переменной такого типа следующим образом:

```
BiFunction<String, String, Integer> comp
    = (first, second) -> Integer.compare(first.length(), second.length());
```

Но это никак не помогает произвести сортировку, поскольку отсутствует метод `Arrays.sort()`, которому требуется тип `BiFunction`. Такое поведение может показаться не совсем обычным для тех, у кого уже имеется некоторый опыт функционального программирования. Но для программирующих на Java это вполне естественно. Такой интерфейс, как, например, `Comparator`, имеет специальное назначение, а не только метод с заданными типами параметров и возвращаемого значения. И эта особенность сохраняется в версии Java 8. Если требуется сделать что-нибудь с лямбда-выражениями, нужно по-прежнему принимать во внимание назначение выражения, имея для этого отдельный функциональный интерфейс.

Функциональные интерфейсы из пакета `java.util.function` применяются лишь в нескольких прикладных программных интерфейсах API версии Java 8, а в будущем они, вероятнее всего, потребуются везде. Но не следует забывать, что лямбда-выражение можно в равной степени преобразовать в функциональный интерфейс, входящий в состав любого применяемого ныне прикладного программного интерфейса API.



НА ЗАМЕТКУ. Любой функциональный интерфейс можно пометить аннотацией `@FunctionalInterface`. Такая мера дает два преимущества: компилятор проверяет, является ли аннотированный элемент интерфейсом с единственным абстрактным методом, а на странице,

формируемой утилитой javadoc, появляется заявление о том, что данный интерфейс является функциональным.

Пользоваться аннотациями совсем не обязательно. Любой интерфейс с единственным абстрактным методом по определению является функциональным. Тем не менее пользоваться аннотацией @FunctionalInterface целесообразно.

И наконец, следует иметь в виду проверяемые исключения, когда лямбда-выражение преобразуется в экземпляр функционального интерфейса. Если в теле лямбда-выражения может быть сгенерировано проверяемое исключение, его нужно объявить в абстрактном методе целевого интерфейса. Например, приведенная ниже строка кода окажется ошибочной.

```
Runnable sleeper = () -> { System.out.println("Zzz"); Thread.sleep(1000); };
// Ошибка: в методе Thread.sleep() может быть сгенерировано
// проверяемое исключение InterruptedException
```

Метод Runnable.run() не может генерировать никаких исключений, и поэтому приведенная выше операция присваивания оказывается недопустимой. Исправить ошибку можно двумя способами. Во-первых, перехватить исключение в теле лямбда-выражения, а во-вторых, присвоить лямбда-выражение интерфейсу, в единственном абстрактном методе которого может быть сгенерировано исключение. Например, в методе call() из интерфейса Callable может быть сгенерировано любое исключение. Следовательно, лямбда-выражение может быть присвоено переменной экземпляра типа Callable<Void> (если добавить в него оператор return null).

Ссылки на методы

Иногда уже имеется метод, в котором выполняется именно такое действие, которое требуется передать для выполнения другому коду. Допустим, требуется лишь выводить событийный объект всякий раз, когда производится щелчок на кнопке. Для этого можно было бы, конечно, сделать следующий вызов:

```
button.setOnAction(event -> System.out.println(event));
```

Но изящнее было бы просто передать метод println() методу setOnAction(). Ниже показано, как это можно сделать.

```
button.setOnAction(System.out::println);
```

Выражение System.out::println является ссылкой на метод, равнозначной лямбда-выражению x -> System.out.println(x). В качестве другого примера допустим, что требуется отсортировать символьные строки независимо от регистра букв. Для этого можно передать следующую ссылку на метод:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

Как следует из этих примеров, оператор :: отделяет имя метода от имени объекта или класса. Для этого имеются три основных варианта:

- объект::метод_экземпляра
- класс::статический_метод
- класс::метод_экземпляра

В двух первых вариантах ссылка на метод равнозначна лямбда-выражению, в котором предоставляются параметры метода. Как упоминалось ранее, ссылка на метод `System.out::println` равнозначна лямбда-выражению `x -> System.out.println(x)`. Аналогично ссылка на метод `Math::pow` равнозначна лямбда-выражению `(x, y) -> Math.pow(x, y)`. А в третьем варианте первый параметр становится целевым для метода. Например, ссылка на метод `String::compareToIgnoreCase` равнозначна лямбда-выражению `(x, y) -> x.compareToIgnoreCase(y)`.



НА ЗАМЕТКУ. Если имеется несколько перегружаемых методов с одинаковым именем, то компилятор попытается выяснить из контекста назначение каждого из них. Например, имеются два варианта метода `Math.max()`: один — для целочисленных значений, другой — для значений с плавающей точкой двойной точности. Выбор одного из этих вариантов зависит от параметров метода того функционального интерфейса, в который преобразуется ссылка на метод `Math::max`. Подобно лямбда-выражениям, ссылки на методы не существуют обособленно. Они всегда преобразуются в экземпляры функциональных интерфейсов.

Параметр `this` можно зафиксировать в ссылке на метод. Например, ссылка на метод `this::equals` равнозначна лямбда-выражению `x -> this.equals(x)`. Допускается также пользоваться параметром `super`. Так, в следующей ссылке на метод параметр `this` используется в качестве целевого и вызывается вариант данного метода из суперкласса:

```
super::метод_экземпляра
```

В приведенном ниже искусственном примере демонстрируется механизм действия ссылки на метод. Когда начинается поток исполнения, происходит обращение к его экземпляру интерфейса `Runnable` и выполняется ссылка на метод `super::greet`, по которой вызывается метод `greet()` из суперкласса.

```
class Greeter {  
    public void greet() {  
        System.out.println("Hello, world!");  
    }  
}  
  
class ConcurrentGreeter extends Greeter {  
    public void greet() {  
        Thread t = new Thread(super::greet);  
        t.start();  
    }  
}
```



НА ЗАМЕТКУ. Во внутреннем классе ссылку `this` на объемлющий класс можно зафиксировать следующим образом: `объемлющий_класс.this:метод` или `объемлющий_класс.super:метод`.

Ссылки на конструкторы

Ссылки на конструкторы подобны ссылкам на методы, за исключением того, что в качестве имени метода указывается оператор `new`. Например, `Button::new` — это ссылка на конструктор класса `Button`, а выбор конкретного конструктора зависит от

контекста. Допустим, имеется список символьных строк. Этот список можно превратить в массив кнопок, вызвав конструктор класса `Button` для каждой символьной строки следующим образом:

```
List<String> labels = ...;
Stream<Button> stream = labels.stream().map(Button::new);
List<Button> buttons = stream.collect(Collectors.toList());
```

Более подробно методы `stream()`, `map()` и `collect()` обсуждаются в главе 2. А до тех пор важно отметить следующее: в методе `map()` конструктор `Button(String)` вызывается для каждого элемента списка. В классе `Button` имеется несколько конструкторов, но компилятор выбирает один из них с параметром типа `String`, поскольку из контекста он заключает, что конструктор вызывается с символьной строкой.

Ссылки на конструкторы можно формировать с типами массивов. Например, `int[]::new` — это ссылка на конструктор с одним параметром, обозначающим длину массива. Она равнозначна лямбда-выражению `x -> new int[x]`.

Ссылки на конструкторы массивов полезны для преодоления некоторых ограничений в Java. В частности, построить массив обобщенного типа `T` нельзя, а выражение `new T[n]` ошибочно, поскольку в результате стирания оно превратится в выражение `Object[n]`. Такое ограничение затрудняет создание библиотек. Допустим, требуется составить массив кнопок. В интерфейсе `Stream` для этой цели имеется метод `toArray()`, возвращающий массив типа `Object`:

```
Object[] buttons = stream.toArray();
```

Но этого явно недостаточно. Пользователю требуется массив кнопок, а не объектов. В библиотеке потоков ввода-вывода этот недостаток устраняется с помощью ссылок на конструкторы. Для этого достаточно передать ссылку на конструктор `Button[]::new` методу `toArray()` следующим образом:

```
Button[] buttons = stream.toArray(Button[]::new);
```

Данный конструктор вызывается в методе `toArray()` для получения массива правильного типа. Он заполняет массив и возвращает его.

Область действия переменных

Нередко доступ к переменным требуется из объемлющего метода или класса в лямбда-выражении. Рассмотрим следующий пример:

```
public static void repeatMessage(String text, int count) {  
    Runnable r = () -> {  
        for (int i = 0; i < count; i++) {  
            System.out.println(text);  
            Thread.yield();  
        }  
    };  
    new Thread(r).start();  
}
```

Рассмотрим далее следующий вызов:

```
repeatMessage("Hello", 1000); // выводит строку «Hello» 1000 раз  
// в отдельном потоке исполнения
```

А теперь проанализируем переменные `count` и `text` в лямбда-выражении. Прежде всего следует заметить, что эти переменные *не* определены в лямбда-выражении. Напротив, они являются переменными параметров метода `repeatMessage()`.

Но если поразмыслить, то в рассматриваемом здесь примере кода происходит нечто не совсем очевидное. В частности, код лямбда-выражения может быть выполнен спустя немало времени после возврата из вызова метода `repeatMessage()`, когда переменные параметров этого метода уже давно отсутствуют. Каким же образом остаются переменные `text` и `count`?

Для того чтобы понять, что же на самом же происходит, следует внести дополнительную ясность в само понятие лямбда-выражения. В частности, лямбда-выражение образуется из следующих трех составляющих:

1. Кодовый блок.
2. Параметры.
3. Значения *свободных* переменных, т.е. таких переменных, которые не являются параметрами и не определены в коде.

В рассматриваемом здесь примере у лямбда-выражения имеются две свободные переменные: `text` и `count`. В структуре данных, представляющей лямбда-выражение, должны храниться значения этих переменных (в данном случае символьная строка `"Hello"` и число `1000`). Эти значения были *записаны* лямбда-выражением. (Подробности этого механизма зависят от конкретной реализации. Например, лямбда-выражение можно преобразовать в объект с единственным методом, чтобы скопировать значения свободных переменных в переменные экземпляра данного объекта.)



НА ЗАМЕТКУ. Формально кодовый блок вместе со значениями свободных переменных называется *замыканием*. В Java замыканиями являются лямбда-выражения. На самом деле внутренние классы уже давно выполняют роль замыканий в Java. А в версии Java 8 замыкания лишь приобрели более привлекательный синтаксис.

Как было показано ранее, в лямбда-выражении можно зафиксировать значение переменной в объемлющей области действия. Для гарантии того, что зафиксированное значение вполне определено, в Java накладывается важное ограничение. Оно состоит в том, что в лямбда-выражении можно обращаться только к тем переменным, значения которых не изменяются. Например, следующий фрагмент кода недопустим:

```
public static void repeatMessage(String text, int count) {
    Runnable r = () -> {
        while (count > 0) {
            count--; // Ошибка: модифицировать зафиксированную переменную нельзя
            System.out.println(text);
            Thread.yield();
        }
    };
    new Thread(r).start();
}
```

Для наложения подобного ограничения имеются веские основания. Модификация переменных в лямбда-выражении не является потокобезопасной. Рассмотрим следующую последовательность параллельно решаемых задач, в каждой из которых обновляется общий счетчик:

```
int matches = 0;
for (Path p : files)
    new Thread(() -> { if (p имеет некоторое свойство) matches++; }).start();
    // модифицировать счетчик matches не разрешается
```

Если бы приведенный выше фрагмент кода был допустим, это имело бы весьма прискорбные последствия. Приращение счетчика `matches++` не является атомарной операцией, и поэтому неизвестно, что произойдет, если попытаться выполнить приращение счетчика одновременно в нескольких потоках.



НА ЗАМЕТКУ. Во внутренних классах можно также фиксировать значения из объемлющей области действия. До версии Java 8 во внутренних классах разрешался лишь доступ к конечным (`final`) локальным переменным. Это правило теперь ослаблено для согласования с лямбда-выражениями. Во внутреннем классе отныне возможен доступ к любой действительно конечной локальной переменной, т.е. к любой переменной, значение которой не изменяется.

Для отлавливания всех ошибок параллельного доступа не следует особенно полагаться на компилятор. Запрет на модификацию распространяется только на локальные переменные. Если `matches` является переменной экземпляра или статической переменной из объемлющего класса, то никакого сообщения об ошибке не последует, даже если результат не определен. Кроме того, вполне допустимо модифицировать общий объект, даже если он ошибочен, как показано в приведенном ниже примере.

```
List<Path> matches = new ArrayList<>();
for (Path p : files)
    new Thread(() -> {
        if (p имеет некоторое свойство) matches.add(p); }).start();
    // модифицировать переменную matches допустимо, но ненадежно
```

Следует иметь в виду, что переменная `matches` действительно является конечной. (Действительно конечной называется такая переменная, которой не присваивается новое значение после ее инициализации.) В данном случае переменная `matches` всегда ссылается на один и тот же объект типа `ArrayList`. Но этот объект модифицируется, что нельзя считать потокобезопасной операцией. Так, если метод `add()` вызывается из нескольких потоков, то конечный результат трудно предсказать.

Для параллельного подсчета и накопления значений существуют безопасные механизмы. В главе 2 поясняется, как пользоваться потоками ввода-вывода для накопления значений с определенными свойствами. А в остальных случаях, возможно, придется воспользоваться потокобезопасными счетчиками и коллекциями. Более подробно данный вопрос рассматривается в главе 6.



НА ЗАМЕТКУ. Как и во внутренних классах, имеется аварийный выход, позволяющий в лямбда-выражении обновлять счетчик, находящийся в объемлющей локальной области действия. С этой целью длина массива указывается равной **1** следующим образом:

```
int[] counter = new int[1];
button.setOnAction(event -> counter[0]++);
```

Разумеется, такой код не является потокобезопасным. И хотя для обратного вызова функции кнопки это не имеет особого значения, в общем случае следует дважды подумать, прежде чем воспользоваться подобным приемом. Пример реализации потокобезопасного общего счетчика приведен в главе 6.

Тело лямбда-выражения имеет такую же область действия, как и у вложенного блока. Затенение и конфликты имен разрешаются по тем же самым правилам. Не допускается объявлять параметр или локальную переменную в лямбда-выражении с таким же самым именем, как и у локальной переменной.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp =
    (first, second) -> Integer.compare(first.length(), second.length());
// Ошибка: переменная first уже определена
```

В методе не допускается наличие двух локальных переменных с одинаковым именем, а следовательно, такие переменные вообще нельзя внедрить в лямбда-выражение. Когда же в лямбда-выражении применяется ключевое слово `this`, делается ссылка на параметр `this` метода, создающего лямбда-выражение. Рассмотрим следующий пример кода:

```
public class Application() {
    public void doWork() {
        Runnable runner = () ->
            { ...; System.out.println(this.toString()); ... };
        ...
    }
}
```

Из выражения `this.toString()` вызывается метод `toString()` для объекта типа `Application`, но не экземпляра интерфейса `Runnable`. В применении ключевого слова `this` в лямбда-выражении нет ничего особенного. Ведь область действия лямбда-выражения не выходит за пределы метода `doWork()`, и поэтому ссылка `this` имеет одинаковое смысловое значение повсюду в этом методе.

Методы по умолчанию

Во многих языках программирования функциональные выражения объединяются со своей библиотекой коллекций. Это нередко приводит к более краткому и удобочитаемому коду, чем в эквивалентных циклах. В качестве примера рассмотрим следующий цикл:

```
for (int i = 0; i < list.size(); i++)
    System.out.println(list.get(i));
```

Но такой код можно написать более рационально. В частности, разработчики библиотек могут предоставить метод `forEach()`, в котором функция применяется к каждому элементу списка. И тогда достаточно сделать следующий вызов:

```
list.forEach(System.out::println);
```

Это вполне допустимо, если библиотека коллекций разработана с самого начала. Но если библиотека коллекций разработана много лет назад, то возникает затруднение. Если в интерфейс `Collection` вводятся новые методы, например `forEach()`, то каждая программа, в которой определяется свой собственный класс, реализующий интерфейс `Collection`, не будет нормально работать до тех пор, пока и в ней не реализуется данный метод. Но в Java это просто неприемлемо.

Разработчики Java решили устраниТЬ данное затруднение раз и навсегда, допустив существование методов интерфейса с конкретными реализациями, называемых *методами по умолчанию*. Такие методы могут быть благополучно введены в существующие интерфейсы. Методы по умолчанию подробно рассматриваются в этом разделе.



НА ЗАМЕТКУ. В версии Java 8 метод `forEach()` был внедрен в интерфейс `Iterable`, являющийся суперинтерфейсом по отношению к интерфейсу `Collection`. Для данной цели был использован механизм, рассматриваемый в этом разделе.

Рассмотрим следующий пример интерфейса:

```
interface Person {
    long getId();
    default String getName() { return "John Q. Public"; }
}
```

В этом интерфейсе определены следующие два метода: абстрактный метод `getId()`, а также метод по умолчанию `getName()`. Конкретный класс, реализующий интерфейс `Person`, должен, разумеется, предоставлять реализацию метода `getId()`, но в нем может быть выбрана реализация или переопределение метода `getName()`.

Методы по умолчанию кладут конец классическому шаблону предоставления интерфейса и абстрактного класса, реализующего большинство или все его методы, в том числе `Collection/AbstractCollection` или `WindowListener/WindowAdapter`. Теперь достаточно реализовать методы в интерфейсе.

Что же произойдет, если один тот же метод определен сначала как метод по умолчанию в одном интерфейсе, а затем в суперклассе или другом интерфейсе? В таких языках программирования, как Scala и C++, имеются сложные правила разрешения подобных неоднозначностей. К счастью, подобные правила в Java оказываются намного более простыми. Они таковы:

1. Одерживают верх суперклассы. Если в суперклассе предоставляется конкретный метод, то методы по умолчанию с одинаковыми именами типами параметров просто игнорируются.
2. Конфликтуют интерфейсы. Если в суперинтерфейсе предоставляется метод по умолчанию, а в другом интерфейсе — метод с таким же самым именем и типами параметров (по умолчанию или нет), то конфликт нужно разрешить, переопределив метод.

Рассмотрим второе правило. Допустим, имеется другой интерфейс с методом `getName()`, как показано ниже.

```
interface Named {
    default String getName()
    { return getClass().getName() + "_" + hashCode(); }
}
```

Что произойдет, если образовать приведенный ниже класс, реализующий оба интерфейса?

```
class Student implements Person, Named {
    ...
}
```

Данный класс наследует два несогласованных метода `getName()`, предоставляемых интерфейсами `Person` и `Named`. Вместо того чтобы отдавать предпочтение какому-либо одному интерфейсу, компилятор Java сообщает об ошибке, оставляя ее обработку программисту для устранения неоднозначности. В таком случае проще предоставить метод `getName()` в классе `Student`. В этом методе можно выбрать один из двух конфликтующих методов:

```
class Student implements Person, Named {  
    public String getName() { return Person.super.getName(); }  
    ...  
}
```

А теперь допустим, что в интерфейсе `Named` не предоставляется реализация метода `getName()` по умолчанию, как показано ниже.

```
interface Named {  
    String getName();  
}
```

Может ли класс `Student` наследовать метод по умолчанию из интерфейса `Person`? Это может быть вполне обоснованно, но разработчики Java отдают предпочтение единобразию. И совершенно не важно, каким образом оба интерфейса вступают в конфликт. Если хотя бы в одном интерфейсе предоставляется реализация метода, то компилятор сообщает об ошибке, а устраниТЬ неоднозначность должен программист.



НА ЗАМЕТКУ. Разумеется, если ни один из интерфейсов не предоставляет общий метод по умолчанию, то возникает бесконфликтная ситуация, характерная до появления версии Java 8. Для реализации этого метода в классе имеются следующие две возможности: реализовать метод или оставить его нереализованным. В последнем случае сам класс оказывается абстрактным.

Итак, мы обсудили конфликты имен в двух интерфейсах. А теперь рассмотрим класс, расширяющий суперкласс и реализующий интерфейс, наследуя один и тот же метод от того и другого. Допустим, что `Person` — это класс, а его подкласс `Student` определяется следующим образом:

```
class Student extends Person implements Named { ... }
```

В таком случае имеет значение только метод из суперкласса, а любой метод по умолчанию из интерфейса просто игнорируется. В данном примере класс `Student` наследует метод `getName()` из класса `Person`, и не имеет никакого значения, предоставляется ли в интерфейсе `Named` вариант метода `getName()` по умолчанию или нет. Это пример того правила, когда класс одерживает верх.

Правило, когда класс одерживает верх, гарантирует совместимость с версией 7. Если ввести методы по умолчанию в интерфейс, это никак не повлияет на код, разработанный до появления этих методов.



ВНИМАНИЕ. Нельзя создать метод по умолчанию, в котором переопределяется один из методов из класса `Object`. Например, нельзя определить метод по умолчанию для метода `toString()` или `equals()`, несмотря на то, что это может оказаться привлекательным для таких интерфейсов, как `List`. В итоге такой метод не может одержать верх над методом `Object.toString()` или `Object.equals()` как следствие из правила, когда одерживают верх классы.

Статические методы в интерфейсах

С версии Java 8 допускается вводить статические методы в интерфейсы. Формальных причин, по которым эта операция была бы незаконной, никогда не существовало. Это просто противоречило самому характеру интерфейсов как абстрактных описаний.

До сих пор статические методы обычно сопутствовали классам. В стандартной библиотеке можно обнаружить пары интерфейсов и служебных классов, как, например, Collection/Collections или Path/Paths.

Рассмотрим класс Paths. В нем имеется лишь пара фабричных методов. Из последовательности символьных строк можно составить путь к файлам, например Paths.get("jdk1.8.0", "jre", "bin"). В версии Java 8 можно было бы ввести этот метод в интерфейс Path следующим образом:

```
public interface Path {  
    public static Path get(String first, String... more) {  
        return FileSystems.getDefault().getPath(first, more);  
    }  
    ...  
}
```

В таком случае класс Paths больше не нужен. Анализируя класс Collections, можно обнаружить два вида методов. Например, метод

```
public static void shuffle(List<?> list)
```

вполне подойдет в качестве метода по умолчанию для интерфейса List, как показано ниже. И тогда можно просто сделать вызов list.shuffle() для любого списка.

```
public default void shuffle()
```

Фабричный метод не действует, потому что отсутствует объект, для которого этот метод можно было бы вызвать. И здесь вступают в действие статические методы. В примере

```
public static <T> List<T> nCopies(int n, T o)  
    // составляет список из n экземпляров o
```

метод мог быть объявлен как статический в интерфейсе List. И тогда можно было бы сделать вызов List.nCopies(10, "Fred") вместо вызова Collections.nCopies(10, "Fred"), чтобы читающему исходный код стало понятно, что в результате получается список типа List.

Маловероятно, чтобы код библиотеки коллекций Java был реорганизован подобным образом, но когда требуется реализовать свои интерфейсы, то больше нет никаких причин предоставлять отдельный сопутствующий класс для служебных методов. В версии Java 8 статические методы были внедрены в довольно большом количестве интерфейсов. Например, в интерфейсе Comparator имеется очень удобный статический метод сравнения, принимающий функцию извлечения ключей и предоставляющий компаратор для сравнения извлекаемых ключей. Так, для сравнения объектов типа Person по имени делается вызов Comparator.comparing(Person::name).

В этой главе было показано, как сравнивать символьные строки по длине с помощью лямбда-выражения (first, second) -> Integer.compare(first.length(),

second.length()). Но с помощью статического метода сравнения можно добиться много большего, просто сделав вызов compare(String::length). Это вполне подходящий способ завершить главу, поскольку он демонстрирует потенциальные возможности для работы с функциями. Метод сравнения превращает функцию, извлекающую ключи, в более сложную функцию для сравнения извлекаемых ключей. Подобные функции более высокого уровня подробнее рассматриваются в главе 3.

Упражнения

1. Вызывается ли код компаратора из метода Arrays.sort() в том же самом потоке исполнения, где вызывается и метод sort(), или же в другом потоке?
2. Используя методы listFiles(FileFilter) и isDirectory() из класса java.io.File, напишите метод, возвращающий все подкаталоги из данного каталога. Воспользуйтесь лямбда-выражением вместо объекта типа FileFilter. Сделайте то же самое с помощью ссылки на метод.
3. Используя метод listFiles(FileFilter) из класса java.io.File, напишите метод, возвращающий все файлы с указанным расширением из заданного каталога. Воспользуйтесь для этой цели лямбда-выражением вместо класса FilenameFilter. Какие переменные из объемлющей области действия фиксируются этим лямбда-выражением?
4. Отсортируйте заданный массив объектов типа File таким образом, чтобы каталоги в нем следовали перед файлами, а в каждой группе элементы были отсортированы по имени пути. Воспользуйтесь для этой цели лямбда-выражением вместо интерфейса Comparator.
5. Возьмите из какого-нибудь своего проекта файл, содержащий ряд экземпляров интерфейсов ActionListener, Runnable и т.п. Замените их лямбда-выражениями. Сколько строк кода вам удастся при этом сэкономить? Станет ли исходный код более удобочитаемым в конечном итоге? Удастся ли вам воспользоваться ссылками на методы?
6. Вам, вероятно, всегда претило обрабатывать проверяемые исключения, возникающие в интерфейсе Runnable. Поэтому напишите метод uncheck(), перехватывающий все проверяемые исключения и превращающий их в непроверяемые, как показано в приведенном ниже примере кода.

```
new Thread(uncheck(
    () -> { System.out.println("Zzz"); Thread.sleep(1000); })).start();
// обратите внимание на отсутствие перехвата
// исключения (InterruptedException)!
```

7. Подсказка: определите сначала интерфейс RunnableEx, в методе run() которого могут быть генерированы исключения. Затем реализуйте метод public static Runnable uncheck(RunnableEx runner). Воспользуйтесь лямбда-выражением в методе uncheck().
8. Почему для этой цели нельзя воспользоваться интерфейсом Callable<Void> вместо интерфейса RunnableEx?
9. Напишите статический метод andThen(), принимающий в качестве параметров два экземпляра интерфейса Runnable и возвращающий экземпляр

интерфейса `Runnable`, который выполняет сначала первый экземпляр, а затем второй. Передайте в методе `main()` два лямбда-выражения вызываемому методу `andThen()`, а затем выполните возвращаемый из него экземпляр.

10. Что происходит, когда лямбда-выражение фиксирует значения в расширенном цикле `for`, аналогичном приведенному ниже?

```
String[] names = { "Peter", "Paul", "Mary" };
List<Runnable> runners = new ArrayList<>();
for (String name : names)
    runners.add(() -> System.out.println(name));
```

11. Насколько это допустимо? Фиксирует ли каждое лямбда-выражение другое значение, или же все они получают последнее значение? Что произойдет, если воспользоваться традиционным циклом `for (int i = 0; i < names.length; i++)`?
12. Сформируйте подкласс `Collection2` из интерфейса `Collection` и введите в него метод по умолчанию `void forEachIf(Consumer<T> action, Predicate<T> filter)`, применявший параметр `action` к каждому элементу, для которого параметр `filter` возвращает логическое значение `true`. Как можно воспользоваться таким методом?
13. Просмотрите методы из класса `Collections`. В каком из интерфейсов вы разместили бы каждый из этих методов? Был бы это метод по умолчанию или статический метод?
14. Допустим, имеется класс, реализующий два интерфейса, `I` и `J`, в каждом из которых присутствует метод `void f()`. Что произойдет, если метод `f()` окажется абстрактным, статическим или методом по умолчанию в интерфейсе `J`? И что произойдет, если создать класс, расширяющий суперкласс `S` и реализующий интерфейс `I`, причем в каждом из них присутствует метод `f()`?
15. В прошлом считалось неудачным решением вводить методы в интерфейс, поскольку это могло бы нарушить уже существующий код. А теперь считается, что вводить новые методы в интерфейс вполне допустимо, при условии, что предоставляется также их реализация по умолчанию. Насколько это надежно? Опишите ситуацию, когда новый метод `stream()` из интерфейса `Collection` становится причиной того, что устаревший код не удается скомпилировать. Что можно сказать о совместимости на уровне двоичных кодов? Будет ли по-прежнему выполняться устаревший код из архивного JAR-файла?

Глава

2

Прикладной программный интерфейс API потоков ввода-вывода

В этой главе...

- От итерации к операциям с потоками ввода-вывода
- Создание потока ввода-вывода
- Методы `filter()`, `map()` и `flatMap()`
- Извлечение подпотоков и объединение потоков ввода-вывода
- Преобразования с сохранением состояния
- Простые операции сведения
- Тип данных `Optional`
- Операции сведения
- Накопление результатов
- Накопление данных в отображениях
- Группирование и разделение
- Потоки ввода-вывода примитивных типов
- Параллельные потоки ввода-вывода
- Функциональные интерфейсы
- Упражнения

Потоки ввода-вывода служат главной абстракцией в версии Java 8 для обработки коллекций значений и указания требуемых действий с предоставлением конкретной реализации права планировать операции. Так, если требуется рассчитать среднее значение в некотором методе, то следует указать, что нужно вызвать метод для каждого элемента и получить среднее значение. Распараллеливание данной операции предоставляется библиотеке потоков ввода-вывода, где для расчета сумм, подсчета каждой части и объединения результатов организуется несколько потоков.

В этой главе рассматриваются следующие основные вопросы.

- Итераторы, подразумевающие конкретную методику обхода коллекций и предпятствующие эффективной организации параллельного выполнения операций.
- Создание потоков ввода-вывода из коллекций, массивов, генераторов или итераторов.
- Выбор и преобразование элементов коллекции с помощью методов `filter()` и `map()` соответственно.
- Другие операции для преобразования потоков ввода-вывода с помощью методов `limit()`, `distinct()` и `sorted()`.
- Получение результата из потока ввода-вывода с помощью операций сведения, выполняемых с помощью метода `count()`, `max()`, `min()`, `findFirst()` или `findAny()`. Некоторые из этих методов возвращают значение типа `Optional`.
- Тип данных `Optional`, служащий в качестве альтернативы обращению с пустыми значениями `null`. Для надежного его применения служат методы `ifPresent()` и `orElse()`.
- Накопление результатов из потоков ввода-вывода в коллекциях, массивах, символьных строках или отображениях.
- Разбиение содержимого потока на группы и получение результата для каждой группы с помощью методов `groupingBy()` и `partitioningBy()` из класса `Collectors`.
- Специализированные потоки для ввода-вывода данных примитивных типов `int`, `long` и `double`.
- Исключение побочных эффектов и наложение ограничений на упорядочение данных при обращении с параллельными потоками ввода-вывода.
- Освоение небольшого количества функциональных интерфейсов для пользования библиотекой потоков ввода-вывода.

От итерации к операциям с потоками ввода-вывода

При обработке коллекции обычно выполняется итерация (т.е. обход) ее элементов и некоторые операции над каждым из них. Допустим, требуется подсчитать все длинные слова в книге. Сначала их нужно разместить в списке, как показано ниже.

```
String contents = new String(Files.readAllBytes(  
    Paths.get("alice.txt")), StandardCharsets.UTF_8);  
    // чтение данных из файла в символьную строку  
List<String> words = Arrays.asList(contents.split("[\\P{L}]+"));  
    // разбить на слова, небуквенные символы считаются разделителями
```

А теперь можно выполнить итерацию:

```
int count = 0;
for (String w : words) {
    if (w.length() > 12) count++;
}
```

Что же здесь не так? Все так, кроме того, что такой код трудно распараллелить. Именно для этого в версии Java 8 были внедрены групповые операции. Аналогичная операция в Java 8 выглядит следующим образом:

```
long count = words.stream().filter(w -> w.length() > 12).count();
```

Метод `stream()` предоставляет поток *ввода-вывода* для списка слов. А метод `filter()` возвращает другой поток ввода-вывода, содержащий только слова длиной больше двенадцати букв. И наконец, метод `count()` сводит этот поток ввода-вывода к нужному результату.

Внешние потоки ввода-вывода очень похожи на коллекции, позволяя преобразовывать и извлекать данные. Но у потоков ввода-вывода и коллекций имеются и следующие существенные отличия.

1. В потоке ввода-вывода не хранятся его элементы. Они могут храниться в базовой коллекции или формироваться по требованию.
2. Операции с потоками ввода-вывода не модифицируют их источник. Вместо этого они возвращают новые потоки, содержащие результат.
3. Операции с потоками ввода-вывода выполняются *по требованию*, когда это возможно. Это означает, что они не выполняются до тех пор, пока не потребуется их результат. Так, если требуется подсчитать лишь первые пять, а не все длинные слова, фильтрация в методе `filter()` прекратится после пятого совпадения. Следовательно, можно организовать даже бесконечные потоки ввода-вывода!

В этой главе рассматривается все, что касается потоков ввода-вывода. Многие считают, что потоковые выражения более удобочитаемы, чем их циклические эквиваленты. Более того, их нетрудно распараллелить. Ниже показано, как организовать параллельный подсчет длинных слов. Достаточно заменить `stream` на `parallelStream` при обращении к потоку ввода-вывода, чтобы библиотека организовала фильтрацию и подсчет длинных слов в параллельном режиме.

```
long count = words.parallelStream().filter(w -> w.length() > 12).count();
```

Потоки ввода-вывода действуют по принципу “что, а не как делать”. В рассматриваемом здесь примере потока ввода-вывода описывается, что нужно сделать, а именно: получить длинные слова и подсчитать их. При этом не указывается, в каком порядке или в каком потоке исполнения это должно произойти. С другой стороны, в цикле, приведенном в начале этого раздела, точно указывается, каким образом должно происходить вычисление, а следовательно, исключается любая возможность для оптимизации.

Обращаясь с потоками ввода-вывода, можно организовать конвейер операций в три этапа.

1. Создание потока ввода-вывода.
2. Указание промежуточных операций для преобразования исходного потока ввода-вывода в другие потоки в один или больше этапов.
3. Выполнение окончной операции для получения результата. Такая операция принуждает к выполнению операций по требованию, которые ей предшествуют. А впоследствии поток ввода-вывода может больше не потребоваться.

В рассматриваемом здесь примере поток ввода-вывода был создан с помощью метода `stream()` или `parallelStream()`. Метод `filter()` преобразовал поток, а в методе `count()` была выполнена окончная операция.



НА ЗАМЕТКУ. Потоковые операции не выполняются над элементами в том порядке, в каком они вызываются в потоках ввода-вывода. В данном примере ничего не происходит до тех пор, пока не вызывается метод `count()`. Когда же в методе `count()` запрашивается первый элемент, в методе `filter()` начинается опрашивание элементов до тех пор, пока не будет найден элемент длиной более 12 букв.

В следующем разделе будет показано, как создается поток ввода-вывода. Три последующие раздела посвящены преобразованиям в потоках ввода-вывода, а пять следующих далее разделов — окончным операциям.

Создание потока ввода-вывода

Как было показано ранее, любую коллекцию можно преобразовать в поток ввода-вывода с помощью метода `stream()`, который был внедрен в интерфейс `Collection` в версии Java 8. Если же имеется массив, то следует применять статический метод `Stream.of()`.

```
Stream<String> words = Stream.of(contents.split("\\P{L}]+"));
// метод split() возвращает массив String[]
```

Метод `of()` имеет переменное число аргументов, и поэтому поток ввода-вывода можно построить из любого количества аргументов следующим образом:

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Для того чтобы создать поток ввода-вывода из части массива, следует вызвать метод `Arrays.stream(array, from, to)`. А для того чтобы создать поток ввода-вывода без элементов, следует вызвать метод `Stream.empty()`, как показано ниже.

```
Stream<String> silence = Stream.empty();
// выводится обобщенный тип <b>String</b>,
// аналогично вызову Stream.<String>empty()
```

В интерфейсе `Stream` имеются два статических метода для создания бесконечных потоков ввода-вывода. Метод `generate()` принимает функцию без аргументов (или формально объект интерфейса `Supplier<T>`; см. далее раздел “Функциональные интерфейсы”). Всякий раз, когда требуется значение из потока ввода-вывода, эта функция вызывается для выдачи значения. Поток ввода-вывода значений констант можно получить следующим образом:

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

а поток ввода-вывода случайных чисел — так, как показано ниже.

```
Stream<Double> randoms = Stream.generate(Math::random);
```

Вместо этого для формирования бесконечных последовательностей вроде 0 1 2 3 ... применяется метод `iterate()`, принимающий начальное случайное значение и функцию (формально `UnaryOperator<T>`), повторно применяя ее к предыдущему результату. Например, в приведенном ниже фрагменте кода первым элементом последовательности оказывается начальное случайное значение `BigInteger.ZERO`. Второй элемент `f(seed)` равен 1 (т.е. имеет большое целое значение), следующий элемент `f(f(seed))` равен 2 и т.д.

```
Stream<BigInteger> integers =
    Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```



НА ЗАМЕТКУ. В прикладной программный интерфейс API версии Java 8 был внедрен целый ряд методов, формирующих потоки ввода-вывода. Например, в классе `Pattern` теперь имеется метод `splitAsStream()`, разбивающий последовательность типа `CharSequence` по регулярному выражению. Так, для разбиения символьной строки на отдельные слова можно воспользоваться следующим оператором:

```
Stream<String> words =
    Pattern.compile("[\\P{L}]+").splitAsStream(contents);
```

Статический метод `Files.lines()` возвращает поток ввода-вывода типа `Stream` всех строк в файле. У интерфейса `Stream` имеется суперинтерфейс `AutoCloseable`. При вызове метода `close()` в потоке ввода-вывода закрывается также базовый файл. Для того чтобы это действительно произошло, лучше воспользоваться оператором `try` с ресурсами, внедренным в версии Java 7, как показано ниже.

```
try (Stream<String> lines = Files.lines(path)) {
    // сделать что-нибудь с методом lines()
}
```

Поток ввода-вывода и базовый файл в нем будут закрыты при выходе из блока `try` нормальным способом или через исключение.

Методы filter(), map() и flatMap()

При преобразовании потоков ввода-вывода данные вводятся из одного потока и после преобразования выводятся в другой поток. Как было показано ранее, преобразование в методе `filter()` образует новый поток ввода-вывода со всеми элементами, совпадающими с определенным условием. В приведенном ниже примере один поток ввода-вывода символьных строк преобразуется в другой поток, содержащий только длинные слова. В качестве аргумента метода `filter()` служит функция `Predicate<T>`, принимающая тип от `T` до `boolean`.

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
Stream<String> longWords = words.filter(w -> w.length() > 12);
```

Зачастую значения в потоке ввода-вывода требуется каким-то образом преобразовать. Для этого следует вызвать метод `map`, передав ему функцию, выполняющую преобразование. Например, все слова могут быть преобразованы в строчные буквы следующим образом:

```
Stream<String> lowercaseWords = words.map(String::toLowerCase);
```

В данном случае метод `map()` вызывается по ссылке на метод. Зачастую вместо этого используется лямбда-выражение, как показано ниже. Результирующий поток ввода-вывода содержит первый символ каждого слова.

```
Stream<Character> firstChars = words.map(s -> s.charAt(0));
```

Если вызывается метод `map()`, функция применяется к каждому элементу и возвращает значения, накопленные в новом потоке ввода-вывода. Допустим, имеется функция, возвращающая не только одно значение, но и поток ввода-вывода значений:

```
public static Stream<Character> characterStream(String s) {
    List<Character> result = new ArrayList<>();
    for (char c : s.toCharArray()) result.add(c);
    return result.stream();
}
```

Например, метод `characterStream("boat")` формирует поток ввода-вывода символов `['b', 'o', 'a', 't']`. Допустим, что этот метод отображается в поток ввода-вывода символьных строк следующим образом:

```
Stream<Stream<Character>> result = words.map(w -> characterStream(w));
```

В итоге будет получен поток потоков ввода-вывода, аналогичный следующему: `[... ['u', 'o', 'u', 'r'], ['b', 'o', 'a', 't'], ...]`. Для того чтобы свести его в поток символов `[... 'u', 'o', 'u', 'r', 'b', 'o', 'a', 't', ...]`, вместо метода `map()` вызывается метод `flatMap`:

```
Stream<Character> letters = words.flatMap(w -> characterStream(w))
// вызывает метод characterStream() для каждого слова и сводит результаты
```



НА ЗАМЕТКУ. Метод `flatMap()` можно обнаружить и в других классах, кроме тех, что реализуют потоки ввода-вывода. Это общий принцип программирования. Допустим, имеются обобщенный тип `G` (например, `Stream`) и функция `f()`, принимающая тип от `T` до `G<U>`, а также функция `g()`, принимающая тип от `U` до `G<V>`. Это главное понятие в теории монад. Тем не менее метод `flatMap()` можно применять, ничего не зная о монадах.

Извлечение подпотоков и объединение потоков ввода-вывода

В результате вызова метода `stream.limit(n)` возвращается новый поток ввода-вывода, который завершается после `n` элементов, а иначе завершается исходный поток ввода-вывода, если он короче. Этот метод особенно удобен для ограничения бесконечных потоков ввода-вывода до определенной величины. Например, в следующей строке кода получается поток ввода-вывода, ограниченный 100 случайными числами:

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

Вызов метода `stream.skip(n)` приводит к совершенно противоположному результату: отбрасыванию первых *n* элементов. Этот метод подходит для рассмотренного выше примера, где подсчитываются длинные слова в книге и где первый элемент оказывается ненужной пустой символьной строкой в силу особенностей работы метода `split()`. От этой пустой строки можно избавиться, вызвав метод `skip()` следующим образом:

```
Stream<String> words = Stream.of(contents.split("[\\P{L}]+")).skip(1);
```

С помощью статического метода `concat()` из класса `Stream` можно соединить два потока ввода-вывода, как показано ниже. Разумеется, первый поток ввода-вывода не должен быть бесконечным, иначе дело вообще не дойдет до второго потока ввода-вывода.

```
Stream<Character> combined = Stream.concat(
    characterStream("Hello"), characterStream("World"));
// производит поток ввода-вывода
// ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
```



СОВЕТ. Метод `peek()` производит еще один поток ввода-вывода с такими же самыми элементами, как и у исходного потока, но в данном случае функция вызывается всякий раз, когда извлекается элемент. Это удобно для отладки, как показано ниже.

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

При доступе к отдельному элементу выводится сообщение. Подобным образом можно проверить, обрабатывается ли по требованию бесконечный поток, возвращаемый методом `iterate()`.

Преобразования с сохранением состояния

Преобразования потоков ввода-вывода, рассматривавшиеся в предыдущих разделах этой главы, выполнялись без *сохранения состояния*. Это означает, что когда элемент извлекается из отфильтрованного или отображенного потока ввода-вывода, результат не зависит от предыдущих элементов. Преобразования могут выполняться и с *сохранением состояния*. В приведенном ниже примере метод `distinct()` возвращает поток ввода-вывода, извлекающий элементы из исходного потока ввода-вывода в том же самом порядке, только подавляя дубликаты. Очевидно, что в потоке ввода-вывода должны запоминаться элементы, которые уже просматривались в нем.

```
Stream<String> uniqueWords
= Stream.of("merrily", "merrily", "merrily", "gently").distinct();
// сохраняется только одно слово "merrily"
```

Метод `sorted()` должен проверить весь поток ввода-вывода и отсортировать его, прежде чем выдать какие-нибудь элементы, причем наименьший элемент может стать последним. Очевидно, что отсортировать бесконечный поток ввода-вывода нельзя. Имеется также несколько методов `sorted()`. Один из них подходит для потоков ввода-вывода элементов типа `Comparable`, а другой принимает объект типа `Comparator`. В приведенном ниже примере символьные строки сортируются таким образом, чтобы первыми следовали самые длинные из них.

```
Stream<String> longestFirst =  
    words.sorted(Comparator.comparing(String::length).reversed());
```

Разумеется, коллекцию можно отсортировать и без потоков ввода-вывода. Метод `sorted()` оказывается удобным в том случае, когда процесс сортировки становится частью конвейера потоков ввода-вывода.



НА ЗАМЕТКУ. Метод `Collections.sort()` сортирует коллекцию на месте, тогда как метод `Stream.sorted()` возвращает новый отсортированный поток.

Простые операции сведения

А теперь, когда было показано, как создавать и преобразовывать потоки ввода-вывода, перейдем к рассмотрению самого важного вопроса: получение результатов из потоковых данных. Методы, рассматриваемые в этом разделе, называются *операциями сведения*. Они сводят поток ввода-вывода к значению, которое может быть в программе. Операции сведения называются *окончными*. После выполнения окончной операции использование потока ввода-вывода прекращается. Выше уже рассматривался простой пример сведения: метод `count()`, возвращающий количество элементов в потоке ввода-вывода.

К числу других простых операций сведения относятся методы `max()` и `min()`, возвращающие наибольшее или наименьшее значение. Но дело в том, что эти методы возвращают значение типа `Optional<T>`, заключающее в оболочку результат или обозначающее его отсутствие, поскольку поток ввода-вывода оказывается пустым. В прошлом в подобных ситуациях обычно возвращалось пустое значение `null`. Но это могло привести к появлению исключений в связи с пустым указателем, когда в непротестированной полностью программе возникала необычная ситуация. В качестве более предпочтительного способа обозначения отсутствующего возвращаемого значения в версии Java 8 внедрен тип `Optional`. Более подробно тип `Optional` рассматривается в следующем разделе, а ниже показано, как получить максимальное значение из потока ввода-вывода.

```
Optional<String> largest = words.max(String::compareToIgnoreCase);  
if (largest.isPresent())  
    System.out.println("largest: " + largest.get());
```

Метод `findFirst()` возвращает первое значение из непустой коллекции. Зачастую он нередко используется вместе с методом `filter()`. Например, в следующем примере обнаруживается первое слово, начинающееся с буквы Q, если таковая имеется:

```
Optional<String> startsWithQ  
= words.filter(s -> s.startsWith("Q")).findFirst();
```

Если устраивает совпадение с любым, а не только с первым словом, то вызывается метод `findAny()`. Это оказывается эффективным при распараллеливании потока ввода-вывода, поскольку при первом совпадении в любой из проверяемых его частей вычисление завершается.

```
Optional<String> startsWithQ  
= words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

Если же требуется лишь выяснить наличие совпадения, то следует вызвать метод `anyMatch()`. Этот метод принимает предикатный аргумент, а следовательно, вызывать метод `filter()` не требуется.

```
boolean aWordStartsWithQ
    = words.parallel().anyMatch(s -> s.startsWith("Q"));
```

Имеются также методы `allMatch()` и `noneMatch()`, возвращающие логическое значение `true`, если с предикатом совпадают все элементы или не совпадает ни один из них. В этих методах всегда проверяется весь поток ввода-вывода, но они извлекают выгоду из параллельного выполнения.

Тип данных Optional

Объект типа `Optional<T>` является оболочкой для объекта типа `T` или же указывает на отсутствие объекта. Он предназначен в качестве более надежной альтернативы, чем ссылка типа `T`, которая делается на объект или пустое значение `null`. Но такой способ оказывается более надежным, если он применяется правильно. В частности, метод `get()` получает заключенный в оболочку элемент, если таковой имеется, или же генерирует исключение типа `NoSuchElementException`, если он отсутствует. Следовательно, способ

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod()
```

оказывается не более надежным, чем такой способ:

```
T value = ...;
value.someMethod();
```

Как было показано в предыдущем разделе, метод `isPresent()` сообщает, содержит ли значение объект типа `Optional<T>`. Но следующий код:

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

оказывается не проще, чем приведенный ниже код. В следующем разделе будет показано, каким образом следует обращаться со значениями типа `Optional`.

```
if (value != null) value.someMethod();
```

Обращение со значениями типа Optional

Самое главное для эффективного применения типа `Optional` — вызвать метод, который *потребляет* правильное значение или *поставляет* альтернативное значение. Для этой цели служит вторая форма метода `ifPresent()`, принимающего функцию. Если значение типа `Optional` существует, оно передается данной функции. В противном случае ничего не происходит. Вместо применения условного оператора `if` достаточно сделать следующий вызов:

```
optionalValue.ifPresent(v -> обработать значение v);
```

Так, если требуется ввести значение во множество, при условии, что оно существует, то достаточно сделать вызов

```
optionalValue.ifPresent(v -> results.add(v));
```

или же вызов

```
optionalValue.ifPresent(results::add);
```

При вызове данной версии метода `ifPresent()` никакого значения не возвращается. Если же требуется обработать результат, то вместо этого лучше воспользоваться методом `map()`, как показано ниже.

```
Optional<Boolean> added = optionalValue.map(results::add);
```

Таким образом, введено одно из следующих трех значений: логическое значение `true` или `false`, заключенное в оболочку типа `Optional`, если присутствовало значение типа `optionalValue`, а иначе — пустое значение типа `Optional`.



НА ЗАМЕТКУ. Данный метод `map()` является аналогом метода `map()` из интерфейса `Stream`, рассмотренного выше, в разделе “Методы `filter()`, `map()` и `flatMap()`”. Значение типа `Optional` достаточно представить в виде потока ввода-вывода нулевой или единичной величины. Результат также оказывается нулевой или единичной величиной, а в последнем случае применяется функция.

Выше было показано, насколько изящно можно потреблять значение типа `Optional`, когда оно присутствует. Другая методика обращения со значениями типа `Optional` состоит в том, чтобы поставлять альтернативное значение, если конкретное значение отсутствует. Зачастую в отсутствие совпадения приходится выбирать значение по умолчанию, возможно, пустую символьную строку, как показано ниже.

```
String result = optionalString.orElse("");  
// заключенная в оболочку символьная строка,  
// а в ее отсутствие — пустая строка ""
```

Кроме того, можно вызвать код для расчета значения по умолчанию следующим образом:

```
String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));  
// функция вызывается только по мере необходимости
```

А в отсутствие значения можно сгенерировать другое исключение, как показано ниже.

```
String result = optionalString.orElseThrow(NoSuchElementException::new);  
// предоставить метод, выдающий объект исключения
```

Формирование значений типа `Optional`

До сих пор обсуждалось потребление объекта типа `Optional`, созданного кем-то другим. Если же требуется написать метод, создающий объект типа `Optional`, то для этой цели служит несколько статических методов. В частности, можно создать метод `Optional.of(result)` или `Optional.empty()`, как показано в приведенном ниже примере.

```
public static Optional<Double> inverse(Double x) {
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

Метод `ofNullable()` предназначен в качестве своего рода моста для перехода от пустых значений типа `null` к значениям типа `Optional`. Так, при вызове метода `Optional.ofNullable(obj)` возвращается ссылка на метод `Optional.of(obj)`, если значение `obj` не является пустым (т.е. `null`), а иначе — ссылка на метод `Optional.empty()`.

Составление функций дополнительных значений методом `flatMap()`

Допустим, имеется метод `f()`, выдающий значение типа `Optional<T>`, а у целевого типа `T` — метод `g()`, выдающий значение типа `Optional<U>`. Если бы это были обычные методы, их можно было бы составить в цепочку вызовов `s.f().g()`. Но в данном случае такое составление не подходит, поскольку вызов метода `s.f()` относится к типу `Optional<T>`, тогда как вызов метода `g()` к данному типу не относится. Вместо этого нужно сделать следующий вызов:

```
Optional<U> = s.f().flatMap(T::g);
```

Если вызов метода `s.f()` присутствует, то вызывается метод `g()`. В противном случае возвращается значение типа `Optional<U>`.

Очевидно, что данный процесс можно повторить, если имеются дополнительные методы или лямбда-выражения, выдающие значения типа `Optional`. В итоге, соединяя вызовы в цепочку вплоть до метода `flatMap()`, можно построить конвейер из отдельных шагов, которые оказываются успешными не частично, а в целом.

Рассмотрим в качестве примера безопасный вариант метода `inverse()`, представленного в предыдущем разделе. Допустим также, что имеется безопасный метод `squareRoot()` для вычисления квадратного корня, как показано ниже.

```
public static Optional<Double> squareRoot(Double x) {
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

В таком случае можно вычислить квадратный корень обратного значения следующим образом:

```
Double result = inverse(x).flatMap(MyMath::squareRoot);
```

или же более предпочтительным способом:

```
Double result =
    Optional.of(-4.0).flatMap(Test::inverse).flatMap(Test::squareRoot);
```

Если метод `inverse()` или `squareRoot()` возвратит ссылку на метод `Optional.empty()`, то результат окажется пустым.



НА ЗАМЕТКУ. Метод `flatMap()` уже демонстрировался в интерфейсе `Stream` (см. раздел “Методы `filter()`, `map()` и `flatMap()`”). В частности, этот метод применялся для составления двух других методов, выдающих потоки ввода-вывода сведением в поток потоков. Метод `Optional.flatMap()` действует таким же образом, если рассматривать значение типа `Optional` в качестве потока ввода-вывода нулевой или единичной величины.

Операции сведения

Если требуется вычислить сумму или объединить элементы из потока ввода-вывода в результат иным способом, то для этой цели можно воспользоваться одним из методов сведения. В простейшем случае такой метод принимает двоичную функцию и применяет ее, начиная с двух первых элементов. Это нетрудно пояснить, если функция вычисляет сумму, как показано ниже.

```
Stream<Integer> values = ...;
Optional<Integer> sum = values.reduce((x, y) -> x + y)
```

В данном случае метод `reduce()` вычисляет сумму $v_0 + v_1 + v_2 + \dots$, где v_i — элементы потока ввода-вывода. Этот метод возвращает значение типа `Optional`, поскольку достоверный результат отсутствует, если поток ввода-вывода пуст.



НА ЗАМЕТКУ. В данном случае можно сделать вызов `values.reduce(Integer::sum)` вместо вызова `values.reduce((x, y) -> x + y)`.

В общем, если в методе `reduce()` присутствует операция сведения *op*, то она сводится к следующему: $v_0 \text{ op } v_1 \text{ op } v_2 \text{ op } \dots$, где $v_i \text{ op } v_{i+1}$ записывается для вызова функции $\text{op}(v_i, v_{i+1})$. Операция сведения должна быть *ассоциативной*, т.е. совершенно не важно, в каком именно порядке объединяются элементы. В математической записи понятие ассоциативной операции выражается следующим образом: $(x \text{ op } y) \text{ op } z = x \text{ op } (y \text{ op } z)$. И благодаря этому становится возможным эффективное сведение в параллельных потоках ввода-вывода.

На практике могут оказаться полезными многие ассоциативные операции, например, сложение и умножение, сцепление символьных строк, получение максимального и минимального значений, объединение и пересечение множеств. Примером операции, не являющейся ассоциативной, служит вычитание: $(6 - 3) - 2 \neq 6 - (3 - 2)$.

Нередко имеется *тождество* *e*, т.е. *e op x = x*, и такой элемент может быть использован для начала вычисления. Например, элемент **0** является тождеством сложения. И тогда вызывается вторая форма метода `reduce()`, как показано ниже. Значение тождества возвращается в том случае, если поток ввода-вывода пуст, и больше не нужно обращаться к классу `Optional`.

```
Stream<Integer> values = ...;
Integer sum = values.reduce(0, (x, y) -> x + y)
// вычисляет сумму 0 + v0 + v1 + v2 + ...
```

А теперь допустим, что имеется поток ввода-вывода и требуется сформировать сумму из некоторых свойств, например, сумму всех длин символьных строк в потоке ввода-вывода. Простая форма метода `reduce()` в данном случае не годится. Для этого требуется функция $(T, T) \rightarrow T$ с одинаковыми типами аргументов и результата. Но в данном случае имеются два типа. Элементы из потока ввода-вывода относятся к типу `String`, а накапливаемый результат — к целочисленному типу. Для подобного случая имеется отдельная форма метода `reduce()`.

Сначала предоставляется функция накопления $(total, word) \rightarrow total + word.length()$. Эта функция вызывается неоднократно, образуя накапливаемую итоговую сумму. Но если вычисление распараллеливается, то таких вычислений может

оказаться несколько, и поэтому придется объединить их результаты. Для этой цели предоставляется вторая функция, а полностью вызов выглядит следующим образом:

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```



НА ЗАМЕТКУ. На практике методом `reduce()`, вероятнее всего, придется пользоваться неоднократно. Поэтому, как правило, проще выполнить преобразование в поток ввода-вывода чисел и воспользоваться одним из его методов для вычисления суммы, максимального или минимального значений (подробнее о потоках ввода-вывода чисел см. в разделе "Потоки ввода-вывода примитивных типов" далее в этой главе). А в данном конкретном примере проще и эффективнее было бы вызвать метод `words.mapToInt(String::length).sum()`, поскольку в этом случае не требуется упаковка.

Накопление результатов

По завершении работы с потоком ввода-вывода нередко требуется лишь просмотреть результаты, а не сводить их к некоторому значению. С этой целью можно вызвать метод `iterator()`, выдающий устаревший итератор для обхода элементов. А с другой стороны, можно вызвать метод `toArray()` и получить массив из элементов потока ввода-вывода.

Создать обобщенный массив во время выполнения нельзя, и поэтому в результате вызова метода `stream.toArray()` возвращается массив типа `Object[]`. Если же требуется получить массив подходящего типа, то следует передать конструктор массива, как показано ниже.

```
String[] result = words.toArray(String[]::new);
// метод words.toArray() возвращает массив типа Object[]
```

А теперь допустим, что требуется накопить результаты в хеш-множестве типа `HashSet`. Если коллекция распараллелена, то элементы нельзя ввести непосредственно в одно хеш-множество типа `HashSet`, поскольку объект типа `HashSet` не является потокобезопасным. Именно по этой причине нельзя воспользоваться методом `reduce()`. Каждая часть должна начинаться с ее собственного пустого хеш-множества, а метод `reduce()` предоставляет только одно значение тождества. Вместо этого следует воспользоваться методом `collect()`, который принимает три аргумента, выполняющих следующие функции.

1. *Поставщик*, предназначенный для получения новых экземпляров целевого объекта, например конструктор хеш-множества.
2. *Накопитель*, добавляющий новый элемент к целевому, например метод `add()`.
3. *Объединитель*, соединяющий два объекта в один, например метод `addAll()`.



НА ЗАМЕТКУ. Целевой объект совсем не обязательно должен быть коллекцией. Им может быть объект типа `StringBuilder` или же объект, отслеживающий подсчет или суммирование.

Ниже показано, каким образом метод `collect()` обращается к хеш-множеству.

```
HashSet<String> result = stream.collect(HashSet::new, HashSet::add,  
HashSet::addAll);
```

На практике делать это совсем не обязательно, поскольку для реализации трех перечисленных выше функций имеется удобный интерфейс `Collector`, а также класс `Collectors` с фабричными методами для общих накопителей. В частности, для накопления потока ввода-вывода в список или множество достаточно сделать один из следующих вызовов:

```
List<String> result = stream.collect(Collectors.toList());
```

или

```
Set<String> result = stream.collect(Collectors.toSet());
```

Если же требуется контролировать вид получаемого множества, в таком случае можно сделать такой вызов:

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

Допустим, требуется накопить все символьные строки в потоке ввода-вывода путем их сцепления. С этой целью можно сделать следующий вызов:

```
String result = stream.collect(Collectors.joining());
```

Если требуется ограничитель между элементами, его можно передать методу `joining()` следующим образом:

```
String result = stream.collect(Collectors.joining(", "));
```

Если же поток ввода-вывода содержит объекты, кроме символьных строк, необходимо сначала преобразовать их в символьные строки так, как показано ниже.

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

А если требуется свести результаты из потока ввода-вывода в сумму, среднее, максимальное или минимальное значение, то следует вызвать один из методов `summarizing(Int|Long|Double)`. Такие методы принимают функцию, преобразующую потоковые объекты в числа и выдающую результат типа `(Int|Long|Double) SummaryStatistics` с помощью методов, предназначенных для получения суммы, среднего, максимального и минимального значения.

```
IntSummaryStatistics summary = words.collect(  
    Collectors.summarizingInt(String::length));  
double averageWordLength = summary.getAverage();  
double maxWordLength = summary.getMax()
```



НА ЗАМЕТКУ. До сих пор было показано, каким образом сводятся или накапливаются значения из потока ввода-вывода. Но их, возможно, требуется лишь вывести на печать или экран или же ввести в базу данных. В таком случае можно воспользоваться методом `forEach()` следующим образом:

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

Передаваемая функция применяется к каждому элементу. В параллельном потоке ввода-вывода ответственность за параллельное выполнение функции возлагается на программиста. Более подробно этот вопрос обсуждается в разделе "Параллельные потоки ввода-вывода" далее в этой главе.

В параллельном потоке ввода-вывода обход элементов может быть произведен в произвольном порядке. Если же требуется обойти их в потоковом порядке, то следует вызвать метод `forEachOrdered()`. Разумеется, для этого, возможно, придется отказаться от большинства или всех выгод параллелизма.

Методы `forEach()` и `forEachOrdered()` относятся к окончательным операциям. Но после их вызова воспользоваться снова потоком ввода-вывода уже нельзя. Если же требуется и далее пользоваться потоком ввода-вывода, в таком случае следует вызвать метод `peek()` [подробнее об этом см. разделе "Извлечение подпотоков и объединение потоков ввода-вывода" ранее в этой главе].

Накопление данных в отображениях

Допустим, имеется поток ввода-вывода типа `Stream<Person>` и требуется накопить элементы в отображении, чтобы в дальнейшем искать отдельных людей по их идентификационному номеру. У метода `Collectors.toMap()` имеются в качестве аргументов две функции, производящие ключи и значения для конкретного отображения, как показано в приведенном ниже примере.

```
Map<Integer, String> idToName = people.collect(  
    Collectors.toMap(Person::getId, Person::getName));
```

В общем случае, когда значения должны быть конкретными элементами, в качестве второй функции применяется метод `Function.identity()`:

```
Map<Integer, Person> idToPerson = people.collect(  
    Collectors.toMap(Person::getId, Function.identity()));
```

Если обнаружится больше одного элемента с одинаковым ключом, накопитель генерирует исключение типа `IllegalStateException`. Такое поведение можно переопределить, предоставив в качестве аргумента третью функцию, определяющую значение по ключу, задаваемому для существующего или нового значения. Эта функция должна возвращать существующее значение, новое значение или их сочетание.

В приведенном ниже примере конструируется отображение, содержащее региональные настройки для каждого языка. В качестве ключа указывается исходное название языка в региональных настройках по умолчанию (например, "German" — «Немецкий»), а в качестве значения — локализованное название языка (в данном случае — "Deutsch"). При этом во внимание не принимается, что один и тот же язык может встретиться дважды (например, немецкий в Германии и Швейцарии), и поэтому оставляется только первое его вхождение.

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());  
Map<String, String> languageNames = locales.collect(  
    Collectors.toMap(  
        l -> l.getDisplayLanguage(),  
        l -> l.getDisplayLanguage(l),  
        (existingValue, newValue) -> existingValue));
```

Но допустим, что требуется выяснить все языки, официально употребляемые в данной стране. Для этого придется воспользоваться отображением типа `Map<String, Set<String>>`. Например, для страны "Switzerland" (Швейцария) в качестве языка задано значение [French, German, Italian] (французский, немецкий, итальянский). Сначала для каждого языка сохраняется одиночное множество, а при обнаружении нового языка для заданной страны составляется объединение из существующего и нового множеств, как показано ниже.

```
Map<String, Set<String>> countryLanguageSets = locales.collect(  
    Collectors.toMap(  
        l -> l.getDisplayCountry(),  
        l -> Collections.singleton(l.getDisplayLanguage()),  
        (a, b) -> { // объединение множеств a и b  
            Set<String> r = new HashSet<>(a);  
            r.addAll(b);  
            return r; }));
```

В следующем разделе будет представлен более простой способ получения того же самого отображения. Если же требуется отображение типа `TreeMap`, то в качестве четвертого аргумента предоставляется конструктор этого отображения. Необходимо также предоставить функцию объединения. Ниже приведен один из примеров, рассматривавшихся в начале этого раздела, но теперь в нем конструируется отображение типа `TreeMap`.

```
Map<Integer, Person> idToPerson = people.collect(  
    Collectors.toMap(  
        Person::getId,  
        Function.identity(),  
        (existingValue, newValue) -> { throw new IllegalStateException(); },  
        TreeMap::new));
```



НА ЗАМЕТКУ. Для каждого из методов `toMap()` имеется эквивалентный метод `toConcurrentMap()`, выдающий параллельное отображение. Единственное параллельное отображение может быть использовано в процессе параллельного накопления. Вместе с параллельным потоком эффективнее пользоваться общим отображением, чем объединять несколько отображений, но в этом случае придется отказаться от упорядочения.

Группирование и разделение

В предыдущем разделе было показано, каким образом накапливаются все языки данной страны, но этот процесс оказался несколько трудоемким. Ведь с этой целью сначала пришлось сформировать одиночное множество для каждого отображаемого значения, а затем указать порядок объединения существующих и новых значений. Формирование групп значений с одинаковыми характеристиками является весьма распространенной операцией, и поэтому для ее непосредственной поддержки служит метод `groupingBy()`.

В качестве примера рассмотрим задачу группирования региональных настроек по странам. Сначала формируется следующее отображение:

```
Map<String, List<Locale>> countryToLocales = locales.collect(  
    Collectors.groupingBy(Locale::getCountry));
```

Функция `Locale::getCountry()` является классификатором группирования. Далее все региональные настройки можно найти по заданному коду страны, как показано в приведенном ниже примере.

```
List<Locale> swissLocales = countryToLocales.get("CH");
// выдает региональные настройки [it_CH, de_CH, fr_CH]
```



НА ЗАМЕТКУ. Напомним вкратце, что в региональных настройках содержится код языка (например, `en` для английского) и код страны (в частности, `US` для Соединенных Штатов). Так, в региональных настройках `en_US` описывается английский язык в Соединенных Штатах, а в региональных настройках `en_IE` – английский язык в Великобритании и Ирландии. Для некоторых стран имеется несколько региональных настроек. Например, в региональных настройках `ga_IE` описывается гэльский язык в Ирландии, а в приведенном выше примере показано, что текущей виртуальной машине Java известны три региональные настройки для Швейцарии.

Если функция классификатора является предикатной (т.е. возвращает логическое значение), то элементы потока ввода-вывода разделяются на два списка: один из них составляется из элементов, для которых функция возвращает логическое значение `true`, а другой – из элементов, для которых функция возвращает логическое значение `false`. В данном случае эффективнее воспользоваться методом `partitioningBy()`, чем методом `groupingBy()`. В приведенном ниже примере все региональные настройки разделяются на те, в которых используется английский язык, и на все остальные.

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```



НА ЗАМЕТКУ. Если вызвать метод `groupingByConcurrent()`, то будет получено параллельное отображение, которое заполняется параллельно, если оно используется вместе с параллельным потоком ввода-вывода. Этот метод действует точно так же, как и метод `toConcurrentMap()`.

Метод `groupingBy()` выдает отображение, значения которого представляют собой списки. Если нужно каким-то образом обработать эти списки, то предоставляется функция так называемого “нисходящего накопителя”. Так, если требуется получить множества вместо списков, то для этой цели можно воспользоваться накопителем `Collectors.toSet()`, демонстрировавшимся в предыдущем разделе и выделенным ниже полужирным.

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
    groupingBy(Locale::getCountry, toSet()));
```



НА ЗАМЕТКУ. В данном примере и во всех оставшихся примерах далее в этой главе осуществляется статический импорт потока ввода-вывода `java.util.stream.Collectors.*` для повышения удобочитаемости выражений в исходном коде.

Для обработки сгруппированных элементов по нисходящей предоставляется ряд других накопителей, в том числе следующие:

- **counting()** — делает подсчет накопленных элементов. Например, следующий накопитель подсчитывает, сколько региональных настроек имеется для каждой страны:

```
Map<String, Long> countryToLocaleCounts = locales.collect(
    groupingBy(Locale::getCountry, counting()));
```

- **summing(Int|Long|Double)** — принимает в качестве аргумента функцию, применяет ее к нисходящим элементам и вычисляет их сумму. В приведенном ниже примере вычисляется суммарное число населения каждого штата в потоке городов.

```
Map<String, Integer> stateToCityPopulation = cities.collect(
    groupingBy(City::getState, summingInt(City::getPopulation)));
```

- **maxBy() и minBy()** — принимают компаратор и выдают максимальный и минимальный нисходящие элементы. В приведенном ниже примере получается самый крупный город в каждом штате.

```
Map<String, City> stateToLargestCity = cities.collect(
    groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
```

- **mapping()** — применяет функцию к нисходящим результатам и требует еще одного накопителя для обработки своих результатов. В приведенном ниже примере города группируются по штатам. В каждом штате получаются названия городов, которые сводятся к максимальной длине.

```
Map<String, Optional<String>> stateToLongestCityName = cities.collect(
    groupingBy(City::getState,
        mapping(City::getName,
            maxBy(Comparator.comparing(String::length)))));
```

- Метод **mapping()** также позволяет найти более изящное решение задачи из предыдущего раздела, чтобы накопить множество из всех языков в стране, как показано ниже.

```
Map<String, Set<String>> countryToLanguages = locales.collect(
    groupingBy(l -> l.getDisplayCountry(),
        mapping(l -> l.getDisplayLanguage(),
            toSet())));
```

- В предыдущем разделе вместо метода **groupingBy()** применялся метод **toMap()**. А в данном случае отпадает необходимость объединять отдельные множества.
- Если функция группирования или отображения возвращает значение типа **int**, **long** или **double**, элементы множества можно накопить в объекте итоговой статистики, как обсуждалось в разделе “Накопление результатов” ранее в этой главе и показано в приведенном ниже примере. Из объектов итоговой статистики по каждой группе можно затем получить сумму, подсчет, минимальное и максимальное значений, возвращаемых функцией.

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary =
    cities.collect(groupingBy(City::getState,
        summarizingInt(City::getPopulation)));
```

- И наконец, методы типа `reducing()` выполняют общее сведение к нисходящим элементам. Имеются следующие три формы этих методов: `reducing(binaryOperator)`, `reducing(identity, binaryOperator)` и `reducing(identity, mapper, binaryOperator)`. В первой форме параметр `identity` принимает пустое значение `null`. (Следует иметь в виду, что она отличается от форм метода `Stream::reduce()`, где метод без параметра `identity` выдает результат типа `Optional`.) В третьей форме применяется функция `mapper()`, а ее значения сводятся.
- Ниже приведен пример, в котором получается разделяемая запятыми символьная строка, состоящая из названий всех городов в каждом штате. Каждый город приводится к своему названию, после чего названия городов соединяются в одну символьную строку, разделяясь запятыми.

```
Map<String, String> stateToCityNames = cities.collect(
    groupingBy(City::getState,
        reducing("", City::getName,
            (s, t) -> s.length() == 0 ? t : s + ", " + t))));
```

- Как и метод `Stream.reduce()`, метод `Collectors.reducing()` требуется редко. В данном случае добиться того же самого результата можно более естественным способом:

```
Map<String, String> stateToCityNames = cities.collect(
    groupingBy(City::getState,
        mapping(City::getName,
            joining(", "))));
```

Откровенно говоря, применение нисходящих накопителей может привести к весьма замысловатым выражениям. Поэтому пользоваться ими следует только вместе с методом `groupingBy()` или `partitioningBy()` для обработки “нисходящих” отображаемых значений. В противном случае достаточно вызвать такие методы, как, например, `map()`, `reduce()`, `count()`, `max()` или `min()` непосредственно в потоках ввода-вывода.

Потоки ввода-вывода примитивных типов

До сих пор в потоке типа `Stream<Integer>` накапливались лишь целые значения, несмотря на то, что такой способ явно неэффективен для заключения каждого целого значения в объект-оболочку. Это же относится и к данным других примитивных типов, в том числе `double`, `float`, `long`, `short`, `char`, `byte` и `boolean`. В библиотеке потоков ввода-вывода имеются специализированные типы данных `IntStream`, `LongStream` и `DoubleStream` для непосредственного хранения примитивных значений, не прибегая к оболочкам. Так, если требуется сохранить данные типа `short`, `char`, `byte` или `boolean`, то следует воспользоваться потоком ввода-вывода типа `IntStream`, `LongStream` и `DoubleStream`. Но разработчики данной библиотеки не считали нужным внедрить для этой цели еще пять типов потоков ввода-вывода.

Для того чтобы создать поток типа `IntStream`, достаточно вызвать методы `IntStream.of()` и `Arrays.stream()`, как показано ниже.

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // значения сохраняются
// в массиве типа int[]
```

Как и в потоках ввода-вывода объектов, в таком потоке можно пользоваться статическими методами `generate()` и `iterate()`. Кроме того, в потоках типа `IntStream` и `LongStream` имеются статические методы `range()` и `rangeClosed()`, формирующие отдельные диапазоны целых значений с единичным шагом:

```
IntStream zeroToNinetyNine = IntStream.range(0, 100);
    // исключая верхний предел
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
    // включая верхний предел
```

В интерфейсе `CharSequence` имеются методы `codePoints()` и `chars()`, выдающие поток типа `IntStream` для ввода-вывода символов в unicode или кодовых единиц в кодировке UTF-16. (Если вы знаете, что такое кодовые единицы, то вам вряд ли стоит пользоваться методом `chars()`). Подробнее об этом можно узнать в главе 3 первого тома книги *Core Java™, Volume I, Ninth Edition* (издательство Prentice Hall, 2013 г.; в русском переводе это издание вышло в двух томах под общим названием *Java. Библиотека профессионала, 9-е издание* в ИД “Вильямс”, 2014 г.). Ниже приведены характерные тому примеры.

```
String sentence = "\uD835\uDD46 is the set of octonions.";
// \uD835\uDD46 – кодировка UTF-16 символа ፩, уникод U+1D546
```

```
IntStream codes = sentence.codePoints();
// поток ввода-вывода шестнадцатеричных значений 1D546 20 69 73 20 . . .
```

Если имеется поток ввода-вывода объектов, его можно преобразовать в поток ввода-вывода данных примитивного типа с помощью методов `mapToInt()`, `mapToLong()` или `maptoDouble()`. Так, если имеется поток ввода-вывода символьных строк и требуется обработать их длину в виде целых значений, это можно сделать и в потоке ввода-вывода типа `IntStream` следующим образом:

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

Для того чтобы преобразовать поток ввода-вывода данных примитивного типа в поток ввода-вывода объектов, метод `boxed()` вызывается следующим образом:

```
Stream<Integer> integers = Integer.range(0, 100).boxed();
```

Как правило, методы в потоках ввода-вывода данных примитивных типов аналогичны методам в потоках ввода-вывода объектов. Тем не менее для них характерны следующие наиболее существенные отличия.

- Методы типа `toArray()` возвращают массивы данных примитивного типа.
- Методы, выдающие результат дополнительного типа, возвращают значение типа `OptionalInt`, `OptionalLong` или `OptionalDouble`. Классы этих типов аналогичны классу `Optional`, но у них имеются методы `getAsInt()`, `getAsLong()` и `getAsDouble()` вместо метода `get()`.
- Имеются также методы `sum()`, `average()`, `max()` и `min()`, возвращающие сумму, среднее, максимальное и минимальное значения соответственно. Эти методы не определены для потоков ввода-вывода объектов.

- Метод `summaryStatistics()` выдает объект типа `IntSummaryStatistics`, `LongSummaryStatistics` или `DoubleSummaryStatistics`, способный одновременно сообщать сумму, среднее, максимальное и минимальное значения в потоке ввода-вывода.



НА ЗАМЕТКУ. В классе `Random` имеются методы `ints()`, `longs()` и `doubles()`, возвращающие потоки ввода-вывода случайных числовых значений примитивных типов.

Параллельные потоки ввода-вывода

Потоки ввода-вывода упрощают распараллеливание групповых операций. Этот процесс выполняется, по существу, автоматически, хотя нужно придерживаться немногих правил. Прежде всего необходимо иметь параллельный поток ввода-вывода. По умолчанию в потоковых операциях создаются последовательные потоки ввода-вывода, за исключением вызова метода `Collection.parallelStream()`. Метод `parallel()` преобразует любой последовательный поток в параллельный, как в приведенном ниже примере.

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

До тех пор, пока поток ввода-вывода находится в параллельном режиме, когда выполняется окончная операция, все промежуточные операции, выполняемые в потоке ввода-вывода по требованию, будут распараллеливаться.

Когда потоковые операции выполняются параллельно, главная цель состоит в том, чтобы возвращался такой же самый результат, как и при последовательном выполнении операций. При этом очень важно, чтобы операции выполнялись без сохранения состояния и в произвольном порядке.

Допустим, требуется подсчитать все короткие слова в потоке ввода-вывода символьных строк. Ниже приведен неудачный пример написания кода для решения подобной задачи.

```
int[] shortWords = new int[12];
words.parallel().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
// Ошибка: состояние гонок!
System.out.println(Arrays.toString(shortWords));
```

Это весьма неудачный код. Функция, передаваемая методу `forEach()`, выполняется параллельно в нескольких потоках исполнения, обновляя общий массив. Это классическое состояние гонок. Если выполнить приведенный выше код несколько раз, то при каждом его выполнении, скорее всего, будет получена другая последовательность подсчетов, причем каждый раз неверно.

В обязанность программиста входит обеспечение потокобезопасности любых функций, передаваемых операциям в параллельном потоке ввода-вывода. В данном примере можно было бы воспользоваться массивом объектов типа `AtomicInteger` для хранения счетчиков (см. упражнение 12 в конце этой главы). С другой стороны, можно было бы просто воспользоваться средствами библиотеки потоков ввода-вывода и сгруппировать символьные строки по длине (см. упражнение 13 в конце этой главы).

По умолчанию потоки ввода-вывода, возникающие из упорядоченных коллекций (массивов и списков), диапазонов значений, генераторов и итераторов или из результата вызова метода `Stream.sorted()`, являются *упорядоченными*. Результаты накапливаются в порядке следования исходных элементов и поэтому полностью предсказуемы. Если выполнить те же самые операции дважды, то будут получены точно такие же результаты.

Упорядочение не исключает распараллеливания. Например, при вызове метода `stream.map(fun)` поток ввода-вывода может быть разделен на *n* частей, каждая из которых обрабатывается параллельно. Полученные результаты затем собираются по порядку.

Некоторые операции могут быть распараллелены более эффективно, если пре-небречь требованием к упорядочению. Вызывая метод `Stream.unordered()`, можно специально указать, что упорядочение не требуется. Этим можно, в частности, выгодно воспользоваться при вызове метода `Stream.distinct()`. В упорядоченном потоке метод `distinct()`, прежде всего, сохраняет равные элементы. Благодаря этому ускоряется распараллеливание, поскольку в том потоке исполнения, где обрабатывается часть потока ввода-вывода, неизвестно, какие именно элементы следует отбросить, до тех пор, пока не будет обработана предыдущая часть. Если же допускается сохранение любого однозначного элемента, все части могут быть обработаны параллельно (с помощью общего множества для отслеживания дубликатов).

Ускорить выполнение метода `limit()` можно также, опустив упорядочение. Если же требуется лишь получить любые *n* элементов из потока ввода-вывода безразлично в каком порядке, то можно сделать следующий вызов:

```
Stream<T> sample = stream.parallel().unordered().limit(n);
```

Как обсуждалось в разделе “Накопление данных в отображениях” ранее в этой главе, объединение отображений обходится недешево с точки зрения потребляемых ресурсов. Именно по этой причине в методе `Collectors.groupingByConcurrent()` используется общее параллельное отображение. Очевидно, что для извлечения выгоды из параллелизма порядок следования отображаемых значений не останется таким же, как и в потоке ввода-вывода. Даже в упорядоченном потоке такой накопитель характеризуется как неупорядоченный, а следовательно, он может быть эффективно использован, не переупорядочивая поток ввода-вывода. Тем не менее поток ввода-вывода должен быть непременно параллельным, как показано ниже.

```
Map<String, List<String>> result = cities.parallel().collect(  
    Collectors.groupingByConcurrent(City::getState));  
    // значения не накапливаются в порядке следования в потоке ввода-вывода
```



ВНИМАНИЕ. Очень важно не модифицировать коллекцию, поддерживающую поток ввода-вывода при выполнении потоковой операции [даже если модификация является потокобезопасной]. Напомним, что в потоках ввода-вывода не накапливаются их собственные данные, поэтому данные всегда находятся в отдельной коллекции. Если модифицировать такую коллекцию, то результат выполнения потоковых операций окажется неопределенным. В документации на набор инструментов JDK такое требование обозначается как *невмешательство*. Оно распространяется как на последовательные, так и на параллельные потоки ввода-вывода.

Точнее говоря, промежуточные потоковые операции выполняются по требованию, и поэтому коллекцию можно видоизменить вплоть до момента выполнения окончной операции. Например, следующий код является корректным:

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
wordList.add("END"); // допустимо
long n = words.distinct().count();
```

А приведенный ниже код некорректен.

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
// Ошибка: вмешательство
```

Функциональные интерфейсы

В этой главе были представлены многие операции, аргументом которых является функция. Например, метод `Streams.filter()` принимает в качестве аргумента функцию

```
Stream<String> longWords = words.filter(s -> s.length() >= 12);
```

В документации, формируемой утилитой javadoc на класс `Stream`, метод `filter()` объявляется следующим образом:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Для того чтобы понять, о чем идет речь в документации, нужно знать, что такое `Predicate`. Это интерфейс с единственным методом, не относящимся к категории по умолчанию и возвращающим логическое значение:

```
public interface Predicate {
    boolean test(T argument);
}
```

На практике обычно передается лямбда-выражение или ссылка на метод, и поэтому имя метода особого значения не имеет. Важнее другое: возврат логического значения. При чтении документации на метод `Stream.filter()` нужно лишь не забывать, что интерфейс `Predicate` — это функция, возвращающая логическое значение.



НА ЗАМЕТКУ. Если присмотреться внимательнее к объявлению метода `Stream.filter()`, то в нем можно обнаружить подстановочный тип `Predicate<? super T>`, характерный для параметров функции. Допустим, у класса `Person` имеется подкласс `Employee`, а также поток ввода-вывода типа `Stream<Employee>`. Этот поток ввода-вывода можно отфильтровать с помощью функционального интерфейса `Predicate<Employee>`, `Predicate<Person>` или `Predicate<Object>` с подстановкой соответствующего типа вместо типа `T`. Подобная гибкость особенно важна для предоставления ссылок на методы. Например, для фильтрации потока типа `Stream<Employee>` можно воспользоваться ссылкой на метод `Person::isAlive()`. И это становится возможным только благодаря подстановке типа параметра в методе `filter()`.

В табл. 2.1 сведены функциональные интерфейсы, применяемые в качестве параметров методов из классов `Stream` и `Collectors`. А другие функциональные интерфейсы приведены в следующей главе.

Таблица 2.1. Функциональные интерфейсы, применяемые в прикладном программном интерфейсе API потоков ввода-вывода

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Описание
<code>Supplier<T></code>	Отсутствует	<code>T</code>	Поставляет значение типа <code>T</code>
<code>Consumer<T></code>	<code>T</code>	<code>void</code>	Потребляет значение типа <code>T</code>
<code>BiConsumer<T, U></code>	<code>T, U</code>	<code>void</code>	Потребляет значения типа <code>T</code> и <code>U</code>
<code>Predicate<T></code>	<code>T</code>	<code>boolean</code>	Функция, возвращающая логическое значение
<code>ToIntFunction<T></code>	<code>T</code>	<code>int</code>	Функция, возвращающая значение типа <code>int</code> , <code>long</code> или <code>double</code>
<code>ToLongFunction<T></code>		<code>long</code>	
<code>ToDoubleFunction<T></code>		<code>double</code>	
<code>IntFunction<R></code>	<code>int</code>	<code>R</code>	Функция с аргументом типа <code>int</code> , <code>long</code> или <code>double</code>
<code>LongFunction<R></code>	<code>long</code>		
<code>DoubleFunction<R></code>	<code>double</code>		
<code>Function<T, R></code>	<code>T</code>	<code>R</code>	Функция с аргументом типа <code>T</code>
<code>BiFunction<T, U, R></code>	<code>T, U</code>	<code>R</code>	Функция с аргументами типа <code>T</code> и <code>U</code>
<code>UnaryOperator<T></code>	<code>T</code>	<code>T</code>	Унарный оператор типа <code>T</code>
<code>BinaryOperator<T></code>	<code>T, T</code>	<code>T</code>	Двоичный оператор типа <code>T</code>

Упражнения

- Напишите параллельный вариант цикла `for` из примера кода, приведенного в разделе “От итерации к операциям с потоками ввода-вывода” этой главы. Организуйте обработку списка по частям в нескольких потоках исполнения, подытоживая результаты по мере их получения. (Обновлять единственный счетчик в потоках исполнения вряд ли целесообразно. Почему?)
- Убедитесь в том, что запрос первых пяти длинных слов не приводит к вызову метода фильтрации, как только будет обнаружено пятое длинное слово. Просто запротоколируйте каждый вызов этого метода.
- Определите разницу во времени при подсчете длинных слов с помощью методов `parallelStream()` и `stream()`. Вызовите метод `System.currentTimeMillis()` до и после вызова каждого из этих методов, а затем выведите разницу во времени. Если у вас быстродействующий компьютер, выберите для подсчета длинных слов достаточно крупный документ (например, роман “Война и мир”).
- Допустим, имеется массив `int[] values = { 1, 4, 9, 16 }`. Что дает вызов метода `Stream.of(values)`? Как получить вместо этого поток ввода-вывода значений типа `int`?

5. Используя метод `Stream.iterate()`, создайте бесконечный поток ввода-вывода случайных чисел, не вызывая метод `Math.random()`, а реализуя непосредственно линейный конгруэнтный генератор. Формирование случайных чисел в таком генераторе начните с исходного выражения $x_0 = seed$, где $seed$ — начальное случайное число, а затем перейдите к выражению $x_n + 1 = (a \cdot x_n + c) \% m$ для получения соответствующих значений a , c и m . Реализуйте метод с параметрами a , c и m и $seed$, выдающий поток ввода-вывода типа `Stream<Long>`. Опробуйте следующие значения параметров данного метода: $a = 25214903917$, $c = 11$ и $m = 2^{48}$.
6. Метод `characterStream()`, представленный в разделе “Методы `filter()`, `map()` и `flatMap()`” этой главы, был составлен несколько неуклюже, заполняя сначала списочный массив, а затем превращая его в поток ввода-вывода. Напишите более изящный вариант этого метода на основе потока ввода-вывода. Это можно, в частности, сделать, создав поток ввода-вывода целых значений в пределах от 0 до `s.length() - 1` и преобразовав его по ссылке на метод `s::charAt()`.
7. Допустим, ваш начальник дал вам задание написать метод `public static <T> boolean isFinite(Stream<T> stream)`. Почему это задание поставлено неудачно? Несмотря на это, попробуйте все же написать такой метод.
8. Напишите метод `public static <T> Stream<T> zip(Stream<T> first, Stream<T> second)`, чередующий элементы из потоков ввода-вывода `first` и `second` вплоть до исчерпания элементов в одном из них.
9. Соедините все элементы из потока ввода-вывода типа `Stream<ArrayList<T>>` в один списочный массив типа `ArrayList<T>`. Покажите, как это можно сделать, используя три формы метода `reduce()`.
10. Организуйте вызов метода `reduce()` для вычисления среднего значения из потока ввода-вывода типа `Stream<Double>`. Почему нельзя просто вычислить сумму и разделить ее на результат вызова метода `count()`?
11. Благодаря тому что параллельные операции типа `set` на разрозненных позициях являются потокобезопасными, в параллельном режиме вполне возможно накопить результаты вычисления из потока ввода-вывода в единственном списочном массиве типа `ArrayList` вместо того, чтобы объединять несколько списочных массивов, при условии, что такой списочный массив построен по величине потока ввода-вывода. Как добиться этого?
12. Подсчитайте все короткие слова в параллельном потоке типа `Stream<String>`, обновив массив объектов типа `AtomicInteger`, как пояснялось ранее в разделе “Параллельные потоки ввода-вывода”. Воспользуйтесь атомарным методом `getAndIncrement()` для надежного приращения каждого счетчика в отдельности.
13. Повторите предыдущее упражнение, но на этот раз отфильтруйте короткие символьные строки и воспользуйтесь методами `Collectors.groupingBy()` и `Collectors.counting()`.

Глава

3

Программирование с помощью лямбда-выражений

В этой главе...

- Отложенное выполнение
- Параметры лямбда-выражений
- Выбор функционального интерфейса
- Возврат функций
- Составление операций
- Отложенность операций
- Распараллеливание операций
- Обработка исключений
- Лямбда-выражения и обобщения
- Одноместные операции
- Упражнения

В двух первых главах был рассмотрен основной синтаксис и семантика лямбда-выражений, а также прикладной программный интерфейс API потоков ввода-вывода, в котором они широко применяются. А в этой главе будет показано, как пользоваться лямбда-выражениями и функциональными интерфейсами в практике программирования на Java.

В этой главе рассматриваются следующие основные вопросы.

- Отложенное выполнение кода до подходящего момента — основная причина применения лямбда-выражения.
- Предоставление любых требующихся входных данных при выполнении лямбда-выражения.
- Выбор по возможности одного из существующих функциональных интерфейсов.
- Польза от написания методов, возвращающих экземпляр функционального интерфейса.
- Порядок составления преобразований для работы с ними.
- Поддержание списка всех ожидающих преобразований при организации их отложенного выполнения.
- Разделение основной задачи на параллельно выполняемые подзадачи, когда лямбда-выражения требуется применять неоднократно.
- Принятие во внимание того, что должно произойти, когда в лямбда-выражении генерируется исключение.
- Применение подстановок `? super` и `? extends` для типов аргументов и возвращаемых типов при обращении с функциональными интерфейсами.
- Предоставление методов `map()` и `flatMap()` при обращении с обобщенными типами, которые могут быть преобразованы функциями.

Отложенное выполнение

Основной целью всех лямбда-выражений является *отложенное выполнение*. Ведь если требуется выполнить какой-нибудь код теперь, то это можно сделать, не заключая его в оболочку лямбда-выражения. Для отложенного выполнения кода имеется немало причин, в том числе выполнение кода в следующих случаях.

- В отдельном потоке.
- Неоднократно.
- В нужный момент при выполнении алгоритма (например, при сравнении во время сортировки).
- Когда что-нибудь произойдет (например, щелчок на кнопке, поступление данных и т.д.)
- Только по мере надобности.

Целесообразно продумать заранее, чего именно требуется добиться, приступая к программированию с помощью лямбда-выражений. Обратимся к простому примеру. Допустим, требуется запротоколировать событие следующим образом:

```
logger.info("x: " + x + ", y: " + y);
```

Что произойдет, если установить уровень протоколирования INFO для подавления информационных сообщений? Символьная строка сообщения будет обработана и передана методу `info()`, где затем должно быть принято решение отбросить данное сообщение. В таком случае не лучше ли было бы выполнить сцепление символьных строк лишь по мере надобности?

Выполнение кода лишь по мере надобности служит одним из веских оснований для применения лямбда-выражений. Стандартным решением в данном случае является заключение кода в оболочку лямбда-выражения без аргументов, как показано ниже.

```
() -> "x: " + x + ", y: " + y
```

Далее нужно написать метод, выполняющий следующие действия.

1. Принятие лямбда-выражения в качестве аргумента.
2. Проверка необходимости вызова лямбда-выражения на выполнение.
3. Вызов лямбда-выражения на выполнение по мере надобности.

Для того чтобы принять лямбда-выражение, нужно выбрать (а в редких случаях — предоставить) функциональный интерфейс. Процесс выбора функционального интерфейса более подробно рассматривается в разделе “Выбор функционального интерфейса” далее этой главе. А до тех пор можно дать следующий полезный совет: выбрать интерфейс типа `Supplier<String>`. В приведенном ниже методе в качестве аргумента предоставляется лямбда-выражение для отложенного протоколирования.

```
public static void info(Logger logger, Supplier<String> message) {  
    if (logger.isLoggable(Level.INFO))  
        logger.info(message.get());  
}
```

В данном примере используется метод `isLoggable()` из класса `Logger`, чтобы принять решение, следует ли протоколировать сообщения типа INFO. Если это следует сделать, то происходит обращение к лямбда-выражению путем вызова абстрактного метода, который в данном случае называется `get()`.



НА ЗАМЕТКУ. Отложенное протоколирование сообщений оказалось настолько удачной идеей, что разработчики библиотеки Java 8 не преминули ею воспользоваться. У метода `info()` и других методов протоколирования теперь имеются варианты, принимающие интерфейс типа `Supplier<String>`. Имеется также возможность непосредственно обратиться к лямбда-выражению следующим образом:

```
logger.info(() -> "x: " + x + ", y: " + y)
```

А за потенциально полезным уточнением данного способа обращайтесь к упражнению 1 в конце этой главы.

Параметры лямбда-выражений

Если требуется предоставить компаратор, то у него, очевидно, должно быть два аргумента, обозначающих сравниваемые значения:

```
Arrays.sort(names,
    (s, t) -> Integer.compare(s.length(), t.length()));
// сравнить символьные строки s и t
```

А теперь рассмотрим другой пример. В следующем методе одно и то же действие повторяется неоднократно:

```
public static void repeat(int n, IntConsumer action) {
    for (int i = 0; i < n; i++) action.accept(i);
}
```

Почему же для действия выбран тип интерфейса `IntConsumer`, а не `Runnable`? Этим действию указывается, где именно должно происходить его повторение, что может послужить для него полезными входными данными. Эти входные данные должны быть зафиксированы действием в качестве параметра следующим образом:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Ниже в качестве примера приведен обработчик событий. Объект `event` несет информацию, которая может понадобиться действию.

```
button.setOnAction(event -> действие);
```

В общем, требуется разработать собственный алгоритм, чтобы передавать любые требующиеся данные в качестве аргументов. Например, при редактировании изображения целесообразно предоставить пользователю функцию, вычисляющую цвет пикселя. Такой функции, возможно, потребуется знать не только текущий цвет пикселя, но и его местоположение в изображении или же цвета соседних пикселей.

Но если такие аргументы требуются редко, то следует предоставить вторую версию метода, не вынуждающую пользователей принимать ненужные аргументы, как показано ниже.

```
public static void repeat(int n, Runnable action) {
    for (int i = 0; i < n; i++) action.run();
}
```

Эта версия может быть вызвана следующим образом:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

Выбор функционального интерфейса

В большинстве языков функционального программирования типы функций являются *структурными*. Для того чтобы указать функцию, преобразующую две символьные строки в целое значение, используется тип выражения, аналогичного следующему: `Function2<String, String, Integer>` or `(String, String) -> int`. А в Java вместо этого объявляется цель функции с помощью функционального интерфейса, например `Comparator<String>`. В теории языков программирования такое действие называется *номинальной типизацией*.

Безусловно, имеется немало случаев, когда требуется принять “любую функцию” без конкретной семантики. Для этой цели существует целый ряд обобщенных типов функций (см. табл. 3.1), и поэтому было бы неплохо воспользоваться одним из них по мере возможности.

Таблица 3.1. Наиболее распространенные функциональные интерфейсы

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода	Описание	Другие методы
Runnable	Отсутствует	void	run	Выполняет действие без аргументов или возвращаемого значения	
Supplier<T>	Отсутствует	T	get	Поставляет значение типа T	
Consumer<T>	T	void	accept	Потребляет значение типа T	chain
BiConsumer<T, U>	T, U	void	accept	Потребляет значения типа T и U	chain
Function<T, R>	T	R	apply	Функция с аргументом типа T	chain
BiFunction<T, U, R>	T, U	R	apply	Функция с аргументами типа T и U	andThen
UnaryOperator<T>	T	T	apply	Унарный оператор для типа T	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	Бинарный оператор для типа T	andThen
Predicate<T>	T	boolean	test	Функция, возвращающая логическое значение	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	Функция с двумя аргументами, возвращающая логическое значение	and, or, negate

Допустим, требуется написать метод для обработки файлов, соответствующих определенному критерию. Следует ли для этого использовать класс `java.io.FileFilter` или функциональный интерфейс `Predicate<File>`? В данном случае настоятельно рекомендуется использовать функциональный интерфейс `Predicate<File>`. Единственной причиной не делать этого служит существование многих полезных методов, производящих экземпляры типа `FileFilter`.



НА ЗАМЕТКУ. У большинства стандартных функциональных интерфейсов имеются неабстрактные методы для производства и объединения функций. Например, вызов метода `Predicate.isEqual(a)` равнозначен ссылке на метод `a::equals()`, если она не пустая. Имеются также методы по умолчанию, `and()`, `or()` и `negate()`, для объединения предикатов. Например, выражение `Predicate.isEqual(a).or(Predicate.isEqual(b))` равнозначно выражению `x -> a.equals(x) || b.equals(x)`.

Рассмотрим еще один пример. Допустим, требуется преобразовать изображение, применив функцию `Color -> Color` к каждому его пикселью. На рис. 3.1 приведено осветленное изображение, полученное в результате следующего вызова:

```
Image brightenedImage = transform(image, Color::brighter);
```



Рис. 3.1. Исходное и преобразованное изображения

Для этой цели имеется стандартный функциональный интерфейс `UnaryOperator<Color>`. Выбор этого интерфейса можно считать удачным решением, поскольку в этом случае отпадает необходимость обращаться к функциональному интерфейсу `ColorTransformer`. Ниже приведена реализация метода `transform()`. Обратите внимание на вызов метода `apply()`.

```
public static Image transform(Image in, UnaryOperator<Color> f) {
    int width = (int) in.getWidth();
    int height = (int) in.getHeight();
    WritableImage out = new WritableImage(width, height);
    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++)
            out.getPixelWriter().setColor(x, y,
                f.apply(in.getPixelReader().getColor(x, y)));
    return out;
}
```



НА ЗАМЕТКУ. В методе `transform()` применяются классы `Color` и `Image` из прикладного программного интерфейса JavaFX, а не из пакета `java.awt`. Подробнее о JavaFX речь пойдет в главе 4.

В табл. 3.2 перечислены имеющиеся 34 спецификации функциональных интерфейсов для примитивных типов `int`, `long` и `double`. Эти спецификации следует использовать в тех случаях, когда имеется возможность для сведения автоупаковки.

Таблица 3.2. Функциональные интерфейсы для примитивных типов

Функциональный интерфейс	Типы параметров	Возвращаемый тип	Имя абстрактного метода
BooleanSupplier	Отсутствует	boolean	getAsBoolean
PSupplier	Отсутствует	P	getAsP
PConsumer	P	void	accept
ObjPConsumer<T>	T, P	void	accept
PFunction<T>	P	T	apply
PToQFunction	P	T	applyAsQ
ToPFunction<T>	T	P	applyAsP
ToPBiFunction<T, U>	T, U	P	applyAsP
PUnaryOperator	P	P	applyAsP
PBinaryOperator	P, P	P	applyAsP
PPredicate	P	boolean	test

Примечание: **P** и **Q** обозначают `int`, `long`, `double`; а **P** и **Q** — `Int`, `Long`, `Double`

Иногда требуется предоставить собственный вариант функционального интерфейса, поскольку в стандартной библиотеке для этого отсутствуют подходящие средства. Допустим, требуется видоизменить цвета в изображении, разрешив пользователям указать функцию `(int, int, Color) -> Color`, вычисляющую новый цвет в зависимости от местоположения (координат *x*, *y*) в изображении. В таком случае можно определить собственный интерфейс следующим образом:

```
@FunctionalInterface
public interface ColorTransformer {
    Color apply(int x, int y, Color colorAtXY);
}
```



НА ЗАМЕТКУ. Абстрактный метод был назван `apply()` потому, что именно под таким именем он применяется в большинстве стандартных функциональных интерфейсов. Можно ли его назвать в таком случае как-то иначе, например `process()`, `transform()` или `getColor()`? Для пользователей кода, манипулирующего цветом, это особого значения не имеет, поскольку они обычно предоставляют лямбда-выражение. Тем не менее реализация методов упрощается, если придерживаться стандартных имен.

Возврат функций

В языке функционального программирования функции занимают привилегированное положение. Подобно числовым значениям, функции можно передавать в качестве аргументов и возвращать из методов. На первый взгляд такая возможность кажется довольно абстрактной, но на практике она весьма полезна. Язык программирования Java не является в достаточной степени функциональным, поскольку в нем применяются функциональные интерфейсы, хотя принцип остается тем же самым. Ранее были продемонстрированы многие методы, принимающие функциональные интерфейсы. А в этом разделе будут рассмотрены методы, возвращаемым типом которых является функциональный интерфейс.

Вернемся к преобразованиям изображений. Если сделать вызов

```
Image brightenedImage = transform(image, Color::brighter);
```

то изображение осветлится на заданную величину. А что, если требуется сделать изображение еще светлее или, наоборот, темнее? Можно ли передать требуемую яркость изображения в качестве дополнительного параметра методу `transform()`? Ниже показано, как это можно реализовать непосредственно в коде.

```
Image brightenedImage = transform(image,
    (c, factor) -> c.deriveColor(0, 1, factor, 1), 1.2);
    // осветлить цвет c на коэффициент 1.2
```

Кроме того, метод `transform()` можно переопределить следующим образом:

```
public static <T> Image transform(Image in, BiFunction<Color, T> f, T arg)
```

Такой прием может оказаться вполне работоспособным (см. упражнение 6 в конце этой главы), но что, если требуется предоставить два или три аргумента? Для этого имеется другой способ. В частности, можно создать метод, возвращающий подходящий функциональный интерфейс типа `UnaryOperator<Color>` с заданной яркостью изображения, как показано ниже.

```
public static UnaryOperator<Color> brighten(double factor) {
    return c -> c.deriveColor(0, 1, factor, 1);
}
```

Далее можно сделать следующий вызов:

```
Image brightenedImage = transform(image, brighten(1.2));
```

Метод `brighten()` возвращает функцию, а формально — экземпляр функционального интерфейса. Такую функцию можно передать другому методу, ожидающему подобный функциональный интерфейс (в данном случае — методу `transform()`).

В общем, не следует смущаться писать методы, производящие функции. В частности, полезно специально настраивать функции, передаваемые методам с помощью функциональных интерфейсов. В качестве примера рассмотрим метод `Arrays.sort()` с аргументом типа `Comparator`. Имеется немало способов сравнения значений, и поэтому можно написать метод, выдающий нужный компаратор (см. упражнение 7 в конце этой главы). А далее можно сделать следующий вызов: `Arrays.sort(значения, comparatorGenerator(аргументы специальной настройки))`.



НА ЗАМЕТКУ. Как будет показано в главе 8, у интерфейса `Comparator` имеется несколько методов, производящих или модифицирующих компараторы.

Составление операций

Функция с единственным аргументом преобразует одно значение в другое. Если же имеются два таких преобразования, то выполнение одного из них после другого также является преобразованием. Рассмотрим следующее манипулирование изображением: сначала осветление, а затем преобразование изображения в полутоновое (рис. 3.2).

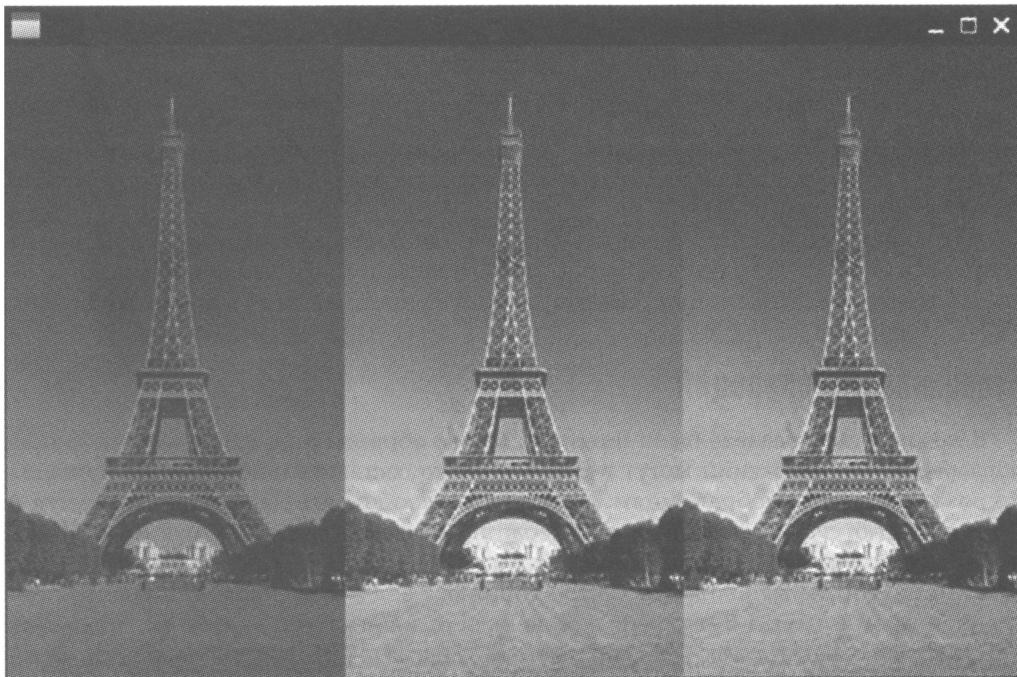


Рис. 3.2. Сначала изображение освещается, а затем превращается в полутононое



НА ЗАМЕТКУ. В печатном варианте все изображения на рис. 3.2 выглядят полностью полутоновыми. Поэтому для просмотра результата на экране цветного монитора следует выполнить приведенный ниже код.

Это нетрудно сделать с помощью метода `transform()` следующим образом:

```
Image image = new Image("eiffel-tower.jpg");
Image image2 = transform(image, Color::brighter);
Image finalImage = transform(image2, Color::grayscale);
```

Но это не совсем эффективно. Необходимо создать промежуточное изображение. Для хранения крупных изображений требуется немало памяти. Лучше было бы сначала составить операции над изображением, а затем выполнить составную операцию применительно к каждому пикселью.

В данном случае операции над изображением являются экземплярами функционального интерфейса типа `UnaryOperator<Color>`. У этого интерфейса имеется метод `compose()`, который не совсем подходит в данном случае, хотя и не совсем уместно используется в упражнении 10, приведенном в конце этой главы. Тем не менее такой метод можно написать самостоятельно, как показано ниже.

```
public static <T> UnaryOperator<T> compose(UnaryOperator<T> op1,
    UnaryOperator<T> op2) {
    return t -> op2.apply(op1.apply(t));
}
```

Далее можно сделать следующий вызов:

```
Image finalImage =  
    transform(image, compose(Color::brighter, Color::grayscale));
```

Это уже намного лучшее решение. Теперь составное преобразование выполняется непосредственно над каждым пикселием, и поэтому потребность в промежуточном изображении отпадает.

Как правило, при построении библиотеки, где пользователи могут производить один эффект за другим, целесообразно предоставить им возможность составлять такие эффекты. Еще один тому пример приведен в упражнении 11 в конце этой главы.

Отложенность операций

В предыдущем разделе было показано, каким образом пользователи метода преобразования изображений могут предварительно составить операции, не прибегая к промежуточным изображениям. Но зачем им вообще это делать? Ведь имеется и другой способ: сначала накопить все операции, а затем объединить их вместе. Именно так и делается в библиотеке потоков ввода-вывода.

Если обработка выполняется по требованию, т.е. в отложенном режиме, то прикладной программный интерфейс API должен отличать промежуточные операции, в ходе которых накапливаются выполняемые задания, от окончательных операций, выдающих результат. Обратившись к примеру обработки изображения, его преобразование можно сделать отложенным, но тогда придется возвратить другой объект, не относящийся к классу `Image`, как показано ниже.

```
LatentImage latent = transform(image, Color::brighter);
```

В классе `LatentImage` просто сохраняется исходное изображение, а также последовательность операций над ним.

```
public class LatentImage {  
    private Image in;  
    private List<UnaryOperator<Color>> pendingOperations;  
    ...  
}
```

В этом классе нужно также определить метод `transform()` следующим образом:

```
LatentImage transform(UnaryOperator<Color> f) {  
    pendingOperations.add(f);  
    return this;  
}
```

Во избежание дублирования методов `transform()` можно последовать примеру из библиотеки потоков ввода-вывода, где требуется первоначальная операция `stream()` для преобразования коллекции в поток ввода-вывода. А поскольку ввести такой метод в класс `Image` нельзя, то можно предоставить конструктор класса `LatentImage` или статический фабричный метод, как показано ниже.

```
LatentImage latent = LatentImage.from(image)  
    .transform(Color::brighter).transform(Color::grayscale);
```

Отложить выполнение задания можно только до определенного момента. В конечном итоге задание должно быть выполнено. С этой целью можно предоставить приведенный ниже метод `toImage()`, выполняющий все операции над изображением и возвращающий результат.

```
Image finalImage = LatentImage.from(image)
    .transform(Color::brighter).transform(Color::grayscale)
    .toImage();
```

Ниже приведена реализация данного метода.

```
public Image toImage() {
    int width = (int) in.getWidth();
    int height = (int) in.getHeight();
    WritableImage out = new WritableImage(width, height);
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            Color c = in.getPixelReader().getColor(x, y);
            for (UnaryOperator<Color> f : pendingOperations) c = f.apply(c);
            out.getPixelWriter().setColor(x, y, c);
        }
    }
    return out;
}
```

ВНИМАНИЕ. На практике реализовать отложенные операции довольно сложно. Обычно имеется определенное сочетание операций, причем не все они могут быть выполнены по требованию, т.е. в отложенном режиме. См. также упражнения 12 и 13 в конце этой главы.

Распараллеливание операций

Если операции выражаются в виде функциональных интерфейсов, то вызывающий код теряет полный контроль над процессом обработки. Но до тех пор, пока выполнение операций приводит к нужному результату,зывающему коду жаловаться особенно не на что. В частности, в библиотеке может быть использовано распараллеливание операций. Так, если требуется обработать изображение, его можно разделить на многие полосы и обработать каждую из них в отдельности.

В приведенном ниже коде демонстрируется простой способ организации параллельной обработки изображения. В этом коде обрабатываются массивы типа `Color[][]`, а не объекты типа `Image`, поскольку интерфейс `PixelWriter` не является потокобезопасным в JavaFX.

```
public static Color[][] parallelTransform(Color[][] in, UnaryOperator<Color> f) {
    int n = Runtime.getRuntime().availableProcessors();
    int height = in.length;
    int width = in[0].length;
    Color[][] out = new Color[height][width];
    try {
        ExecutorService pool = Executors.newCachedThreadPool();
        for (int i = 0; i < n; i++) {
            int fromY = i * height / n;
            int toY = (i + 1) * height / n;
            pool.submit(() -> {

```

```

        for (int x = 0; x < width; x++)
            for (int y = fromY; y < toY; y++)
                out[y][x] = f.apply(in[y][x]);
    });
}
pool.shutdown();
pool.awaitTermination(1, TimeUnit.HOURS);
}
catch (InterruptedException ex) {
    ex.printStackTrace();
}
return out;
}

```

Это, конечно, лишь проверка самого принципа. Главная трудность состоит в поддержке операций над изображением, объединяющих многие пиксели.

В общем, получив в свое распоряжение объект функционального интерфейса, его придется вызывать неоднократно. И попутно следует выяснить, можно ли выгодно воспользоваться распараллеливанием операций.

Обработка исключений

При написании метода, принимающего лямбда-выражения, нужно уделить внимание обработке и уведомлению об исключениях, которые могут возникнуть при выполнении лямбда-выражения.

Если в лямбда-выражении генерируется исключение, оно передается вызывающему коду. Этим выполнение лямбда-выражений ничем особым не отличается. Оно просто состоит в вызове методов для некоторых объектов, реализующих функциональный интерфейс. Зачастую уместно организовать всплытие исключений до уровня вызывающего кода.

Рассмотрим следующий пример:

```

public static void doInOrder(Runnable first, Runnable second) {
    first.run();
    second.run();
}

```

Если при выполнении метода `first.run()` генерируется исключение, то выполнение метода `doInOrder()` прекращается, а до вызова метода `second.run()` дело так и не доходит. И тогда в вызывающем коде придется обрабатывать исключение. Но допустим, что задания выполняются асинхронно, как показано ниже.

```

public static void doInOrderAsync(Runnable first, Runnable second) {
    Thread t = new Thread() {
        public void run() {
            first.run();
            second.run();
        }
    };
    t.start();
}

```

Если при выполнении метода `first.run()` генерируется исключение, исполнение потока прекращается, а метод `second.run()` вообще не выполняется. Но сразу же

происходит возврат из метода `doInOrderAsync()` и задание выполняется в отдельном потоке, и поэтому метод не может генерировать исключение повторно. В подобных случаях имеет смысл предоставить обработчик исключений, как показано ниже.

```
public static void doInOrderAsync(Runnable first, Runnable second,
    Consumer<Throwable> handler) {
    Thread t = new Thread() {
        public void run() {
            try {
                first.run();
                second.run();
            } catch (Throwable t) {
                handler.accept(t);
            }
        }
    };
    t.start();
}
```

А теперь допустим, что поток исполнения `first` поставляет результат, потребляемый в потоке исполнения `second`. И в этом случае можно по-прежнему воспользоваться обработчиком исключений, как показано ниже. С другой стороны, можно объявить поток исполнения `second` типа `BiConsumer<T, Throwable>` и обрабатывать исключение из потока исполнения `first`, как показано ниже (см. также упражнение 16 в конце этой главы).

```
public static <T> void doInOrderAsync(Supplier<T> first,
    Consumer<T> second, Consumer<Throwable> handler) {
    Thread t = new Thread() {
        public void run() {
            try {
                T result = first.get();
                second.accept(result);
            } catch (Throwable t) {
                handler.accept(t);
            }
        }
    };
    t.start();
}
```

Зачастую неудобно, если в методах из функциональных интерфейсов не допускаются проверяемые исключения. Разумеется, создаваемые самостоятельно методы могут принимать функциональные интерфейсы, в методах которых допускаются проверяемые исключения, например, интерфейс `Callable<T>` вместо `Supplier<T>`. У функционального интерфейса `Callable<T>` имеется метод, объявляемый следующим образом: `T call() throws Exception`. Если же требуется какой-нибудь эквивалент функциональному интерфейсу `Consumer` или `Function`, то его придется создать самостоятельно. Иногда в качестве выхода из этого положения предлагается воспользоваться обобщенным методом-оболочкой следующим образом:

```
public static <T> Supplier<T> unchecked(Callable<T> f) {
    return () -> {
        try {
            return f.call();
        }
```

```

    }
    catch (Exception e) {
        throw new RuntimeException(e);
    }
    catch (Throwable t) {
        throw t;
    }
};

}
}

```

В таком случае приведенный ниже вызов метода-оболочки можно передать функциональному интерфейсу `Supplier<String>`, даже если метод `readAllBytes()` генерирует исключение типа `IOException`.

```
unchecked(() -> new String(Files.readAllBytes(
    Paths.get("/etc/passwd")), StandardCharsets.UTF_8))
```

Это лишь возможное, но не окончательное решение. Например, такой метод не может сформировать функциональный интерфейс `Callable<T>` или `Function<T, U>`. Поэтому вариант метода-оболочки `unchecked()` для каждого функционального интерфейса придется реализовать самостоятельно.

Лямбда-выражения и обобщения

Как правило, в лямбда-выражениях вполне допускаются обобщенные типы. Ранее демонстрировался целый ряд примеров, в которых применялись обобщенные механизмы, в том числе метод-оболочка `unchecked()` из предыдущего примера. При этом следует иметь в виду рассматриваемые ниже особенности.

Одно из неблагоприятных последствий стирания типов состоит в том, что во время выполнения нельзя построить обобщенный массив. Например, метод `toArray()` из функционального интерфейса `Collection<T>` или `Stream<T>` не может вызвать конструктор `T[] result = new T[n]`. Следовательно, такие методы возвращают массивы типа `Object[]`. В прошлом в качестве выхода из этого положения предоставлялся второй метод, принимавший массив. Этот массив заполнялся или применялся для создания нового массива посредством рефлексии. Например, в функциональном интерфейсе `Collection<T>` имеется метод `toArray(T[] a)`. Новую возможность теперь предоставляют лямбда-выражения, а именно: передачу конструктора. Ниже показано, как это делается в отношении потоков ввода-вывода:

```
String[] result = words.toArray(String[]::new);
```

При реализации такого метода выражением конструктора является `IntFunction<T[]>`, поскольку размер массива передается конструктору. В прикладном коде конструктор можно вызвать следующим образом: `T[] result = constr. apply(n)`.

В этом отношении лямбда-выражения помогают преодолеть ограничения, присущие обобщенным типам. К сожалению, в другой типичной ситуации лямбда-выражения страдают собственным ограничением. Для того чтобы стало понятнее это ограничение, следует вспомнить понятие дисперсии типов.

Допустим, что `Employee` является подтипов `Person`. Должен ли в таком случае список типа `List<Employee>` быть особым случаем списка типа `List<Person>?`

По-видимому, должен, хотя это и ненадежный вариант. Рассмотрим следующий пример кода:

```
List<Employee> staff = ...;
List<Person> tenants = staff; // недопустимо, но можно предположить
tenants.add(new Person("John Q. Public"));
// вводит объект типа Person в список staff!
```

Следует иметь в виду, что `staff` и `tenants` являются ссылками на один и тот же список. Для того чтобы исключить такого рода ошибку, необходимо запретить преобразование типа `List<Employee>` в тип `List<Person>`. В таком случае параметр типа `T` в определении списка типа `List<T>` оказывается *инвариантным*.

Если бы список типа `List` был неизменяемым, как это имеет место в языке функционального программирования, то данное ограничение было бы устранено и список мог бы стать *ковариантным*. Именно это и сделано в таких языках программирования, как Scala. Но с момента внедрения обобщений в Java появилось очень мало неизменяемых обобщенных классов, и вместо этого разработчики языка решили воспользоваться другим понятием: подстановками, или используемой по месту дисперсией.

В методе может быть принято решение принять список типа `List<? extends Person>`, если требуется только прочитать данные из списка. В таком случае этому методу можно передать список типа `List<Person>` или `List<Employee>`. С другой стороны, этот метод может принять список типа `List<? super Employee>`, если требуется только записать данные в список. Записывать служащих в список типа `List<Person>` вполне допустимо, а следовательно, такой список можно передать. В общем, чтение данных является ковариантным (т.е. допускаются подтипы), а запись данных — контравариантной (т.е. допускаются супертипы). Используемая по месту дисперсия пригодна лишь для изменяемых структур данных. Это дает каждой службе возможность выбрать подходящую дисперсию типов, если таковая имеется.

Но использовать дисперсию по месту для типов функций затруднительно. Ведь тип функции *всегда* оказывается контравариантным в ее аргументах и ковариантным в возвращаемом ею значении. Так, если имеется функциональный интерфейс `Function<Person, Employee>`, его можно благополучно передать любому вызывающему коду, где требуется такой интерфейс. Он будет вызываться только со служащими, тогда как функция способна обработать любой субъект. Таким образом, в вызывающем коде предполагается возврат субъекта, но на самом деле получается нечто большее.

Если в Java объявляется обобщенный функциональный интерфейс, то нельзя указать, что аргументы функции всегда контравариантны, а возвращаемые типы — ковариантны. Вместо этого объявление функционального интерфейса придется повторять при каждом его применении. В качестве примера ниже приведено объявление функционального интерфейса `Stream<T>`, сформированное с помощью утилиты `javadoc`.

```
void forEach(Consumer<? super T> action)
Stream<T> filter(Predicate<? super T> predicate)
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Как правило, подстановка `? super` применяется для указания типов аргументов, а подстановка `? extends` — для указания возвращаемых типов. Таким образом, аргумент типа `Consumer<Object>` можно передать методу `forEach()` для потока ввода-вывода типа `Stream<String>`. Если же требуется потребить любой объект, то

ничто не помешает потребить символьные строки. Но подстановки не всегда доступны. Рассмотрим следующую строку кода:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

В данном примере T — это тип аргумента, а BinaryOperator — возвращаемый тип, и поэтому тип не изменяется. По существу, контравариантность и ковариантность исключают друг друга. При реализации метода, принимающего лямбда-выражения с обобщенными типами, достаточно дополнить подстановкой ? extends тип любого аргумента, не относящийся к возвращаемому типу, а подстановкой ? super тип любой возвращаемый тип, не относящийся также к типу аргумента.

В качестве примера рассмотрим метод doInOrderAsync() из предыдущего раздела. Вместо следующего объявления этого метода:

```
public static <T> void doInOrderAsync(Supplier<T> first,  
    Consumer<T> second, Consumer<Throwable> handler)
```

следует сделать такое объявление:

```
public static <T> void doInOrderAsync(Supplier<? extends T> first,  
    Consumer<? super T> second, Consumer<? super Throwable> handler)
```

Одноместные операции

При обращении с обобщенными типами и функциями, выдающими значения этих типов, полезно предоставить методы, позволяющие составлять эти функции, т.е. выполнять их одну за другой. В этом разделе будет продемонстрирован шаблон проектирования, предназначенный для подобного составления операций.

Рассмотрим обобщенный тип G<T> с одним параметром типа, например, List<T> (от нуля до большого числа значений типа T), Optional<T> (от нуля до одного значения типа T) или Future<T> (значение типа T, которое будет доступно впоследствии). Кроме того, рассмотрим функцию T → U или объект типа Function<T, U>.

Нередко имеет смысл применить эту функцию к обобщенному типу G<T> (т.е. List<T>, Optional<T>, Future<T> и т.д.). Конкретный механизм действия зависит от характера обобщенного типа G. Например, применение функции f() к объекту типа List с элементами e₁, …, e_n означает создание списка с элементами f(e₁), …, f(e_n).

Применение функции f() к объекту типа Optional<T>, содержащему значение v, означает создание объекта типа Optional<U>, содержащего вызов функции f(v). Но если функция f() применяется к пустому объекту типа Optional<T> без какого-нибудь значения, то в результате получается пустой объект типа Optional<U>. А применение функции f() к интерфейсу Future<T> просто означает ее применение, когда это доступно. В результате получается функциональный интерфейс Future<U>.

По традиции такая операция обычно называется map. Для классов Stream и Optional имеется метод map(). А для класса CompletableFuture, рассматриваемого в главе 6, имеется операция, выполняющая те же действия, что и метод map(), но называемая thenApply. И хотя для простого функционального интерфейса Future<V> метод map() отсутствует, его эквивалент нетрудно предоставить (см. упражнение 21 в конце этой главы).

Рассматривавшийся до сих пор механизм был довольно прост. Но он значительно усложняется, если вместо функций T → U обратиться к функциям T → G<U>.

В качестве примера рассмотрим получение веб-страницы по заданному URL. Для выборки веб-страницы требуется некоторое время, и поэтому вызов функции будет выглядеть следующим образом: URL → Future<String>. А теперь допустим, что имеется функциональный интерфейс Future<URL>, т.е. URL, доступный через некоторое время. Очевидно, что имеет смысл отобразить функцию на этот объект, а именно: дождаться момента, когда URL станет доступным, предоставить ему функцию, а затем дождаться появления символьной строки. По традиции такая операция называется flatMap.

Название flatMap происходит от множеств. Допустим, имеются две “многозначные” функции, вычисляющие ряд возможных результатов. Как же составить эти функции? Если функция $f(x)$ обозначает множество $\{y_1, \dots, y_n\}$, то функцию $g()$ можно применить к каждому элементу этого множества, получив в итоге $\{g(y_1), \dots, g(y_n)\}$. Но каждый элемент $g(y)$ является множеством, поэтому множество множеств нужно “свести” таким образом, чтобы получить множество всех возможных значений обеих функций.

Операция flatMap имеется и для класса Optional<T>. При наличии функции $T \rightarrow \text{Optional}<U>$ операция flatMap извлекает значение из оболочки объекта типа Optional и применяет функцию в отсутствие исходного или целевого варианта. Эта операция делает то же самое, что и операция flatMap над множествами нулевого или единичного размера.

Как правило, при разработке обобщенного типа G<T> и функции $T \rightarrow U$ нужно сначала решить, стоит ли определять операцию map, приводящую к типу G<U>. А затем следует обобщить функции $T \rightarrow G<U>$ и предоставить операцию flatMap, если это уместно.

 **НА ЗАМЕТКУ.** Такие операции играют важную роль в теории монад, но знать эту теорию совсем не обязательно, чтобы понять операции map и flatMap. Принцип отображения функции прост и полезен, и основное значение этого раздела — ознакомить с ним читателя.

Упражнения

1. Расширьте способ отложенного протоколирования условным протоколированием. Типичный вызов при этом будет следующим: `logIf(Level.FINEST, () -> i == 10, () -> "a[10] = " + a[10])`. Не вычисляйте условие, если регистратор не будет протоколировать сообщение.
2. При использовании класса ReentrantLock требуется организовать блокировку и разблокировку по следующему принципу:
`myLock.lock();
try {
 некоторое действие
} finally {
 myLock.unlock();
}`

Предоставьте метод `withLock()` таким образом, чтобы его можно было вызвать, как показано ниже.

```
withLock(myLock, () -> { некоторое действие })
```

3. В версии Java 1.4 были внедрены утверждения с помощью ключевого слова `assert`. Почему утверждения не предоставлялись в виде библиотечного средства? Можно ли было реализовать их в виде библиотечного средства в версии Java 8?
4. Сколько функциональных интерфейсов со словом `Filter` в своем имени можно обнаружить в прикладном программном интерфейсе API? Какой из них более ценен, чем функциональный интерфейс `Predicate<T>`?
5. Ниже приведен конкретный пример применения функционального интерфейса `ColorTransformer`. В этом примере изображение заключается в рамку, как показано ниже.



Сначала реализуйте вариант метода `transform()`, приведенного в разделе “Выбор функционального интерфейса”, заменив функциональный интерфейс `UnaryOperator<Color>` на функциональный интерфейс `ColorTransformer`. Затем вызовите этот метод с подходящим лямбда-выражением, чтобы заключить изображение в серую рамку толщиной 10 пикселей, заменяя пиксели на границе изображения.

6. Завершите следующий метод из раздела “Возврат функций”:

```
static <T> Image transform(Image in, BiFunction<Color, T> f, T arg)
```

7. Напишите метод, формирующий компаратор типа `Comparator<String>` для сравнения символьных строк в обычном и обратном порядке, с учетом и без учета регистра, пробелов или того и другого. Этот метод должен возвращать лямбда-выражение.

8. Обобщите пример из упражнения 5, написав статический метод, выдающий функциональный интерфейс `ColorTransformer`, заключающий изображение в цветную рамку произвольной длины.
9. Напишите метод `lexicographicComparator(String... fieldNames)`, выдающий компаратор для сравнения заданных полей в указанном порядке. Например, при вызове метода `lexicographicComparator("lastname", "firstname")` принимаются два объекта и с помощью рефлексии получаются значения из поля `lastname`. Если они разные, то возвращается разность, а иначе происходит обращение к полю `firstname`. Если же совпадают значения во всех полях, то возвращается нулевое значение.
10. Почему нельзя сделать приведенный ниже вызов?

```
UnaryOperator op = Color::brighter;
Image finalImage = transform(image, op.compose(Color::grayscale));
```

Обратите особое внимание на тип, возвращаемый из метода `compose()` функционального интерфейса `UnaryOperator<T>`. Почему это не годится для метода `transform()`? Что это говорит о полезности структурных и номинальных типов для составления функций?

11. Реализуйте статические методы, способные составлять два объекта типа `ColorTransformer`, а также статический метод, превращающий функциональный интерфейс `UnaryOperator<Color>` в функциональный интерфейс `ColorTransformer`, игнорирующий координаты `x`, `y`. Затем воспользуйтесь этими методами, чтобы заключить освещенное изображение в серую рамку (см. выше упражнение 5).
12. Расширьте класс `LatentImage`, представленный в разделе “Отложенность операций”, чтобы поддерживать оба функциональных интерфейса, `UnaryOperator<Color>` и `ColorTransformer`. Подсказка: адаптируйте первый из них во второй.
13. Фильтры свертки, например размытия и обнаружения краев, вычисляют пиксель, исходя из соседних пикселей. Для размытия изображения замените значение каждого цвета средним значением цвета этого и восьми соседних пикселей. А для обнаружения краев замените значение с каждого цвета значением $4c - n - e - s - w$, где n , e , s , и w обозначают значения цвета остальных пикселей на север, восток, юг и запад от данного пикселя. Обратите внимание на то, что эти фильтры не могут быть реализованы отложенными, используя прием, обсуждавшийся в разделе “Отложенность операций”, поскольку для них требуется обработать изображение из предыдущей стадии (или, по крайней мере, соседние пиксели). Расширьте отложенную обработку изображения таким образом, чтобы выполнять подобные операции. Усовершенствуйте вычисление на предыдущей стадии, когда определяется одна из этих операций.
14. Для отложенной обработки изображения по пикселям внесите изменения в операции преобразования таким образом, чтобы они передавались объекту типа `PixelReader`, из которого они могут читать другие пиксели в изображении. Например, `(x, y, reader) -> reader.get(width - x, y)` – это операция зеркального отображения. Фильтры свертки из предыдущего упражнения нетрудно реализовать с точки зрения подобных операций преобразования. Простейшая операция может иметь следующую форму: `(x, y, reader) ->`

`reader.get(x, y).grayscale()`, и ее можно дополнить адаптированным вариантом из функционального интерфейса `UnaryOperator<Color>`. Объект типа `PixelReader` находится на особом уровне в конвейере операций. Организуйте кеширование прочитанных недавно пикселей на каждом уровне конвейера. Если у объекта типа `PixelReader` запрашивается пиксель, его поиск осуществляется в кеше (или в исходном изображении на нулевом уровне). Если же найти пиксель не удастся, то формируется другой объект типа `PixelReader`, запрашивающий предыдущее преобразование.

15. Объедините отложенное вычисление из раздела “Отложенные операции” с параллельным вычислением из раздела “Распараллеливание операций”.
16. Реализуйте метод `doInOrderAsync()` из раздела “Обработка исключений” со вторым параметром типа `BiConsumer<T, Throwable>`. Предоставьте правдоподобный пример его применения. Нужен ли ему третий параметр?
17. Реализуйте метод `doInParallelAsync(Runnable first, Runnable second, Consumer<Throwable>)`, выполняющий потоки `first` и `second` параллельно, вызывая обработчик исключений, если метод `run()` в любом из этих потоков исполнения генерирует исключение.
18. Реализуйте версию метода `unchecked()` из раздела “Обработка исключений” таким образом, чтобы в нем формировался функциональный интерфейс `Function<T, U>` из лямбда-выражения, генерирующего проверяемые исключения. Имейте в виду, что для этого вам придется подобрать и предоставить функциональный интерфейс, абстрактный метод которого генерирует произвольные исключения.
19. Проанализируйте метод `<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)` из интерфейса `Stream<T>`. Следует ли обобщенный тип `U`, объявленный как `? super U` в первом аргументе этого метода, объявить как `BiFunction`? Объясните, почему это следует или не следует делать.
20. Предоставьте статический метод `<T, U> List<U> map(List<T>, Function<T, U>)`.
21. Предоставьте статический метод `<T, U> Future<U> map(Future<T>, Function<T, U>)`. Возвратите объект анонимного класса, реализующий все методы из интерфейса `Future`. Организуйте вызов функции в методах типа `get()`.
22. Имеется ли операция `flatMap` для класса `CompletableFuture`? Если имеется, то каково ее назначение?
23. Определите операцию `map` для класса `Pair<T>`, представляющего пару объектов обобщенного типа `T`.
24. Можно ли определить операцию `flatMap` и соответствующий метод для класса `Pair<T>`? Если можно, то каково назначение этой операции? А если нельзя, то почему

Глава

4

Прикладной программный интерфейс JavaFX

В этой главе...

- Краткая история программирования ГПИ средствами Java
- Внедрение JavaFX
- Обработка событий
- Свойства JavaFX
- Привязки
- Компоновка
- Язык разметки FXML
- Таблицы стилей CSS
- Анимация и спецэффекты
- Декоративные элементы управления
- Упражнения

Прикладной программный интерфейс JavaFX является рекомендуемым инструментальным средством для разработки графического пользовательского интерфейса (ГПИ) приложений средствами Java. Теперь JavaFX входит в комплект версий платформы Java, поддерживаемых компанией Oracle. В этой главе поясняются основы разработки ГПИ средствами JavaFX. Если у вас имеется опыт разработки клиентских платформ с ГПИ, обладающим расширенными функциональными возможностями, то из этой главы вы узнаете, как лучше всего перейти от Swing к JavaFX. А если у вас нет такого опыта, то все равно прочитайте хотя бы бегло эту главу, чтобы понять из приведенных в ней примеров, как разрабатывать графические приложения средства-ми JavaFX.

В этой главе рассматриваются следующие основные вопросы.

- Граф сцены, состоящий из узлов, которые в свою очередь могут содержать узлы.
- Сцена, отображаемая на подмостках (в окне верхнего уровня, поверхности апле-та или на всем экране).
- Элементы управления (например, кнопки), инициирующие события, хотя большинство событий в JavaFX наступают в результате изменения свойств.
- Свойства JavaFX, инициирующие события в результате изменений или недей-ствительности данных.
- Обновление одного свойства, привязанного к другому свойству, когда последнее изменяется.
- Применение в JavaFX панелей компоновки, действующих аналогично диспетче-рам компоновки в Swing.
- Определение компоновки средствами языка разметки FXML.
- Изменение внешнего вида приложения с помощью таблиц стилей CSS.
- Простые средства реализации анимации и спецеффектов.
- Усовершенствованные элементы управления, доступные в JavaFX, в том числе для построения диаграмм, проигрывания мультимедийных данных и просмо-тра встроенными средствами WebKit.

Краткая история программирования ГПИ средствами Java

Когда был разработан язык программирования Java, Интернет пребывал в зача-точном состоянии, а персональные компьютеры существовали лишь в настольном варианте. Бизнес-приложения были реализованы на платформе так называемых “толстых клиентов” — программ со многими кнопками, ползунками и текстовыми полями, предназначавшихся для взаимодействия с сервером. Такое решение считалось намного более изящным, чем “немые” (т.е. непрограммируемые) терминальные приложения из предшествовавшей эпохи. В состав Java 1.0 была включена библиотека AWT — набор инструментальных средств для разработки ГПИ независимо от конкретной платформы. Эта библиотека предназначалась для обслуживания клиентов зарождавшейся в то время Всемирной паутины (или просто веб), чтобы исключить затраты на сопровождение и обновление приложений на каждом настольном ком-пьютере.

Разработка библиотеки AWT преследовала следующий благородный замысел: предоставить общий интерфейс для программирования машинно-зависимых кнопок, ползунков, текстовых полей и прочих элементов управления ГПИ в различных операционных системах. Но реализовать этот замысел на практике не удалось до конца. В каждой операционной системе выявились незначительные отличия в функциональных возможностях виджетов ГПИ, и поэтому первоначальный замысел "написано один раз, а выполняется везде" превратился в нечто другое: "написано много раз, а отлаживается везде".

Затем была разработана библиотека Swing, главный замысел которой состоял в том, чтобы не использовать виджеты, зависящие от отдельных платформ, а отображать свои. Благодаря этому обеспечивался общий стиль оформления ГПИ на каждой платформе. При желании пользователи могли выбрать машинно-зависимый стиль оформления, характерный для каждой платформы, а виджеты Swing должны были отображаться согласованно с машинно- зависимыми виджетами. Разумеется, все это происходило медленно, на что жаловались пользователи. Когда же появились более быстродействующие компьютеры, пользователи стали жаловаться на то, что библиотека Swing оказалась в конечном итоге хуже "родных" виджетов конкретных платформ, будучи перенасыщенной анимационными эффектами и спецэффектами. Более того, средства Flash стали все чаще применяться для создания ГПИ с еще более изощренными эффектами, которые вообще не применялись в "родных" элементах управления.

В 2007 году компания Sun Microsystems внедрила новую технологию под названием JavaFX в качестве альтернативы технологии Flash. Эта технология выполнялась на виртуальной машине Java, но использовала свой язык программирования под названием JavaFX Script. Этот язык был оптимизирован для программирования анимационных эффектов и спецэффектов. Но программисты стали жаловаться на необходимость изучать этот новый язык и поэтому старались в своем большинстве держаться подальше от данной технологии. С появлением Java 7 в качестве обновления версии 6 технология JavaFX версии 2.2 вошла в комплект вместе с JDK и JRE, а в версии Java 8 она получила название JavaFX 8, отражающее результаты многократных переработок.

Разумеется, технология Flash ныне постепенно отходит в небытие, а ГПИ приложений были значительно усовершенствованы и нашли широкое распространение на мобильных устройствах. Но до сих пор имеются случаи, когда "толстый" клиент на настольном компьютере повышает производительность труда пользователя. Кроме того, код Java теперь выполняется на процессорах с архитектурой ARM. Имеются также встроенные системы, которым требуется ГПИ, в том числе информационные киоски и автомобильные средства отображения.

Предлагая технологию JavaFX, компания Oracle рекомендует применять ее в подобных приложениях. Почему же компания Oracle не внедрила лучшие части JavaFX в Swing? Для того чтобы повысить эффективность работы Swing на современных графических аппаратных средствах, эту библиотеку пришлось бы полностью переделать. Поэтому в компании Oracle было решено, что делать этого не стоит, а лучше объявить о прекращении дальнейшей разработки Swing.

В этой главе рассматриваются основы создания ГПИ в JavaFX. При этом главное внимание уделяется бизнес-приложениям с кнопками, ползунками и текстовыми полями, а не броским эффектам, послужившим первоначально к разработке JavaFX.

Применение JavaFX

Начнем с самого простого примера программы, выводящей сообщение (рис. 4.1). Как и в Swing, для этой цели используется метка следующим образом:

```
Label message = new Label("Hello, JavaFX!");
```



НА ЗАМЕТКУ. Следует иметь в виду, что в названии компонентов JavaFX отсутствует префикс **J**, характерный для компонентов Swing. Этот префикс используется в Swing для того, чтобы отличать, например, компонент **JLabel** от компонента **Label** в AWT.



Рис. 4.1. Результат вывода из программы сообщения "Hello, JavaFX!" средствами JavaFX

Увеличим размер шрифта, как показано ниже. Для этой цели служит конструктор класса **Font**, создающий объект, который по умолчанию представляет шрифт размером **100** пунктов.

```
message.setFont(new Font(100));
```

Все, что требуется отобразить средствами JavaFX, размещается на *сцене*, которую можно оформить и оживить "актерами", в роли которых выступают элементы управления и формы. Рассматриваемой здесь программе никакого оформления или анимации не требуется, но все же нужна сцена. А сцена должна находиться на *подмостках*, которые представляют собой окно верхнего уровня, если программа выполняется на рабочем столе операционной системы, или прямоугольную область, если программа выполняется в виде апплета. Подмостки передаются в виде параметра методу **start()**, который необходимо переопределить в подклассе, производном от класса **Application**, как показано в приведенном ниже примере.

```
public class HelloWorld extends Application {  
    public void start(Stage stage) {  
        Label message = new Label("Hello, JavaFX!");  
        message.setFont(new Font(100));  
        stage.setScene(new Scene(message));  
        stage.setTitle("Hello");  
        stage.show();  
    }  
}
```



НА ЗАМЕТКУ. Как следует из приведенного выше примера, для запуска JavaFX-приложения на выполнение метод **main()** не требуется. Но в предыдущих версиях JavaFX метод **main()** требовалось включать в форму, как показано ниже.

```
public class MyApp extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
    ...  
}
```

Обработка событий

Графические пользовательские интерфейсы управляются событиями. Когда пользователи выбирают кнопки щелчком на них, регулируют ползунки и выполняют другие действия, ГПИ реагирует и обновляется. Как и в Swing, отдельный элемент управления (например, кнопка) снабжается обработчиком событий, чтобы получать уведомления о выборе кнопки щелчком на ней. Все это существенно упрощается благодаря лямбда-выражениям, как выделено полужирным в приведенном ниже примере кода.

```
Button red = new Button("Red");  
red.setOnAction(event -> message.setTextFill(Color.RED));
```

При выборе кнопки щелчком на ней вызывается лямбда-выражение. В данном случае задается красный цвет текста.

Но обработка событий, возникающих в элементах управления JavaFX, происходит иначе. В качестве примера рассмотрим ползунок, показанный на рис. 4.2. Когда перемещается ползунок, регулируемое значение изменяется. Но принимать низкоуровневые события, инициируемые ползунком для уведомления об изменениях в этом элементе управления, не следует. Вместо этого у ползунка имеется *свойство* JavaFX, называемое *value*, а также инициирующее события об изменениях в нем. Более подробно свойства будут рассматриваться в следующем разделе, а ниже показано, как организовать прием событий от свойств и отрегулировать размер шрифта, которым набрано выводимое сообщение.

```
slider.valueProperty().addListener(property  
-> message.setFont(new Font(slider.getValue())));
```

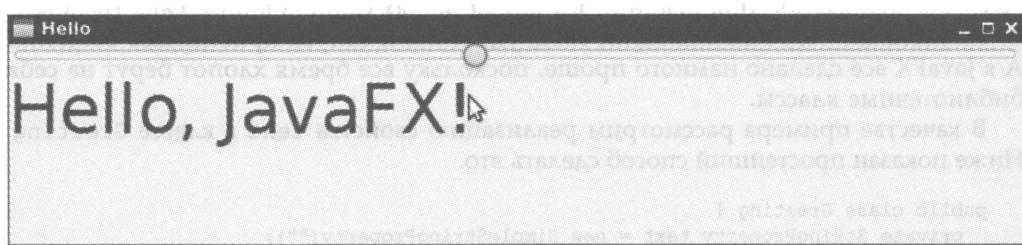


Рис. 4.2. Обработка событий от ползунка

Прием событий от свойств весьма распространен в JavaFX. Так, если требуется изменить часть пользовательского интерфейса при вводе текста пользователем в текстовом поле, достаточно ввести приемник событий в свойство *text*.



НА ЗАМЕТКУ. Иное дело — кнопки. Щелчок на кнопке совсем не изменяет одно из ее свойств.

Свойства JavaFX

Свойство — это атрибут класса, значение которого можно читать или записывать. Как правило, свойство поддерживается полем, а метод получения и установки просто читает и записывает данные в свойство соответственно. Но метод получения и установки может также принимать другие действия, в том числе чтение значений из базы данных или отправку уведомлений об изменениях. Во многих языках программирования имеется удобный синтаксис для вызова методов получения и установки значений свойств. Как правило, если свойство указывается в правой части выражения присваивания, то вызывается метод получения. А если свойство указывается в левой части выражения присваивания, то вызывается метод установки. Ниже приведен характерный тому пример.

```
value = obj.property;  
// во многих языках программирования, кроме Java,  
// здесь вызывается метод получения  
obj.property = value; // а здесь вызывается метод установки
```

К сожалению, подобный синтаксис отсутствует в Java. Тем не менее свойства поддерживаются в Java с версии 1.1. В спецификации JavaBeans поясняется, что свойство должно быть выведено из пары методов получения и установки. Например, считается, что в классе с методами `String getText()` и `void setText(String newValue)` имеется свойство `text`. А классы `Introspector` и `BeanInfo` из пакета `java.beans` позволяют перечислять все свойства класса.

В спецификации JavaBeans определяются также *привязанные свойства*, где объекты инициируют события об изменениях в свойствах при вызове методов установки. В JavaFX эта часть спецификации не используется. Вместо этого свойство в JavaFX, помимо методов получения и установки, снабжается третьим методом, возвращающим объект класса, реализующего интерфейс `Property`. Например, у свойства `text` в JavaFX имеется метод `Property<String> textProperty()`. К объекту свойства можно присоединить приемник событий. Этим JavaFX отличается от прежней технологии JavaBeans. В JavaFX объект свойства, а не компонент JavaBeans, посыпает уведомления об изменениях. И для таких изменений имеют веские основания. Для реализации привязанных свойств в JavaBeans требовался шаблонный код, выполнявший ввод, удаление и запуск приемников событий. А в JavaFX все сделано намного проще, поскольку все бремя хлопот берут на себя библиотечные классы.

В качестве примера рассмотрим реализацию свойства `text` в классе `Greeting`. Ниже показан простейший способ сделать это.

```
public class Greeting {  
    private StringProperty text = new SimpleStringProperty("");  
    public final StringProperty textProperty() { return text; }  
    public final void setText(String newValue) { text.set(newValue); }  
    public final String getText() { return text.get(); }  
}
```

Класс `StringProperty` заключает символьную строку в оболочку. В нем имеются методы для получения и установки заключенных в оболочку значений, а также для управления приемниками событий. Как видите, для реализации свойства в JavaFX требуется некоторый шаблонный код, и, к сожалению, в Java нет никакой

возможности сформировать такой код автоматически. Но, по крайней мере, об управлении приемниками событий можно не беспокоиться. И хотя объявлять как `final` методы получения и установки значений свойств совсем не обязательно, тем не менее разработчики JavaFX рекомендуют поступать именно так.



НА ЗАМЕТКУ. По такому шаблону объект свойства требуется для каждого свойства независимо от того, принимаются ли где-нибудь события об изменениях в нем. В упражнении 2, приведенном в конце этой главы, используется полезная оптимизация для шаблона, состоящая в создании объектов свойств по требованию.

В предыдущем примере был определен класс `StringProperty`. Для свойства примитивного типа в качестве оболочки используется один из следующих классов: `IntegerProperty`, `LongProperty`, `DoubleProperty`, `FloatingProperty` или `BooleanProperty`. Для этой цели имеются также классы `ListProperty`, `MapProperty` и `SetProperty`, а для всего остального — обобщенный класс `ObjectProperty<T>`. Все эти классы являются абстрактными и имеют конкретные подклассы `SimpleIntegerProperty`, `SimpleObjectProperty<T>` и т.д.



НА ЗАМЕТКУ. Если требуется лишь управлять приемниками событий, методы свойств могут возвращать объекты типа `ObjectProperty<T>` или даже интерфейс `Property<T>`. А более специализированные классы полезны для вычислений с помощью свойств, как поясняется в разделе “Привязки” далее в этой главе.



НА ЗАМЕТКУ. У классов свойств имеются методы `getValue()` и `setValue()`, помимо методов `get()` и `set()`. В классе `StringProperty` метод `get()` равнозначен методу `getValue()`, а метод `set()` — методу `setValue()`. Но для примитивных типов они разные. Например, в классе `IntegerProperty` метод `getValue()` возвращает объект типа `Integer`, а метод `get()` — значение типа `int`. Как правило, используются методы `get()` и `set()`, если только не написать обобщенный код специально для обращения со свойствами любого типа.

К свойству могут быть присоединены две разновидности приемников событий. В частности, приемник событий типа `ChangeListener` уведомляется, когда значение свойства было изменено, а приемник событий типа `InvalidationListener` вызывается, когда значение свойства *может* быть изменено. Главное отличие заключается в следующем: вычисляется ли свойство *по требованию*. Как будет показано в следующем разделе, одни свойства вычисляются из других, причем вычисление выполняется лишь по мере необходимости. В результате обратного вызова интерфейса `ChangeListener` сообщается старое и новое значение, а это означает, что в нем придется вычислять новое значение. В интерфейсе `InvalidationListener` новое значение не вычисляется, но это означает, что обратный вызов может быть получен, когда значение фактически не изменилось.

В большинстве случаев это отличие оказывается несущественным. Ведь совершенно не важно, будет ли новое значение получено в виде параметра обратного вызова или же из свойства. И, как правило, не стоит беспокоиться о вычисляемых свойствах, которые остались без изменения, несмотря на то, что изменилось одно из их входных значений. В примере кода из предыдущего раздела интерфейс `InvalidationListener` применялся ради упрощения кода.



ВНИМАНИЕ. Пользоваться интерфейсом ChangeListener для обращения со свойствами не так-то просто. Можно было бы сделать следующий вызов:

```
slider.valueProperty().addListener((property, oldValue, newValue)
    -> message.setFont(new Font(newValue)));
```

Но такой вариант не пройдет. В классе DoubleProperty реализуется интерфейс Property<Number>, а не Property<Double>. Следовательно, типом параметров oldValue и newValue является Number, а не Double. Это означает, что их придется распаковать следующим образом:

```
slider.valueProperty().addListener((property, oldValue, newValue)
    -> message.setFont(new Font(newValue.doubleValue())));
```

Привязки

Главное назначение свойств в JavaFX состоит в уведомлении о *привязке* — автоматическом обновлении одного свойства при изменении другого. В качестве примера рассмотрим приложение, приведенное на рис. 4.3. Когда пользователь редактирует верхний адрес, обновляется также нижний адрес.

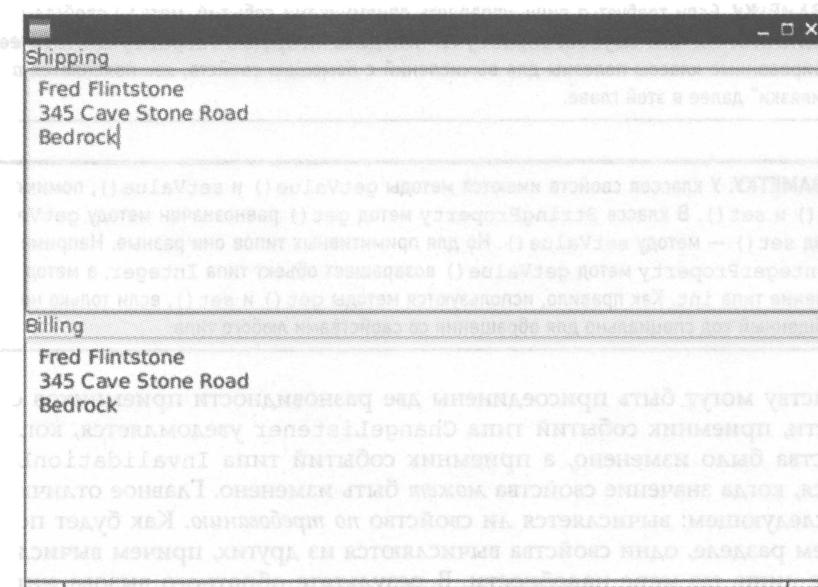


Рис. 4.3. Привязанное текстовое свойство обновляется автоматически

Такой результат достигается привязкой одного свойства к другому следующим образом:

```
billing.textProperty().bind(shipping.textProperty());
```

Приемник событий об изменениях подспудно присоединяется к текстовому свойству объекта shipping, устанавливающего текстовое свойство объекта billing. С другой стороны, можно сделать следующий вызов:

```
billing.textProperty().bindBidirectional(shipping.textProperty());
```

Если изменяется любое из этих свойств, то обновляется и другое. Для отмены привязки свойств следует вызвать метод `unbind()` или `unbindBidirectional()`.

Механизм привязки разрешает типичное затруднение, возникающее при программировании пользовательского интерфейса. В качестве примера рассмотрим поле даты и селектор даты из календаря. Когда пользователь выбирает дату из календаря, поле даты должно быть обновлено автоматически, а вместе с ним и свойство даты в модели.

Как правило, одно свойство зависит от другого, но эта взаимосвязь сложнее, чем кажется на первый взгляд. Например, круг обычно требуется расположить по центру сцены (рис. 4.4). Это означает, что значение свойства `centerX` должно составлять половину ширины сцены, определяемой свойством `width`.

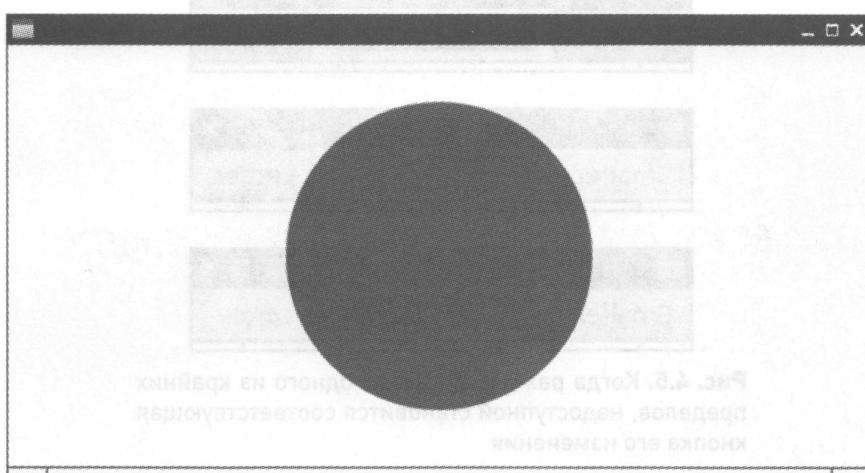


Рис. 4.4. Центр этого круга привязан к половине ширины и высоты сцены

Для достижения такого результата необходимо произвести вычисленное свойство. И для этой цели в классе `Bindings` имеются соответствующие статические методы. Например, при вызове метода `Bindings.divide(scene.widthProperty(), 2)` вычисляется свойство, значение которого составляет половину ширины сцены. Это свойство автоматически обновляется при изменении ширины сцены. Остается лишь привязать это вычисленное свойство к свойству `centerX` круга следующим образом:

```
circle.centerXProperty().bind(Bindings.divide(scene.widthProperty(), 2));
```



НА ЗАМЕТКУ. С другой стороны, можно вызывать метод `scene.widthProperty().divide(2)`. Применение статических методов из класса `Bindings` в более сложных выражениях повышает удобочитаемость исходного кода, особенно если сделать сначала следующее объявление:

```
import static javafx.beans.binding.Bindings.*;
```

а затем такой вызов:

```
divide(scene.widthProperty(), 2).
```

Рассмотрим более реалистичный пример. Допустим, требуется сделать недоступными кнопки Smaller (Мельче) и Larger (Крупнее), когда размер слишком мал или велик (рис. 4.5). Когда ширина меньше или равна нулю, недоступной становится кнопка Smaller. А когда ширина больше или равна 100, недоступной становится кнопка Larger. Ниже показано, как все это реализуется непосредственно в коде.

```
smaller.disableProperty().bind(
    Bindings.lessThanOrEqualTo(gauge.widthProperty(), 0));
larger.disableProperty().bind(
    Bindings.greaterThanOrEqualTo(gauge.widthProperty(), 100));
```

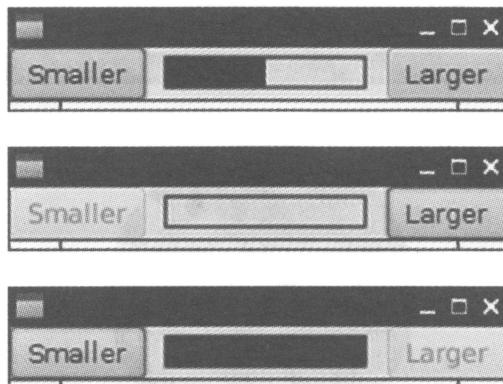


Рис. 4.5. Когда размер достигает одного из крайних пределов, недоступной становится соответствующая кнопка его изменения

В табл. 4.1 перечислены все операции и соответствующие им методы, поддерживаемые в классе Bindings. В качестве одного или обоих аргументов указываются объекты, в классах которых реализуется интерфейс Observable или производные от него интерфейсы. В интерфейсе Observable предоставляются методы для ввода и удаления приемника событий типа `InvalidationListener`. В интерфейсе `ObservableValue` вводится приемник событий типа `ChangeListener` и метод `getValue()`, а в производных от него интерфейсах — методы для получения значения подходящего типа. Например, метод `get()` из интерфейса `ObservableStringValue` возвращает значение типа `int`. Типы, возвращаемые из методов класса `Bindings`, являются интерфейсами, производными от интерфейса `Binding`, который, в свою очередь, является производным от интерфейса `Observable`. Интерфейсу `Binding` известны все свойства, от которых он зависит. На практике разбираться во всех этих интерфейсах совсем не обязательно. Для привязки одного свойства к другому с целью добиться желаемого результата достаточно употребить эти свойства в нужном сочетании.

Таблица 4.1. Операции и соответствующие им методы, поддерживаемые в классе Bindings

Имя метода	Аргументы
<code>add, subtract, multiply,</code> <code>divide, max, min</code>	Оба типа <code>ObservableNumberValue, int, long, float, double</code>
<code>negate</code>	Типа <code>ObservableNumberValue</code>

Окончание табл. 4.1

Имя метода	Аргументы
<code>greaterThan,</code> <code>greaterThanOrEqual,</code> <code>lessThan, lessThanOrEqual</code>	Оба типа <code>ObservableNumberValue, int, long, float, double</code> или же типа <code>ObservableStringValue, String</code>
<code>equal, notEqual</code>	Оба типа <code>ObservableObjectValue, ObservableNumberValue, int, long, float, double, Object</code>
<code>equalIgnoreCase,</code> <code>notEqualIgnoreCase</code>	Оба типа <code>ObservableStringValue, String</code>
<code>isEmpty, isNotEmpty</code>	Типа <code>Observable(List Map Set StringValue)</code>
<code>isNull, isNotNull</code>	Типа <code>ObservableObjectValue</code>
<code>length</code>	Типа <code>ObservableStringValue</code>
<code>size</code>	Типа <code>Observable(List Map Set)</code>
<code>and, or</code>	Оба типа <code>ObservableBooleanValue</code>
<code>not</code>	Типа <code>ObservableBooleanValue</code>
<code>convert</code>	Типа <code>ObservableValue</code> , преобразуемого в символьную строку для привязки
<code>concat</code>	Последовательность объектов, значения которых сцепляются в символьные строки, возвращаемые методом <code>toString()</code> . Если любой из объектов относится к изменяющемуся типу <code>ObservableValue</code> , то изменяется и результат сцепления строк
<code>format</code>	Дополнительные региональные настройки, символьная строка, отформатированная средствами класс <code>MessageFormat</code> , последовательность форматируемых объектов. Если любой из объектов относится к изменяющемуся типу <code>ObservableValue</code> , то изменяется и отформатированная строка
<code>valueAt (double float integer long) ValueAt stringValueAt</code>	Типа <code>ObservableList</code> и индекс или типа <code>ObservableMap</code> и ключ
<code>create(Boolean Double Float Integer Long Object String) Binding</code>	Типа <code>Callable</code> и список зависимостей
<code>select select(Boolean Double Float Integer Long String)</code>	Типа <code>Object</code> или <code>ObservableValue</code> и последовательность имен открытых свойств, дающих свойство <code>obj.p₁, p₂, ..., p_n</code>
<code>when</code>	Дает построитель условного оператора. Привязка <code>when(b)</code> . <code>then(v₁) . otherwise(v₂)</code> дает <code>v₁</code> или <code>v₂</code> в зависимости от того, имеет ли аргумент <code>b</code> типа <code>ObservableBooleanValue</code> логическое значение <code>true</code> . Здесь <code>v₁</code> или <code>v₂</code> может принимать регулярное или наблюдаемое значение. Условное значение вычисляется повторно всякий раз, когда изменяется наблюдаемое значение

Привязка вычисленного значения с помощью методов из класса `Bindings` может получиться довольно вычурной. Впрочем, имеется другой, более простой способ получения вычисленных привязок. Для этого достаточно разместить вычисляемое выражение в лямбда-выражении и предоставить список зависимых свойств. При

изменении любых свойств лямбда-выражение вычисляется повторно, как показано в приведенном ниже примере. В упражнении 5, приведенном в конце этой главы, предлагается еще более изящный способ вычисления привязок по требованию с помощью лямбда-выражений.

```
larger.disableProperty().bind(  
    createBooleanBinding(  
        () -> gauge.getWidth() >= 100, // это выражение вычисляется . . .  
        gauge.widthProperty())); // . . . когда изменяется данное свойство
```



НА ЗАМЕТКУ. Компилятор языка JavaFX Script анализирует выражения привязки и автоматически определяет зависимые свойства. В рассмотренном выше примере кода было объявлено следующее: **disable bind gauge.width >= 100**, и поэтому компилятор присоединил приемник события к свойству **gauge.width property**. А программирующему на Java подобные сведения приходится предоставлять вручную.

Компоновка

Если ГПИ содержит несколько элементов управления, они должны быть расположены на экране как с точки зрения выполняемых ими функций, так и с точки зрения удобства применения. Компоновку элементов управления ГПИ можно осуществить с помощью специального инструментального средства, предназначенного для разработки пользовательского интерфейса. Пользователь такого инструментального средства (зачастую графический дизайнер) перетаскивает графические изображения элементов управления в представлении конструирования, располагая их, изменения размеры и настраивая. Но такой способ построения ГПИ может вызвать определенные трудности, например, при изменении размеров элементов, поскольку метки могут иметь разную длину при интернационализации программы на разных языках.

С другой стороны, компоновка может быть достигнута программным способом, т.е. написанием кода в методе установки, вводящим элементы управления в пользовательский интерфейс на отдельных позициях. Именно так и было сделано в библиотеке Swing с помощью объектов диспетчеров компоновки.

Еще один способ состоит в том, чтобы определить компоновку на декларативном, непроцедурном языке. Например, веб-страницы компонуются средствами HTML и CSS. Аналогично на платформе Android имеется отдельный язык XML для определения компоновок.

В JavaFX поддерживаются все три упомянутых выше способа. В частности, SceneBuilder выполняет в JavaFX роль визуального конструктора ГПИ. Это инструментальное средство можно загрузить по адресу www.oracle.com/technetwork/java/javafx/overview. На рис. 4.6 показан моментальный снимок экрана SceneBuilder. Это инструментальное средство здесь не рассматривается, но, уяснив основные принципы компоновки ГПИ, вы сможете без особого труда пользоваться SceneBuilder на практике.

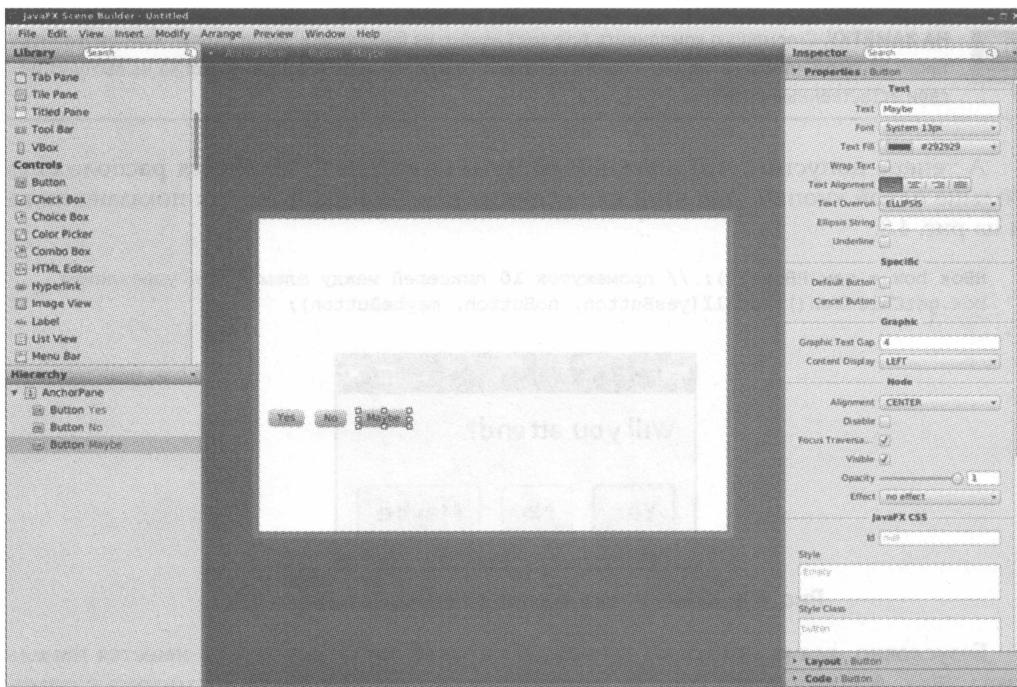


Рис. 4.6. Инструментальное средство SceneBuilder в JavaFX

Программная компоновка осуществляется во многом так же, как и в Swing. Но вместо диспетчеров компоновки, вводимых на произвольных панелях, в данном случае используются *панели* — контейнеры с правилами компоновки. Например, на панели типа BorderPane имеются пять областей: северная, западная, южная, восточная и центральная. В приведенном ниже примере показано, каким образом кнопки располагаются в этих областях панели, а результат приведен на рис. 4.7.

```
BorderPane pane = new BorderPane();
pane.setTop(new Button("Top"));
pane.setLeft(new Button("Left"));
pane.setCenter(new Button("Center"));
pane.setRight(new Button("Right"));
pane.setBottom(new Button("Bottom"));
stage.setScene(new Scene(pane));
```

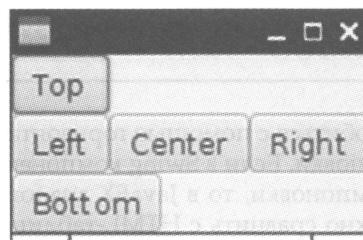


Рис. 4.7. Компоновка кнопок на панели типа BorderPane



НА ЗАМЕТКУ. С помощью компонента `BorderLayout` из библиотеки Swing кнопки компоновались таким образом, чтобы заполнить собой всю компонуемую область. А в JavaFX кнопка не выходит на свои естественные размеры.

А теперь допустим, что в южной области компоновки требуется расположить больше одной кнопки. Для этой цели служит панель типа `HBox`, как показано ниже и на рис. 4.8.

```
HBox box = new HBox(10); // промежуток 10 пикселей между элементами управления  
box.getChildren().addAll(yesButton, noButton, maybeButton);
```

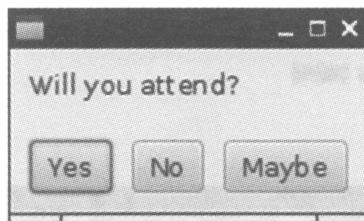


Рис. 4.8. Компоновка кнопок с помощью панели `HBox`

Безусловно, для компоновки элементов управления по вертикали имеется панель типа `VBox`. А компоновка кнопок, приведенная на рис. 4.8, была достигнута с помощью следующего фрагмента кода:

```
VBox pane = new VBox(10);  
pane.getChildren().addAll(question, buttons);  
pane.setPadding(new Insets(10));
```

Обратите внимание на свойство `padding`. Без него метка и кнопки будут сливаться с границей окна, не отделяясь от нее заданным промежутком.



ВНИМАНИЕ. Размеры в JavaFX указываются в пикселях. В данном примере для заполнения и расположения кнопок с пробелами выбрана величина промежутка, равная 10 пикселям. Но в настоящее время указывать размеры в пикселях уже стало неуместно, поскольку плотность расположения пикселей может заметно отличаться. Это затруднение можно, в частности, преодолеть, вычисляя размеры в корневых круглых шпациях [rem], как это делается по стандарту CSS3. (Корневая круглая шпация обозначает высоту исходного шрифта в корне документа.) В приведенном ниже примере кода показано, как это делается.

```
final double rem = new Text("").getLayoutBounds().getHeight();  
pane.setPadding(new Insets(0.8 * rem));
```

Но это все, чего можно добиться с помощью горизонтальных и вертикальных прямоугольных областей компоновки. Если в Swing компонент `GridBagLayout` составлял основу всех диспетчеров компоновки, то в JavaFX аналогичную роль выполняет панель типа `GridPane`. Ее можно сравнить с HTML-таблицей. На этой панели можно задать выравнивание всех ячеек по горизонтали и по вертикали, а при желании — сделать так, чтобы ячейки заполняли несколько рядов и столбцов. В качестве примера на рис. 4.9 показана компоновка диалогового окна регистрации.

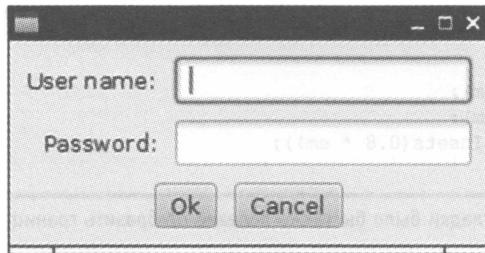


Рис. 4.9. Панель типа GridPane позволяет расположить элементы управления в этом диалоговом окне регистрации

Имейте в виду следующее:

- метки "User name" (Имя пользователя) и "Password" (Пароль) выравниваются по правому краю;
- кнопки располагаются на панели типа HBox, охватывающем два ряда.

Когда на панели типа GridPane вводится дочерний элемент, следует указать индексы ряда и столбца его расположения (именно в таком порядке, т.е. аналогично координатам x , y). В приведенном ниже примере кода показано, как это делается.

```
pane.add(usernameLabel, 0, 0);
pane.add(username, 1, 0);
pane.add(passwordLabel, 0, 1);
pane.add(password, 1, 1);
```

Если же дочерний элемент охватывает несколько столбцов или рядов, следует также указать охватываемую область после места его расположения. Например, в следующей строке кода указывается, что панель кнопок охватывает два столбца и один ряд:

```
pane.add(buttons, 0, 2, 2, 1);
```

А если требуется, чтобы дочерний элемент охватывал все оставшиеся ряды или столбцы, следует воспользоваться свойством GridPane.REMAINDER. Для выравнивания дочернего элемента по горизонтали достаточно вызвать метод setHalignment(), передав ссылку на дочерний элемент, а также константу LEFT, CENTER или RIGHT из перечисления типа HPos, как показано ниже. Аналогично для вертикального выравнивания достаточно вызвать метод setValignment(), передав ему константу TOP, CENTER или BOTTOM из перечисления типа VPos.

```
GridPane.setHalignment(usernameLabel, HPos.RIGHT);
```

НА ЗАМЕТКУ. Вызовы этих статических методов выглядят не совсем изящно в коде Java. Тем не менее они имеют смысл в языке разметки FXML, как поясняется в следующем разделе.

ВНИМАНИЕ. Не выравнивайте по центру панель типа HBox с кнопками, расположеннымными сеткой. Эта прямоугольная область компоновки распространяется по всей ширине, поэтому центровка не изменяет расположение данной области. Поэтому лучше отцентровать не саму панель типа HBox, а его содержимое, как показано ниже.

```
buttons.setAlignment(Pos.CENTER);
```

Ряды и столбцы, а также саму таблицу требуется также отделить пробелами следующим образом:

```
pane.setHgap(0.8 * em);
pane.setVgap(0.8 * em);
pane.setPadding(new Insets(0.8 * em));
```



СОВЕТ. Для целей отладки было бы также полезно отобразить границы ячеек (рис. 4.10), сделав следующий вызов:

```
pane.setGridLinesVisible(true);
```

Если же требуется отобразить границы отдельного дочернего элемента (например, для того, чтобы было видно, заполняет ли он всю ячейку), их следует задать. И это проще всего сделать с помощью таблицы стилей CSS следующим образом:

```
buttons.setStyle("-fx-border-color: red;");
```

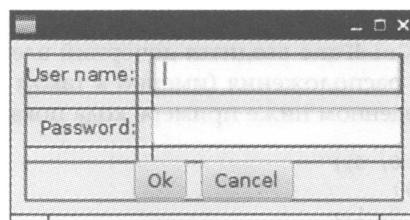


Рис. 4.10. При отладке компоновки на панели типа *GridPane* используются видимые линии сетки

Упомянутых выше панелей должно быть достаточно для компоновки ГПИ большинства приложений. В табл. 4.2 приведены все компоновки, создаваемые средствами JavaFX.

Таблица 4.2. Компоновки в JavaFX

Класс панели	Описание
HBox, VBox	Выравнивает дочерние элементы по горизонтали или по вертикали
GridPane	Располагает дочерние элементы в виде табличной сетки аналогично компоненту GridBagLayout в Swing
TilePane	Располагает дочерние элементы в виде сетки и одинакового размера аналогично компоненту GridLayout в Swing
BorderPane	Предоставляет для компоновки северную, восточную, южную, западную и центральную области аналогично компоненту BorderLayout в Swing
FlowPane	Располагает дочерние элементы рядами, создавая новые ряды, если недостаточно свободного места, аналогично компоненту FlowLayout в Swing
AnchorPane	Дочерние элементы могут быть расположены на абсолютных позициях или относительно границ панели. Такой режим выбирается по умолчанию в инструменте компоновки SceneBuilder
StackPane	Располагает дочерние элементы друг над другом. Такая компоновка может оказаться полезной для оформления компонентов, например, для размещения кнопки на цветном прямоугольнике



НА ЗАМЕТКУ. В этом разделе рассматривается построение пользовательских интерфейсов ручным вложением панелей и элементов управления. В языке JavaFX Script имелся синтаксис "построителя" для описания таких вложенных структур, называемых "графом сцены". В версии JavaFX 2 применялись классы построителей для имитации данного синтаксиса. Ниже показано, как построить подобными средствами диалоговое окно регистрации.

```
GridPane pane = GridPaneBuilder.create()
    .hgap(10)
    .vgap(10)
    .padding(new Insets(10))
    .children(
        usernameLabel = LabelBuilder.create()
            .text("User name:")
            .build(),
        passwordLabel = LabelBuilder.create()
            .text("Password:")
            .build(),
        username = TextFieldBuilder.create().build(),
        password = PasswordFieldBuilder.create().build(),
        buttons = HBoxBuilder.create()
            .spacing(10)
            .alignment(Pos.CENTER)
            .children(
                okButton = ButtonBuilder.create()
                    .text("Ok")
                    .build(),
                cancelButton = ButtonBuilder.create()
                    .text("Cancel")
                    .build())
            .build())
    .build();
```

Такой код выглядит довольно громоздким, и тем не менее он не избавляет от необходимости указывать ограничения на сетку. Построители элементов ГПИ стали нерекомендованными к применению в версии JavaFX 8 не из-за громоздкости кода, а из-за трудностей реализации. Ради экономии кода в построителях применяется дерево наследования, распараллеливающее наследование соответствующих узлов. Например, класс GridPaneBuilder расширяет класс PaneBuilder, поскольку класс GridPane расширяет класс Pane. Но при этом возникает вопрос: что именно должен возвращать метод PaneBuilder.children()? Если он возвращает только объект типа PaneBuilder, то пользователь должен проявить особую аккуратность, настроив сначала свойства подкласса, а затем и свойства суперкласса.

Разработчики постарались разрешить данное затруднение с помощью обобщений. Методы из обобщенного класса PaneBuilder возвращают объект типа B, а следовательно, класс GridPaneBuilder может расширять класс PaneBuilder<GridPaneBuilder>. Но дело в том, что сам класс GridPaneBuilder является обобщенным, и поэтому вполне допустимо его расширение GridPaneBuilder<GridPaneBuilder> или даже GridPaneBuilder<GridPaneBuilder<- нечто>>. Подобное зацикливание было преодолено специальными приемами, но они не совсем надежны и вряд ли окажутся работоспособными в последующих версиях Java. Поэтому от построителей пришлось отказаться. Если же построители элементов ГПИ вас вполне устраивают, воспользуйтесь языком Scala или Groovy и его привязками к JavaFX (<https://code.google.com/p/scalafx;> <http://groovyfx.org/>).

Язык разметки FXML

Для описания компоновок в JavaFX применяется язык разметки FXML под названием FXML. Это язык разметки рассматривается здесь в некоторых подробностях потому, что он основывается на ряде принципов, интерес к которым выходит за рамки JavaFX, а реализуется он обычным способом. Ниже приведен пример разметки в коде FXML диалогового окна регистрации из предыдущего примера.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<GridPane hgap="10" vgap="10">
    <padding>
        <Insets top="10" right="10" bottom="10" left="10"/>
    </padding>
    <children>
        <Label text="User name:" GridPane.columnIndex="0" GridPane.rowIndex="0"
               GridPane.halignment="RIGHT" />
        <Label text="Password: " GridPane.columnIndex="0" GridPane.rowIndex="1"
               GridPane.halignment="RIGHT" />
        <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
        <PasswordField GridPane.columnIndex="1" GridPane.rowIndex="1" />
        <HBox GridPane.columnIndex="0" GridPane.rowIndex="2"
              GridPane.columnSpan="2" alignment="CENTER" spacing="10">
            <children>
                <Button text="Ok" />
                <Button text="Cancel" />
            </children>
        </HBox>
    </children>
</GridPane>
```

Если проанализировать приведенный выше код FXML-разметки, то в нем можно обнаружить команды обработки `<?import ... *?>` для импорта пакетов Java. (Как правило, команды обработки в XML служат спасительным выходом для тех приложений, где требуется обрабатывать XML-документы.)

А теперь рассмотрим структуру данного FXML-документа. Прежде всего, вложение панели типа `GridPane`, меток и текстовых полей, панели типа `HBox` и ее дочерних элементов в виде кнопок отражает вложение, организованное в коде Java из предыдущего раздела. Большинство атрибутов соответствуют методам установки свойств. Например, приведенная ниже строка кода означает следующее: “построить панель типа `GridPane` и установить свойства `hgap` и `vgap`”.

```
<GridPane hgap="10" vgap="10">
```

Если атрибут начинается с имени класса и статического метода, этот метод вызывается. Например, следующая строка кода означает, что будут вызваны статические методы `GridPane.setColumnIndex(thisTextField, 1)` и `GridPane.setRowIndex(thisTextField, 0)`:

```
<TextField GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```



НА ЗАМЕТКУ. Как правило, элемент разметки FXML сначала конструируется своим конструктором по умолчанию, а затем специально настраивается с помощью вызываемых методов установки свойств или статических методов в духе спецификации JavaBeans. Некоторые исключения из данного правила будут рассмотрены далее в этой главе.

Если свойство слишком трудно выразить в виде символьной строки, то вместо атрибутов применяются вложенные элементы. Рассмотрим следующий пример:

```
<GridPane hgap="10" vgap="10">
  <padding>
    <Insets top="10" right="10" bottom="10" left="10"/>
  </padding>
  ...

```

Свойство padding содержит объект типа Insets, построенный с помощью дочернего элемента разметки `<Insets ...>`, в котором определяется порядок установки его свойств. И наконец, для списочных свойств имеется особое правило. Например, children — это списочное свойство, и в результате обращения к нему в следующем коде разметки вводятся кнопки в список, возвращаемый методом `getChildren()`.

```
<HBox ...>
  <children>
    <Button text="Ok" />
    <Button text="Cancel" />
  </children>
</HBox>
```

Файлы FXML-документов можно составить вручную или же воспользоваться для этой цели инструментальным средством SceneBuilder, упоминавшимся в предыдущем разделе. Получив такой файл, его можно загрузить следующим образом:

```
public void start(Stage stage) {
  try {
    Parent root = FXMLLoader.load(getClass().getResource("dialog.fxml"));
    stage.setScene(new Scene(root));
    stage.show();
  } catch (IOException ex) {
    ex.printStackTrace();
    System.exit(0);
  }
}
```

Разумеется, такой код пока еще не является полезным сам по себе. Пользовательский интерфейс отображается, но программа не имеет доступа к значениям, предоставляемым пользователем. Установить связь между элементами управления пользовательского интерфейса и программой можно, в частности, воспользовавшись атрибутами id, как это обычно делается в JavaScript. Атрибуты id предоставляются в FXML-файле следующим образом:

```
<TextField id="username" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
```

А в программе элемент управления находится следующим образом:

```
TextField username = (TextField) root.lookup("#username");
```

Но имеется и лучший способ. В частности, аннотацию @FXML можно использовать для “внедрения” управляющих объектов в класс контроллера. А в классе контроллера должен быть реализован интерфейс Initializable. В методе initialize() из класса контроллера средства привязки связываются с обработчиками событий. В качестве контроллера может служить любой класс — даже само JavaFX-приложение. Например, в приведенном ниже коде реализуется контроллер для управления диалоговым окном регистрации.

```
public class LoginDialogController implements Initializable {
    @FXML private TextField username;
    @FXML private PasswordField password;
    @FXML private Button okButton;

    public void initialize(URL url, ResourceBundle rb) {
        okButton.disableProperty().bind(
            Bindings.createBooleanBinding(
                () -> username.getText().length() == 0
                || password.getText().length() == 0,
                username.textProperty(),
                password.textProperty()));
        okButton.setOnAction(event ->
            System.out.println("Verifying " + username.getText()
                + ":" + password.getText()));
    }
}
```

В файле FXML-документа следует предоставить имена переменных экземпляра контроллера для соответствующих элементов управления в этом файле. Для этой цели служит атрибут fx:id, а не атрибут id, как выделено ниже полужирным.

```
<TextField fx:id="username" GridPane.columnIndex="1"
    GridPane.rowIndex="0"/>
<PasswordField fx:id="password" GridPane.columnIndex="1"
    GridPane.rowIndex="1" />
<Button fx:id="okButton" text="Ok" />
```

В корневом элементе необходимо также объявить класс контроллера. А для этой цели служит атрибут fx:controller, как показано ниже. Обратите внимание на атрибут для внедрения пространства имен FXML.

```
<GridPane xmlns:fx="http://javafx.com/fxml" hgap="10" vgap="10"
    fx:controller="LoginDialogController">
```



НА ЗАМЕТКУ. Если у контроллера отсутствует конструктор по умолчанию (возможно, потому что он инициализирован по ссылке на бизнес-сервис), то его можно установить программным способом, как показано ниже.

```
FXMLLoader loader = new FXMLLoader(getClass().getResource(...));
loader.setController(new Controller(service));
Parent root = (Parent) loader.load();
```



ВНИМАНИЕ. Если контроллер устанавливается программным способом, то для этой цели вполне подходит исходный код из предыдущей врезки. Приведенный ниже код будет скомпилирован, но в нем будет вызван статический метод FXMLLoader.load() в обход построенного загрузчика.

```
FXMLLoader loader = new FXMLLoader();
loader.setController(...);
Parent root = (Parent) loader.load(getClass().getResource(...));
// Ошибка: вызывается статический метод
```

При загрузке файла FXML-документа конструируется граф сцены, а ссылки на именованные объекты управления внедряются в аннотированные поля объекта контроллера. Затем вызывается его метод `initialize()`.

Большую часть инициализации вполне возможно сделать в файле FXML-документа. В частности, можно определить простые привязки и задать аннотированные методы контроллера в качестве приемников событий. Не вдаваясь в подробности, заметим, что документация на соответствующий синтаксис доступна по адресу http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html. По-видимому, визуальный дизайн лучше отделить от поведения программы, чтобы разработчик пользовательского интерфейса мог заниматься графическим оформлением программы, а программист — реализацией ее поведения.



НА ЗАМЕТКУ. В файл FXML-документа можно также ввести сценарии на JavaScript или другом языке написания сценариев. Этот вопрос вкратце обсуждается в главе 7.

Таблицы стилей CSS

В JavaFX допускается изменение внешнего вида пользовательского интерфейса с помощью таблиц стилей CSS, что, как правило, удобнее, чем предоставление атрибутов FXML-разметки или вызов методов в Java. Таблицу стилей CSS можно загрузить программно и применить ее к графу сцены следующим образом:

```
Scene scene = new Scene(pane);
scene.getStylesheets().add("scene.css");
```

В таблице стилей можно обращаться к любым элементам управления, имеющим идентификатор. В качестве примера рассмотрим, как организовать управление внешним видом панели типа `GridPane`. Идентификатор этой панели устанавливается в коде следующим образом:

```
GridPane pane = new GridPane();
pane.setId("pane");
```

А любое заполнение промежутков или расстановка устанавливается не в коде, а в таблице CSS, как показано ниже.

```
#pane {
    -fx-padding: 0.5em;
    -fx-hgap: 0.5em;
    -fx-vgap: 0.5em;
    -fx-background-image: url("metal.jpg")
}
```

К сожалению, вместо известных атрибутов CSS придется освоить и применять специальные атрибуты JavaFX, имена которых начинаются с префикса `-fx-` и обра- зуются из имен свойств в нижнем регистре и дефисов вместо смешанного написания.

Например, свойство `textAlignment`, определяющее выравнивание текста, превращается в соответствующий атрибут `-fx-text-alignment`. Все атрибуты CSS, поддерживаемые в JavaFX, можно найти по адресу <http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>.

Пользоваться таблицами CSS удобнее, чем загромождать код деталями компоновки ГПИ. Кроме того, можно без особого труда пользоваться круглыми шпациями как единицами измерения, независящими от разрешения. Разумеется, пользование таблицами CSS способно принести не только пользу, но и нанести вред. В частности, следует удерживаться от искушения применять неуместные фоновые текстуры в диалоговых окнах регистрации (рис. 4.11).

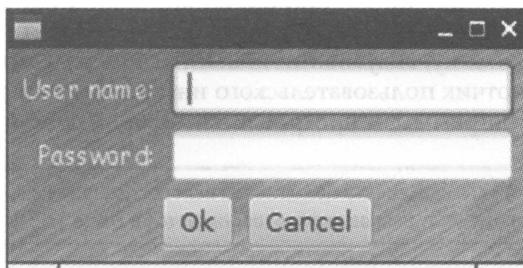


Рис. 4.11. Неуместное применение фоновой текстуры для стилизации пользовательского интерфейса средствами CSS

Вместо стилизации по отдельным идентификаторам можно воспользоваться классами стилей. С этой целью класс стиля вводится в узловой объект следующим образом:

```
HBox buttons = new HBox();
buttons.getStyleClass().add("buttonrow");
```

Затем он стилизуется с помощью следующего обозначения класса CSS:

```
.buttonrow {
    -fx-spacing: 0.5em;
}
```

Каждый класс элемента управления или формы в JavaFX относится к классу CSS, имя которого образуется из имени класса Java, приведенного в нижний регистр. Например, все узлы типа `Label` относятся к классу `label`. В качестве примера ниже показано, как заменить на `Comic Sans` шрифт всех меток, хотя на самом деле делать этого не нужно.

```
.label {
    -fx-font-family: "Comic Sans MS";
}
```

Кроме того, таблицы стилей CSS можно применять в FXML-компонентовках. Для этого достаточно присоединить таблицу стилей к корневой панели следующим образом:

```
<GridPane id="pane" stylesheets="scene.css">
```

Код FXML-разметки снабжается атрибутами `id` или `styleClass`, как показано в приведенном ниже примере.

```
<HBox styleClass="buttonrow">
```

Затем большую часть стилизации можно указать в таблице стилей CSS, а FXML-разметку использовать только для компоновки. К сожалению, стилизацию нельзя полностью удалить из FXML-разметки. Так, в настоящее время отсутствует способ, позволяющий указывать выравнивание ячейки сеточной компоновки в таблице стилей CSS.



НА ЗАМЕТКУ. Стили CSS можно применять и программным способом, например, так, как показано ниже.

```
buttons.setStyle("-fx-border-color: red;");
```

Такой способ может оказаться удобным для отладки, но, как правило, лучше пользоваться внешними таблицами стилей.

Анимация и спецэффекты

На момент появления JavaFX спецэффекты были весьма распространены, и поэтому в JavaFX имеются средства, позволяющие без особого труда отбрасывать тени, размывать изображение или перемещать объекты. В Интернете можно найти немало примеров, демонстрирующих беспорядочное движение всплывающих пузырьков, нервно скачущий текст и прочие анимационные эффекты. В этой связи было бы полезно дать некоторые рекомендации по поводу применения подобных анимационных эффектов в прикладных программах. В качестве примера на рис. 4.12 показано приложение, в котором размеры кнопки Yes (Да) постепенно увеличиваются, кнопка No (Нет) сливаются с фоном, а кнопка Maybe (Может быть) вращается.

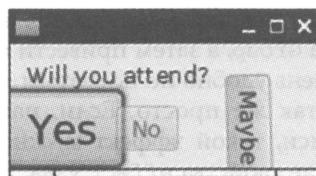


Рис. 4.12. Кнопки, которые постепенно увеличиваются, обесцвечиваются и вращаются

В JavaFX определен целый ряд монтажных *переходов*, изменяющих свойства узлов во времени. В качестве примера ниже показано, каким образом в коде реализуется увеличение размеров узла наполовину в направлении обеих осей координат *x* и *y* в течение трех секунд.

```
ScaleTransition st = new ScaleTransition(Duration.millis(3000));  
st.setByX(1.5);  
st.setByY(1.5);  
st.setNode(yesButton);  
st.play();
```

В качестве узла для монтажного перехода может служить любой узел графа сцены, например, кружок в анимации мыльных пузырьков или более привлекательная кнопка Yes в диалоговом окне, приведенном на рис. 4.12. Установленный однажды монтажный переход завершается по достижении своей цели. Его циклическое повторение можно организовать следующим образом:

```
st.setCycleCount(Animation.INDEFINITE);
st.setAutoReverse(true);
```

Теперь узел будет то увеличиваться, то уменьшаться до бесконечности. При монтажном переходе типа FadeTransition постепенно изменяется непрозрачность узла. Ниже показано, каким образом в коде реализуется постепенное слияние кнопки No с фоном.

```
FadeTransition ft = new FadeTransition(Duration.millis(3000));
ft.setFromValue(1.0);
ft.setToValue(0);
ft.setNode(noButton);
ft.play();
```

Все узлы JavaFX могут вращаться вокруг своего центра. Так, при монтажном переходе типа RotateTransition изменяется свойство rotate вращаемого узла. В приведенном ниже фрагменте кода осуществляется анимация кнопки Maybe.

```
RotateTransition rt = new RotateTransition(Duration.millis(3000));
rt.setByAngle(180);
rt.setCycleCount(Animation.INDEFINITE);
rt.setAutoReverse(true);
rt.setNode(maybeButton);
rt.play();
```

Монтажные переходы можно составлять, сочетая объекты типа ParallelTransition и SequentialTransition, чтобы выполнять эти переходы параллельно или последовательно. Если же требуется анимация нескольких узлов, их достаточно разместить в групповом узле типа Group, а затем привести в движение. И для того чтобы добиться такого поведения, очень удобно пользоваться классами JavaFX.

Спецэффекты создаются так же просто. Если, например, требуется отбрасывать тени от нарядной надписи, такой эффект можно создать с помощью класса DropShadow, задав его в качестве свойства effect узла. На рис. 4.13 приведен результат применения эффекта отбрасывания тени к узлу типа Text, а ниже показано, каким образом данный эффект реализуется в коде.

```
DropShadow dropShadow = new DropShadow();
dropShadow.setRadius(5.0);
dropShadow.setOffsetX(3.0);
dropShadow.setOffsetY(3.0);
dropShadow.setColor(Color.GRAY);

Text text = new Text();
text.setFill(Color.RED);
text.setText("Drop shadow");
text.setFont(Font.font("sans", FontWeight.BOLD, 40));
text.setEffect(dropShadow);
```

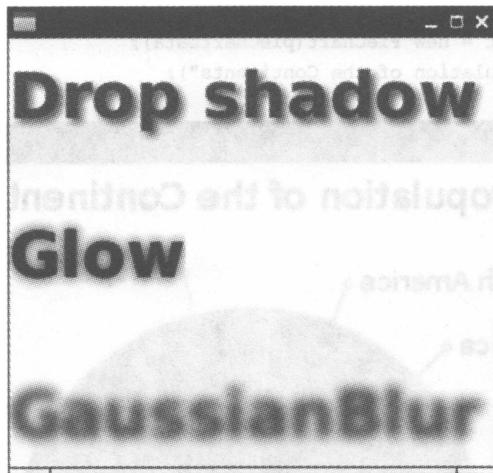


Рис. 4.13. Спецэффекты, получаемые средствами JavaFX

Эффекты свечения или размытия, приведенные на рис. 4.13, реализуются так же просто. Ниже показано, как это делается.

```
text2.setEffect(new Glow(0.8));
text3.setEffect(new GaussianBlur());
```

Правда, эффект свечения выглядит не очень привлекательно, а эффект размытия вряд ли найдет широкое применение в большинстве прикладных программ. Тем не менее нельзя не отметить, насколько просто получить такие спецэффекты.

Декоративные элементы управления

Разумеется, в JavaFX, как и в Swing, имеются комбинированные списки, панели вкладок, деревья, таблицы, а также элементы управления пользовательского интерфейса, отсутствующие в Swing, в том числе селектор дат и меню-гармошка. Для подробного их описания потребовалась бы отдельная книга. Поэтому, для того чтобы развеять ностальгию по Swing, в этом разделе будут рассмотрены три декоративных элемента управления, функциональные возможности которых выходят далеко за рамки того, что может предложить Swing.

На рис. 4.14 показана одна из многих диаграмм, которые можно построить средствами JavaFX. И для этого не потребуется специально устанавливать сторонние библиотеки.

А в приведенном ниже фрагменте кода показано, насколько просто построить круговую диаграмму средствами JavaFX.

```
ObservableList<PieChart.Data> pieChartData =
    FXCollections.observableArrayList(
        new PieChart.Data("Asia", 4298723000.0),
        new PieChart.Data("North America", 355361000.0),
        new PieChart.Data("South America", 616644000.0),
        new PieChart.Data("Europe", 742452000.0),
        new PieChart.Data("Africa", 1110635000.0),
```

```
new PieChart.Data("Oceania", 38304000.0));
final PieChart chart = new PieChart(pieChartData);
chart.setTitle("Population of the Continents");
```

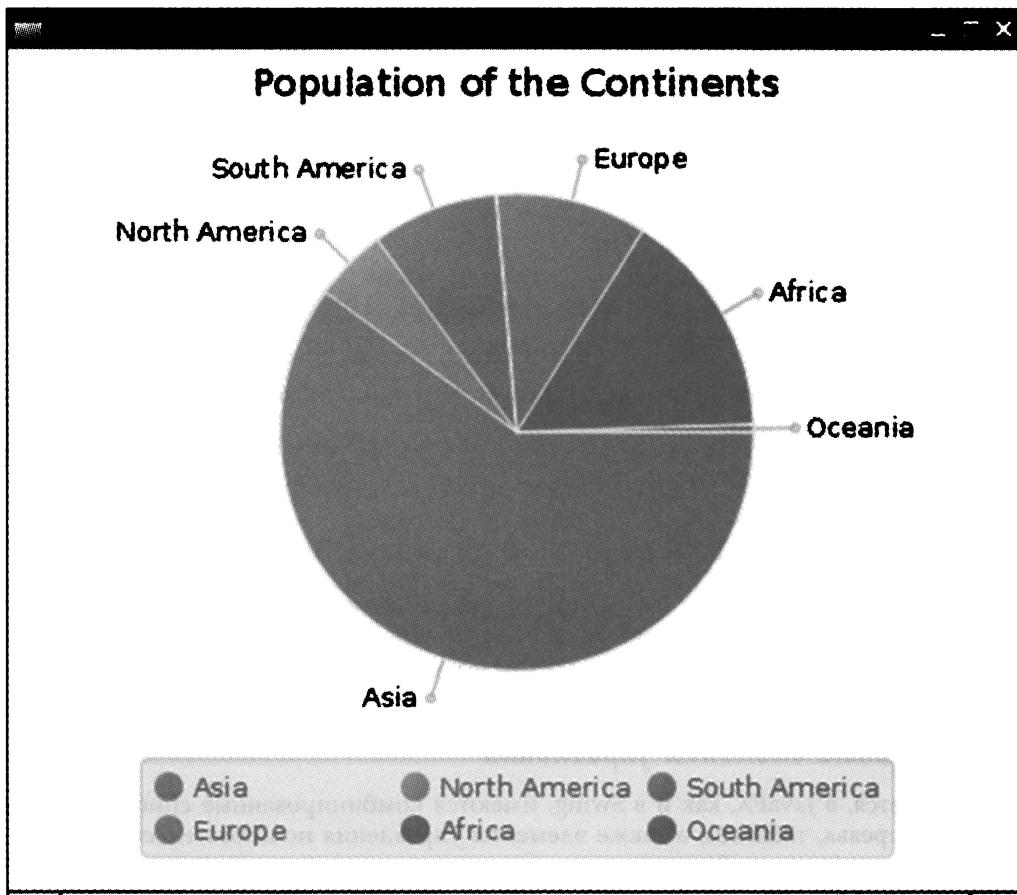


Рис. 4.14. Круговая диаграмма, построенная средствами JavaFX

Вообще в JavaFX имеется с полдесятка разновидностей диаграмм, которые можно применять в готовом виде или же специально настраивать. Подробнее об этом можно узнать из документации, доступной по адресу <http://docs.oracle.com/javafx/2/charts/jfxpub-charts.htm>.

В Swing можно было отображать HTML-разметку в компоненте `JEditorPane`, но воспроизведение настоящих HTML-документов, как правило, поддерживалось слабо. И это понятно, ведь реализовать подобные функциональные возможности на уровне большинства браузеров не так-то просто, поскольку они служат надстройкой над открытым кодом механизма `WebKit`. То же самое происходит и в JavaFX. В частности, класс `WebView` отображает встроенное машинно-зависимое окно `WebKit`, как показано на рис. 4.15.

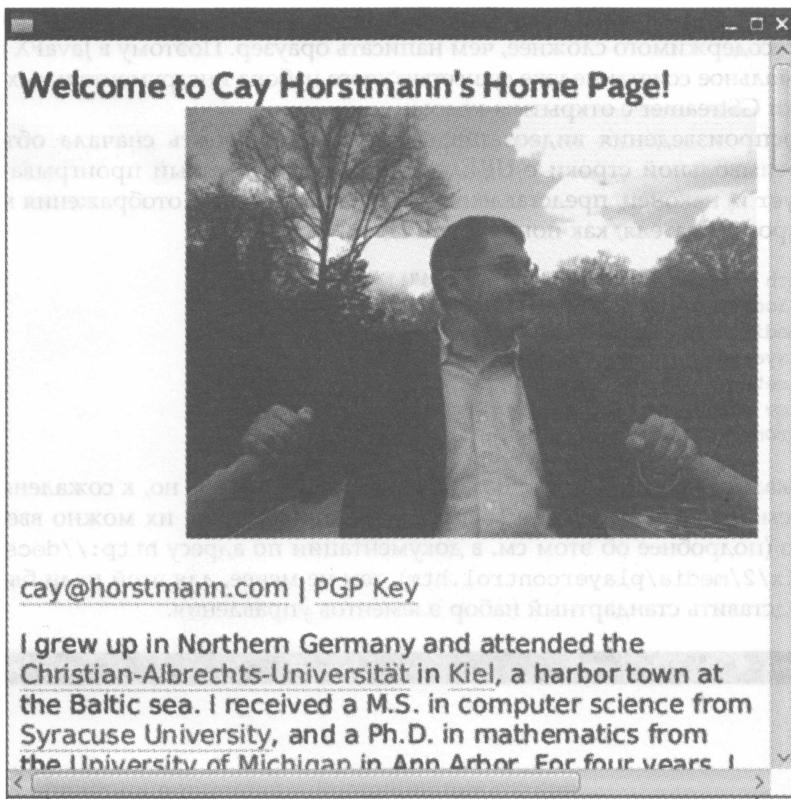


Рис. 4.15. Просмотр веб-страницы

Ниже приведен фрагмент кода для отображения веб-страницы, приведенной на рис. 4.15, средствами JavaFX.

```
String location = "http://horstmann.com";
WebView browser = new WebView();
WebEngine engine = browser.getEngine();
engine.load(location);
```

Браузер действует, реагируя обычным образом на выбор ссылок щелчком кнопкой мыши. Действует также и код сценариев JavaScript. Но если требуется отобразить строку состояния или всплывающие сообщения из сценария JavaScript, то придется установить обработчики уведомлений и реализовать отображение строки состояния или всплывающих сообщений самостоятельно.



НА ЗАМЕТКУ. В классе WebView не поддерживается ни один из подключаемых модулей, и поэтому его нельзя использовать для отображения Flash-анимации, документов формата PDF и даже аплетов.

До появления JavaFX воспроизведение мультимедийного содержимого было достойно сожаления. В частности, дополнительная загрузка такого содержимого была доступна в прикладном программном интерфейсе Java Media Framework, но это

мало привлекало разработчиков. Разумеется, реализовать воспроизведение мультимедийного содержимого сложнее, чем написать браузер. Поэтому в JavaFX используется оптимальное сочетание уже существующего набора инструментальных средств и библиотеки GStreamer с открытым кодом.

Для воспроизведения видеозаписей следует построить сначала объект типа `Media` из символьной строки с URL, затем мультимедийный проигрыватель типа `MediaPlayer` и, наконец, представление типа `MediaView` для отображения мультимедийного проигрывателя, как показано ниже.

```
Path path = Paths.get("moonlanding.mp4");
String location = path.toUri().toString();
Media media = new Media(location);
MediaPlayer player = new MediaPlayer(media);
player.setAutoPlay(true);
MediaView view = new MediaView(player);
view.setOnError(e -> System.out.println(e));
```

Как показано на рис. 4.16, видеозапись воспроизводится, но, к сожалению, отсутствуют элементы управления ее воспроизведением. И хотя их можно ввести самостоятельно (подробнее об этом см. в документации по адресу <http://docs.oracle.com/javafx/2/media/playercontrol.htm>), тем не менее, для этой цели было бы не плохо представить стандартный набор элементов управления.



Рис 4.16. Воспроизведение видеозаписи средствами JavaFX



НА ЗАМЕТКУ. Несмотря на все сказанное выше, в библиотеке GStreamer отсутствуют средства для обработки отдельных файлов видеозаписи. Обработчик ошибок в коде отображает сообщения GStreamer для целей диагностики ошибок, возникающих при воспроизведении.

На этом краткий обзор функциональных возможностей JavaFX завершается. С JavaFX связывается будущее разработки настольных приложений на Java. Эта технология еще не отлажена полностью в силу той поспешности, с которой пришлось переходить на нее из исходного языка написания сценариев. Но, безусловно, пользоваться средствами JavaFX не труднее, чем Swing, а имеющиеся среди них элементы управления намного более полезны и привлекательны, чем те, что когда-либо имелись в Swing.

Упражнения

1. Напишите программу с текстовым полем и соответствующей меткой. Как и в программе, выводящей сообщение "Hello, JavaFX!" средствами JavaFX, метка должна содержать это сообщение, выделенное шрифтом размером 100 пунктов. Инициализируйте текстовое поле тем же самым сообщением. А метку обновляйте по мере редактирования содержимого текстового поля.
2. Разработайте класс со многими свойствами JavaFX, например, для составления таблицы или диаграммы. Скорее всего, в конкретном приложении к большинству этих свойств не будут присоединены приемники событий, поэтому не стоит снабжать каждое свойство отдельным объектом. Покажите, каким образом можно устанавливать свойство по требованию, сначала с помощью обычного поля, в котором должно храниться значение свойства, а затем с помощью объекта свойства, но только в тот момент, когда метод `xxxProperty()` вызывается в первый раз.
3. Разработайте класс со многими свойствами JavaFX, большинство исходных значений которых вообще не изменяются. Покажите, каким образом свойство может быть установлено по умолчанию, когда задается неисходное значение или же когда метод `xxxProperty()` вызывается в первый раз.
4. Усовершенствуйте программу из раздела "Привязки" этой главы таким образом, чтобы круг остался отцентрованным и всегда касался, по крайней мере, двух сторон сцены.
5. Напишите следующие методы:

```
public static <T, R> ObservableValue<R> observe(
    Function<T, R> f, ObservableValue<T> t)
public static <T, U, R> ObservableValue<R> observe(
    BiFunction<T, U, R> f, ObservableValue<T> t, ObservableValue<U> u)
```

Они должны возвращать наблюдаемые значения, для которых метод `getValue()` возвращает значение лямбда-выражения, а приемники событий о недостоверности и изменениях данных запускаются в тот момент, когда любые вводимые данные становятся недостоверными или изменяются, как показано в приведенном ниже примере.

```
larger.disableProperty().bind(observe(
    t -> t >= 100, gauge.widthProperty()));
```

6. Отцентруйте верхнюю и нижнюю кнопки, приведенные на рис. 4.7.
7. Придумайте, как установить границу элемента управления, не прибегая к таблице стилей CSS.
8. Для синтаксического анализа файлов FXML-документов в JavaFX отсутствует соответствующая логика, поэтому придумайте пример загрузки объекта с рядом вложенных объектов, но не имеющего никакого отношения к JavaFX, а также установите свойства, используя синтаксис FXML. Было бы лучше, если бы вы воспользовались для этой цели внедрением кода.
9. Организуйте анимацию окружности, представляющей планету, чтобы она перемещалась по эллиптической орбите. Воспользуйтесь для этого классом `PathTransition`.
10. Используя средство просмотра веб-содержимого, реализуйте браузер со строкой для ввода URL и кнопкой возврата к прежней веб-странице. *Подсказка:* воспользуйтесь для этой цели методом `WebEngine.getHistory()`.

Глава

5

Новый прикладной программный интерфейс API для даты и времени

В этой главе...

- Временная шкала
- Местные даты
- Корректоры дат
- Местное время
- Поясное время
- Форматирование и синтаксический анализ даты и времени
- Взаимодействие с устаревшим кодом
- Упражнения

Время летит как стрела, и ничто не мешает легко установить исходный момент и отсчитать время вперед и назад в секундах. Почему же так трудно обращаться со временем? Все дело в человеческом факторе. Было бы ли проще, если бы мы стали договариваться друг с другом следующим образом: «Встретимся в 1371409200, и не опаздывай!» Но ведь текущее время должно быть каким-то образом связано с временем суток и года. Именно здесь и возникают главные трудности. В версии Java 1.0 имелся класс Date, который в прошлом, хотя, возможно, и наивно содержал большинство методов, ставших не рекомендованными к употреблению в версии Java 1.1, когда в ней был внедрен класс Calendar. Его прикладной программный интерфейс API не был идеальным, его экземпляры были изменяемыми, и он не имел никакого отношения к таким понятиям, как потерянные секунды. И наконец, с третьей попытки свершилось чудо, когда в версии Java 8 был внедрен пакет java.time, в котором были устранены имеющиеся в прошлом недостатки и который должен отныне служить для обращения со временем в коде Java. В этой главе будет показано, что именно затрудняет расчет времени и как в новом прикладном программном интерфейсе API для даты и времени устраняются подобные затруднения.

В этой главе рассматриваются следующие основные вопросы.

- Все объекты классов из пакета `java.time` являются неизменяемыми.
- Класс Instant представляет точку на временной шкале (подобно объекту типа Date).
- В Java время суток составляет ровно 86400 секунд, т.е. исключает потерянные секунды.
- Класс Duration представляет разность между двумя моментами времени.
- Класс LocalDateTime не содержит сведений о часовом поясе.
- Методы из класса TemporalAdjuster выполняют типичные календарные расчеты, например, поиск первого вторника в месяце.
- Класс ZonedDateTime представляет момент времени в заданном часовом поясе (аналогично классу GregorianCalendar).
- Применение класса Period вместо класса Duration при смене часового пояса, чтобы принять во внимание переход на летнее время.
- Форматирование и синтаксический анализ дат и времени средствами класса DateTimeFormatter.

Временная шкала

По традиции основополагающей единицей измерения времени является секунда, получаемая из расчета суточного времени вращения Земли вокруг своей оси. Сутки составляют 24 часа, или $24 \times 60 \times 60 = 86400$ секунд, в течение которых Земля совершает полный оборот вокруг своей оси. Таким образом, величина секунды определяется точными астрономическими измерениями. К сожалению, Земля совершает незначительные колебательные движения, и поэтому требовалось более точное определение секунды, и в 1967 году такое определение секунды появилось. Оно соответствовало историческому определению секунды и было получено исходя из внутреннего свойства атомов цезия 133. С тех пор была создана целая сеть атомных часов для отсчета официального времени.

С тех пор официальные приборы для хранения времени синхронизируют абсолютное время с вращением Земли. Сначала официальные секунды немного корректировались, но с 1972 года стали периодически вводиться так называемые "потерянные" секунды. (Теоретически вполне возможно, что секунды придется удалять, но на практике до этого дело пока еще не дошло.) В настоящее время снова возникли дискуссии по поводу изменений в системе отсчета времени. Очевидно, что потерянные секунды вносят немало неудобств в точный отчет времени, и поэтому во многих вычислительных системах применяется так называемое "слаживание", когда требуется искусственно замедлить или ускорить время до потерянной секунды, чтобы время суток по-прежнему составляло 86400 секунд. Такой прием оказывается вполне работоспособным, поскольку местное время на компьютере не является совершенно точным, а следовательно, компьютеры должны периодически синхронизироваться с внешней службой времени.

В спецификации на прикладной программный интерфейс API для даты и времени в Java требуется применение временной шкалы, соответствующей приведенным ниже условиям. Благодаря этому язык Java может гибче приспосабливаться к возможным в будущем изменениям в системе отсчета точного времени.

- Содержит 86400 секунд в сутках.
- Точно соответствует официальному времени в полдень каждого дня.
- Близко соответствуетциальному времени в другое время суток и точно определяется.

В Java класс `Instant` представляет точку на временной шкале. В качестве начала отсчета времени, называемого *эпохой*, произвольно выбрана полночь 1 января 1970 года на главном меридиане, проходящем через Гринвичскую королевскую обсерваторию в Лондоне. Именно такое соглашение было принято для отчета времени в Unix/POSIX. От этого начального момента время отсчитывается по 86400 секунд каждые сутки с точностью порядка наносекунд в ту или иную сторону.

Значения типа `Instant` отсчитываются назад на миллиарды лет (вплоть до наименьшего значения `Instant.MIN`). Этого недостаточно для выражения возраста вселенной (около 13,5 миллиарда лет), но в то же время достаточно для большинства практических целей. В конечном итоге миллиард лет назад Земля была покрыта льдом и населена микроскопическими предшественниками современных растений и животных. А наибольшее значение `Instant.MAX` соответствует 31 декабря 1000000000 года. В результате вызова статического метода `Instant.now()` получается текущий момент времени. А с помощью методов `equals()` и `compareTo()` оба момента времени сравниваются обычным образом, чтобы использовать их в качестве меток времени.

Для определения разности между двумя экземплярами типа `Instant` служит метод `Duration.between()`. В следующем примере кода демонстрируется реализация алгоритма измерения текущего времени:

```
Instant start = Instant.now();
runAlgorithm();
Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);
long millis = timeElapsed.toMillis();
```

Класс `Duration` определяет промежуток между двумя моментами времени. Продолжительность этого промежутка времени определяется в соответствующих единицах в результате вызова методов `toNanos()`, `toMillis()`, `toSeconds()`, `toMinutes()`, `toHours()` или `toDays()`.

Для внутреннего хранения промежутков времени требуется нечто большее, чем значение типа `long`. Количество секунд содержится в значении типа `long`, а количество наносекунд — в дополнительном значении типа `int`. Если требуется произвести расчет времени с точностью до наносекунд, то сначала придется указать весь диапазон значений промежутка типа `Duration`, а затем воспользоваться одним из методов, перечисленных в табл. 5.1. В противном случае достаточно вызвать метод `toNanos()` и выполнить расчет времени с помощью значений типа `long`.



НА ЗАМЕТКУ. Для переполнения значения типа `long` промежуток в наносекундах должен составлять почти 300 лет.

Таблица 5.1. Арифметические операции над моментами и промежутками времени и соответствующие им методы

Метод	Описание
<code>plus, minus</code>	Добавляют или вычтывают промежуток времени из данного момента времени (объекта типа <code>Instant</code>) или промежутка времени (объекта типа <code>Duration</code>)
<code>plusNanos, plusMillis,</code> <code>plusSeconds,</code> <code>plusMinutes, plusHours,</code> <code>plusDays</code>	Добавляют количество единиц времени к данному моменту времени (объекту типа <code>Instant</code>) или промежутку времени (объекту типа <code>Duration</code>)
<code>minusNanos, minusMillis,</code> <code>minusSeconds,</code> <code>minusMinutes,</code> <code>minusHours, minusDays</code>	Вычтывают количество единиц времени из данного момента времени (объекта типа <code>Instant</code>) или промежутка времени (объекта типа <code>Duration</code>)
<code>multipliedBy, dividedBy,</code> <code>negated</code>	Возвращают промежуток времени, получаемый умножением или делением данного промежутка времени (объекта типа <code>Duration</code>) на заданное значение типа <code>long</code> или на <code>-1</code> . Следует иметь в виду, что масштабировать можно только промежутки, но не моменты времени
<code>isZero, isNegative</code>	Проверяют, является ли промежуток времени (объект типа <code>Duration</code>) нулевым или отрицательным

Так, если требуется проверить, является ли один алгоритм расчета времени более быстродействующим, чем другой, можно написать следующий код:

```
Duration timeElapsed2 = Duration.between(start2, end2);
boolean overTenTimesFaster =
    timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();
// или timeElapsed.toNanos() * 10 < timeElapsed2.toNanos()
```



НА ЗАМЕТКУ. Классы `Instant` и `Duration` являются неизменяемыми, а все их методы, например `multipliedBy()` или `minus()`, возвращают новый экземпляр.

Местные даты

А теперь перейдем от абсолютного времени к привычному для человека времени. Такое время представлено в новом прикладном программном интерфейсе Java API локальными датой и временем, а также поясным временем. Локальная дата и местное время обозначают дату и/или время суток, но не привязаны к часовому поясу. Примером локальной даты служит 14 июня 1903 года (день рождения Алонсо Черча – изобретателя лямбда-выражений). А поскольку такая дата не отражает ни время суток, ни часовой пояс, то она не соответствует точному моменту времени. С другой стороны, 16 июня 1969 года, 09:32:00 по восточному поясному времени (момент запуска космического корабля “Аполлон 11”) являются поясными датой и временем, обозначающими точный момент на временной шкале.

Для многих видов расчетов требуется указывать часовые пояса, но в некоторых случаях они служат помехой. Допустим, требуется запланировать встречу каждую неделю на 10:00. Если сложить 7 дней (т.е. $7 \times 24 \times 60 \times 60$ секунд) с последним поясным временем и, возможно, учесть переход на летнее время, то встреча может состояться на час раньше или позже!

Именно по этой причине разработчики рассматриваемого здесь прикладного программного интерфейса API рекомендуют не пользоваться поясным временем, если только не требуется действительно представить экземпляры абсолютного времени. Дни рождения, праздники, сроки исполнения и прочие события обычно лучше представить в виде локальных дат или времени.

Класс `LocalDate` представляет локальную дату с учетом года, месяца и дня месяца. Для составления локальной даты служат статические методы `now()` или `of()`, употребляемые следующим образом:

```
LocalDate today = LocalDate.now(); // сегодняшняя дата
LocalDate alonzoBirthday = LocalDate.of(1903, 6, 14);
alonzoBirthday = LocalDate.of(1903, Month.JUNE, 14);
// используется перечисление типа Month
```

В отличие от нестандартных соглашений, принятых в Unix и классе `java.util.Date`, где месяцы отсчитываются от нуля, а годы – от 1900 года, в данном случае предоставляются обычные числа для обозначения месяцев года. С другой стороны, можно воспользоваться перечислением типа `Month`. В табл. 5.2 представлены наиболее полезные методы для работы с объектами типа `LocalDate`.

Таблица 5.2. Методы из класса `LocalDate`

Метод	Описание
<code>now, of</code>	Эти статические методы составляют локальную дату (объект типа <code>LocalDate</code>), исходя из текущего времени или заданного года, месяца и дня
<code>plusDays, plusWeeks, plusMonths, plusYears</code>	Добавляют количество дней, недель, месяцев или лет к данной локальной дате (объекту типа <code>LocalDate</code>)
<code>minusDays, minusWeeks, minusMonths, minusYears</code>	Вычитают количество дней, недель, месяцев или лет из данной локальной даты (объекта типа <code>LocalDate</code>)
<code>plus, minus</code>	Добавляют промежуток или период времени (объект типа <code>Duration</code> или <code>Period</code>)

Окончание табл. 5.2

Метод	Описание
<code>withDayOfMonth,</code> <code>withDayOfYear,</code> <code>withMonth, withYear</code>	Возвращают новую локальную дату (объект типа <code>LocalDate</code>), значение которой изменено с учетом заданного дня месяца, дня года, месяца или года
<code>getDayOfMonth</code>	Получает день месяца (число в пределах от 1 до 31)
<code>getDayOfYear</code>	Получает день года (число в пределах от 1 до 366)
<code>getDayOfWeek</code>	Получает день недели, возвращая соответствующее значение из перечисления типа <code>DayOfWeek</code>
<code>getMonth, getMonthValue</code>	Получает месяц в виде значения из перечисления типа <code>Month</code> или в виде числа в пределах от 1 до 12
<code>getYear</code>	Получает год в виде числа в пределах от -999,999,999 до 999,999,999
<code>until</code>	Получает период времени между двумя датами в виде объекта типа <code>Period</code> или объекта типа <code>ChronoUnit</code> , обозначающего количество единиц измерения времени
<code>isBefore, isAfter</code>	Сравнивает данную локальную дату с другой локальной датой (объекты типа <code>LocalDate</code>)
<code>isLeapYear</code>	Возвращает логическое значение <code>true</code> , если год високосный, т.е. делится на 4 или на 400, но не на 100. Данный алгоритм применяется ко всем прошедшим годам, хотя исторически он неточен. (Високосные годы были введены в 45 году до н.э., а правила делимости на 100 или 400 — в 1582 году вместе с реформой григорианского календаря. Но то, чтобы охватить этой реформой весь мир, потребовалось 300 лет.)

Например, День программиста приходится на 256-й день года. Ниже показано, насколько просто рассчитать этот день.

```
LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);
// 13 сентября, но в високосный год это будет 12 сентября
```

Напомним, что разность между двумя моментами времени определяется промежутком времени, представленным объектом типа `Duration`. Его эквивалентом для локальных дат является период времени (объект типа `Period`), обозначающий количество прошедших лет, месяцев или дней. Так, для того чтобы получить день недели в следующем году, достаточно сделать следующий вызов: `birthday.plus(Period.ofYears(1))`. Разумеется, для этого достаточно сделать и такой вызов: `birthday.plusYears(1)`. Но вызов `birthday.plus(Duration.ofDays(365))` не даст правильного результата в високосный год.

Метод `until()` выдает разность между двумя локальными датами. Например, в приведенной ниже строке кода получается период времени 5 месяцев и 21 день.

```
independenceDay.until(christmas)
```

Но это не очень удобно, поскольку количество дней в месяце изменяется. Поэтому для определения количества дней лучше сделать следующий вызов:

```
independenceDay.until(christmas, ChronoUnit.DAYS) // 174 дня
```



ВНИМАНИЕ. Некоторые методы, приведенные в табл. 5.2, способны формировать несуществующие даты. Так, если добавить один месяц к дате 31 января, то получится дата 31 февраля. Вместо

того чтобы генерировать исключение, эти методы возвращают последний достоверный день месяца. Например, в обеих приведенных ниже строках кода получается одна и та же дата 29 февраля 2016 года.

```
LocalDate.of(2016, 1, 31).plusMonths(1)
и
LocalDate.of(2016, 3, 31).minusMonths(1)
```

Метод `getDayOfWeek()` выдает день недели в виде значения из перечисления типа `DayOfWeek`. В частности, значение `DayOfWeek.MONDAY` равно **1**, а значение `DayOfWeek.SUNDAY` — **7**. Например, следующий вызов дает **1**:

```
LocalDate.of(1900, 1, 1).getDayOfWeek().getValue()
```

В классе перечисления `DayOfWeek` имеются служебные методы `plus()` и `minus()` для вычисления дней недели по модулю **7**. Например, в результате вызова `DayOfWeek.SATURDAY.plus(3)` получается день недели `DayOfWeek.TUESDAY`, т.е. вторник.

 **НА ЗАМЕТКУ.** Выходные дни недели фактически приходятся на конец недели. Но совсем иначе дело обстоит в классе `java.util.Calendar`, где воскресенью соответствует числовое значение **1**, а субботе — числовое значение **7**.

Помимо класса `LocalDate`, для описания частных дат имеются также классы `MonthDay`, `YearMonth` и `Year`. Например, дата 25 декабря (без указания года) может быть представлена в виде дня месяца (объекта типа `MonthDay`).

Корректоры дат

Для применения в целях планирования нередко требуется рассчитывать даты вроде первого вторника каждого месяца. Для обычной коррекции дат в классе `TemporalAdjusters` предоставляется целый ряд статических методов. Результат коррекции даты с помощью соответствующего метода обычно передается методу `with()`. Например, дата первого вторника каждого месяца может быть рассчитана следующим образом:

```
LocalDate firstTuesday = LocalDate.of(year, month, 1).with(
    TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

Как всегда, метод `with()` возвращает новый объект типа `LocalDate` без модификации исходного объекта. Доступные методы коррекции дат приведены в табл. 5.3.

Таблица 5.3. Методы коррекции дат из класса `TemporalAdjusters`

Метод	Описание
<code>next(weekday)</code> , <code>previous(weekday)</code>	Возвращают следующую или предыдущую дату, приходящуюся на заданный день недели
<code>nextOrSame(weekday)</code> , <code>previousOrSame(weekday)</code>	Возвращают следующую или предыдущую дату, приходящуюся на заданный день недели, начиная с указанной даты

Окончание табл. 5.3

Метод	Описание
<code>dayOfWeekInMonth(n, weekday)</code>	Возвращает n-й день недели в месяце
<code>lastInMonth(weekday)</code>	Возвращает последний день недели в месяце
<code>firstDayOfMonth(), firstDayOfNextMonth(), firstDayOfNextYear(), lastDayOfMonth(), lastDayOfPreviousMonth(), lastDayOfYear()</code>	Возвращают дату, описанную в имени метода

Реализовав интерфейс TemporalAdjuster, можно сформировать свой корректор дат. В качестве примера ниже приведен корректор дат для расчета следующего дня недели.

```
TemporalAdjuster NEXT_WORKDAY = w -> {
    LocalDate result = (LocalDate) w;
    do {
        result = result.plusDays(1);
    } while (result.getDayOfWeek().getValue() >= 6);
    return result;
};

LocalDate backToWork = today.with(NEXT_WORKDAY);
```

Следует иметь в виду, что параметр лямбда-выражения относится к типу Temporal и должен быть приведен к типу LocalDate. Этого приведения типов можно избежать с помощью метода ofDateAdjuster(), ожидающего лямбда-выражение типа UnaryOperator<LocalDate> в качестве своего аргумента, как показано ниже.

```
TemporalAdjuster NEXT_WORKDAY = TemporalAdjusters.ofDateAdjuster(w -> {
    LocalDate result = w; // без приведения типов
    do {
        result = result.plusDays(1);
    } while (result.getDayOfWeek().getValue() >= 6);
    return result;
});
```

Местное время

Класс LocalTime представляет время дня, например 15:30:00. Экземпляр этого класса можно получить с помощью метода now() или of() следующим образом:

```
LocalTime rightNow = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30); // или LocalTime.of(22, 30, 0)
```

В табл. 5.4 перечислены типичные операции над местным временем и соответствующие им методы. В частности, операции plus и minus охватывают 24 часа в сутки. В приведенном ниже примере устанавливается местное время подъема.

```
LocalTime wakeup = bedtime.plusHours(8); // подъем в 6:30:00
```

Таблица 5.4. Методы из класса LocalTime

Метод	Описание
<code>now, of</code>	Эти статические методы составляют местное время (объект типа <code>LocalTime</code>), исходя из текущего времени или заданного количества часов и минут, а дополнительно — секунд и наносекунд
<code>minusHours, minusMinutes, minusSeconds, minusNanos</code>	Добавляют количество часов, минут, секунд или наносекунд к данному местному времени (объекту типа <code>LocalTime</code>)
<code>minusHours, minusMinutes, minusSeconds, minusNanos</code>	Вычитают количество часов, минут, секунд или наносекунд из данного местного времени (объекта типа <code>LocalTime</code>)
<code>plus, minus</code>	Добавляют промежуток времени (объект типа <code>Duration</code>)
<code>withHour, withMinute, withSecond, withNano</code>	Возвращают новое местное время (объект типа <code>LocalTime</code>), значение которого изменено с учетом заданного количества часов, минут, секунд или наносекунд
<code>getHour, getMinute, getSecond, getNano</code>	Получают количество часов, минут, секунд или наносекунд данного местного времени (объекта типа <code>LocalTime</code>)
<code>toSecondOfDay, toNanoOfDay</code>	Возвращают количество часов, минут, секунд или наносекунд от полуночи до данного местного времени (объекта типа <code>LocalTime</code>)
<code>isBefore, isAfter</code>	Сравнивают данное местное время с другим местным временем (объекты типа <code>LocalTime</code>)



НА ЗАМЕТКУ. В классе `LocalTime` время до полудня и пополудни не учитывается. Эта неприятная обязанность возлагается на средство форматирования (см. раздел “Форматирование и синтаксический анализ даты и времени” далее в этой главе).

Имеется также класс `LocalDateTime`, представляющий дату и время. Этот класс пригоден для хранения моментов времени в фиксированном часовом поясе, например, для планирования учебных занятий или событий. Но если требуется произвести расчеты с учетом перехода на летнее время или с учетом пользователей, находящихся в разных часовых поясах, то следует воспользоваться рассматриваемым далее классом `ZonedDateTime`.

Поясное время

Часовые пояса еще больше усложняют расчет времени, чем нерегулярность вращения Земли, поскольку они полностью являются изобретением человека. На практике отсчет времени производится от времени по Гринвичу, а следовательно, одни люди едят свой завтрак в 02:00, а другие — в 22:00, т.е. с разницей во времени на четыре часовых пояса, хотя это никак не сказывается на их пищеварении. И к этому еще добавляется переход на летнее время.

Какими бы причудливыми ни казались часовые пояса просвещенным, они являются фактом реальности, с которым приходится считаться. Так, разрабатывая календарное приложение, нужно принимать во внимание интересы тех пользователей, которые перемещаются из одного конца света в другой. Если вас пригласили на конференцию, которая должна состояться в Нью-Йорке в 10:00, а вы находитесь в Берлине, то вас должны предупредить о том, что начало конференции указано по местному времени.

Агентство по распределению номеров Интернета (IANA – Internet Assigned Numbers Authority) ведет базу данных всех известных в мире часовых поясов (<https://www.iana.org/time-zones>), которая обновляется несколько раз в год. Ее массовые обновления связаны с изменением правил перехода на летнее время. База данных IANA используется в Java.

Каждому часовому поясу присваивается свой идентификатор, например America/New_York или Europe/Berlin. Для выяснения имеющихся часовых поясов вызывается метод ZoneId.getAvailableIds(). На момент написания этой книги насчитывалось около 600 идентификаторов часовых поясов.

Получая идентификатор часового пояса, статический метод ZoneId.of(id) выдает объект типа ZoneId. Этот объект можно использовать для преобразования объекта типа LocalDateTime в объект типа ZonedDateTime в результате вызова метода local.atZone(zoneId). С другой стороны, объект типа ZonedDateTime можно построить, вызвав статический метод ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId), как показано ниже.

```
ZonedDateTime apollo11Launch = ZonedDateTime.of(
    1969, 7, 16, 9, 32, 0, 0, ZoneId.of("America/New_York"));
// 1969-07-16T09:32-04:00[America/New_York]
```

Это особый момент времени. Для получения объекта типа Instant вызывается метод apollo11Launch.toInstant(). С другой стороны, если имеется момент времени, то для получения объекта типа ZonedDateTime, обозначающего время на уровне Гринвичской королевской обсерватории, следует вызвать метод instant.atZone(ZoneId.of("UTC")) или же воспользоваться другим объектом ZoneId, чтобы получить момент времени в каком-нибудь другом конце света.



НА ЗАМЕТКУ. Сокращение UTC обозначает “всемирное скоординированное время” и выбрано в качестве компромисса между английским и французским названиями одного и того же понятия. UTC – это время на уровне Гринвичской королевской обсерватории без учета перехода на летнее время.

Методы из класса ZonedDateTime практически такие же, как и в классе LocalDateTime (табл. 5.5). Большинство из них просты, но переход на летнее время несколько усложняет их применение.

Таблица 5.5. Методы из класса ZonedDateTime

Метод	Описание
now, of, ofInstant	Эти статические методы строят объект типа ZonedDateTime из текущего времени или года, месяца, дня, количества часов, минут, секунд, наносекунд (т.е. объектов типа LocalDate и LocalTime) и часового пояса (объекта типа ZoneId) или же из объектов типа Instant и ZoneId
plusDays, plusWeeks, plusMonths, plusYears, plusHours, plusMinutes, plusSeconds, plusNanos	Добавляют количество единиц измерения времени к данном объекту типа ZonedDateTime

Метод	Описание
<code>minusDays, minusWeeks, minusMonths, minusYears, minusHours, minusMinutes, minusSeconds, minusNanos</code>	Вычитают количество единиц измерения времени из данного объекта типа <code>ZonedDateTime</code>
<code>plus, minus</code>	Добавляют промежуток или период времени (объект типа <code>Duration</code> или <code>Period</code>)
<code>withDayOfMonth, withDayOfYear, withMonth, withYear, withHour, withMinute, withSecond, withNano</code>	Возвращают новый объект типа <code>ZonedDateTime</code> , в котором значение одной из единиц измерения времени изменено на заданное
<code>withZoneSameInstant, withZoneSameLocal</code>	Возвращают новый объект типа <code>ZonedDateTime</code> в заданном часовом поясе, чтобы представить один и тот же момент времени или то же самое местное время
<code>getDayOfMonth</code>	Получает день месяца (число в пределах от 1 до 31)
<code>getDayOfYear</code>	Получает день года (число в пределах от 1 до 366)
<code>getDayOfWeek</code>	Получает день недели, возвращая соответствующее значение из перечисления типа <code>DayOfWeek</code>
<code>getMonth, getMonthValue</code>	Получает месяц в виде значения из перечисления типа <code>Month</code> или в виде числа в пределах от 1 до 12
<code>getYear</code>	Получает год в виде числа в пределах от -999,999,999 до 999,999,999
<code>getHour, getMinute, getSecond, getNano</code>	Получают количество минут, секунд или наносекунд в данном объекте типа <code>ZonedDateTime</code>
<code>getOffset</code>	Получает смещение относительно универсального времени UTC в виде экземпляра типа <code>ZoneOffset</code> . Смещения могут изменяться в пределах от -12:00 до +14:00. У некоторых часовых поясов имеются дробные смещения. При переходе на летнее время смещения изменяются
<code>toLocalDate, toLocalTime, toInstant</code>	Получают локальную дату, местное время или соответствующий момент времени
<code>isBefore, isAfter</code>	Сравнивают объекты типа <code>ZonedDateTime</code> друг с другом

При переходе на летнее время часы переводятся на один час вперед. Что же произойдет, если сформировать момент времени, приходящийся на пропущенный час? Например, в 2013 году переход на летнее время в Центральной Европе произошел 31 марта в 2:00. Если попытаться сформировать несуществующий момент времени 31 марта в 2:30, то на самом деле будет получено время 3:30, как показано ниже.

```
ZonedDateTime skipped = ZonedDateTime.of(
    LocalDate.of(2013, 3, 31),
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
// формирует момент времени 31 марта в 3:30
```

С другой стороны, при переходе на зимнее время часы переводятся на один час назад, и в одно и то же локальное время оказываются два момента времени! В итоге при формировании момента времени в этом промежутке получается более ранний из двух моментов времени. Час спустя количество часов и минут, определяющих текущий момент времени, остается тем же самым, но смещение часового пояса изменяется, как показано ниже.

```
ZonedDateTime ambiguous = ZonedDateTime.of(
    LocalDate.of(2013, 10, 27), // переход на зимнее время
    LocalTime.of(2, 30),
    ZoneId.of("Europe/Berlin"));
// 2013-10-27T02:30+02:00[Europe/Berlin]
ZonedDateTime anHourLater = ambiguous.plusHours(1);
// 2013-10-27T02:30+01:00[Europe/Berlin]
```

Следует также обратить внимание на коррекцию даты при переходе на летнее и зимнее время. Так, если назначается встреча на следующую неделю, то семидневный промежуток добавлять не следует:

```
ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));
// Внимание! Этот код не годится, поскольку в нем
// не учитывается переход на летнее время
```

Вместо этого нужно воспользоваться классом `Period` следующим образом:

```
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // Годится!
```



ВНИМАНИЕ. Имеется также класс `OffsetDateTime`, представляющий моменты времени со смещением относительно универсального времени UTC, но без соблюдения правил смены часовых поясов. Этот класс предназначен для специального применения, не требующего соблюдения упомянутых правил, например, в некоторых сетевых протоколах. А для расчета времени, привычного для человека, следует пользоваться классом `ZonedDateTime`.

Форматирование и синтаксический анализ даты и времени

В классе `DateTimeFormatter` предоставляются следующие три разновидности средств форматирования для вывода значения даты и времени.

- Предопределенные стандартные средства форматирования (табл. 5.6).
- Средства форматирования с учетом региональных настроек.
- Средства форматирования со специальными шаблонами.

Таблица 5.6. Предопределенные средства форматирования

Средство форматирования	Описание	Пример
<code>BASIC_ISO_DATE</code>	Год, месяц, день, смещение часового пояса без разделителей	19690716-0500
<code>ISO_LOCAL_DATE</code> , <code>ISO_LOCAL_TIME</code> , <code>ISO_LOCAL_DATE_TIME</code>	Разделители - , :, T	1969-07-16, 09:32:00, 1969-07-16T09:32:00
<code>ISO_OFFSET_DATE</code> , <code>ISO_OFFSET_TIME</code> , <code>ISO_OFFSET_DATE_TIME</code>	Аналогично <code>ISO_LOCAL_XXX</code> , но без смещения часового пояса	1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00

Окончание табл. 5.6

Средство форматирования	Описание	Пример
<code>ISO_ZONED_DATE_TIME</code>	Со смещением и идентификатором часового пояса	<code>1969-07-16T09:32:00-05:00[America/New_York]</code>
<code>ISO_INSTANT</code>	Обозначается в формате UTC идентификатором Z часового пояса	<code>1969-07-16T14:32:00Z</code>
<code>ISO_DATE</code> , <code>ISO_TIME</code> , <code>ISO_DATE_TIME</code>	Аналогично <code>ISO_OFFSET_DATE</code> , <code>ISO_OFFSET_TIME</code> , <code>ISO_ZONED_DATE_TIME</code> , но сведения о часовом поясе указываются дополнительно	<code>1969-07-16-05:00, 09:32:00-05:00, 1969-07-16T09:32:00-05:00[America/New_York]</code>
<code>ISO_ORDINAL_DATE</code>	Год и день года для локальной даты (объекта типа <code>LocalDate</code>)	<code>1969-197</code>
<code>ISO_WEEK_DATE</code>	Год, неделя и день недели для локальной даты (объекта типа <code>LocalDate</code>)	<code>1969-W29-3</code>
<code>RFC_1123_DATE_TIME</code>	Стандарт для отметок времени, кодируемых в формате RFC 822 и обновляемых до четырех цифр года в формате RFC 1123	<code>Wed, 16 Jul 1969 09:32:00 -0500</code>

Для того чтобы воспользоваться одним из стандартных средств форматирования даты и времени, достаточно вызвать его метод `format()` следующим образом:

```
String formatted = DateTimeFormatter.ISO_DATE_TIME.format(apollo11Launch);
// 1969-07-16T09:32:00-05:00[America/New_York]
```

Стандартные средства форматирования предназначены, главным образом, для машиночитаемых меток времени. А для представления дат и времени в удобочитаемом для человека виде служат средства форматирования даты и времени с учетом региональных настроек. Для форматирования даты и времени с учетом региональных настроек имеются четыре стиля: `SHORT`, `MEDIUM`, `LONG` и `FULL` (табл. 5.7).

Таблица 5.7. Стили форматирования с учетом региональных настроек

Стиль	Дата	Время
<code>SHORT</code>	7/16/69	9:32 AM
<code>MEDIUM</code>	Jul 16, 1969	9:32:00 AM
<code>LONG</code>	July 16, 1969	9:32:00 AM EDT
<code>FULL</code>	Wednesday, July 16, 1969	9:32:00 AM EDT

Такие средства форматирования создаются статическими методами `ofLocalizedDate()`, `ofLocalizedTime()` и `ofLocalizedDateTime()`. Ниже приведен характерный тому пример.

```
DateTimeFormatter formatter =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);
String formatted = formatter.format(apollo11Launch);
// July 16, 1969 9:32:00 AM EDT
```

В этих методах применяются региональные настройки по умолчанию. Для смены региональных настроек достаточно вызвать метод `withLocale()`:

```
formatted = formatter.withLocale(Locale.FRENCH).format(apollo11Launch);
// 16 juillet 1969 09:32:00 EDT
```



НА ЗАМЕТКУ. Класс `java.time.format.DateTimeFormatter` предназначен для замены класса `java.util.DateFormat`. Если же требуется получить экземпляр класса `DateFormat` для обратной совместимости, то следует вызвать метод `formatter.toFormat()`.

И наконец, можно задействовать свой формат, указав шаблон. В приведенном ниже примере форматируется дата в виде **Wed 1969-07-16 09:32**.

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

Каждая буква обозначает отдельное поле для указания времени, а количество повторений этой буквы определяет конкретный формат по правилам, которые естественным, хотя и не совсем понятным образом сформировались с течением времени. В табл. 5.8 приведены наиболее полезные элементы шаблона форматирования.

Таблица 5.8. Наиболее употребительные обозначения в форматах даты и времени

Перечислимые константы типа <code>ChronoField</code> и прочее	Перевод	Примеры
<code>ERA</code>	Эра	<code>G: AD, GGGG: Anno Domini, GGGGG: A</code>
<code>YEAR_OF_ERA</code>	Год эры	<code>yy: 69, yyyy: 1969</code>
<code>MONTH_OF_YEAR</code>	Месяц года	<code>M: 7, MM: 07, MMM: Jul, MMMM: July, MMMMM: J</code>
<code>DAY_OF_MONTH</code>	День месяца	<code>d: 6, dd: 06</code>
<code>DAY_OF_WEEK</code>	День недели	<code>e: 3, E: Wed,EEE: Wednesday, EEEE: W</code>
<code>HOUR_OF_DAY</code>	Час суток	<code>H: 9, HH: 09</code>
<code>CLOCK_HOUR_OF_AM_PM</code>	Астрономический час до полудня и полополудни	<code>K: 9, KK: 09</code>
<code>AMPM_OF_DAY</code>	Время суток до полудня и полополудни	<code>a: AM</code>
<code>MINUTE_OF_HOUR</code>	Минута часа	<code>mm: 02</code>
<code>SECOND_OF_MINUTE</code>	Секунда минуты	<code>ss: 00</code>
<code>NANO_OF_SECOND</code>	Наносекунда секунды	<code>nnnnnn: 000000</code>
Идентификатор часового пояса		<code>VV: America/New_York</code>
Наименование часового пояса		<code>z: EDT, zzzz: Eastern Daylight Time</code>
Смещение часового пояса		<code>x: -04, xx: -0400, xxx: -04:00, XXX: то же самое, но вместо нуля употребляется буква Z</code>
Локализованное смещение часового пояса		<code>O: GMT-4, OOOO: GMT-04:00</code>

Для синтаксического анализа значения даты и времени из символьной строки служит один из специально предназначенных для этого статических методов. Сначала в

приведенном ниже примере кода применяется стандартное средство форматирования **ISO_LOCAL_DATE**, а затем одно из специальных средств форматирования.

```
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");
ZonedDateTime apollo11Launch =
    ZonedDateTime.parse("1969-07-16 03:32:00-0400",
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));
```

Взаимодействие с устаревшим кодом

Вновь разработанному в Java прикладному программному интерфейсу API даты и времени придется каким-то образом взаимодействовать с уже существующими классами и, в частности, с общеупотребительными классами `java.util.Date`, `java.util.GregorianCalendar` и `java.sql.Date/Time/Timestamp`. Класс `Instant` является близким аналогом класса `java.util.Date`. В версии Java 8 в этот класс внедрены два метода. Во-первых, метод `toInstant()`, преобразующий объект типа `Date` в объект типа `Instant`, а во-вторых, статический метод `from()`, выполняющий преобразование в обратном направлении.

Аналогично класс `ZonedDateTime` является близким аналогом класса `java.util.GregorianCalendar`, в котором появились методы преобразования, внедренные в версии Java 8. В частности, метод `toZonedDateTime()` преобразует объект типа `GregorianCalendar` в объект типа `ZonedDateTime`, а статический метод `from()` выполняет преобразование в обратном направлении.

Еще один ряд преобразований имеется для классов даты и времени из пакета `java.sql`. Кроме того, объект типа `DateTimeFormatter` может быть передан устаревшему коду, в котором используется класс `java.text.Format`. Эти преобразования перечислены в табл. 5.9.

Таблица 5.9. Преобразования классов из пакета `java.time` в устаревшие классы

Класс	Преобразование в устаревший класс	Преобразование из устаревшего класса
<code>Instant</code> ↔ <code>java.util.Date</code>	<code>Date.from(instant)</code>	<code>date.toInstant()</code>
<code>ZonedDateTime</code> ↔ <code>java.util.GregorianCalendar</code>	<code>GregorianCalendar.from(zonedDateTime)</code>	<code>cal.toZonedDateTime()</code>
<code>Instant</code> ↔ <code>java.sql.Timestamp</code>	<code>TimeStamp.from(instant)</code>	<code>timestamp.toInstant()</code>
<code>LocalDateTime</code> ↔ <code>java.sql.Timestamp</code>	<code>Timestamp.valueOf(localDateTime)</code>	<code>timeStamp.toLocalDateTime()</code>
<code>LocalDate</code> ↔ <code>java.sql.Date</code>	<code>Date.valueOf(localDate)</code>	<code>date.toLocalDate()</code>
<code>LocalTime</code> ↔ <code>java.sql.Time</code>	<code>Time.valueOf(localTime)</code>	<code>time.toLocalTime()</code>
<code>DateTimeFormatter</code> → <code>java.text.DateFormat</code>	<code>formatter.toFormat()</code>	Отсутствует
<code>java.util.TimeZone</code> → <code>ZoneId</code>	<code>Timezone.getTimeZone(id)</code>	<code>timeZone.toZoneId()</code>
<code>java.nio.file.attribute.FileTime</code> → <code>Instant</code>	<code>FileTime.from(instant)</code>	<code>fileTime.toInstant()</code>

Упражнения

1. Рассчитайте дату Дня программиста, не прибегая к методу `plusDays()`.
2. Что получится, если добавить один год, четыре года или четыре раза по одному году к локальной дате, получаемой в результате вызова метода `LocalDate.of(2000, 2, 29)`?
3. Реализуйте метод `next()`, принимающий функциональный интерфейс `Predicate<LocalDate>` и возвращающий корректор дат, который выдает следующую дату, удовлетворяющую предикату. Например, в приведенной ниже строке кода рассчитывается следующий рабочий день.
`today.with(next(w -> getDayOfWeek().getValue() < 6))`
4. Напишите эквивалент команды `cal`, отображающей в Unix календарь на месяц. Например, по команде `java Cal 3 2013` должен быть показан приведенный ниже календарь, где 1 марта 2013 года приходится на пятницу. (Обозначьте выходные дни в конце недели.)

	1	2	3			
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

5. Напишите программу, выводящую количество дней, которые вы прожили.
6. Напишите программу, перечисляющую все пятницы, которые пришлись на 13-е число в XX веке.
7. Реализуйте класс `TimeInterval`, представляющий промежуток времени, подходящий для календарных событий (например, встречи, намеченной на указанную дату с 10:00 до 11:00). Предоставьте метод для проверки перекрытия двух промежутков времени.
8. Напишите программу, получающую смещения нынешней даты во всех поддерживаемых часовых поясах для текущего момента времени, преобразовав результат вызова метода `ZoneId.getAvailableIds()` в поток ввода-вывода и воспользовавшись потоковыми операциями.
9. Выявите с помощью потоковых операций все часовые пояса со смещениями не на целые часы.
10. Самолет авиарейсом из Лос-Анджелеса во Франкфурт отправляется в полет в 3:05 пополудни местного времени, находится в полете 10 часов 50 минут. Когда он прибывает во Франкфурт? Напишите программу, выполняющую подобные расчеты времени.
11. Самолет обратным авиарейсом из Франкфурта отправляется в полет в 14:05 и прибывает в Лос-Анджелес в 16:40. Сколько времени длится полет? Напишите программу, выполняющую подобные расчеты времени.
12. Напишите программу, устраняющую затруднение, описанное в начале раздела “Поясное время” ранее в этой главе. Эта программа должна выбирать назначенные встречи в разных часовых поясах и предупреждать пользователя о тех встречах, которые должны состояться в следующий час по местному времени.

Глава

6

Усовершенствования параллелизма

В этой главе...

- Атомарные значения
- Усовершенствования в классе ConcurrentHashMap
- Параллельные операции с массивами
- Завершаемые будущие действия
- Упражнения

Программировать параллельные операции нелегко, но делать это еще труднее без подходящих инструментальных средств. В первых версиях Java имелась минимальная поддержка параллелизма, и поэтому программистам приходилось усердно бороться с взаимоблокировками и состояниями гонок в создаваемом прикладном коде. Надежный пакет `java.util.concurrent` для поддержки параллелизма появился только в версии Java 5. В этот пакет вошли потокобезопасные коллекции и пулы потоков исполнения, позволявшие многим разработчикам писать параллельно выполняющиеся прикладные программы, не прибегая к блокировкам и не запуская потоки на исполнение. К сожалению, пакет `java.util.concurrent` составлен из утилит, полезных для прикладного программирования, а также инструментальных средств, пригодных для написания библиотек, но одни из них в недостаточной степени отделены от других в этом пакете. В этой главе основное внимание уделяется конкретным потребностям разработчиков прикладных программ в отношении параллелизма.

В этой главе рассматриваются следующие основные вопросы.

- Упрощение обновления атомарных переменных с помощью методов `updateAndGet()` и `accumulateAndGet()`.
- Повышенная эффективность классов `LongAccumulator` и `DoubleAccumulator` по сравнению с классами `AtomicLong` и `AtomicDouble` в конфликтной ситуации.
- Упрощение обновления элементов коллекции типа `ConcurrentHashMap` благодаря методам `compute()` и `merge()`.
- Внедрение групповых операций `search`, `reduce`, `forEach` в разных вариантах в класс `ConcurrentHashMap` для обращения с ключами, значениями, ключами и значениями, а также с записями.
- Применение класса `ConcurrentHashMap` вместо класса `Set` благодаря представлению множества.
- Внедрение в класс `Arrays` методов для параллельной сортировки, заполнения и выполнения префиксных операций.
- Составление асинхронных операций с использованием завершаемых будущих действий.

Атомарные значения

В версии Java 5 появился пакет `java.util.concurrent.atomic`, в котором представляются классы свободной от блокировок модификации переменных. Например, последовательность чисел можно безопасно сформировать следующим образом:

```
public static AtomicLong nextNumber = new AtomicLong();
// выполняется в некотором потоке . . .
long id = nextNumber.incrementAndGet();
```

Метод `incrementAndGet()` атомарно инкрементирует объект типа `AtomicLong` и возвращает значение после приращения. Это означает, что операции для получения конкретного значения, сложения с 1, установки и получения нового значения не могут быть прерваны. При этом гарантируется, что вычисляется и возвращается правильное значение, даже если к одному и тому же экземпляру осуществляется параллельный доступ из нескольких потоков исполнения.

Имеются методы для атомарной установки, добавления и вычитания значений, но если требуется более сложное обновление, то придется вызвать метод `compareAndSet()`. Допустим, требуется отслеживать наибольшее значение одновременно в разных потоках исполнения. Следующий код не сработает:

```
public static AtomicLong largest = new AtomicLong();
    // выполняется в некотором потоке . .
largest.set(Math.max(largest.get(), observed));
    // Ошибка: состояние гонок!
```

Такое обновление не является атомарным. Вместо этого следует вычислить новое значение и вызвать метод `compareAndSet()` в цикле, как показано ниже.

```
do {
    oldValue = largest.get();
    newValue = Math.max(oldValue, observed);
} while (!largest.compareAndSet(oldValue, newValue));
```

Если объект `largest` обновляется в другом потоке исполнения, то вполне возможно, что этот поток выиграл состязание за доступ к данному объекту. В таком случае метод `compareAndSet()` возвратит логическое значение `false`, не устанавливая новое значение, а в цикле будет предпринята очередная попытка прочитать обновленное значение и изменить его. В конечном итоге замена существующего значения на новое будет выполнена успешно. И хотя такой прием кажется трудоемким, тем не менее метод `compareAndSet()` назначает операцию процессора, которая выполняется быстрее, чем при использовании блокировки потоков исполнения.

В версии Java 8 организовывать шаблонный цикл больше не нужно. Вместо этого для обновления конкретного значения достаточно предоставить лямбда-выражение, и тогда обновление этого значения произойдет автоматически. Так, в рассматриваемом здесь примере можно сделать вызов

```
largest.updateAndGet(x -> Math.max(x, observed));
```

или

```
largest.accumulateAndGet(observed, Math::max);
```

Метод `accumulateAndGet()` принимает в качестве своего аргумента двоичный оператор для объединения атомарного значения с предоставляемым аргументом. Имеются также методы `getAndUpdate()` и `getAndAccumulate()`, возвращающие прежнее значение.



НА ЗАМЕТКУ. Эти методы предоставляются также для классов `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray` и `AtomicReferenceFieldUpdater`.

Если имеется большое количество потоков, из которых осуществляется одновременный доступ к одним и тем же атомарным значениям, это отрицательно сказывается на производительности, поскольку для оптимистических обновлений требуется слишком много попыток. Для разрешения подобного затруднения в версии Java 8 предоставляются классы `LongAdder` и `LongAccumulator`. В частности, класс `LongAdder`

состоит из многих переменных, суммарное значение которых оказывается текущим. В нескольких потоках исполнения могут обновляться разные слагаемые, а новые слагаемые автоматически предстаются при увеличении количества потоков исполнения. Это достаточно эффективный способ в обычной ситуации, когда значение суммы не требуется до тех пор, пока будет выполнена вся работа. При этом можно добиться значительного повышения производительности (см. упражнение 3 в конце главы).

Если предполагается высокая степень состязания среди потоков исполнения за доступ к данным, то вместо класса `AtomicLong` следует просто использовать класс `LongAdder`. Методы в этом классе именуются несколько иначе. Так, для приращения счетчика вызывается метод `increment()`, сложения величин — метод `add()`, а для получения итоговой суммы — метод `sum()`. Ниже приведен характерный тому пример.

```
final LongAdder adder = new LongAdder();
for (...)

    pool.submit(() -> {
        while (...) {
            ...
            if (...) adder.increment();
        }
    });
...
long total = adder.sum();
```



НА ЗАМЕТКУ. Разумеется, метод `increment()` не возвращает прежнее значение. Ведь если сделать это, то будут сведены на нет все преимущества повышения производительности благодаря разбиению суммы на несколько слагаемых.

Принцип выполнения операции произвольного накопления данных обобщен в классе `LongAccumulator`. В конструкторе этого класса предоставляется сама операция, а также ее нейтральный элемент. Для внедрения новых значений вызывается метод `accumulate()`, а для получения текущего значения — метод `get()`. Так, приведенный ниже пример кода дает такой же результат, как и предыдущий пример, в котором используется класс `LongAdder`.

```
LongAccumulator adder = new LongAccumulator(Long::sum, 0);
// выполняется в некотором потоке . . .
adder.accumulate(value);
```

В аккумуляторе имеются переменные a_1, a_2, \dots, a_n . Каждая переменная инициализируется нейтральным элементом (в данном примере — 0). Когда метод `accumulate()` вызывается со значением v , одна из переменных автоматически обновляется следующим образом: $a_i = a_i \text{ op } v$, где op — операция накопления, записанная в инфиксной форме. В рассматриваемом здесь примере при вызове метода `accumulate()` вычисляется выражение $a_i = a_i + v$ для некоторой величины i . При вызове метода `get()` получается следующий результат: $a_1 \text{ op } a_2 \text{ op } \dots \text{ op } a_n$. В данном случае это сумма значений, накопленных в переменных следующим образом: $a_1 + a_2 + \dots + a_n$.

Если выбрать другую операцию, то можно вычислить максимальное или минимальное значение (см. упражнение 4 в конце этой главы). В общем, такая операция должна быть ассоциативной и коммутативной. Это означает, что ее конечный результат не должен зависеть от порядка, в котором объединяются промежуточные

значения. Имеются также классы DoubleAdder и DoubleAccumulator, действующие аналогичным образом, но только со значениями типа double.



НА ЗАМЕТКУ. Еще одним полезным дополнением версии Java 8 является класс StampedLock, предназначенный для реализации оптимистичных операций чтения данных. В целом пользоваться блокировками при разработке прикладных программ не рекомендуется, тем не менее это можно сделать следующим образом. Сначала вызывается метод tryOptimisticRead(), в результате чего получается некоторый "отпечаток". Затем читаются данные и проверяется достоверность этого отпечатка (т.е. тот факт, что никакой другой поток исполнения не получил блокировку на запись данных). Если отпечаток все еще достоверный, то можно благополучно пользоваться считанными данными. В противном случае следует получить блокировку на чтение данных, чтобы заблокировать их запись в любых других потоках исполнения. В приведенном ниже примере показано, как все это реализуется непосредственно в коде.

```
public class Vector {
    private int size;
    private Object[] elements;
    private StampedLock lock = new StampedLock();

    public Object get(int n) {
        long stamp = lock.tryOptimisticRead();
        Object[] currentElements = elements;
        int currentSize = size;
        if (!lock.validate(stamp)) {
            // какой-то другой поток получил блокировку на запись
            stamp = lock.readLock(); // получить пессимистическую блокировку
            currentElements = elements;
            currentSize = size;
            lock.unlockRead(stamp);
        }
        return n < currentSize ? currentElements[n] : null;
    }
    ...
}
```

Усовершенствования в классе ConcurrentHashMap

В классическом программировании бытует следующее мнение: "Если вам доступна только одна структура данных, сделайте из нее хеш-таблицу". Начиная с версии Java 5, класс ConcurrentHashMap стал основополагающим компонентом параллельного программирования. Разумеется, класс ConcurrentHashMap является потокобезопасным, т.е. элементы могут добавляться и удаляться в нескольких потоках исполнения без ущерба для внутренней структуры. Кроме того, класс ConcurrentHashMap является довольно эффективным, позволяя параллельно обновлять отдельные части таблицы в нескольких потоках исполнения без их взаимной блокировки.



НА ЗАМЕТКУ. В некоторых приложениях применяются параллельные хеш-отображения, причем настолько огромные, что определить их размеры с помощью метода size() невозможно, поскольку он возвращает значение типа int. Что же в таком случае делать с отображением, насчитывающим больше миллиарда записей? С этой целью в версии Java 8 был внедрен метод mappingCount(), возвращающий размер отображения в виде значения типа long.



НА ЗАМЕТКУ. Все записи в хеш-отображении хранятся с одинаковым хеш-кодом в одной и той же области памяти, как в корзине. В некоторых приложениях применяются неудачные хеш-функции, и в конечном итоге все записи оказываются в небольшом количестве областей памяти, что заметно снижает производительность. И тогда применение даже обоснованных в общем хеш-функций, например, из класса `String`, может оказаться проблематичным. Например, совершающий атаку злоумышленник способен замедлить выполнение программы, подделав большое количество символьных строк, хеширование которых приводит к одному и тому же значению. Начиная с версии Java 8, области памяти, выделяемые под параллельное хеш-отображение, организуются в виде деревьев, а не списков, когда тип ключа реализует интерфейс `Comparable`, гарантируя производительность $O(\log(n))$.

Обновление значений

В исходной версии класса `ConcurrentHashMap` имелось лишь несколько методов для атомарных обновлений, что несколько затрудняло программирование. Допустим, требуется подсчитать, насколько часто употребляются некоторые средства. В качестве примера допустим, что требуется подсчитать частоту употребления слов в нескольких потоках исполнения.

Можно ли воспользоваться для этой цели хеш-отображением типа `ConcurrentHashMap<String, Long>`? Рассмотрим приведенный ниже пример кода для приращения счетчика. Очевидно, что такой код не является потокобезопасным. Ведь одновременно в другом потоке исполнения может быть обновлен тот же самый подсчет.

```
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue);
// Ошибка: значение oldValue может быть и не заменено
```



НА ЗАМЕТКУ. Некоторых программистов удивляет, что потокобезопасная предположительно структура данных допускает выполнение операций, которые не являются потокобезопасными. Но это приводит к двум совершенно противоположным выводам. Если обычное хеш-множество типа `HashMap` модифицируется в нескольких потоках исполнения, то они способны разрушить его внутреннюю структуру (массив связных списков). При этом некоторые связи могут исчезнуть или даже зациклиться, сделав структуру данных непригодной к употреблению. Ничего подобного вообще не может произойти с хеш-отображением типа `ConcurrentHashMap`. В рассмотренном выше примере код получения и установки значений не в состоянии нарушить структуру данных. Но поскольку последовательность операции не является атомарной, то результат их выполнения непредсказуем.

В качестве выхода из данного положения можно, в частности, воспользоваться операцией `replace`, в ходе которой известное прежнее значение заменяется новым, как было показано в предыдущем разделе и демонстрируется ниже.

```
do {
    oldValue = map.get(word);
    newValue = oldValue == null ? 1 : oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

С другой стороны, можно воспользоваться хеш-отображением типа `ConcurrentHashMap<String, AtomicLong>` или, начиная с версии Java 8, хеш-отображением типа `ConcurrentHashMap<String, LongAdder>`. И тогда код обновления будет выглядеть следующим образом:

```
map.putIfAbsent(word, new LongAdder());
map.get(word).increment();
```

В первой строке данного кода гарантируется наличие объекта типа `LongAdder`, который можно инкрементировать автоматически. А поскольку метод `putIfAbsent()` возвращает отображаемое значение (уже существующее или вновь установленное), то обе приведенные выше строки кода можно объединить в одну следующим образом:

```
map.putIfAbsent(word, new LongAdder()).increment();
```

В версии Java 8 предоставляются методы, позволяющие сделать более удобными автоматические обновления. В частности, метод `compute()` вызывается с ключом и функцией для вычисления нового значения. Эта функция получает ключ и связанное с ним значение, а в его отсутствие — пустое значение `null`, и вычисляет новое значение. В качестве примера ниже показано, каким образом обновляется отображение целочисленных счетчиков.

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```



НА ЗАМЕТКУ. В хеш-отображении типа `ConcurrentHashMap` пустые значения не допускаются. Тем не менее имеется немало методов, в которых пустые значения используются для того, чтобы обозначить отсутствие заданного ключа в отображении.

Имеются также разновидности методов `computeIfPresent()` и `computeIfAbsent()`, вычисляющих новое значение только в том случае, если прежнее значение уже имеется или еще отсутствует. Отображение счетчиков типа `LongAdder` может быть обновлено следующим образом:

```
map.computeIfAbsent(word, k -> new LongAdder()).increment();
```

Это очень похоже на вызов упоминавшегося ранее метода `putIfAbsent()`. Но на этот раз конструктор класса `LongAdder` вызывается только в том случае, если действительно требуется новый счетчик.

При вводе ключа в отображение обычно требуется сделать что-нибудь особенное. И для этой цели очень удобным оказывается метод `merge()`. В качестве своего параметра он принимает исходное значение, используемое в отсутствие ключа. В противном случае вызывается дополнительно предоставляемая функция, объединяющая существующее значение с исходным. (В отличие от метода `compute()`, эта функция не обрабатывает ключ.) Ниже приведены характерные примеры вызова метода `merge()`.

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue + newValue);
```

или просто

```
map.merge(word, 1L, Long::sum);
```

Короче просто не бывает. Еще один примечательный пример применения метода `merge()` демонстрируется в упражнении 5 в конце этой главы.



НА ЗАМЕТКУ. Если функция, передаваемая для вычисления или объединения значений, возвращает пустое значение `null`, то существующее значение удаляется из отображения.



ВНИМАНИЕ. Пользуясь методом `compute()` или `merge()`, имейте в виду, что предоставляемая вами функция не должна выполнять большой объем работы. Ведь во время выполнения этой функции в одном потоке обновления иных частей того же самого отображения могут быть заблокированы в другом потоке. Безусловно, данная функция, в свою очередь, не должна обновлять другие части отображения.

Групповые операции

В версии Java 8 предоставляются групповые операции над параллельными хеш-отображениями, которые можно надежно выполнять даже в том случае, когда доступ к одному и тому же отображению осуществляется из разных потоков исполнения. В ходе групповых операций производится обход отображения и обработка обнаруженных по ходу элементов. При этом не требуется фиксировать моментальный снимок отображения во времени. Результат выполнения групповой операции следует рассматривать как некоторое приближение истинного состояния данного отображения, кроме тех случаев, когда становится известно, что отображение не модифицируется во время выполнения групповой операции.

Имеются три разновидности групповых операций:

- **search** — применяет функцию к каждому ключу и/или значению до тех пор, пока функция не выдаст непустой результат. В таком случае поиск прекращается, а функция возвращает полученный результат.
- **reduce** — объединяет все ключи и/или значения, используя предоставляемую функцию накопления.
- **forEach** — применяет функцию ко всем ключам и/или значениям.

Для каждой групповой операции предусмотрены четыре варианта:

- **операцияKeys**: оперирует ключами.
- **операцияValues**: оперирует значениями.
- **операция**: оперирует ключами и значениями.
- **операцияEntries**: оперирует объектами типа `Map.Entry`.

Вместе с каждой операцией нужно указывать *порог параллелизма*. Так, если отображение содержит больше элементов, чем этот порог, групповая операция распараллеливается. Если групповую операцию требуется выполнять в одном потоке, то для нее следует указать порог `Long.MAX_VALUE`. А если для выполнения групповой операции требуется сделать доступным максимальное количество потоков исполнения, то для нее следует указать порог 1.

Рассмотрим сначала методы типа `search()`. Ниже приведены их разновидности.

```
U searchKeys(long threshold, BiFunction<? super K, ? extends U> f)
U searchValues(long threshold, BiFunction<? super V, ? extends U> f)
```

```
U search(long threshold, BiFunction<? super K, ? super V, ? extends U> f)
U searchEntries(long threshold, BiFunction<Map.Entry<K, V>, ? extends U> f)
```

Допустим, требуется найти первое слово, встречающееся больше 1000 раз. Для такого поиска потребуются ключи и значения, как показано ниже.

```
String result = map.search(threshold, (k, v) -> v > 1000 ? k : null);
```

Далее в переменной `result` устанавливается первое совпадение искомого слова или пустое значение `null`, если функция поиска возвращает пустое значение `null` для всех вводимых данных. У методов типа `forEach()` имеются две разновидности. В одной из них просто применяется функция *потребителя* для каждой записи в отображении, как показано в приведенном ниже примере.

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

Во второй разновидности принимается дополнительная функция *преобразователя*. Сначала выполняется эта функция, а затем ее результат передается функции потребителя:

```
map.forEach(threshold,
    (k, v) -> k + " -> " + v, // преобразователь
    System.out::println); // потребитель
```

Функцию преобразователя можно использовать как фильтр. Всякий раз, когда функция преобразователя возвращает пустое значение `null`, это значение незаметно пропускается. В приведенном ниже фрагменте кода выводятся только записи с крупными значениями.

```
map.forEach(threshold,
    (k, v) -> v > 1000 ? k + " -> " + v : null, // фильтр и преобразователь
    System.out::println); // пустые значения не передаются потребителю
```

В операциях `reduce` входные данные объединяются с функцией накопителя. Например, в следующей строке кода вычисляется сумма всех значений:

```
Long sum = map.reduceValues(threshold, Long::sum);
```

Как и при выполнении операции `forEach`, в данном случае можно предоставить функцию преобразователя. В следующем фрагменте кода вычисляется длина самого длинного ключа:

```
Integer maxlenlength = map.reduceKeys(threshold,
    String::length, // преобразователь
    Integer::max); // накопитель
```

Функция преобразователя может действовать подобно фильтру, возвращая пустое значение `null` для исключения ненужных вводимых данных. В приведенном ниже фрагменте кода подсчитывается количество записей со значением больше 1000.

```
Long count = map.reduceValues(threshold,
    v -> v > 1000 ? 1L : null,
    Long::sum);
```



НА ЗАМЕТКУ. Если отображение пусто или все его записи отфильтрованы, в результате выполнения операции `reduce` возвращается пустое значение `null`. Если же в отображении остается лишь один элемент, то возвращается результат его преобразования, а функция накопителя не применяется.

Имеются специальные разновидности операции `reduce` с суффиксом `ToInt`, `ToLong` и `ToDouble` для вывода значений типа `int`, `long` и `double`. Сначала нужно преобразовать вводимое значение примитивного типа, а затем указать значение по умолчанию и функцию накопителя. Значение по умолчанию возвращается, когда отображение пусто. Ниже приведен характерный тому пример.

```
long sum = map.reduceValuesToLong(threshold,
    Long::longValue, // преобразователь в примитивный тип
    0, // значение по умолчанию для пустого отображения
    Long::sum); // накопитель примитивных типов
```



ВНИМАНИЕ. Эти специальные разновидности операции `reduce` действуют иначе, чем эквивалентные им объекты, где во внимание принимается только один элемент. Вместо возврата преобразованного элемента последний накапливается вместе со значением по умолчанию. Следовательно, значение по умолчанию должно быть нейтральным элементом накопителя.

Представления множеств

Допустим, что вместо отображения требуется крупное потокобезопасное множество. Но для этой цели отсутствует класс `ConcurrentHashSet`, пытаясь создавать который самостоятельно было бы нецелесообразно. Можно было бы, конечно, воспользоваться классом `ConcurrentHashMap` с фиктивными значениями, но в таком случае получилось бы отображение, а не множество, над которым нельзя выполнять операции, определяемые в интерфейсе `Set`.

При вызове статического метода `newKeySet()` получается множество типа `Set<K>`, которое, по существу, служит оболочкой для отображения типа `ConcurrentHashMap<K, Boolean>`. (Все значения в этом отображении являются логическими `Boolean.TRUE`, но это на самом деле не важно, поскольку оно используется лишь как множество.) В следующей строке кода подобным образом получается множество символьных строк:

```
Set<String> words = ConcurrentHashMap.<String>newKeySet();
```

Безусловно, если уже имеется отображение, то метод `keySet()` выдает множество ключей, которое является изменяемым. Если удалить элементы из этого множества, то ключи (и их значения) удалятся из отображения. Но вводить элементы в такое множество не имеет смысла, поскольку для них будут отсутствовать соответствующие вводимые значения. В версии Java 8 в класс `ConcurrentHashMap` внедрен второй метод `keySet()` со значением, используемым по умолчанию при вводе элементов во множество. Так, если в приведенном ниже примере слово "Java" отсутствовало во множестве `words`, то теперь оно имеет единичное значение.

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

Параллельные операции с массивами

В классе `Arrays` теперь имеется целый ряд распараллелиемых операций. В частности, статический метод `Arrays.parallelSort()` позволяет отсортировать массив примитивных значений или объектов. Ниже приведен характерный тому пример.

```
String contents = new String(Files.readAllBytes(
    Paths.get("alice.txt")), StandardCharsets.UTF_8);
    // прочитать файл в символьную строку
String[] words = contents.split("[\\P{L}]+");
    // разбить строку по небуквенным символам
Arrays.parallelSort(words);
```

При сортировке объектов для их сравнения можно предоставить компаратор типа `Comparator`. Кроме того, во всех методах можно указать границы диапазона сортировки, как показано ниже.

```
values.parallelSort(values.length / 2, values.length);
    // отсортировать верхнюю половину диапазона значений
```



НА ЗАМЕТКУ. На первый взгляд, наличие слова `parallel` в именах упомянутых выше методов кажется излишним, поскольку пользователя не должно интересовать, каким образом происходит сортировка. Тем не менее разработчики прикладного программного интерфейса API стремились ясно показать, что сортировка распараллеливается. Таким образом, пользователи предупреждаются о том, чтобы избегать компараторов с побочными эффектами.

Метод `parallelSetAll()` заполняет массив значениями, вычисляемыми из функции. Эта функция получает индекс элемента массива и вычисляет значение в данном месте массива, как показано ниже.

```
Arrays.parallelSetAll(values, i -> i % 10);
    // заполняет массив values рядом чисел 0 1 2 3 4 5 6 7 8 9 0 1 2 ...
```

Выгоды от распараллеливания такой операции вполне очевидны. Имеются ее варианты для обработки массивов всех примитивных типов и объектов.

И наконец, имеется метод `parallelPrefix()`, заменяющий каждый элемент массива накоплением префикса заданной ассоциативной операции. Рассмотрим в качестве примера массив `[1, 2, 3, 4, ...]` и операцию умножения (`*`). После выполнения метода `Arrays.parallelPrefix(values, (x, y) -> x * y)` содержимое этого массива оказывается следующим:

`[1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, ...]`

Как ни странно, такое вычисление можно распараллелить. Сначала соединяются соседние элементы, как показано ниже.

`[1, 1 × 2, 3 × 4, 5, 5 × 6, 7, 7 × 8]`

Элементы, не выделенные полужирным, остаются без изменения. Очевидно, что такое вычисление можно выполнить параллельно в отдельных частях массива. На следующем этапе элементы, не выделенные полужирным, обновляются перемножением на элементы, отстоящие от них на одну или две позиции в массиве:

`[1, 1 × 2, 1 × 2 × 3, 1 × 2 × 3 × 4, 5, 5 × 6, 5 × 6 × 7, 5 × 6 × 7 × 8]`

И это можно сделать параллельно. Процесс завершается после $\log(n)$ этапов. Такой алгоритм выгодно отличается от прямолинейного вычисления, если имеется достаточно большое количество процессоров. Он обычно применяется на специализированном оборудовании, пользователи которого могут приспособить данный алгоритм для решения различных задач, проявив некоторую смекалку. Простой тому пример приведен в упражнении 9 в конце этой главы.

Завершаемые будущие действия

В библиотеке `java.util.concurrent` предоставляется интерфейс `Future<T>` с целью обозначить значение обобщенного типа `T`, которое станет в какой-то момент доступным в будущем. Но до сих пор применение будущих действий было довольно ограниченным. В этом разделе будет показано, каким образом **завершаемые будущие действия** позволяют составлять асинхронные операции.

Будущие действия

Напомним вкратце, что собой представляют будущие действия. С этой целью рассмотрим следующий метод:

```
public void Future<String> readPage(URL url)
```

Этот метод читает веб-страницу в отдельном потоке исполнения, на что требуется некоторое время. При вызове

```
Future<String> contents = readPage(url);
```

этот метод сразу же возвращает с веб-страницы содержимое типа `Future<String>`. А теперь допустим, что требуется извлечь все URL с веб-страницы, чтобы построить поисковый робот. Для этой цели имеется класс `Parser` с методом

```
public static List<URL> getLinks(String page)
```

Как же применить такой метод к будущему объекту? К сожалению, это можно сделать только одним способом: сначала вызвать метод `get()` для будущего объекта, чтобы получить его значение, как только оно станет доступным, а затем обработать результат, как показано ниже.

```
String page = contents.get();
List<URL> links = Parser.getLinks(page);
```

Но ведь вызов метода `get()` оказывается блокирующим. На самом деле он ничем не лучше вызова метода `public String readPage(URL url)`, блокирующего до тех пор, пока не появится результат.

Справедливости ради следует заметить, что поддержка будущих действий существовала в библиотеке `java.util.concurrent` и раньше, хотя существенные ее элементы отсутствовали. Но в то же время нельзя было сказать следующее: “Когда становится доступным результат, его можно обработать соответствующим образом”. Это важное средство было внедрено в классе `CompletableFuture<T>`.

Составление будущих действий

Попробуем изменить метод `readPage()` таким образом, чтобы он возвращал объект типа `CompletableFuture<String>`. В отличие от простого интерфейса `Future`, в классе `CompletableFuture` имеется метод `thenApply()`, которому можно передать функцию для последующей обработки:

```
CompletableFuture<String> contents = readPage(url);
CompletableFuture<List<String>> links =
    contents.thenApply(Parser::getLinks);
```

Метод `thenApply()` ничего вообще не блокирует, а только возвращает очередное будущее действие. Как только завершится первое будущее действие, его результат передается методу `getLinks()` и возвращаемое им значение становится окончательным результатом.

Составляемость является главным аспектом класса `CompletableFuture`. Составление будущих действий позволяет устраниить серьезное затруднение, возникающее при разработке асинхронных приложений. Традиционный подход к обработке неблокирующих вызовов заключается в применении обработчиков событий. При этом разработчик регистрирует обработчик для следующего действия по завершении предыдущего. Разумеется, если следующее действие окажется асинхронным, последующее за ним действие должно выполняться в другом обработчике событий. И даже если разработчик придерживается принципа "выполнить сначала первый этап, затем второй этап и далее третий этап", то логика программы все равно становится распределенной по нескольким местам. Дело еще больше усложняется, когда приходится внедрять обработку ошибок. Допустим, что на втором этапе требуется реализовать функции регистрации пользователя. В таком случае данный этап, возможно, придется повторить, поскольку пользователь может ошибиться при вводе своих учетных данных. Попытка реализовать такую управляющую логику в ряде обработчиков событий или хотя бы понять, как она реализована, может вызвать серьезные затруднения.

При наличии завершаемых будущих действий достаточно указать, что именно и в каком порядке требуется сделать. Разумеется, все это не произойдет сразу, но самое главное, что весь соответствующий код оказывается в одном месте.

Конвейер составления

Как пояснялось в главе 2, конвейер потоков ввода-вывода начинается с создания потока ввода-вывода, затем претерпевает ряд преобразований и завершается окончной операцией. Это же справедливо и для конвейера будущих действий.

Сначала формируется объект типа `CompletableFuture` — как правило, с помощью статического метода `supplyAsync()`. Этому методу требуется предоставить функциональный интерфейс `Supplier<T>`, т.е. функцию, не имеющую параметров и выдающую результат типа `T`. Эта функция вызывается в отдельном потоке исполнения. В данном примере конвейер начинается следующим образом:

```
CompletableFuture<String> contents
    = CompletableFuture.supplyAsync(() -> blockingReadPage(url));
```

Имеется также статический метод `runAsync()`, принимающий объект типа `Runnable` и выдающий будущее действие типа `CompletableFuture<Void>`. Это

удобно в том случае, если требуется лишь запланировать одно действие после другого, не организуя обмен данными между ними.



НА ЗАМЕТКУ. Все методы, имена которых оканчиваются на **Async**, имеются в двух вариантах. В первом из них предоставляемое действие выполняется в общем пуле типа `ForkJoinPool`. А у второго метода имеется параметр типа `java.util.concurrent.Executor`, и для выполнения действия в нем используется заданный исполнитель.

Далее для выполнения другого действия в том же самом или другом потоке можно вызвать метод `thenApply()` или `thenApplyAsync()`. Любому из этих методов предоставляется функция, а в результате получается будущее действие типа `CompletableFuture<U>`, где `U` — тип, возвращаемый функцией. В качестве примера ниже показано, каким образом формируется двухстадийный конвейер для чтения и обработки содержимого веб-страницы.

```
CompletableFuture<List<String>> links  
= CompletableFuture.supplyAsync(() -> blockingReadPage(url))  
    .thenApply(Parser::getLinks);
```

Можно организовать и дополнительные стадии обработки содержимого веб-страницы. В конечном итоге результаты придется где-нибудь сохранить. А в данном случае результат просто выводится, как показано ниже.

```
CompletableFuture<Void> links  
= CompletableFuture.supplyAsync(() -> blockingReadPage(url))  
    .thenApply(Parser::getLinks)  
    .thenAccept(System.out::println);
```

Метод `thenAccept()` принимает функциональный интерфейс `Consumer`, т.е. функцию с возвращаемым типом `void`. В идеальном случае метод `get()` для будущего действия не вызывается. На последней стадии работы конвейера результат просто сохраняется там, где он и должен оставаться.



НА ЗАМЕТКУ. Вычисление не начинается явным образом. Статический метод `supplyAsync()` начинает его автоматически, а остальные методы обусловливают его продолжение.

Составление асинхронных операций

Имеется немало методов для работы с завершаемыми будущими действиями. Рассмотрим сначала те из них, которые обращаются с единственным будущим действием. Для каждого метода, перечисленного в табл. 6.1, имеются также варианты типа `Async`, которые здесь не рассматриваются. Как отмечалось в предыдущем разделе, в одном из них используется общий пул типа `ForkJoinPool`, а у другого имеется параметр типа `Executor`. В табл. 6.1 для обозначения громоздких функциональных интерфейсов употребляется сокращение `T -> U` вместо `Function<? super T, U>`. Разумеется, они не являются конкретными типами Java.

Таблица 6.1. Методы для ввода действия в объект типа CompletableFuture<T>

Метод	Параметр	Описание
<code>thenApply</code>	<code>T -> U</code>	Применяет функцию к результату
<code>thenCompose</code>	<code>T -> CompletableFuture<U></code>	Вызывает функцию для результата и выполняет вызываемое будущее действие
<code>handle</code>	<code>(T, Throwable) -> U</code>	Обрабатывает результат или ошибку
<code>thenAccept</code>	<code>T -> void</code>	Действует аналогично методу <code>thenApply()</code> , но возвращает результат типа <code>void</code>
<code>whenComplete</code>	<code>(T, Throwable) -> void</code>	Действует аналогично методу <code>handle()</code> , но возвращает результат типа <code>void</code>
<code>thenRun</code>	<code>Runnable</code>	Выполняет исполняемый поток в виде экземпляра интерфейса <code>Runnable</code> с возвратом результата типа <code>void</code>

Метод `thenApply()` уже демонстрировался ранее. В результате приведенных ниже вызовов возвращается будущее действие. Функция `f()` применяется к результату выполнения будущего действия `future` сначала в одном потоке, а затем в другом, аналогичном потоке исполнения.

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

Вместо функции `T -> U` метод `thenCompose()` принимает функцию `T -> CompletableFuture<U>`. На первый взгляд, это кажется абстрактным, но может быть довольно естественным. Рассмотрим действие чтения веб-страницы по заданному URL. Вместо того чтобы предоставлять метод

```
public String blockingReadPage(URL url)
```

более изящным оказывается наличие приведенного ниже метода, возвращающего будущее действие.

```
public CompletableFuture<String> readPage(URL url)
```

А теперь допустим, что имеется еще один, приведенный ниже метод, получающий URL из вводимых пользователем данных, возможно, в диалоговом окне, где ответ не обнаруживается до тех пор, пока пользователь не щелкнет на кнопке ОК. И это событие наступит в будущем.

```
public CompletableFuture<URL> getURLInput(String prompt)
```

В данном случае имеются две функции: `T -> CompletableFuture<U>` и `U -> CompletableFuture<V>`. Очевидно, что они составляются в функцию `T -> CompletableFuture<V>` в результате вызова второй функции по завершении первой. Именно это и делается в методе `thenCompose()`.

Действие третьего метода из табл. 6.1 сосредоточено на другом, игнорировавшемся до сих пор аспекте: отказе. Когда в будущем действии типа `CompletableFuture` генерируется исключение, оно перехватывается и заключается в оболочку непроверяемого исключения типа `ExecutionException` при вызове метода `get()`. Но метод `get()`

может вообще не вызываться. Для обработки исключения служит метод `handle()`. Предоставляемая ему функция вызывается с результатом (или пустым значением `null` в его отсутствие), а также с исключением (или пустым значением `null` в его отсутствие), и этого должно быть достаточно, чтобы справиться с ситуацией. Остальные методы выдают результаты типа `void` и обычно используются в конце конвейера обработки.

А теперь обратимся к методам, объединяющим несколько будущих действий (табл. 6.2). В первых трех методах будущие действия типа `CompletableFuture<T>` и `CompletableFuture<U>` выполняются параллельно, а получаемые результаты объединяются. В трех следующих методах оба будущих действия типа `CompletableFuture<T>` выполняются параллельно. Как только завершается одно из них, его результат передается далее, тогда как результат другого просто игнорируется.

И наконец, статические методы `allOf()` и `anyOf()` принимают переменное число завершаемых будущих действий и выдают действие типа `CompletableFuture<Void>`, которое завершается тогда, когда завершаются все эти действия или любое из них. Дальнейшего распространения результатов не происходит.

Таблица 6.2. Методы объединения нескольких составляемых объектов

Метод	Параметры	Описание
<code>thenCombine</code>	<code>CompletableFuture<U>, (T, U) -> V</code>	Исполняет и объединяет результаты с заданной функцией
<code>thenAcceptBoth</code>	<code>CompletableFuture<U>, (T, U) -> void</code>	Действует аналогично методу <code>thenCombine()</code> , но возвращает результат типа <code>void</code>
<code>runAfterBoth</code>	<code>CompletableFuture<?>, Runnable</code>	Выполняет исполняемый поток после обоих будущих действий
<code>applyToEither</code>	<code>CompletableFuture<T>, T -> V</code>	Когда становится доступным результат одного или другого действия, этому методу следует передать заданную функцию
<code>acceptEither</code>	<code>CompletableFuture<T>, T -> void</code>	Действует аналогично методу <code>applyToEither()</code> , но возвращает результат типа <code>void</code>
<code>runAfterEither</code>	<code>CompletableFuture<?>, Runnable</code>	Выполняет исполняемый поток после одного или другого будущего действия
<code>static allOf</code>	<code>CompletableFuture<?>...</code>	Завершается, возвращая результат типа <code>void</code> по завершении всех заданных будущих действий
<code>static anyOf</code>	<code>CompletableFuture<?>...</code>	Завершается, возвращая результат типа <code>void</code> по завершении любого из заданных будущих действий



НА ЗАМЕТКУ. Формально говоря, методы, рассматриваемые в этом разделе, принимают параметры типа `CompletionStage`, но не `CompletableFuture`. И это тип интерфейса почти сорока абстрактными методами, реализуемыми в настоящий момент только в классе `CompletableFuture`. Большинство программистов редко реализуют данный интерфейс, и поэтому он здесь не рассматривается.

Упражнения

- Напишите программу, отслеживающую самую длинную символьную строку, наблюдаемую в целом ряде потоков исполнения. Воспользуйтесь для этой цели классом `AtomicReference` и соответствующим накопителем.
- Помогает ли класс `LongAdder` получению последовательности возрастающих идентификаторов? Объясните, почему это помогает или, наоборот, не помогает.
- Сформируйте 1000 потоков исполнения, в каждом из которых счетчик инкрементируется 100000 раз. Сравните эффективность применения для этой цели классов `AtomicLong` и `LongAdder`.
- Воспользуйтесь классом `LongAccumulator` для вычисления максимального или минимального накапливающего элемента.
- Напишите прикладную программу, в которой все слова читаются в нескольких потоках из коллекции файлов. Воспользуйтесь отображением типа `ConcurrentHashMap<String, Set<File>>` для отслеживания файлов, в которых встречается каждое слово. Воспользуйтесь также методом `merge()` для обновления полученного в итоге отображения.
- Повторите предыдущее упражнение, но на этот раз воспользуйтесь методом `computeIfAbsent()`. В чем преимущество такого подхода?
- Найдите в хеш-отображении типа `ConcurrentHashMap<String, Long>` ключ с максимальным значением, произвольно разбивая связи. Подсказка: воспользуйтесь методом `reduceEntries()`.
- На сколько крупным должен быть массив, чтобы метод `Arrays.parallelSort()` выполнялся на вашем компьютере быстрее, чем метод `Arrays.sort()`?
- Методом `parallelPrefix()` можно воспользоваться для распараллеливания процесса вычисления чисел Фибоначчи. Для этой цели используется тот факт, n -е число Фибоначчи является коэффициентом в левом верхнем углу матрицы F^n , где . Создайте массив, заполнив его матрицами 2×2 . Определите класс `Matrix` с методом умножения. Воспользуйтесь методом `parallelSetAll()` для создания массива матриц, а также методом `parallelPrefix()` для их перемножения.
- Напишите программу, сначала запрашивающую у пользователя URL, а затем читающую веб-страницу по этому URL и далее отображающую все ссылки. Воспользуйтесь на каждой стадии данного процесса классом `CompletableFuture`, но не вызывайте метод `get()`. Во избежание преждевременного завершения программы сделайте следующий вызов:
`ForkJoinPool.commonPool().awaitQuiescence(10, TimeUnit.SECONDS);`
- Напишите приведенный ниже метод, асинхронно повторяющий действие до тех пор, пока не будет получено значение, принимаемое функцией `until()`, которая также должна выполняться асинхронно.
`public static <T> CompletableFuture<T> repeat(
 Supplier<T> action, Predicate<T> until)`

Проверьте этот метод с помощью функции, читающей учетные данные пользователя в виде объекта типа `java.net.PasswordAuthentication`, а также функции, имитирующей проверку достоверности учетных данных, переходя на секунду в режим ожидания, а затем проверяя соответствие учетных данных паролю "`secret`". Подсказка: воспользуйтесь рекурсией.

Глава

7

Интерпретатор Nashorn языка JavaScript

В этой главе...

- Выполнение интерпретатора Nashorn из командной строки
- Выполнение интерпретатора Nashorn из кода Java
- Вызов методов
- Построение объектов
- Символьные строки
- Числа
- Обращение с массивами
- Списки и отображения
- Лямбда-выражения
- Расширение классов и реализация интерфейсов Java
- Исключения
- Написание сценариев командного процессора
- Nashorn и JavaFX
- Упражнения

Многие годы в комплект Java входил открытый интерпретатор Rhino языка JavaScript, написанный на Java. Он назывался Rhino потому, что на обложке весьма почитаемой книги по Java красовалось изображение носорога. Интерпретатор Rhino вполне справлялся со своими функциями, но не очень быстро. Поэтому разработчики из компании Oracle решили создать намного более эффективный интерпретатор JavaScript, используя новые команды виртуальной машины JVM, предназначенные для динамических языков.

Это послужило толчком к зарождению проекта Nashorn, что по-немецки означает “носорог”. Интерпретатор Nashorn действует очень быстро, позволяя интегрировать Java с JavaScript на высокопроизводительной виртуальной машине. Он также полностью соответствует стандарту ECMAScript для языка JavaScript. Если вам требуется предоставить пользователям своего приложения возможность написать сценарий или же вас заинтриговала простота использования таких сред реактивного программирования, как node.js, обратитесь к интерпретатору Nashorn в Java 8. В этом случае вы не только извлекаете выгоды из разумно разработанного языка написания сценариев (т.е. JavaScript), но и получаете возможность полностью воспользоваться потенциалом внутреннего механизма виртуальной машины Java.

В этой главе рассматриваются следующие основные вопросы.

- Nashorn является преемником интерпретатора Rhino языка JavaScript, обеспечивая более высокую производительность и соответствие стандарту на JavaScript.
- Nashorn является удобной средой для экспериментирования с прикладным программным интерфейсом Java API.
- Выполнение сценариев JavaScript по команде интерпретатора `jjs` или из кода Java через прикладной программный интерфейс API для написания сценариев.
- Применение предопределенных объектов JavaScript для доступа к наиболее распространенным пакетам или функции `Java.type()` для доступа к любому пакету.
- Трудности переноса символьных строк и чисел из кода JavaScript в код Java и обратно.
- Удобный синтаксис JavaScript для обращения со списками и отображениями в Java, а также со свойствами JavaBeans.
- Преобразование функций JavaScript в интерфейсы Java способом, очень похожим на применение лямбда-выражений.
- Расширение классов и реализация интерфейсов Java на JavaScript, хотя и с некоторыми ограничениями.
- Поддержка в Nashorn написания сценариев командного процессора на JavaScript.
- Написание JavaFX-программ на JavaScript, хотя и с не такой интеграцией, как хотелось бы.

Выполнение интерпретатора Nashorn из командной строки

В версии Java 8 предоставляется интерпретатор `jjs` языка JavaScript, запускаемый из командной строки. Достаточно запустить его на выполнение и выдать соответствующие команды JavaScript, как показано ниже.

```
$ jjs
jjs> 'Hello, World'
Hello, World
```

В итоге получается цикл, называемый “чтением–вычислением–печатью” (REPL) в таких языках, как Lisp, Scala и пр. Всякий раз, когда вводится выражение, выводится его значение:

```
jjs> 'Hello, World!'.length
13
```



НА ЗАМЕТКУ. Напомним, что в JavaScript символьные строки могут быть ограничены пустыми строками '... ' или "... ". В этой главе употребляются символьные строки JavaScript, заключенные в одиночные кавычки, чтобы дать наглядное представление о том, что речь идет о коде JavaScript, а не Java.

Имеется также возможность определять и вызывать функции JavaScript следующим образом:

```
jjs> function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }
function factorial(n) { return n <= 1 ? 1 : n * factorial(n - 1) }
jjs> factorial(10)
3628800
```

Кроме того, можно вызывать методы Java, как показано ниже. Если затем ввести **contents** в режиме командной строки, то появится содержимое веб-страницы.

```
var input = new java.util.Scanner(
    new java.net.URL('http://horstmann.com').openStream())
input.useDelimiter('$')
var contents = input.next()
```

Обратите внимание, насколько оперативно происходит обновление кода. При этом можно вообще не думать об исключениях, экспериментируя с кодом в динамическом режиме. Не будучи уверенными, что мне удастся прочитать содержимое полностью, я попытался установить ограничитель \$, и, как ни странно, это сработало. Для исследования прикладного программного интерфейса API намного проще воспользоваться циклом REPL, чем писать метод `public static void main()`, компилировать код или создавать проект в ИСР. Безусловно, управлять кодом Java из сценария JavaScript не совсем обычно, хотя и удобно. Обратите также внимание, каким образом определяются типы переменных `input` и `contents`.



COBET. С обновлением цикла REPL в JavaScript, подобно его эквиваленту в Scala, связаны некоторые неудобные моменты. Для цикла REPL в Scala имеется возможность завершения команд. После нажатия клавиши <Tab> получается список возможных вариантов завершения текущего выражения. Общеизвестно, насколько трудно добиться этого в таких динамически типизированных языках, как JavaScript. И самое главное упущение — отсутствие повторного вызова в режиме командной строки. После нажатия клавиши ↑ должна появиться предыдущая команда. В противном случае попробуйте установить утилиту `rlwrap` и выполнить команду `rlwrap jjs`. С другой стороны, интерпретатор `jjs` можно выполнить в редакторе Emacs, и сделать это совсем не трудно. С этой целью запустите редактор Emacs и нажмите комбинацию клавиш <Alt+x> или <Esc+x> для перехода в режим командной строки и введите `jjs`. Затем введите, как обычно, выражение. Для перехода

на предыдущую или следующую строку нажмите комбинацию клавиш **<Alt+p>** или **<Alt+n>** соответственно, а для перемещения в пределах текущей строки — клавишу **<←>** или **<→>**. Отредактируйте команду, а затем нажмите клавишу **<Enter>**, чтобы выполнить команду.

Выполнение интерпретатора Nashorn из кода Java

В предыдущем разделе было продемонстрировано применение интерпретатора Nashorn при написании сценариев JavaScript для экспериментирования с прикладным программным интерфейсом Java API из цикла REPL, запускаемого по команде **jjs**. Другое применение состоит в том, чтобы дать пользователям прикладных программ возможность выполнять сценарии, что весьма характерно для настольных систем. Например, все приложения Microsoft Office допускают написание сценариев на языке VB Script, производном от языка Basic. И многие пользователи Microsoft Office пишут такие сценарии, что привязывает их к конкретному пакету для автоматизации учрежденческих работ. Следовательно, если вы хотите привязать пользователей к своему настольному или серверному приложению на Java, вам придется предоставить аналогичные возможности.

Для выполнения сценариев JavaScript из кода Java служит механизм, внедренный еще в версии Java 6. С помощью этого механизма можно выполнять сценарии на любом языке виртуальной машины Java, в том числе Groovy, JRuby или Jython. Имеются также механизмы выполнения сценариев на языках, действующих за пределами виртуальной машины Java, например PHP или Scheme.

Для выполнения сценария из кода Java нужно получить объект типа `ScriptEngine`. Если механизм выполнения сценариев зарегистрирован, то его достаточно получить по имени. В версии Java 8 такой механизм доступен по имени "nashorn". Ниже показано, как им пользоваться.

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");
Object result = engine.eval("'Hello, World!'.length");
System.out.println(result);
```

Кроме того, прочитать сценарий можно из потока чтения типа `Reader` следующим образом:

```
Object result = engine.eval(Files.newBufferedReader(path));
```

Для того чтобы сделать объект Java доступным в сценариях, следует вызвать метод `put()` из интерфейса `ScriptEngine`. Например, для того чтобы сделать подмостки JavaFX видимыми, их можно заполнить с помощью следующего кода со сценарием JavaScript:

```
public void start(Stage stage) {
    engine.put("stage", stage);
    engine.eval(script); // код сценария, где доступен объект stage
}
```

Вместо объявления переменных в глобальной области действия их можно собрать в объект типа `Bindings` и передать этот объект методу `eval()`, как показано ниже.

Это удобно, если ряд привязок не должен сохраняться для последующих вызовов метода eval().

```
Bindings scope = engine.createBindings();
scope.put("stage", stage);
engine.eval(script, scope);
```

Вызов методов

В предыдущем разделе было показано, каким образом объекты Java становятся доступными для сценариев JavaScript с помощью механизма выполнения сценариев. В таком случае появляется и другая возможность: вызывать методы для предоставляемых переменных. Так, если в коде Java делается вызов

```
engine.put("stage", stage);
```

то из кода сценария JavaScript можно сделать следующий вызов:

```
stage.setTitle('Hello')
```

На самом деле вполне допустим и такой синтаксис:

```
stage.title = 'Hello'
```

В интерпретаторе Nashorn поддерживается удобный синтаксис для методов получения и установки свойств. Так, если в левой части оператора = оказывается выражение stage.title, оно преобразуется в вызов метода setTitle(), а иначе — в вызов метода stage.getTitle(). Для доступа к свойствам можно даже воспользоваться принятым в JavaScript синтаксисом квадратных скобок:

```
stage['title'] = 'Hello'
```



НА ЗАМЕТКУ. В JavaScript ставить точку с запятой в конце строки кода необязательно. Тем не менее многие программирующие на JavaScript ставят ее на всякий случай. Но в этой главе точки с запятой опускаются для того, чтобы вам было легче различать фрагменты кода JavaScript и Java.

В JavaScript отсутствует понятие перегрузки методов. Следовательно, допускается только один метод с заданным именем, хотя у него может быть произвольное количество параметров любого типа. В Nashorn предпринимается попытка выбрать подходящий метод Java, исходя из количества и типов параметров.

Практически всегда имеется только один метод Java, соответствующий предоставленным параметрам. Если же такой метод отсутствует, то имеется возможность выбрать подходящий метод вручную, используя следующий довольно необычный синтаксис:

```
list['remove(Object)'](1)
```

В данном случае указывается метод remove (Object), удаляющий из списка объект типа Integer, находящийся на позиции 1. (Имеется также метод remove (int), удаляющий объект на позиции 1.)

Построение объектов

Если требуется построить объекты в сценарии JavaScript вместо того, чтобы их предоставлял механизм выполнения сценариев, то нужно знать, как получать доступ к пакетам в Java. Для этого имеются два механизма.

Прежде всего, имеются глобальные объекты `java`, `javax`, `javafx`, `com`, `org` и `edu`, предоставляющие доступ к объектам пакетов и классов посредством записи через точку. Ниже приведены характерные тому примеры.

```
var javaNetPackage = java.net // объект типа JavaPackage
var URL = java.net.URL // объект типа JavaClass
```

Если требуется доступ к пакету, имя которого не начинается с одного из упомянутых выше идентификаторов, то его можно обнаружить в объекте типа `Package`, например `Package.ch.cern`. С другой стороны, можно вызвать функцию `Java.type()` следующим образом:

```
var URL = Java.type('java.net.URL')
```

Это немного быстрее, чем обращаться к классу `java.net.URL` непосредственно, и к тому же обеспечивает лучший контроль ошибок. (Если сделать опечатку, введя, например, `java.net.Url`, интерпретатор Nashorn расценит это как пакет.) Но если требуется быстродействие и качественная обработка ошибок, то вряд ли стоит обращаться для этого к языку написания сценариев. Поэтому лучше придерживаться более краткой формы.



НА ЗАМЕТКУ. В документации на Nashorn предлагается определять объекты вначале файла сценария аналогично импорту пакетов в начале файла исходного кода Java, как показано ниже.

```
var URL = Java.type('java.net.URL')
var JMath = Java.type('java.lang.Math')
// исключает конфликты с объектом типа Math в коде JavaScript
```

Получив объект класса, можно вызвать его статические методы следующим образом:

```
JMath.floorMod(-3, 10)
```

Для построения объекта следует передать объект класс оператору `new` в коде JavaScript. Любые параметры передаются конструктору класса обычным образом:

```
var URL = java.net.URL
var url = new URL('http://horstmann.com')
```

Если же эффективность кода не имеет особого значения, то можно сделать и такой вызов:

```
var url = new java.net.URL('http://horstmann.com')
```



НА ЗАМЕТКУ. Если функция `Java.type()` применяется вместе с оператором `new`, то потребуются дополнительные круглые скобки, как показано ниже.

```
var url = new (Java.type('java.net.URL'))('http://horstmann.com')
```

Если же требуется указать внутренний класс, это можно сделать, используя запись через точку следующим образом:

```
var entry = new java.util.AbstractMap.SimpleEntry('hello', 42)
```

С другой стороны, если для этой цели вызывается функция `Java.type()`, то следует употреблять разделительный знак `$`, как это делается в виртуальной машине Java:

```
var Entry = Java.type('java.util.AbstractMap$SimpleEntry')
```

Символьные строки

Символьные строки в Nashorn, безусловно, являются объектами JavaScript. Рассмотрим, например, следующий вызов:

```
'Hello'.slice(-2) // получается подстрока 'lo'
```

В данном случае вызывается метод `slice()` языка JavaScript. А в Java такой метод отсутствует.

Но вполне работоспособным оказывается и приведенный ниже вызов, несмотря на то, что в JavaScript отсутствует метод `compareTo()`, где вместо этого применяется оператор сравнения `<`.

```
'Hello'.compareTo('World')
```

В данном случае символьная строка JavaScript преобразуется в символьную строку Java. В общем, символьная строка JavaScript преобразуется в символьную строку Java всякий раз, когда она передается методу в Java.

Следует также иметь в виду, что любой объект JavaScript преобразуется в символьную строку, когда он передается методу в коде Java с помощью параметра типа `String`. Рассмотрим следующий пример кода:

```
var path = java.nio.file.Paths.get('/home/)  
// регулярное выражение JavaScript преобразуется  
// в символьную строку Java!
```

где `/home/` является регулярным выражением. Методу `Paths.get()` требуется объект типа `String`, и он его получает, хотя это и не имеет никакого смысла в данном случае. Обвинять в этом интерпретатор Nashorn не следует, поскольку он придерживается общей для JavaScript тенденции превращать все в символьную строку, если таковая ожидается. Аналогичное преобразование происходит с числовыми и логическими значениями. Например, вполне допустимым считается вызов `'Hello'.slice('-2')`, где символьная строка `'-2'` незаметно преобразуется в число `-2`. Именно такие средства превращают программирование на динамически типизированных языках в захватывающее приключение.

Числа

В JavaScript отсутствует явная поддержка целых чисел. Тип `Number` в этом языке программирования аналогичен типу `double` в Java. Когда числовое значение передается коду Java, где ожидается значение типа `int` или `long`, любая дробная часть

числа незаметно удаляется. Например, вызов 'Hello'.slice(-2.99) аналогичен вызову 'Hello'.slice(-2).

Ради эффективности в Nashorn по возможности поддерживаются целочисленные вычисления, но такое отличие, как правило, прозрачно. Впрочем, ниже приведен один из примеров, когда это правило не соблюдается.

```
java.lang.String.format('Unit price: %.2f', Number(price))
```

Обращение с массивами

Для того чтобы построить массив в Java, следует прежде всего создать объект класса:

```
var intArray = Java.type('int[]')
var StringArray = Java.type('java.lang.String[]')
```

Затем необходимо вызвать оператор new и предоставить длину массива следующим образом:

```
var numbers = new intArray(10) // массив примитивного типа int[]
var names = new StringArray(10) // массив ссылок типа String
```

Далее употребляется синтаксис квадратных скобок обычным образом:

```
numbers[0] = 42
print(numbers[0])
```

В итоге длина массива получается в виде свойства numbers.length. Для обращения ко всем значениям в массиве names организуется следующий цикл:

```
for each (var elem in names)
    сделать что-нибудь с переменной elem
```

Это равнозначно усовершенствованному циклу for в Java. Если же требуются значения индексов, то для этой цели организуется такой цикл:

```
for (var i in names)
    сделать что-нибудь с переменной i и элементом массива names[i]
```



ВНИМАНИЕ. Несмотря на то что приведенный выше цикл выглядит как усовершенствованный цикл for в Java, в нем происходит обращение ко всем значениям индексов. Массивы в JavaScript могут быть разреженными. Допустим, что массив JavaScript требуется инициализировать следующим образом:

```
var names = []
names[0] = 'Fred'
names[2] = 'Barney'
```

И тогда в цикле for (var i in names) print(i) выводятся значения 0 и 2.

Массивы в Java и JavaScript заметно разнятся. Если предоставляется массив JavaScript там, где предполагается массив Java, то в Nashorn выполняется преобразование. Но иногда для этого требуется дополнительная помощь. Так, если имеется

массив JavaScript, то для его преобразования в эквивалентный массив Java вызывается метод `Java.to()`, как показано ниже.

```
var javaNames = Java.to(names, StringArray) // массив типа String[]
```

С другой стороны, метод `Java.from()` вызывается для преобразования массива Java в массив JavaScript. Ниже показано, как это делается.

```
var jsNumbers = Java.from(numbers)
jsNumbers[-1] = 42
```

Метод `Java.to()` необходимо применять для разрешения неоднозначности при перегрузке. Например, приведенный ниже вызов оказывается неоднозначным, поскольку интерпретатору Nashorn неясно, следует ли выполнять преобразование в массив типа `int[]` или `Object[]`.

```
java.util.Arrays.toString([1, 2, 3])
```

В подобных случаях нужно сделать следующий вызов:

```
java.util.Arrays.toString(Java.to([1, 2, 3], Java.type('int[]')))
```

или просто такой вызов:

```
java.util.Arrays.toString(Java.to([1, 2, 3], 'int[]'))
```

Списки и отображения

В Nashorn предоставляется синтаксическое удобство для обращения со списками и отображениями в Java. Вместе с любым объектом типа `List` в Java может употребляться оператор квадратных скобок для вызова методов `get()` и `set()`, как выделено ниже полужирным.

```
var names = java.util.Arrays.asList('Fred', 'Wilma', 'Barney')
var first = names[0]
names[0] = 'Duke'
```

Оператор квадратных скобок подходит и для обращения с отображениями в Java, как показано ниже.

```
var scores = new java.util.HashMap
scores['Fred'] = 10 // делается вызов scores.put('Fred', 10)
```

Для обращения ко всем элементам отображения в коде JavaScript можно организовать циклы `for each` следующим образом:

```
for (var key in scores) ...
for each (var value in scores) ...
```

Если ключи и значения требуется обработать совместно, достаточно организовать циклическое обращение ко множеству записей следующим образом:

```
for each (var e in scores.entrySet())
    обработать ключ e.key и значение e.value
```



НА ЗАМЕТКУ. Цикл `for each` подходит для любого класса Java, реализующего интерфейс `Iterable`.

Лямбда-выражения

В JavaScript имеются анонимные функции. Ниже приведен пример такой функции.

```
var square = function(x) { return x * x }
// в правой части выражения указана анонимная функция
var result = square(2)
// оператор () служит для вызова функции
```

Синтаксически такая анонимная функция очень похожа на лямбда-выражение в Java. Только вместо стрелки после списка параметров в данном случае употребляется ключевое слово `function`.

Анонимную функцию можно употреблять как функциональный интерфейс, указываемый в качестве аргумента метода Java, подобно тому, как это делается в лямбда-выражениях. Ниже приведен характерный тому пример.

```
java.util.Arrays.sort(words,
    function(a, b) { return java.lang.Integer.compare(a.length, b.length) })
// сортирует массив по возрастанию длины
```

В Nashorn поддерживается сокращение для функций, тело которых состоит из единственного выражения. В определении таких функций можно опустить фигурные скобки и ключевое слово `return`, как показано ниже. И здесь обращает на себя внимание сходство с лямбда-выражением `(a, b) -> Integer.compare(a.length, b.length)` в Java.

```
java.util.Arrays.sort(words,
    function(a, b) java.lang.Integer.compare(a.length, b.length))
```



НА ЗАМЕТКУ. Такая сокращенная запись, называемая замыканием выражения, не является официальной частью стандарта ECMAScript 5.1 на язык JavaScript. Тем не менее она поддерживается и в реализации JavaScript в браузере Mozilla.

Расширение классов и реализация интерфейсов Java

Для расширения класса или реализации интерфейса Java служит функция `Java.extend()`. Этой функции предоставляется объект суперкласса или интерфейса и объект JavaScript с методами, которые требуется переопределить или реализовать.

В приведенном ниже примере демонстрируется итератор, формирующий бесконечную последовательность случайных чисел. При этом переопределяются два метода: `next()` и `hasNext()`. Для каждого из этих методов предоставляется реализация анонимной функции в коде JavaScript.

```
var RandomIterator = Java.extend(java.util.Iterator, {
    next: function() Math.random(),
```

```
hasNext: function() true
}) // RandomIterator – это объект класса
var iter = new RandomIterator()
// использовать его для построения экземпляра
```



НА ЗАМЕТКУ. При вызове функции `Java.extend()` можно указать любое количество суперинтерфейсов, а также суперкласс. Все объекты классов следует указать перед объектом с реализуемыми методами.

Еще одно расширение синтаксиса Nashorn позволяет определить анонимные подклассы интерфейсов или абстрактных классов. Если после выражения `new` Объект-КлассJava в конструкторе следует объект JavaScript, то возвращается объект расширенного класса. Ниже приведен характерный тому пример.

```
var iter = new java.util.Iterator {
    next: function() Math.random(),
    hasNext: function() true
}
```

Если супертип относится к абстрактному классу, содержащему единственный абстрактный метод, то можно даже не предоставлять имя метода. Вместо этого функция передается так, как если бы это был параметр конструктора:

```
var task = new java.lang.Runnable(function() { print('Hello') })
// задача – это объект анонимного класса,
// реализующего интерфейс Runnable
```



ВНИМАНИЕ. Упомянутый выше синтаксис конструктора нельзя использовать при расширении конкретного класса. Например, в выражении `new java.lang.Thread(function() { print('Hello') })` вызывается конструктор класса `Thread` (в данном случае – конструктор `Thread(Runnable)`). При вызове конструктора с помощью оператора `new` возвращается объект класса `Thread`, а не его подкласса.

Если в подклассе требуются переменные экземпляра, их следует ввести в объект JavaScript. В приведенном ниже примере демонстрируется итератор, формирующий ряд из десяти случайных чисел. Следует иметь в виду, что в методах `next()` и `hasNext()` языка JavaScript делается ссылка `this.count` на переменную экземпляра.

```
var iter = new java.util.Iterator {
    count: 10,
    next: function() { this.count--; return Math.random() },
    hasNext: function() this.count > 0
}
```

Из суперкласса можно вызвать метод при его переопределении, хотя сделать это непросто. В результате вызова функции `Java.super(obj)` формируется объект, для которого можно вызвать метод из суперкласса того класса, которому принадлежит объект `obj`, но этот объект должен быть доступен. Ниже показано, как этого добиться.

```
var arr = new (Java.extend(java.util.ArrayList)) {
    add: function(x) {
        print('Adding ' + x);
        return Java.super(arr).add(x)
    }
}
```

При вызове функции `arr.add('Fred')` сообщение выводится перед тем, как значение вводится в списочный массив. Следует иметь в виду, что для вызова функции `Java.super(arr)` требуется переменная `arr`, в которой устанавливается значение, возвращаемое оператором `new`. А при вызове функции `Java.super(this)` получается только объект JavaScript, определяющий метод, но не его заместитель в Java. Механизм действия функции `Java.super()` оказывается полезным лишь для определения отдельных объектов, но не подклассов.



НА ЗАМЕТКУ. Вместо вызова функции `Java.super(arr).add(x)` можно также воспользоваться следующим синтаксисом: `arr.super$add(x)`.

Исключения

Когда метод Java генерирует исключение, его можно перехватить в коде JavaScript обычным образом:

```
try {
    var first = list.get(0)
    ...
} catch (e) {
    if (e instanceof java.lang.IndexOutOfBoundsException)
        print('list is empty')
}
```

Обратите внимание на наличие в приведенном выше примере кода только одного оператора `catch` для перехвата исключений, в отличие от кода Java, где можно перехватывать исключения по типу. И это также в духе динамических языков программирования, где все запросы типов происходят во время выполнения.

Написание сценариев командного процессора

Если требуется автоматизировать задачу, повторно выполняемую на компьютере, то для этого, скорее всего, придется ввести команды в *сценарий командного процессора*, воспроизводящий ряд команд на уровне операционной системы. На моем компьютере имеется каталог `~/bin`, заполненный сценариями командного процессора, предназначенными для самых разных целей: выгрузки файлов на мой веб-сайт, в мой блог, фотоархив или FTP-сайт моего издателя; приведения размеров изображений к принятым в блоге нормам; массовой рассылки сообщений по электронной почте моим студентам; регулярного резервного копирования данных на моем компьютере в два часа ночи и прочих целей.

Теперь это сценарии командного процессора `bash`. Но в прошлом, когда я пользовался Windows, это были командные файлы. Что же в них не так? Дело в том, что рано или поздно в таких сценариях возникает потребность в ветвлении и циклах.

По ряду причин большинство разработчиков командных процессоров плохо разбирались в основах проектирования языков программирования. В частности, переменные, ветвления, циклы и функции реализованы в командном процессоре bash из рук вон плохо, а командный язык в Windows в этом отношении еще хуже. У меня имеется несколько сценариев командного процессора bash, которые поначалу были скромных размеров, но со временем разрослись настолько, что стали практически неуправляемыми. И это типичный недостаток подобных сценариев.

А почему бы не написать подобные сценарии на Java? А потому, что язык Java слишком многословен. Если вызывать внешние команды с помощью метода `Runtime.exec()`, то придется управлять стандартными потоками ввода-вывода данных и сообщений об ошибках. Поэтому для написания сценариев командного процессора разработчики Nashorn предлагают в качестве альтернативы язык JavaScript. Его синтаксис относительно прост, а интерпретатор Nashorn предоставляет некоторые удобства специально для программирующих на уровне командного процессора.

Выполнение команд из командного процессора

Для того чтобы воспользоваться в Nashorn расширениями, предназначенными для написания сценариев, выполните команду

```
jjs -scripting
```

или

```
jrunscript
```

После этого можно выполнять команды из командного процессора, заключая их в обратные кавычки, как показано ниже.

```
`ls -al`
```

Данные и сообщения об ошибках последней команды выводятся в стандартные потоки и фиксируются в переменных `$OUT` и `$ERR` соответственно. А код завершения команды фиксируется в переменной `$EXIT`. (Нулевым кодом принято обозначать успешное завершение, а ненулевыми кодами — состояния ошибок.) Для того чтобы зафиксировать стандартный вывод, достаточно заключить команду в обратные кавычки, а результат ее выполнения присвоить переменной следующим образом:

```
var output = `ls -al`
```

Если для команды требуется предоставить стандартный ввод, достаточно воспользоваться командой

```
$EXEC(команда, ввод)
```

Например, приведенная ниже команда передает вывод из команды `ls -al` команде `grep -v class`.

```
$EXEC('grep -v class', `ls -al`)
```

Это не совсем конвейер, но его, если требуется, нетрудно реализовать (см. упражнение 6 в конце этой главы).

Интерполяция символьных строк

Выражения в конструкции `${...}` вычисляются в символьных строках, заключенных в двойные и обратные кавычки. Это так называемая *интерполяция символьных строк*. В следующем примере:

```
var cmd = "javac -classpath ${classpath} ${mainclass}.java"
$EXEC(cmd)
```

или просто

```
`javac -classpath ${classpath} ${mainclass}.java`
```

содержимое переменных `classpath` и `mainclass` вставляется в исполняемую команду.

В конструкции `${...}` могут указываться произвольные выражения:

```
var message = "The current time is ${java.time.Instant.now()}"
// устанавливает в сообщении символьную строку, например,
// «The current time is 2013-10-12T21:48:58.545Z»
// (Текущее время: 2013-10-12T21:48:58.545Z)
```

Символьные строки также интерполируются в документах, встраиваемых в сценарии. Такие документы оказываются полезными, когда многие строки читаются по команде из стандартного потока ввода, тогда как автору сценария не требуется вводить их в отдельный файл. В качестве примера ниже показано, каким образом инструментальное средство администрирования GlassFish снабжается командами.

```
name='myapp'
dir='/opt/apps/myapp'
$EXEC("asadmin", <<END)
start-domain
start-database
deploy ${name} ${dir}
exit
END
```

Конструкция `<<END` означает следующее: “Вставить символьную строку, начинаяющуюся со следующей строки и завершающуюся строкой END”. (Вместо строки END можно указать любой идентификатор, отсутствующий в символьной строке.) Следует также иметь в виду, что наименование и местоположение приложения интерполируются. Интерполяция символьных строк и встраиваемые документы доступны только в режиме сценариев.

Ввод данных в сценарий

Сценарий можно снабдить аргументами командной строки. В строке команды `jjs` можно ввести несколько файлов сценариев, и поэтому эти файлы нужно отделить от аргументов разделителем `--`, как показано ниже.

```
jjs script1.js script2.js -- arg1 arg2 arg3
```



НА ЗАМЕТКУ. Такая команда выглядит несколько громоздкой. Поэтому если имеется только один файл сценария, то его можно выполнить вместе с аргументами командной строки следующим образом:

```
jrunscript -f script.js arg1 arg2 arg3
```

 **СОВЕТ.** В первой строке сценария сначала может быть указана последовательность символов `#!`, а затем местоположение интерпретатора сценариев. Ниже приведены характерные тому примеры.

```
#!/opt/java/bin/jjs
```

или

```
#!/opt/java/bin/jrunscript -f
```

В таком случае файл сценария можно сделать исполняемым и просто выполнить его следующим образом:

```
путь/script.js
```

Если сценарий начинается с последовательности символов `#!`, то режим сценариев активизируется автоматически.

 **ВНИМАНИЕ.** Если у сценария имеются аргументы, а вслед за последовательностью символов `#!` указывается команда `jjs`, то пользователям такого сценария придется ввести команду

```
путь/script.js -- arg1 arg2 arg3
```

И это пользователям, конечно, не понравится. Поэтому воспользуйтесь лучше командой `jrunscript`.

В файле сценария аргументы командной строки вводятся в массив `arguments` следующим образом:

```
var deployCommand = "deploy ${arguments[0]} ${arguments[1]}"
```

Вместо массива `arguments` в команде `jjs` (но не `jrunscript`) можно использовать переменную `$ARG`. Если эта переменная используется вместе с интерполяцией символьных строк, то для указания аргументов потребуются два знака `$`, как показано ниже.

```
var deployCommand = "deploy ${$ARG[0]} ${$ARG[1]}"
```

С помощью объекта `ENV` в сценарии можно также получить доступ к переменным окружения командного процессора следующим образом:

```
var javaHome = $ENV.JAVA_HOME
```

Используя функцию `readLine()`, в режиме сценариев можно предоставить пользователю подсказку для ввода данных:

```
var username = readLine('Username: ')
```

И наконец, для завершения сценария служит функция `exit()`. Эту функцию можно снабдить дополнительным кодом завершения следующим образом:

```
if (username.length == 0) exit(1)
```

 **ВНИМАНИЕ.** Для вывода приглашения ввести пароль достаточно сделать вызов

```
var password = java.lang.System.console().readPassword('Password: ')
```

Nashorn и JavaFX

Интерпретатор Nashorn предоставляет удобный способ запуска JavaFX-приложений. Для этого достаточно ввести в сценарий команды, которые обычно вводятся в метод `start()` из подкласса, производного от класса `Application`, а в качестве параметра типа `Stage` указать переменную `$STAGE`. Не нужно даже вызывать метод `show()` для объекта типа `Stage`, поскольку это делается автоматически. В качестве примера ниже приведена программа из главы 4, выводящая сообщение "Hello, JavaFX!", но теперь написанная на JavaScript в виде сценария.

```
var message = new javafx.scene.control.Label("Hello, JavaFX!");
message.font = new javafx.scene.text.Font(100);
$STAGE.scene = new javafx.scene.Scene(message);
$STAGE.title = "Hello";
```

Выполните этот сценарий, указав его вместе с параметром `-fx` в следующей команде:

```
jjs -fx myJavaFxApp.js
```

Вот, собственно, и все. Метка с сообщением "Hello, JavaFX!" отображается шрифтом размером 100 пунктов в окне с заголовком "Hello", как показано на рис. 7.1. Теперь вместо шаблонного кода

```
message.setFont(new Font(100))
```

можно употребить следующую удобную запись свойства:

```
message.font = new Font(100)
```

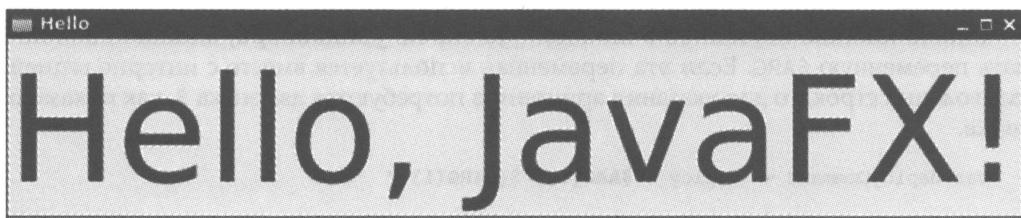


Рис. 7.1. JavaFX-приложение, выводящее сообщение "Hello, JavaFX!", но написанное на JavaScript в виде сценария



НА ЗАМЕТКУ. Если требуется переопределить методы `init()` или `stop()` из класса `Application`, определяющие, помимо метода `start()`, жизненный цикл приложения, их следует включить в сценарий на самом верхнем его уровне. А затем с помощью параметра `-fx` можно получить подкласс, производный от класса `Application`, вместе с методами сценария.

А теперь перейдем к обработке событий. Как пояснялось в главе 4, большинство событий в JavaFX обрабатываются приемниками, присоединяемыми к свойствам JavaFX. Но в JavaScript дело обстоит сложнее. Напомним, что у свойства JavaFX имеютсь два интерфейса приемников событий: `InvalidationListener` и `ChangeListener`, и у обоих имеется метод `addListener()`. Этот метод можно вызвать в коде Java с лямбда-выражением, предоставив компилятору возможность самому определять по типам параметров, какой из двух приемников событий следует добавить к свойству

JavaFX. А в JavaScript параметры функции не имеют типа. Допустим, имеется ползунок для регулирования размера шрифта. Этот ползунок требуется снабдить приемником событий, чтобы обновлять размер шрифта при установке ползунка на нужном значении. В приведенном ниже фрагменте кода предпринимается попытка реализовать такую возможность.

```
slider.valueProperty().addListener(function(property)
    message.font = new Font(slider.value))
    // Ошибка: в Nashorn нельзя определить тип
    // добавляемого приемника событий
```

К сожалению, этот код неработоспособен, поскольку интерпретатору Nashorn неизвестно, какого типа приемник событий требуется добавить: `InvalidationListener` или `ChangeListener`. И тем более ему неизвестно, что тип приемника событий особого значения не имеет. Поэтому для устранения подобного недоразумения выбор типа приемника событий придется сделать вручную, как показано ниже.

```
slider.valueProperty().addListener(
    new javafx.beans.InvalidationListener(function(property)
        message.font = new Font(slider.value)))
```

Это более громоздкий код, чем его эквивалент в Java. Очевидно, что такой код плохо сообразуется с лаконичностью языка написания сценариев, но тут уж ничего не поделаешь. Разработчики JavaFX решили перегрузить метод `addListener()`, что было вполне благородно в контексте версии Java 7 и в основном согласуется с лямбда-выражениями в версии Java 8. Но совместимость с языками написания сценариев, вероятно, не была главной заботой разработчиков JavaFX, в частности потому, что они просто пренебрели другим языком написания сценариев.

Но при разработке прикладного программного интерфейса Java API не следует забывать о законе Атвуда, который гласит: “Любое приложение, которое можно написать на JavaScript, в конечном итоге будет написано на JavaScript”. Следовательно, прикладной программный интерфейс Java API следует разрабатывать таким образом, чтобы он был вполне доступным из кода JavaScript.

Необходимо обратить внимание на еще один прискорбный факт, касающийся поддержки JavaFX в Nashorn. В прошлом компоновку сцены можно было легко описать на языке JavaFX Script следующим образом:

```
Frame {
    title: "Hello"
    content: Label {
        text: "Hello, World!"
    }
}
```

Очень похоже на код JavaScript, не так ли? Разработчики Nashorn и JavaFX стирают эту границу, превращая ее в JavaScript! Теперь компоновку пользовательского интерфейса и обработку событий можно написать на JavaScript, и закон Атвуда будет соблюден.

Упражнения

1. Выберите ту часть прикладного программного интерфейса Java API, которую требуется исследовать, например, класс `ZonedDateTime`. Проведите ряд

экспериментов в режиме командной строки **jjS**, конструируя объекты, вызывая методы и наблюдая возвращаемые значения. Насколько это удобнее написания тестовых программ на Java?

2. Выполните команду **jjS** и, воспользовавшись библиотекой потоков ввода-вывода, найдите в диалоговом режиме решение следующей задачи: вывести из файла все неодинаковые длинные слова (больше 12 символов) в отсортированном порядке. Сначала должны быть прочитаны все слова, затем отобраны длинные слова и т.д. Сравните такой диалоговый режим работы с обычным режимом разработки прикладных программ.
3. Выполните команду **jjS** и сделайте следующий вызов:

```
var b = new java.math.BigInteger('1234567890987654321')
```

Затем отобразите содержимое переменной **b**, просто введя **b** и нажав клавишу **<Enter>**. Что вы получите в итоге? Каково значение, возвращаемое из вызова **b.mod(java.math.BigInteger.TEN)**? Почему содержимое переменной **b** отображается так необычно? Каким образом отобразить фактическое значение переменной **b**?

4. Составьте в коде JavaScript нелитеральную символьную строку, выделив подстроку из другой строки или вызвав метод **getClass()**. Какой класс вы получите в итоге? Затем передайте объект методу **java.lang.String.class.cast()**. Объясните, к чему это приведет?
5. В конце раздела “Расширение классов и реализация интерфейсов Java” ранее в этой главе было показано, как расширить класс **ArrayList**, чтобы протоколировать каждый вызов метода **add()**. Но это годится только для одного объекта. Напишите на JavaScript функцию, выполняющую роль фабрики для подобных объектов, чтобы можно было составить любое количество списочных массивов для протоколирования.
6. Напишите на JavaScript функцию **pipe()**, принимающую последовательность команд из командного процессора и организующую конвейерную передачу данных с выхода одной команды на вход другой и возвращающую конечный результат. Например, **pipe('find .', 'grep -v class', 'sort')**. Подсказка: организуйте повторный вызов команды **\$EXEC**.
7. Решение задачи из предыдущего упражнения недостаточно эффективно по сравнению с настоящим конвейером в Unix, поскольку выполнение следующей команды начинается только по завершении предыдущей. Устраните этот недостаток, используя класс **ProcessBuilder**.
8. Напишите сценарий для вывода значений всех переменных окружения.
9. Напишите сценарий **nextYear.js**, в котором определяется возраст пользователя и выводится сообщение “Next year, you will be ...” (В следующем году вам будет...) с прибавлением 1 к исходным данным. Возраст может быть указан в режиме командной строки или в переменной окружения **AGE**. Если же возраст никак не указан, выведите приглашение пользователю указать свой возраст.
10. Напишите на JavaScript прикладную JavaFX-программу, читающую данные из выбранного вами источника и строящую на их основании круговую диаграмму. Насколько труднее или легче было бы написать такую программу на Java и почему?

Глава

8

Разные полезные средства

В этой главе...

- Символьные строки
- Числовые классы
- Новые математические функции
- Коллекции
- Обращение с файлами
- Аннотации
- Различные незначительные изменения
- Упражнения

Версия Java 8 выпущена со значительными и важными усовершенствованиями как самого языка, так и его библиотеки. Но в этой версии внесены также многочисленные мелкие и полезные изменения повсюду в библиотеке Java. В документации на прикладной программный интерфейс Java API можно тщательно изучены все примечания типа `@since 1.8` (т.е. начиная с версии 1.8) к изменениям, которые ради удобства разделены на отдельные категории. Из этой главы вы узнаете, что нового появилось в Java 8 для символьных строк, чисел, математических расчетов, коллекций, файлов, регулярных выражений и технологии JDBC.

В этой главе рассматриваются следующие основные вопросы.

- Упрощение соединения символьных строк с ограничителем: вызов метода `String.join(", ", a, b, c)` вместо `a + ", " + b + ", " + c`.
- Поддержка беззнаковых арифметических операций.
- Методы из класса `Math` для обнаружения переполнения в целочисленных операциях.
- Вызов метода `Math.floorMod(x, n)` вместо `x % n`, если `x` имеет отрицательное значение.
- Новые методы модификации `removeIf()` в классе `Collection` и `replaceAll()` и `sort()` в классе `List`.
- Отложенное чтение потока ввода символьных строк методом `Files.lines()`.
- Отложенное перечисление файлов в каталоге методом `Files.list()` и их рекурсивный обход методом `Files.walk()`.
- Реализованная наконец-то официальная поддержка кодировки `Base64`.
- Повторение аннотаций и их применение к используемым типам данных.
- Удобная поддержка проверки пустых параметров в классе `Objects`.

Символьные строки

Соединение нескольких строк, а также их разделение ограничителем вроде `", "` или `"/"` относится к типичным задачам обращения с символьными строками. В версии Java 8 были внедрены дополнительные средства для решения подобных задач. Теперь символьные строки можно получать из массива или последовательности символов типа `Iterable<? extends CharSequence>`, как показано в приведенных ниже примерах.

```
String joined = String.join("/", "usr", "local", "bin"); // "usr/local/bin"
System.out.println(joined);
String ids = String.join(", ", ZoneId.getAvailableZoneIds());
System.out.println(ids);
// все идентификаторы часовых поясов, разделенные запятыми
```

Действие метода `join()` следует рассматривать как противоположное действию метода `String.split()`. Это единственный метод, введенный в класс `String` в версии Java 8.



НА ЗАМЕТКУ. Как упоминалось в главе 2, в интерфейсе `CharSequence` предоставляется полезный метод экземпляра `codePoints()`, возвращающий поток ввода-вывода значений в unicode, а также менее полезный метод `chars()`, возвращающий поток ввода-вывода кодовых единиц в кодировке UTF-16.

Числовые классы

Начиная с версии Java 5, каждый из семи классов-оболочек примитивных типов данных (т.е. не логического типа Boolean) имеет статическое поле SIZE, предоставляющее разрядность типа данных в битах. Приятно сообщить, что в этих классах-оболочках теперь появилось поле BYTES, сообщающее размерность в байтах для тех типов данных, которые делятся на восемь. Все восемь классов-оболочек примитивных типов теперь содержат статический метод hashCode(), возвращающий тот же самый хеш-код, что и метод экземпляра, но без необходимости упаковки.

У классов-оболочек пяти примитивных типов данных — Short, Integer, Long, Float и Double — имеются статические методы sum(), max() и min(), которые могут оказаться полезными в качестве функций сведения в операциях с потоками ввода-вывода. А в классе Boolean для тех же самых целей имеются статические методы logicalAnd(), logicalOr() и logicalXor().

Для числовых типов данных теперь поддерживаются беззнаковые арифметические операции. Например, вместо того чтобы представлять значение типа Byte в пределах от **-128** до **127**, можно вызвать статический метод Byte.toUnsignedInt(b) и получить значение в пределах от **0** до **255**. В общем, в числовых операциях без знака теряются отрицательные значения, но в то же время диапазон положительных значений расширяется в два раза. В классах Byte и Short имеются методы toUnsignedInt(), а в классах Byte, Short и Integer — методы toUnsignedLong().

В классах Integer и Long имеются методы compareUnsigned(), divideUnsigned() и remainderUnsigned() для обработки значений без знака. А для операций сложения, вычитания и умножения специальных методов не требуется. Операторы + и - вполне справляются со своей задачей, но уже и в отношении значений без знака. Умножение целых значений без знака больше величины Integer.MAX_VALUE может привести к переполнению, поэтому следует вызвать метод toUnsignedLong() и перемножить их как значения типа long.



НА ЗАМЕТКУ. Для обработки чисел без знака требуется ясное представление о двоичной арифметике и двоичном представлении отрицательных чисел. В языках С и С++ совместное использование типов данных со знаком и без знака служит типичной причиной едва уловимых ошибок. А в языке Java много лет назад было принято более мудрое решение отказаться от такого смешанного использования числовых типов данных и оставить только числа со знаком. Главная причина для применения чисел со знаком состоит в необходимости обращаться с форматами файлов или сетевыми протоколами, где такие числа требуются.

В классах Float и Double имеются статические методы типа isFinite. Так, в результате вызова метода Double.isFinite(x) возвращается логическое значение true, если параметр x не содержит плюс или минус бесконечность либо его значение не является числом. В прошлом для получения того же самого результата приходилось вызывать методы экземпляра isInfinite() и isNaN().

И наконец, в классе BigInteger имеются методы экземпляра типа (long|int|short|byte)ValueExact, возвращающие значение типа long, int, short или byte и генерирующие исключение типа ArithmeticException, если это значение не находится в целевых пределах.

Новые математические функции

В классе Math предоставляется несколько методов для “точных” арифметических операций, генерирующих исключение при переполнении результата. Например, операция умножения **100000 * 100000** незаметно приводит к неверному результату **1410065408**, тогда как вызов метода `multiplyExact(100000, 100000)` — к генери-рованию исключения. Для этих целей предоставляются методы типа (`add|subtract|multiply|increment|decrement|negate`)`Exact` с параметрами типа `int` и `long`. В частности, метод `toIntExact()` преобразует значение типа `long` в эквивалентное значение типа `int`.

Методы `floorMod()` и `floorDiv()` нацелены на разрешение давнишней проблемы с целочисленными остатками. Рассмотрим в качестве примера выражение `n % 2`. Как известно, вычисление этого выражения дает нулевой результат, если `n` — четное число, или единичный результат, если `n` — нечетное число. Но если `n` — отрицательное число, то результат равен **-1**. Почему? Когда появились первые компьютеры, пришлось составлять правила для операций деления целых чисел с остатком, когда операнды являются отрицательными числами. Математикам давно известно оптимальное (так называемое “евклидово”) правило, которое предписывает всегда оставлять остаток, больший или равный нулю. Но вместо того чтобы обратиться за справкой к учебнику по математике, создатели первых компьютеров выработали свои правила, которые казались им вполне обоснованными, а на самом деле неудобными.

Рассмотрим эту проблему подробнее. Допустим, требуется рассчитать положение стрелки часов, откорректировав его путем нормализации числа в пределах от **0** до **11**. На первый взгляд, сделать это нетрудно в следующем выражении: `(position + adjustment) % 12`. Но что, если значение переменной `adjustment` отрицательно? В таком случае может быть получено отрицательное число. Следовательно, придется внедрить ветвление или воспользоваться выражением `((position + adjustment) % 12 + 12) % 12`. Но в любом случае это хлопотно. Новый метод `floorMod()` позволяет сделать это проще. В частности, вызов метода `floorMod(position + adjustment, 12)` всегда дает числовое значение в пределах от **0** до **11**.



НА ЗАМЕТКУ. К сожалению, метод `floorMod()` дает отрицательные результаты для отрицательных делителей, но на практике такая ситуация возникает нечасто.

Метод `nextDown()`, определенный как для параметров типа `double`, так и для параметров типа `float`, дает следующее по величине число с плавающей точкой, меньшее заданного числа. Так, если предполагается получить число меньше `b`, но на самом деле придется рассчитать значение, точно равное `b`, то можно возвратить результат вызова метода `Math.nextDown(b)`. (Соответствующий метод `Math.nextUp()` существует с версии Java 6.)



НА ЗАМЕТКУ. Все методы, рассматриваемые в этом разделе, существуют также в классе `StrictMath`.

Коллекции

Самое большое изменение в библиотеке коллекций, разумеется, состоит в поддержке потоков ввода-вывода, рассматривавшихся в главе 2. Хотя имеются и менее значимые изменения.

Методы, введенные в классы коллекций

В табл. 8.1 сведены различные методы, введенные в классы и интерфейсы коллекций, начиная с версии Java 8. К их числу не относятся методы `stream()`, `parallelStream()` и `spliterator()`.

Таблица 8.1. Методы, введенные в классы и интерфейсы коллекций в версии Java 8

Класс или интерфейс	Новые методы
<code>Iterable</code>	<code>forEach</code>
<code>Collection</code>	<code>removeIf</code>
<code>List</code>	<code>forEach</code> , <code>replace</code> , <code>replaceAll</code> , <code>remove(key, value)</code> [удаляет элемент только в том случае, если ключ отображается на значение], <code>putIfAbsent</code> , <code>compute</code> , <code>computeIfAbsent Present</code> , <code>merge</code>
<code>Iterator</code>	<code>forEachRemaining</code>
<code>BitSet</code>	<code>stream</code>

В связи с изложенным выше возникает следующий резонный вопрос: зачем в интерфейсе `Stream` присутствует столько методов, принимающих лямбда-выражения, но лишь один такой метод, `removeIf()`, введен в интерфейс `Collection`? Если просмотреть методы в интерфейсе `Stream`, то можно обнаружить, что большинство из них возвращает единственное значение или поток ввода-вывода преобразованных значений, отсутствующих в исходном потоке. Исключением из этого правила являются методы `filter()` и `distinct()`. Действие метода `removeIf()` следует рассматривать как противоположное действию метода `filter()`, т.е. он удаляет на месте все совпадающие элементы. Реализовывать метод `distinct()` в произвольных коллекциях было бы неэффективно.

В интерфейсе `List` имеется метод `replaceAll()`, который является уместным эквивалентом метода `map()`, а также весьма полезный метод `sort()`. В интерфейсе `Map` имеется ряд методов, имеющих особое значение для параллельного доступа к отображениям. Более подробно эти методы рассматриваются в главе 6.

В интерфейсе `Iterator` имеется метод `forEachRemaining()`, исчерпывающий итератор, снабжая функцию оставшимися элементами для итератора. И наконец, в классе `BitSet` имеется метод, выдающий все члены множества в виде потока ввода-вывода значений типа `int`.

Компараторы

В интерфейс `Comparator` внедрен целый ряд новых полезных методов, выгодно использующих тот факт, что в интерфейсах теперь могут присутствовать конкретные методы. В частности, статический метод `comparing()` принимает функцию, извлекающую ключ и преобразующую обобщенный тип `T` в сравниваемый тип (например,

`String`). Эта функция применяется к сравниваемым объектам, после чего происходит их сравнение по возвращаемым ключам. Допустим, имеется массив объектов типа `Person`. Ниже показано, как отсортировать их по имени.

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

Компараторы можно связывать в цепочку с помощью метода `thenComparing()` для разрываания связей. Ниже приведен характерный тому пример. Если у двух человек одинаковая фамилия, то используется второй компаратор.

```
Arrays.sort(people,
    Comparator.comparing(Person::getLastName)
        .thenComparing(Person::getFirstName));
```

Имеется несколько вариантов этих методов. В частности, для ключей, извлекаемых методами `comparing()` и `thenComparing()`, можно указать соответствующий компаратор. В качестве примера ниже демонстрируется сортировка людей по длине их имен.

```
Arrays.sort(people, Comparator.comparing(Person::getName,
    (s, t) -> Integer.compare(s.length(), t.length())));
```

Более того, у обоих методов, `comparing()` и `thenComparing()`, имеются варианты, в которых исключается упаковка значений типа `int`, `long` или `double`. В частности, предыдущую операцию проще выполнить следующим образом:

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

Если разрабатываемые вами функции могут возвращать пустое значение `null`, то вам, вероятнее всего, понравятся методы адаптеров `nullsFirst()` и `nullsLast()`. Эти статические методы принимают существующий компаратор и модифицируют его, чтобы он не генерировал исключение, когда ему встречаются пустые значения `null`, но классифицировал их больше или меньше обычных значений. Допустим, что метод `getMiddleName()` возвращает пустое значение `null`, если у человека отсутствует второе имя или отчество. В таком случае можно сделать следующий вызов `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

Методу `nullsFirst()` требуется компаратор (в данном случае для сравнения символьных строк). Метод `naturalOrder()` создает компаратор для любого класса, реализующего интерфейс `Comparable`. Именно такой компаратор предоставляется в результате вызова `Comparator.<String>naturalOrder()`. Ниже показано, каким образом организуется вызов для сортировки людей по потенциально пустым вторым именам. Для того чтобы сделать выражение более удобочитаемым, используется статический импорт `java.util.Comparator.*`. Следует иметь в виду, что тип для метода `naturalOrder()` выводится автоматически.

```
Arrays.sort(people, comparing(Person::getMiddleName,
    nullsFirst(naturalOrder())));
```

Статический метод `reverseOrder()` выдает результаты в порядке, обратном естественному. А для обращения любого компаратора служит метод экземпляра `reversed()`. Например, вызов метода `naturalOrder().reversed()` аналогичен вызову метода `reverseOrder()`.

Класс Collections

В версии Java 6 были внедрены классы NavigableSet и NavigableMap, в которых выгодно используется упорядочение элементов или ключей, предоставляя эффективные методы для обнаружения наименьшего элемента, большего или равного заданному значению v, либо наибольшего элемента, меньшего или равного заданному значению v. Помимо других коллекций, теперь эти классы поддерживаются в классе Collections вместе с соответствующими методами (unmodifiable|synchronized|checked|empty)Navigable(Set|Map).

Появился также класс-оболочка checkedQueue, которым явно пренебрегали раньше. Напомним, что у оболочек интерфейса checked имеется параметр типа Class, а при вставке элемента неверного типа они генерируют исключение типа ClassCastException. Эти классы предназначены для целей отладки. Допустим, что требуется объявить интерфейс Queue<Path>, а в другом месте кода при попытке привести тип String к типу Path возникает исключение типа ClassCastException.

Это могло произойти потому, что очередь была передана при вызове метода void getMoreWork (Queue q) без параметра типа. Затем где-то в очереди q был вставлен объект типа String. (Этот факт не был обнаружен компилятором, поскольку обобщенный тип быть подавлен.) Намного позже объект типа String был извлечен из очереди в предположении, что он относится к типу Path, и в этом проявилась ошибка. Если временно заменить очередь q на очередь CheckedQueue (new LinkedList<Path>, Path.class), то вставка в нее каждого элемента будет проверяться во время выполнения, что позволит выявить ошибочный код вставки.

И наконец, имеются методы типа emptySorted(Set|Map), предоставляющие упрощенные экземпляры отсортированных коллекций. Они аналогичны методам типа empty(Set|Map), внедренным еще в версии Java 5.

Обращение с файлами

В версии Java 8 внедрено небольшое количество служебных методов, применяющих потоки ввода-вывода для чтения данных из файлов и просмотра содержимого каталогов. И наконец-то появился официальный способ кодирования и декодирования по основанию Base64.

Потоки ввода-вывода строк

Для чтения строк из файла по требованию служит метод Files.lines(). Он выдает поток ввода символьных строк по очереди, как показано ниже. И как только обнаруживается первая строка, содержащая слово password, чтение остальных строк из исходного файла прекращается.

```
Stream<String> lines = Files.lines(path);
Optional<String> passwordEntry =
    lines.filter(s -> s.contains("password")).findFirst();
```



НА ЗАМЕТКУ. В отличие от класса FileReader, переносимость которого доставляла немало хлопот, поскольку он открывал файлы с локальной кодировкой символов, метод Files.lines() по умолчанию выбирает кодировку UTF-8. Впрочем, имеется возможность предоставить и другую кодировку в качестве аргумента типа Charset.

Исходный файл в конечном итоге придется закрыть. К сожалению, интерфейс Stream расширяет интерфейс AutoCloseable. Потоки ввода-вывода, обсуждавшиеся в главе 2, не требовали закрытия любых ресурсов. Но метод Files.lines() производит поток ввода-вывода, метод close() которого закрывает файл. Самый простой способ убедиться в том, что файл действительно закрыт, — воспользоваться блоком оператора try с ресурсами, внедренного в версии Java 7, как показано ниже.

```
try (Stream<String> lines = Files.lines(path)) {
    Optional<String> passwordEntry
        = lines.filter(s -> s.contains("password")).findFirst();
    ...
} // здесь будет закрыт поток ввода-вывода, а следовательно, и файл
```

Когда один поток ввода-вывода порождает другой, методы close() связываются в цепочку. Следовательно, приведенный выше код можно переписать и так, как показано ниже. Когда закрывается поток ввода filteredLines, вместе с ним закрывается базовый поток ввода, а затем исходный файл.

```
try (Stream<String> filteredLines
      = Files.lines(path).filter(s -> s.contains("password"))) {
    Optional<String> passwordEntry = filteredLines.findFirst();
    ...
}
```



НА ЗАМЕТКУ. Если требуется получить уведомление о закрытии потока ввода-вывода, то можно присоединить обработчик onClose(). Ниже показано, как проверить, что закрытие потока ввода filteredLines действительно приводит к закрытию базового потока.

```
try (Stream<String> filteredLines
      = Files.lines(path).onClose(() -> System.out.println("Closing"))
            .filter(s -> s.contains("password"))) { ... }
```

Если возникает исключение типа IOException при вводе строк в поток, это исключение заключается в оболочку исключения типа UncheckedIOException, которое генерируется при выполнении операции в потоке ввода-вывода. (Такое ухищрение необходимо потому, что операции в потоках ввода-вывода не объявляются для генерирования любых проверяемых исключений.)

Если требуется прочитать строки из другого источника, кроме файла, то для этой цели вызывается метод BufferedReader.lines() следующим образом:

```
try (BufferedReader reader
      = new BufferedReader(new InputStreamReader(url.openStream())));
    Stream<String> lines = reader.lines();
    ...
}
```

С помощью этого метода закрытие результирующего потока ввода не приводит к закрытию потока чтения. Именно по этой причине в заголовочной части оператора try должен быть размещен объект буферизированного потока чтения типа BufferedReader, а не объект потока ввода.



НА ЗАМЕТКУ. Около десяти лет назад в версии Java 5 был внедрен класс Scanner для замены громоздкого класса BufferedReader. К сожалению, разработчики прикладного программного интерфейса API в версии Java 8 решили внедрить метод lines() в класс BufferedReader, а не в класс Scanner.

Потоки ввода-вывода содержимого каталогов

Статический метод `Files.list()` возвращает поток типа `Stream<Path>` для ввода содержимого каталога. Это содержимое считывается по требованию, что дает возможность повысить эффективность процесса обработки каталогов с большим количеством записей.

При чтении содержимого каталога задействуются системные ресурсы, которые нужно закрывать. Поэтому следует воспользоваться блоком оператора `try`, как показано ниже.

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {  
    ...  
}
```



НА ЗАМЕТКУ. Подспудно в потоке ввода содержимого каталога используется другой поток типа `DirectoryStream` – интерфейс, внедренный в версии Java 7 для повышения эффективности обхода крупных каталогов. Этот интерфейс не имеет никакого отношения к потокам ввода-вывода, внедренным в версии Java 8. Он лишь расширяет интерфейс `Iterable`, а следовательно, может быть использован в усовершенствованном цикле `for` следующим образом:

```
try (DirectoryStream stream = Files.newDirectoryStream(pathToDirectory)) {  
    for (Path entry : stream) {  
        ...  
    }  
}
```

В версии Java 8 для тех же самых целей достаточно вызвать метод `Files.list()`.

В методе `Files.list()` вхождение в подкаталоги не предусматривается. Для обработки всех подкаталогов, присутствующих в каталоге, лучше воспользоваться методом `Files.walk()`:

```
try (Stream<Path> entries = Files.walk(pathToRoot)) {  
    // содержит все порожденные элементы каталога,  
    // обход которых совершается в глубину  
}
```

Для того чтобы ограничить глубину обхода дерева, достаточно вызвать метод `Files.walk(pathToRoot, depth)`. У обоих вариантов метода `walk()` имеется параметр типа `FileVisitOption...` с переменным количеством аргументов, но в настоящее время можно предоставить только один аргумент `FOLLOW_LINKS` для отслеживания символических ссылок.



НА ЗАМЕТКУ. Если требуется отфильтровать пути, возвращаемые при обходе каталога, включив в критерий фильтрации атрибуты файлов, хранящихся в каталоге, в том числе размер, дату и время создания или тип (файла, каталога, символьской ссылки), то вместо метода `walk()` следует воспользоваться методом `find()`. Этот метод вызывается с предикатной функцией, принимающей путь и объект типа `BasicFileAttributes`. Единственное преимущество такого способа состоит в его эффективности. А поскольку содержимое каталога все равно читается, то атрибуты файлов легко доступны.

Кодировка Base64

Кодировка Base64 позволяет преобразовать последовательность байтов в более длинную последовательность печатаемых символов в коде ASCII. Она используется для кодирования двоичных данных в сообщениях электронной почты и элементарной аутентификации по сетевому протоколу HTTP. В течение многих лет в наборе инструментальных средств JDK для этой цели предоставлялся закрытый, а следовательно, неиспользуемый класс `java.util.prefs.Base64`, а также недокументированный класс `sun.misc.BASE64Encoder`. И наконец, в версии Java 8 появились стандартные средства кодирования и декодирования по основанию Base64.

В кодировке Base64 для кодирования шести битов информации используются следующие 64 символа:

- 26 прописных букв английского алфавита от A до Z.
- 26 строчных букв английского алфавита от A до Z.
- 10 цифр от 0 до 9.
- 2 основных знака + и / или вспомогательные знаки - и _ (как безопасный вариант для обозначения URL и имен файлов).

Как правило, закодированная символьная строка не содержит разрывы строк, но по стандарту MIME, применяемому для электронной почты, требуется указывать разрыв строки "\r\n" через каждые 76 символов. Для кодирования запрашивается объект типа `Base64.Encoder` с одним из следующих статических методов из класса `Base64: getEncoder(), getUrlEncoder() или getMimeEncoder()`.

В классе `Base64.Encoder` имеются методы для кодирования массива байтов или буфера типа `ByteBuffer` из новой системы ввода-вывода (NIO). Ниже приведен характерный пример такого кодирования.

```
Base64.Encoder encoder = Base64.getEncoder();
String original = username + ":" + password;
String encoded =
    encoder.encodeToString(original.getBytes(StandardCharsets.UTF_8));
```

С другой стороны, можно заключить поток вывода в оболочку таким образом, чтобы автоматически кодировались все направляемые в него данные:

```
Path originalPath = ..., encodedPath = ...;
Base64.Encoder encoder = Base64.getMimeEncoder();
try (OutputStream output = Files.newOutputStream(encodedPath)) {
    Files.copy(originalPath, encoder.wrap(output));
}
```

Для декодирования все операции выполняются в обратном порядке:

```
Path encodedPath = ..., decodedPath = ...;
Base64.Decoder decoder = Base64.getMimeDecoder();
try (InputStream input = Files.newInputStream(encodedPath)) {
    Files.copy(decoder.wrap(input), decodedPath);
}
```

Аннотации

Аннотации представляют собой дескрипторы, которые вставляются в исходный код и могут быть обработаны некоторыми инструментальными средствами. В версии Java SE аннотации применяются для таких простых целей, как пометка нерекомендованных к употреблению языковых средств или подавление сообщений. Аннотации играют намного более важную роль в версии Java EE, где они применяются для настройки практически любого свойства приложения, замены громоздкого шаблонного кода и специальной настройки XML-разметки, которая вызывала немало трудностей в прежних выпусках Java EE.

В Java 8 были внесены следующие два усовершенствования в обработку аннотаций: повторяющиеся аннотации, а также аннотации к использованию типов. Это должно упростить аннотации к параметрам методов.

Повторяющиеся аннотации

Изначально аннотации предназначались для пометки методов и полей с целью их обработки. Ниже приведен характерный тому пример.

```
@PostConstruct public void fetchData() { ... } // вызов после построения  
@Resource("jdbc:derby:sample") private Connection conn;  
// вставить здесь ресурс
```

В данном контексте не имеет смысла применять аннотацию дважды. Вставить одно и то же поле двумя способами нельзя. Разумеется, разные аннотации к одному и тому же элементу вполне допустимы и весьма распространены, как показано ниже.

```
@Stateless @Path("/service") public class Service { ... }
```

Очень быстрый рост применения аннотаций привел к таким ситуациям, где могла бы потребоваться одна и та же аннотация. Например, для обозначения составного ключа в базе данных приходится указывать несколько столбцов:

```
@Entity  
@PrimaryKeyJoinColumn(name="ID")  
@PrimaryKeyJoinColumn(name="REGION")  
public class Item { ... }
```

Но это было невозможно, и поэтому аннотации были упакованы в контейнер следующим образом:

```
@Entity  
@PrimaryKeyJoinColumns({  
    @PrimaryKeyJoinColumn(name="ID")  
    @PrimaryKeyJoinColumn(name="REGION")  
})  
public class Item { ... }
```

Это очень неудобно, но больше не требуется в версии Java 8. И это все, что вам как пользователю аннотаций нужно знать. Если поставщик используемой вами библиотеки разрешил применение повторяющихся аннотаций, вам только останется воспользоваться ими.

Дело несколько усложняется для тех, кто реализует библиотеку. Ведь у интерфейса AnnotatedElement имеется следующий метод, получающий аннотацию к типу T, если таковой присутствует:

```
public <T extends Annotation> T getAnnotation(Class<T> annotationClass)
```

Что же должен делать этот метод, если имеется несколько аннотаций к одному и тому же типу, — возвращать только первую из них? Это могло бы привести к всевозможным видам нежелательного поведения унаследованного кода.

Для разрешения подобного затруднения создатель повторяющейся аннотации должен сделать следующее.

1. Составить аннотацию к аннотации в виде @Repeatable.
2. Предоставить контейнерную аннотацию.

Например, для разработки библиотеки, предназначеннной для простого блочного тестирования, возможно, придется определить повторяемую аннотацию @TestCase, чтобы использовать ее следующим образом:

```
@TestCase(params="4", expected="24")
@TestCase(params="0", expected="1")
public static long factorial(int n) { ... }
```

Ниже показано, каким образом может быть определена аннотация. Всякий раз, когда пользователь предоставляет две аннотации или более @TestCase, они автоматически заключаются в оболочку аннотации @TestCase.

```
@Repeatable(TestCase.class)
interface TestCase {
    String params();
    String expected();
}

interface TestCases {
    TestCase[] value();
}
```

Если в коде обработки аннотаций вызывается метод element.getAnnotation(TestCase.class) для элемента, представляющего метод factorial(), то из него возвращается пустое значение null. Дело в том, что элемент фактически аннотируется контейнерной аннотацией TestCase.

При реализации процессора повторяемых аннотаций оказывается проще воспользоваться методом getAnnotationsByType(). При вызове метода element.getAnnotationsByType(TestCase.class) просматривается любой контейнер TestCases и предоставляется массив аннотаций типа TestCase.



НА ЗАМЕТКУ. Все описанное выше имеет отношение к обработке аннотаций во время выполнения средствами прикладного программного интерфейса API. Для обработки аннотаций на уровне источника используется пакет javax.lang.model и javax.annotation.processing из прикладного программного интерфейса API. В подобных прикладных программных интерфейсах API отсутствует поддержка для просмотра контейнера. Поэтому приходится обрабатывать как отдельные аннотации, если они предоставляются однократно, так и контейнерные, если одна и та же аннотация предоставляется неоднократно.

Аннотации к использованию типов

До версии Java 8 аннотация применялась к *объявлению*. Объявление является частью кода, внедряющего новое имя. Ниже приведены два примера, в которых объявляемое имя выделено полужирным.

```
@Entity public class Person { ... }
@SuppressWarnings("unchecked") List<Person> people =
    query.getResultList();
```

В версии Java 8 имеется возможность аннотировать любое *использование типа*. Это может быть полезным в сочетании с инструментальными средствами, которые проверяют типичные программные ошибки. К числу типичных ошибок относится генерирование исключения типа `NullPointerException`, поскольку программист не предусмотрел то обстоятельство, что ссылка может оказаться пустой `null`. А теперь допустим, что переменные, которые вообще не должны быть пустыми, аннотированы как `@NonNull`. Инструментальное средство может проверить, что следующий код написан неверно:

```
private @NonNull List<String> names = new ArrayList<>();
...
names.add("Fred"); // нельзя сгенерировать исключение NullPointerException
```

Разумеется, инструментальное средство должно обнаруживать любой оператор, выполнение которого может привести к пустому значению `null` переменной `names`:

```
names = null; // средство проверки пустых значений уведомляет,
               // что этот оператор содержит ошибку
names = readNames(); // верно, если метод readNames()
                     // возвращает результат @NonNull String
```

На первый взгляд, размещать подобные аннотации повсюду в прикладном коде довольно хлопотно, но на практике подобных хлопот можно отчасти избежать благодаря простому эвристическому анализу. В средстве проверки пустых значений из библиотеки Checker Framework (<http://types.cs.washington.edu/checker-framework>) предполагается, что любые нелокальные переменные неявно аннотированы как `@NonNull`, но локальные переменные могут иметь пустое значение `null`, если только в коде не проявляется иное. Если метод может возвратить пустое значение `null`, его следует аннотировать как `@Nullable`. И такое аннотирование может оказаться не хуже, чем документирование поведения, приводящего к пустым значениям. (В документации на прикладной программный интерфейс Java API имеется более 5 тысяч упоминаний об исключении типа `NullPointerException`.)

В предыдущем примере переменная `names` объявлена как `@NonNull`. Такое аннотирование было возможно до версии Java 8. Но как выразить, что список элементов должен быть непустым? Логически это должно выглядеть так, как показано ниже. Такое аннотирование было невозможно до версии Java 8, но теперь оно стало допустимым.

```
private List<@NonNull String> names;
```



НА ЗАМЕТКУ. Такие аннотации не являются частью стандартной спецификации Java. В настоящее время отсутствуют стандартные аннотации, имеющие смысловые значения для использования типов. Все примеры, приведенные в этом разделе, взяты из библиотеки Checker Framework или специально придуманы автором книги.

Аннотации к использованию типов могут появиться в следующих местах прикладного кода.

- Рядом с аргументами обобщенных типов: `List<@NotNull String>, Comparator.<@NotNull String> reverseOrder()`.
- В любом месте массива: `@NotNull String[][] words` (элемент массива `words[i][j]` не является пустым), `String @NotNull [][] words` (массив `words` не является пустым), `String[] @NotNull [] words` (элемент массива `words[i]` не является пустым).
- В суперклассах и реализуемых интерфейсах: `class Image implements @Rectangular Shape`.
- В вызовах конструкторов: `new @Path String("/usr/bin")`.
- При приведении типов и проверках типа `instanceof @Path String`: `input, if (input instanceof @Path String)`. (Аннотации указываются только для обработки внешними инструментальными средствами. Они не оказывают никакого влияния на поведение кода при приведении типов и проверках типа `instanceof`.)
- В определениях исключений: `public Person read() throws @Localized IOException`.
- При указании границ типов и вместе с метасимволами: `List<@ReadOnly ? extends Person>, List<? Extends @ReadOnly> Person`.
- В ссылках на методы и конструкторы: `@Immutable Person::getName`.

Впрочем, имеется несколько мест в прикладном коде, которые не подлежат аннотированию. Соответствующие примеры приведены ниже.

```
@NotNull String.class // Недопустимо: литерал класса аннотировать нельзя
import java.lang.@NotNull String; // Недопустимо: импорт аннотировать нельзя
```

Нельзя также аннотировать сами аннотации. Например, аннотацию `@NotNull String name` нельзя снабдить аннотацией `@NotNull`. (Можно предоставить отдельную аннотацию, но ее нельзя применить к объявлению переменной `name`.)

Практическое применение такого рода аннотаций зависит от живучести инструментальных средств. Если вас действительно интересует истинный потенциал расширенного контроля типов, то начать рекомендуется с учебного материала по Checker Framework, доступного по адресу <http://types.cs.washington.edu/checker-framework/tutorial>.

Рефлексия параметров метода

Имена параметров теперь доступны посредством рефлексии. Это многообещающая возможность, поскольку она позволяет сократить шаблонный код аннотаций. Рассмотрим в качестве примера следующий вызов типичного метода из прикладного программного интерфейса JAX-RS:

```
Person getEmployee(@PathParam("dept") Long dept, @QueryParam("id") Long id)
```

В большинстве случаев имена параметров совпадают с именами аннотированных аргументов, или же их можно сделать одинаковыми. Если бы процессор аннотаций

умел читать имена параметров, то можно было бы просто написать приведенную ниже строку кода. В версии Java 8 это стало возможным благодаря новому классу `java.lang.reflect.Parameter`.

```
Person getEmployee(@PathParam Long dept, @QueryParam Long id)
```

К сожалению, для того чтобы вся необходимая информация появилась в файле класса, исходный код должен быть скомпилирован по команде `javac -parameters SourceFile.java`. Можно только надеяться, что создатели аннотаций охотно возьмут данный механизм на вооружение, чтобы когда-нибудь можно было бы обойтись без указанного параметра компилятора.

Различные незначительные изменения

В завершение этой главы упоминаются различные мелкие изменения в версии Java 8, которые могут оказаться полезными для вас. В этом разделе рассматриваются новые средства, внедренные в классы `Objects`, `Logger` и `Locale`, а также изменения в регулярных выражениях и технологии JDBC.

Проверки пустых значений

В класс `Objects` внедрены статические предикатные методы `isNull()` и `nonNull()`, которые могут оказаться полезными для работы с потоками ввода-вывода. Например, при вызове следующего метода проверяется, содержит ли поток ввода-вывода пустое значение `null`:

```
stream.anyMatch(Object::isNull)
```

А при вызове приведенного ниже метода удаляются все пустые значения.

```
stream.filter(Object::nonNull)
```

Отложенные сообщения

В методах `log()`, `logp()`, `severe()`, `warning()`, `info()`, `config()`, `fine()`, `finer()` и `finest()`, внедренных в класс `java.util.Logger`, теперь поддерживаются сообщения, составляемые по требованию. В качестве примера ниже приведен вызов одного из таких методов.

```
logger.finest("x: " + x + ", y:" + y);
```

Строка сообщения форматируется даже на таком уровне протоколирования, когда она вообще не используется. Поэтому лучше сделать следующий вызов:

```
logger.finest(() -> "x: " + x + ", y:" + y);
```

Теперь лямбда-выражение вычисляется только на уровне протоколирования FINEST, когда затраты на дополнительный вызов лямбда-выражения могут оказаться ниже всех остальных.

В классе `Objects`, описанном в главе 9, имеется также вариант метода `requireNonNull()`, в котором строка сообщений составляется по требованию. Ниже приведен пример вызова этого метода.

```
this.directions = Objects.requireNonNull(directions,
    () -> "directions for " + this.goal + " must not be null");
```

В общем случае, когда значение переменной `directions` не является пустым, ссылка `this.directions` просто устанавливается равной `directions`. Если же значение переменной `directions` оказывается пустым, то вызывается лямбда-выражение и генерируется исключение типа `NullPointerException` с сообщением, составляемым из возвращаемой строки.

Регулярные выражения

В версии Java 7 были внедрены именованные фиксируемые группы. Например, следующее регулярное выражение является достоверным:

```
(?<city>[\p{L} ]+), \s*(?<state>[A-Z]{2})
```

А в версии Java 8 появилась возможность использовать имена в методах `start()`, `end()` и `group()` из класса `Matcher` следующим образом:

```
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) {
    String city = matcher.group("city");
    ...
}
```

В классе `Pattern` теперь имеется метод `splitAsStream()`, разделяющий последовательность символов типа `CharSequence` по регулярному выражению, как показано ниже.

```
String contents =
    new String(Files.readAllBytes(path), StandardCharsets.UTF_8);
Stream<String> words =
    Pattern.compile("[\P{L}]+").splitAsStream(contents);
```

Последовательности небуквенных символов служат в качестве разделителей слов. Метод `asPredicate()` может быть использован для фильтрации символьных строк, совпадающих с регулярным выражением. В приведенном ниже примере показано, как это делается.

```
Stream<String> acronyms =
    words.filter(Pattern.compile("[A-Z]{2,}").asPredicate());
```

Региональные настройки

В региональных настройках указывается все, что нужно знать для представления информации пользователю с учетом языка, форматов дат, денежных сумм и прочих местных особенностей. Например, американские пользователи предпочитают даты, отформатированные следующим образом: `December 24, 2013` или `12/24/2013`, а немецкие пользователи — даты, отформатированные так: `24. Dezember 2013` или `24.12.2013`.

Раньше региональные настройки просто состояли из местоположения, языка и его разновидностей (например, двух систем правописания в таких исключительных случаях, как в норвежском языке). Но когда такие исключения стали правилом, Инженерная группа по развитию Интернета (Internet Engineering Task Force) издала памятную записку BCP 47 "Best Current Practices" (Современный передовой опыт; <http://tools.ietf.org/html/bcp47>), чтобы навести в данной области какой-то порядок. В настоящее время региональные настройки составляются из пяти компонентов.

1. Язык, указываемый двумя или тремя строчными буквами, например **en** (английский) или **de** (Deutsch — немецкий в Германии).
2. Письмо, указываемое четырьмя буквами, начиная с заглавной, например **Latn** (латынь), **Cyr1** (кириллица) или **Hant** (символы традиционного китайского письма). Это удобно, потому что в ряде языков, например, сербском, применяется латинское или кириллическое письмо, а некоторые китайские пользователи предпочитают традиционные символы упрощенным.
3. Страна, указываемая двумя прописными буквами или тремя цифрами, например **US** (Соединенные Штаты) или **CH** (Швейцария).
4. Разновидность языка (дополнительно). Коды разновидностей языков больше не применяются. Например, новонорвежское правописание норвежского языка теперь выражается другим кодом языка **nn** вместо кода **NY** разновидности норвежского языка, выражаемого кодом **no**.
5. Расширение (дополнительно). Расширения описывают локальные предпочтения для календарей (например, японского), чисел (тайских чисел) и т.д. В стандарте на уникод определяются некоторые из этих расширений. Такие расширения начинаются с символов **u-** и двухбуквенного кода, обозначающего, имеет ли расширение отношение к календарю (**ca**), числам (**nu**) и т.д. Например, расширение **u-nu-thai** обозначает применение тайских цифр. Другие расширения обозначаются совершенно произвольно и начинаются с символов **x-**, например **x-java**.

Тем не менее региональные настройки можно составлять и старым способом, например, следующим образом: `new Locale("en", "US")`. Но, начиная с версии Java 7, достаточно сделать следующий вызов: `Locale.forLanguageTag("en-US")`. В версии Java 8 внедрены методы для поиска региональных настроек, совпадающих с потребностями пользователей.

Языковые пределы — это символьная строка, обозначающая желательные для пользователя региональные характеристики с помощью метасимволов *****. Например, немецкоговорящий пользователь из Швейцарии может предпочесть отображение всей информации на немецком языке, но с учетом швейцарских особенностей. Это можно выразить с помощью двух объектов типа `Locale.LanguageRange`, обозначаемых символьными строками "**de**" и "***-CH**". При построении объекта типа `Locale.LanguageRange` можно дополнительно указать весовой коэффициент от **0** до **1**.

При наличии списка взвешенных языковых пределов и коллекции региональных настроек метод `filter()` составляет список совпадающих региональных настроек в убывающем порядке совпадения:

```
List<Locale.LanguageRange> ranges = Stream.of("de", "*-CH")
    .map(Locale.LanguageRange::new)
    .collect(Collectors.toList());
```

```
// список, содержащий объекты типа Locale.LanguageRange
// для заданных символьных строк
List<Locale> matches = Locale.filter(ranges,
    Arrays.asList(Locale.getAvailableLocales()));
// совпадающие региональные настройки:
// de, de-CH, de-AT, de-DE, de-GR, fr-CH, it_CH
```

Статический метод `lookup()` лишь обнаруживает наиболее подходящие региональные настройки следующим образом:

```
Locale bestMatch = Locale.lookup(ranges, locales);
```

В данном случае больше всего подходят региональные настройки `de`, что не очень интересно. Но если параметр `locales` содержит более ограниченный ряд региональных настроек, например, тех, в которых документ доступен, то такой механизм может оказаться полезным.

Технология JDBC

В версии Java 8 технология JDBC была обновлена до версии 4.2. В нее были внесены незначительные изменения. В частности, в классах `Date`, `Time` и `Timestamp` из пакета `java.sql` появились методы для взаимного преобразования в аналоги `LocalDate`, `LocalTime` и `LocalDateTime` этих классов из пакета `java.time`. В классе `Statement` теперь имеется метод `executeLargeUpdate()` для обновления таблицы базы данных, если количество строк в ней превышает величину `Integer.MAX_VALUE`.

В версии JDBC 4.1, входившей в состав Java 7, определен обобщенный метод `getObjetc(столбец, тип)` для классов `Statement` и `ResultSet`, где `тип` — это экземпляр типа `Class`. Например, в следующей строке кода:

```
URL url = result.getObject("link", URL.class)
```

извлекается объект `DATALINK` типа `URL`. Теперь предоставляется и соответствующий метод `setObject()`.

Упражнения

1. Напишите программу для сложения, вычитания, деления и сравнения чисел в пределах от 0 до $2^{32} - 1$, используя значения типа `int` и беззнаковые операции. Покажите, почему для этого нужны методы `divideUnsigned()` и `remainderUnsigned()`.
2. Для какого целого значения `n` метод `Math.negateExact(n)` генерирует исключение? (Подсказка: имеется только одно такое значение.)
3. По алгоритму Евклида, которому уже больше двух тысяч лет, вычисляется наибольший общий делитель двух чисел: $\text{gcd}(a, b) = a$, если b равно нулю, а иначе — $\text{gcd}(b, \text{rem}(a, b))$, где `rem` — остаток. Очевидно, что наибольший общий делитель `gcd` не должен быть отрицательным, даже если отрицательно a или b , поскольку его отрицание будет тогда большим делителем. Реализуйте данный алгоритм с помощью оператора `%`, метода `floorMod()` и функции `rem()`, производящей (неотрицательный) математический остаток. Какое из этих трех средств доставит меньше хлопот при обращении с отрицательными числовыми значениями?

4. Метод `Math.nextDown(x)` возвращает следующее по величине числовое значение с плавающей точкой, меньшее заданного числового значения x , на тот случай, если в ходе некоторого случайного процесса будет достигнуто точное значение x , хотя предполагалось получить числовое значение меньше x . Может ли это произойти на самом деле? Рассмотрим следующее выражение: `double r = 1 - generator.nextDouble()`, где `generator` — это экземпляр класса `java.util.Random`. Может ли оно вообще дать единичный результат и может ли вызов `generator.nextDouble()` привести к нулевому результату? Как говорится в документации, результат может быть получен в пределах от 0 включительно до 1 исключительно. Но если имеется 2^{53} чисел с плавающей точкой, то будет ли вообще достигнут нулевой результат? Конечно, будет. Генератор случайных чисел вычисляет следующее начальное случайное значение следующим образом: $\text{next}(s) = s \cdot m + a \% N$, где $m = 25214903917$, $a = 11$ и $N = 2^{48}$. Обратная величина m по модулю N составляет $v = 246154705703781$, а следовательно, случайное значение, предшествующее начальному случайному значению, можно вычислить следующим образом: $\text{prev}(s) = (s - a) \cdot v \% N$. Для получения числового значения типа `double` формируются два случайных числа, и каждый раз берутся старшие 26 и 27 битов. Если s равно 0, то $\text{next}(s)$ равно 11, следовательно, достигается требуемый результат: два последовательных числа, старшие биты которых равны нулю. А теперь станем действовать в обратном направлении, начав со следующего выражения: $s = \text{prev}(\text{prev}(\text{prev}(0)))$. Конструктор класса `Random` задает случайное значение $s = (\text{начальное_случайное_значение}^m \% N)$, поэтому достаточно предоставить ему случайное значение $s = \text{prev}(\text{prev}(\text{prev}(0)))^m = 164311266871034$, чтобы получить нулевой результат после двух вызовов метода `nextDouble()`. Но и это слишком очевидно. Поэтому сформируйте миллион предшествующих случайных значений, используя поток ввода-вывода, и выберите среди них наименьшее начальное случайное значение. *Подсказка:* после 376050 вызовов метода `nextDouble()` вы получите нулевой результат.
5. В начале главы 2 было показано, как подсчитать длинные слова в списке, сделав следующий вызов: `words.stream().filter(w -> w.length() > 12).count()`. Сделайте то же самое с помощью лямбда-выражения, но не прибегая к потокам ввода-вывода. Какая из этих операций будет выполнена быстрее при обработке длинного списка?
6. Пользуясь только методами из класса `Comparator`, определите для объектов типа `Point2D` компаратор, выполняющий полное упорядочение, т.е. возвращающий нулевое значение только в том случае, если объекты равны. *Подсказка:* сравнивайте сначала координаты x , а затем координаты y . Сделайте то же самое для объектов типа `Rectangle2D`.
7. Перепишите выражение `nullsFirst(naturalOrder()).reversed()`, не вызывая метод `reversed()`.
8. Напишите программу, демонстрирующую преимущества класса `CheckedQueue`.
9. Напишите методы, преобразующие объект типа `Scanner` в поток ввода слов, строк, целых чисел или числовых значений типа `double`. *Подсказка:* просмотрите исходный код метода `BufferedReader.lines()`.

10. Разархивируйте файл `src.zip` из набора инструментальных средств JDK. Используя метод `Files.walk()`, найдите все файлы исходного кода на Java, содержащие ключевые слова `transient` и `volatile`.
11. Напишите программу, получающую содержимое веб-страницы, защищенной паролем. Сделайте сначала следующий вызов: `URLConnection connection = url.openConnection();`. Затем составьте символьную строку `имя_пользователя:пароль` и закодируйте ее в кодировке Base64. И далее сделайте следующие один за другим вызовы: `connection.setRequestProperty("Authorization", "Basic " + закодированная_строка)`, `connection.connect()` и `connection.getInputStream()`.
12. Реализуйте аннотацию `TestCase` и напишите программу, загружающую класс с такими аннотациями и вызывающую аннотированные методы, чтобы проверить, дают ли они ожидаемые результаты. В качестве параметров и возвращаемых значений допускаются целочисленные значения.
13. Повторите предыдущее упражнение, но на этот раз постройте процессор аннотаций на уровне исходного кода, формирующий программу, при выполнении которой в ее методе `main()` производятся тесты. (Введение в обработку аннотаций на уровне исходного кода см. в главе 10 второго тома книги *Core Java, 9th Edition, Volume 2*, упоминавшейся во введении к этой книге.)
14. Продемонстрируйте применение метода `Objects.requireNonNull()`, приводящее к выводу более полезных сообщений об ошибках.
15. Воспользовавшись методами `Files.lines()` и `Pattern.asPredicate()`, напишите программу, действующую аналогично утилите `grep`, которая выводит все строки, содержащие результаты совпадения с регулярным выражением.
16. Воспользуйтесь регулярным выражением с именованными фиксируемыми группами для синтаксического анализа строки, содержащей названия города, штата, области или региона и почтовый индекс, который может состоять как из 5, так из 9 цифр.

Глава

9

Недостаточно освещенные языковые средства в Java 7

В этой главе...

- Изменения в обработке исключений
- Обращение с файлами
- Реализация методов `equals()`, `hashCode()` и `compareTo()`
- Требования к безопасности
- Прочие изменения
- Упражнения

После выпуска версии Java 7 большинство обозревателей сосредоточили все свое внимание на анализе новых языковых средств: строковых метках ветвления в операторе `switch`, двоичных литералах, знаках подчеркивания в литералах, усовершенствованном выведении типов и т.д. В этой главе рассматривается ряд изменений в библиотеке Java, которые были недостаточно освещены и, на мой взгляд, оказались намного более полезными в повседневной практике программирования на Java, чем ветвление по строковым меткам в операторе `switch` или двоичные литералы. Речь, в частности, пойдет об одном языковом средстве Java, очень полезном для повседневной практики программирования: операторе `try` с ресурсами.

В этой главе рассматриваются следующие основные вопросы.

- Применение оператора `try` с ресурсами и объекта класса, реализующего интерфейс `AutoCloseable`.
- Повторное генерирование первичного исключения оператором `try` с ресурсами, если закрываемый ресурс генерирует другое исключение.
- Перехват несвязанных исключений с помощью единственного оператора `catch`.
- Общий суперкласс `ReflectiveOperationException` для исключений в рефлексивных операциях.
- Применение интерфейса `Path` вместо класса `File`.
- Чтение и запись всех символов или всех строк из текстового файла по единственной команде.
- Статические методы из класса `Files` для копирования, перемещения, удаления и создания файлов и каталогов.
- Применение метода `Objects.equals()` для безопасной проверки на равенство пустым значениям.
- Упрощение реализации метода `hashCode()` с помощью метода `Objects.hash()`.
- Применение статического метода `compare()` в компараторе для сравнения чисел.
- Продолжающаяся поддержка аплетов и приложений Java Web Start в корпоративных средах и их беспersпективность для рядовых пользователей.
- Простое и удобное для всех изменение: символьную строку "+1" теперь можно преобразовать в целочисленное значение, не генерируя исключение.
- Изменения в классе `ProcessBuilder` с целью упростить переадресацию стандартных потоков ввода-вывода и сообщений об ошибках.

Изменения в обработке исключений

В начале этой главы будут рассмотрены языковые средства, внедренные в версии Java 7 для обработки исключений, поскольку они имеют решающее значение для написания надежных программ. Прежде чем перейти к рассмотрению более мелких изменений в обработке исключений, опишем вкратце оператор `try` с ресурсами.

Оператор `try` с ресурсами

В версии Java 7 предоставляется удобное сокращение следующего образца кода:

```
открыть ресурс
try {
    работать с ресурсом
}
finally {
    закрыть ресурс
}
```

при условии, что ресурс принадлежит классу, реализующему интерфейс `AutoCloseable`. В этом интерфейсе имеется следующий единственный метод:

```
void close() throws Exception
```



НА ЗАМЕТКУ. Имеется также интерфейс `Closeable`, производный от интерфейса `Closeable`. И у него имеется единственный метод `close()`, но он объявляется для генерирования исключения типа `IOException`.

В простейшей форме оператор `try` с ресурсами выглядит следующим образом:

```
try (Resource res = ...) {
    работать с ресурсом res
}
```

При выходе из блока оператора `try` метод `res.close()` вызывается автоматически. Ниже приведен типичный пример применения оператора `try` с ресурсами для чтения всех слов из файла.

```
try (Scanner in = new Scanner(Paths.get("/usr/share/dict/words"))) {
    while (in.hasNext())
        System.out.println(in.next().toLowerCase());
}
```

При нормальном выходе из блока оператора `try` или в отсутствие исключения метод `in.close()` вызывается точно так же, как и при использовании блока оператора `finally`. В операторе `try` с ресурсами можно указать несколько ресурсов, как показано в приведенном ниже примере.

```
try (Scanner in = new Scanner(Paths.get("/usr/share/dict/words"));
      PrintWriter out = new PrintWriter("/tmp/out.txt")) {
    while (in.hasNext())
        out.println(in.next().toLowerCase());
}
```

Независимо от того, каким образом происходит выход из блока оператора `try`, оба ресурса `in` и `out` закрываются, если они сконструированы. Как ни странно, реализовать такую возможность до версии Java 7 было трудно (см. упражнение 1 в конце этой главы).



НА ЗАМЕТКУ. Оператор `try` с ресурсами может и сам содержать операторы `catch` и `finally`. Эти операторы выполняются после закрытия ресурсов. Но на практике вряд ли стоит загромождать ими единственный оператор `try`.

Подавляемые исключения

Всякий раз, когда приходится иметь дело с вводом или выводом данных, встает нелегкая задача закрытия ресурса после исключения. Допустим, возникает исключение типа `IOException`, а когда закрывается ресурс, вызывается метод `close()`, генерирующий другое исключение.

Какое же исключение на самом деле перехватывается? В Java исключение, генерируемое в блоке оператора `finally`, снимает предыдущее исключение. И это действительно не очень удобно. Ведь пользователя может больше интересовать первоначальное исключение.

Оператор `try` с ресурсами изменяет это поведение на обратное. Когда исключение генерируется в методе `close()` одного из объектов типа `AutoCloseable`, первоначальное исключение генерируется повторно, а исключения, возникающие в результате вызова метода `close()`, перехватываются и присоединяются в качестве "подавляемых" исключений. Это очень полезный механизм, реализовать который вручную было бы затруднительно (см. упражнение 2 в конце этой главы).

При перехвате первичного исключения можно извлечь вторичные исключения, вызвав метод `getSuppressed()`:

```
try {  
    ...  
} catch (IOException ex) {  
    Throwable[] secondaryExceptions = ex.getSuppressed();  
}
```

Если же подобный механизм требуется реализовать самостоятельно в редких (как можно надеяться) случаях, когда нельзя воспользоваться оператором `try` с ресурсами, то нужно сделать следующий вызов:

```
ex.addSuppressed(secondaryException);
```



НА ЗАМЕТКУ. У классов `Throwable`, `Exception`, `RuntimeException` и `Error` имеются конструкторы, в которых можно запретить подавляемые исключения и трассировки стека. Когда подавляемые исключения запрещаются, вызов метода `addSuppressed()` не имеет никакого действия, а метод `getSuppressed()` возвращает массив нулевой длины. Когда же запрещаются трассировки стека, вызовы метода `fillInStackTrace()` также не имеют никакого действия, а метод `getStackTrace()` возвращает массив нулевой длины. Это может оказаться полезным для обработки ошибок, возникающих в виртуальной машине при недостатке оперативной памяти, или для тех языков программирования, которые действуют в виртуальной машине и в которых исключения применяются для разрыва вложенных вызовов методов.



ВНИМАНИЕ. Обнаружение вторичных исключений действует лишь в том случае, если оно не саботируется активно. В частности, если используется класс `Scanner` и если терпит неудачу сначала ввод данных, а затем и закрытие ресурса, то в классе `Scanner` перехватывается исключение, возникающее при вводе данных, закрывается ресурс и далее перехватывается исключение, возникающее при закрытии ресурса, а после этого генерируется совершенно другое исключение, никак не связанное с подавляемыми исключениями.

Перехват нескольких исключений

В версии Java SE 7 появилась возможность перехватывать несколько типов исключений в одном и том же операторе `catch`. Допустим, что для обработки отсутствующих файлов и неизвестных хостов (т.е. веб-узлов) предусмотрено одно и то же действие. В таком случае операторы `catch` можно объединить следующим образом:

```
try {
    код, способный генерировать исключения
}
catch (FileNotFoundException | UnknownHostException ex) {
    чрезвычайное действие для обработки отсутствующих файлов
    и неизвестных ресурсов
}
catch (IOException ex) {
    чрезвычайное действие для обработки всех остальных
    ошибок ввода-вывода
}
```

К такому средству следует прибегать лишь в том случае, если перехватываются исключения, типы которых не относятся к подклассам друг друга. Перехват нескольких исключений не только упрощает прикладной код, но и повышает его эффективность. Формируемые при этом байт-коды содержат единственный кодовый блок для общего оператора `catch`.



НА ЗАМЕТКУ. Если перехватывается несколько исключений, то переменная исключения `ex` неявно становится конечной. Например, новое значение нельзя присвоить переменной `ex` в теле оператора `catch FileNotFoundException | UnknownHostException ex { ... }`.

Упрощение обработки исключений для рефлексивных методов

В прошлом при вызове рефлексивного метода приходилось перехватывать несколько несвязанных друг с другом проверяемых исключений. Допустим, требуется сконструировать класс и вызвать его метод `main()`, как показано ниже.

```
Class.forName(className).getMethod("main").invoke(null, new String[] {});
```

Выполнение такого оператора может привести к исключению типа `ClassNotFoundException`, `NoSuchMethodException`, `IllegalAccessException` или `InvocationTargetException`. Разумеется, для перехвата этих исключений можно было бы воспользоваться средством, описанным в предыдущем разделе, объединив их в одном операторе `catch` следующим образом:

```
catch (ClassNotFoundException | NoSuchMethodException
      | IllegalAccessException | InvocationTargetException ex) { ... }
```

Но это все-таки очень неудобно. Откровенно говоря, не предоставить общий суперкласс для связанных вместе исключений считается неудачным проектным решением. И такие проектные просчеты были устранены в версии Java 7. В частности, для

перехвата всех подобных исключений в единственном обработчике был внедрен суперкласс ReflectiveOperationException. В следующей строке кода демонстрируется его применение в операторе catch:

```
catch (ReflectiveOperationException ex) { ... }
```

Обращение с файлами

Оператор try с ресурсами является моим излюбленным языковым средством в версии Java 7, но изменения в обработке файлов оказались не такими совершенными. Операции, которые раньше считались трудоемкими, например, чтение данных из файла в символьную строку или копирование одного файла в другой, теперь выполняются так просто, как они должны были выполняться все время. Это часть тех усилий, которые были направлены на разработку новой системы ввода-вывода NIO2, призванной обновить библиотеку NIO, внедренную в версии Java 1.4, выпущенной в 2002 году. (Было бы неверно включать слово new в название программного продукта, поскольку все новое неизбежно устаревает, и такое название выглядело бы неуместно.)

Прежде чем показывать, насколько просто теперь выполняются операции над файлами, следует упомянуть об интерфейсе Path, заменяющем класс File. Затем речь пойдет о том, как организуется чтение и запись данных в файлы. И наконец, будут рассмотрены операции над файлами и каталогами.

Пути

Интерфейс Path представляет имена каталогов, после которых, возможно, следуют имена файлов. Первой в пути оказывается корневая составляющая, например / или C:\. Допустимые корневые составляющие зависят от конкретной файловой системы. Путь, который начинается с корневой составляющей, называется *абсолютным*, а иначе — *относительным*. В приведенном ниже примере составляются абсолютный и относительный пути. Для составления абсолютного пути предполагается наличие Unix-подобной файловой системы, развернутой на компьютере.

```
Path absolute = Paths.get("/", "home", "cay");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

Статический метод Paths.get() получает одну или несколько символьных строк, соединяя их с помощью разделителя путей в используемой по умолчанию файловой системе (/ в Unix-подобной файловой системе или \ в Windows). Затем в этом методе выполняется синтаксический анализ полученного результата и генерируется исключение типа InvalidPathException, если результат (т.е. объект типа Path) оказывается недействительным путем в данной файловой системе. Методу Paths.get() можно также предоставить символьную строку с разделителями следующим образом:

```
Path homeDirectory = Paths.get("/home/cay");
```



НА ЗАМЕТКУ. Как и объект типа File, объект типа Path совсем не обязательно должен соответствовать файлу, который фактически существует. Ведь это всего лишь абстрактная последовательность имен. Для создания файла нужно сначала составить путь к нему, а затем вызвать соответствующий метод.

Объединение или разрешение путей относится к типичным операциям над файлами. В результате вызова метода `p.resolve(q)` возвращается путь согласно следующим правилам:

- если q — это абсолютный путь, то результатом оказывается путь q ;
 - в противном случае результат представляет собой сочетание путей “ p , а затем q ”, согласно правилам, принятым в данной файловой системе.

Допустим, что в приложении требуется найти файл его конфигурации относительно начального каталога. Ниже показано, каким образом объединяются соответствующие пути:

```
Path configPath = homeDirectory.resolve("myprog/conf/user.properties");
    // то же самое, что и вызов метода
    // homeDirectory.resolve(Paths.get("myprog/conf/user.properties"));
```

Имеется также служебный метод `resolveSibling()`, разрешающий путь относительно его родительского пути, выдавая родственный путь. Так, если объект `workPath` обозначает путь к каталогу `home/cay/myprog/work`, то в результате приведенного ниже вызова возвращается путь к каталогу `/home/cay/myprog/temp`.

```
Path tempPath = workPath.resolveSibling("temp");
```

Метод `relativize()` выполняет действия, противоположные действиям метода `resolve()`. В результате вызова `p.relativize(r)` получается путь `q`, из которого, в свою очередь, получается путь `r` при разрешении относительно пути `p`. Например, в результате вызова

```
Paths.get("/home/cay").relativize(Paths.get("/home/fred/myprog"))
```

получается путь `../fred/myprog`, при условии, что двоеточие `..` обозначает родительский каталог в данной файловой системе.

Метод `normalize()` удаляет любые лишние составляющие . и .. из пути (или все, что может считаться лишним в данной файловой системе). Например, в результате нормализации пути `/home/cay/..../fred/.myprog` получается путь `/home/fred/myprog`.

Метод `toAbsolutePath()` выдает абсолютный путь для заданного пути. Если путь уже не является абсолютным, то он разрешается относительно пользовательского каталога, т.е. того каталога, из которого вызывается виртуальная машина JVM. Так, если запустить программу на выполнение из каталога `/home/cay/myprog`, то в результате вызова метода `Paths.get("config").toAbsolutePath()` возвращается каталог `/home/cay/myprog/config`.

В интерфейсе Path имеется немало полезных методов для разделения и объединения одних путей с другими. В следующем примере кода демонстрируется применение наиболее полезных из этих методов.



НА ЗАМЕТКУ. Иногда может возникнуть потребность для взаимодействия с устаревшими прикладными программными интерфейсами API, в которых вместо интерфейса Path применяется класс File. Так, в интерфейсе Path имеется метод toFile(), тогда как в классе File – аналогичный метод toPath().

Чтение и запись данных в файлы

Класс Files позволяет легко и быстро выполнять типичные операции над файлами. Например, содержимое всего файла нетрудно прочитать следующим образом:

```
byte[] bytes = Files.readAllBytes(path);  
byte[] bytes = Files.readAllBytes(path);
```

Если требуется прочитать данные из файла в символьную строку, то с этой целью достаточно вызвать сначала метод readAllBytes(), как показано выше, а затем конструктор

```
String content = new String(bytes, StandardCharsets.UTF_8);
```

Но если требуется прочитать данные из файла в последовательный ряд строк, то достаточно вызвать такой метод:

```
List<String> lines = Files.readAllLines(path);
```

С другой стороны, если требуется записать символьную строку в файл, то нужно вызвать следующий метод:

```
Files.write(path, content.getBytes(StandardCharsets.UTF_8));
```

Имеется также возможность записать коллекцию строк следующим образом:

```
Files.write(path, lines);
```

А для присоединения заданного файла служит приведенный ниже вызов.

```
Files.write(path, lines, StandardOpenOption.APPEND);
```



НА ЗАМЕТКУ. По умолчанию во всех методах из класса Files, предназначенных для чтения или записи символов в файлы, используется кодировка UTF-8. В тех редких случаях, когда требуется другая кодировка, в качестве аргумента можно предоставить объект типа Charset. С другой стороны, в конструкторе класса String и методе getBytes() используется кодировка, принятая на данной платформе по умолчанию. В распространенных настольных операционных системах по-прежнему применяются устаревшие 8-разрядные кодировки, несовместимые с UTF-8, и поэтому кодировку приходится указывать всякий раз, когда выполняется взаимное преобразование символьных строк и байтов.

Как правило, при обращении с текстовыми файлами умеренной длины их содержимое проще обработать в виде одной символьной строки или списка символьных строк. Если же файлы крупные или двоичные, то для обращения с ними по-прежнему следует пользоваться уже знакомыми вам потоками ввода-вывода или чтения-записи, как показано ниже.

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader reader = Files.newBufferedReader(path);
Writer writer = Files.newBufferedWriter(path);
```

Приведенные выше служебные методы избавляют от необходимости обращаться к потокам ввода-вывода типа `FileInputStream`, `FileOutputStream`, `BufferedReader` или `BufferedWriter`. Иногда может потребоваться поток типа `InputStream`, например, для ввода веб-содержимого по заданному URL и сохранения этого содержимого в файле. Так, для копирования содержимого из потока ввода в файл делается следующий вызов:

```
Files.copy(in, path);
```

А для копирования файла в поток вывода делается такой вызов:

```
Files.copy(path, out);
```

Создание файлов и каталогов

Для создания нового каталога вызывается следующий метод:

```
Files.createDirectory(path);
```

Все составляющие пути, кроме последней, должны уже существовать. А для создания промежуточных каталогов можно сделать такой вызов:

```
Files.createDirectories(path);
```

Пустой файл можно создать следующим образом:

```
Files.createFile(path);
```

При вызове данного метода генерируется исключение, если файл уже существует. Проверка существования и создание файла являются атомарными операциями. Если файл не существует, он создается перед тем, как у кого-нибудь другого появится шанс сделать то же самое.

При вызове метода `path.exists()` проверяется, существует ли заданный файл или каталог, но ведь он может прекратить свое существование к тому моменту, когда происходит возврат из метода. Для создания временного файла или каталога в заданном или специальном месте файловой системы имеются служебные методы, примеры применения которых приведены ниже.

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

Здесь `dir` — это объект типа `Path`, а `prefix` и `suffix` — символьные строки, которые могут быть пустыми. Например, в результате вызова метода `Files.createTempFile(null, ".txt")` может быть возвращен путь вроде `/tmp/1234405522364837194.txt`.



НА ЗАМЕТКУ. Для чтения файлов из каталога служат методы `Files.list()` и `Files.walk()`, описанные в главе 8.

Копирование, перемещение и удаление файлов

Для того чтобы скопировать файл из одного места в другое, достаточно вызвать следующий метод:

```
Files.copy(fromPath, toPath);
```

Для перемещения файла (т.е. его копирования в другое место и удаления оригинала) достаточно вызвать приведенный ниже метод. С помощью этого же метода можно переместить пустой каталог.

```
Files.move(fromPath, toPath);
```

Если целевой файл существует, то операция копирования или перемещения файла завершится неудачно. Если же требуется перезаписать существующий целевой файл, то следует указать параметр `REPLACE_EXISTING`. А если требуется скопировать все атрибуты файла, то нужно указать параметр `COPY_ATTRIBUTES`. Оба эти параметра предоставляются следующим образом:

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,  
          StandardCopyOption.COPY_ATTRIBUTES);
```

Можно также указать, что операция перемещения должна быть атомарной. В этом случае гарантируется одно из двух: перемещение файла завершится успешно или же исходный файл останется существовать. Параметр `ATOMIC_MOVE` указывается следующим образом:

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

И наконец, для удаления файла достаточно вызвать следующий метод:

```
Files.delete(path);
```

Этот метод генерирует исключение, если файл не существует. Поэтому может быть лучше воспользоваться приведенным ниже методом. Методы удаления могут быть также использованы для удаления пустого каталога.

```
boolean deleted = Files.deleteIfExists(path);
```



НА ЗАМЕТКУ. Отдельного служебного метода для удаления или копирования непустого каталога не существует. Подробнее о коде для достижения подобных целей можно узнать из документации на интерфейс `FileVisitor` в прикладном программном интерфейсе API.

Реализация методов equals(), hashCode() и compareTo()

В версии Java 7 внедрено несколько методов, упрощающих обработку пустых значений в вездесущих методах equals() и hashCode(), а также сравнение числовых значений в методе compareTo().

Безопасная проверка на равенство пустым значениям

Допустим, требуется реализовать метод equals() для следующего класса:

```
public class Person {  
    private String first;  
    private String last;  
    ...  
}
```

Прежде всего, привести тип параметра данного метода к типу Person довольно хлопотно. Ниже приведен характерный тому пример.

```
public boolean equals(Object otherObject) {  
    if (this == otherObject) return true;  
    if (otherObject == null) return false;  
    if (getClass() != otherObject.getClass()) return false;  
    Person other = (Person) otherObject;  
    ...  
}
```

К сожалению, этих хлопот не стало меньше. Тем не менее ситуация улучшилась. Вместо того чтобы проверять, содержат ли поля first или last пустое значение null, достаточно сделать следующий вызов:

```
return Objects.equals(first, other.first) &&  
    Objects.equals(last, other.last);
```

В результате вызова метода Objects.equals(a, b) возвращается логическое значение true, если параметры a и b имеют пустое значение null; логическое значение false, если только параметр a имеет пустое значение null; а иначе — результат вызова a.equals(b).

 **НА ЗАМЕТКУ.** В общем, метод Objects.equals(a, b) целесообразно вызывать, если прежде был вызван метод a.equals(b).

Вычисление хеш-кодов

Рассмотрим вычисление хеш-кодов для класса из предыдущего раздела. Метод Objects.hashCode() возвращает код 0, если аргумент пустой. Следовательно, тело метода hashCode() можно реализовать следующим образом:

```
return 31 * Objects.hashCode(first) + Objects.hashCode(last);
```

Но ситуация изменилась к лучшему. В версии Java 7 был внедрен метод `Objects.hash()` с переменным числом аргументов, позволяющий указать любую последовательность значений, а их хеш-коды объединяются, как показано ниже.

```
return Objects.hash(first, last);
```



НА ЗАМЕТКУ. В методе `Objects.hash()` просто вызывается метод `Arrays.hash()`, существующий с версии Java 5. Но это не метод с переменным числом аргументов, что делает его менее удобным.



НА ЗАМЕТКУ. Всегда существовал способ вызова метода `toString()` в виде `String.valueOf(obj)` для безопасной обработки пустых значений. Если параметр `obj` имеет пустое значение, то возвращается символьная строка `"null"`. Если же это вас не устраивает, вы можете вызвать метод `Object.toString()` и предоставить значение, которое должно быть использовано вместо пустого, например, следующим образом: `Object.toString(obj, "")`.

Сравнение числовых типов

При сравнении целочисленных значений в компараторе возникает искушение возвратить их разность, поскольку разрешается возвращать любое отрицательное или положительное число — важен только знак. Допустим, требуется реализовать класс `Point` со следующим методом:

```
public int compareTo(Point other) {  
    int diff = x - other.x;  
    // риск переполнения присутствует  
    if (diff != 0) return diff;  
    return y - other.y;  
}
```

Но при этом возникает следующее затруднение: если значение `x` велико, а значение `other.x` — отрицательно, то вычисление их разности приведет к переполнению. Это доставит немало хлопот при использовании метода `compareTo()` (см. упражнение 8 в конце главы). Начиная с версии Java 7, с этой целью лучше воспользоваться статическим методом `Integer.compare()`, как показано ниже.

```
public int compareTo(Point other) {  
    int diff = Integer.compare(x, other.x);  
    // риск переполнения отсутствует  
    if (diff != 0) return diff;  
    return Integer.compare(y, other.y);  
}
```

В прошлом некоторые пользовались выражением `new Integer(x).compareTo(other.x)`, но это приводило к двум упакованным целочисленным значениям. У статического метода `Integer.compare()` имеется параметр типа `int`. Аналогичные статические методы были внедрены и в классы `Long`, `Short`, `Byte`, `Boolean`. Так, если требуется сравнить два значения типа `char`, их можно благополучно вычесть, поскольку данная операция теперь не приведет к переполнению. (Это же справедливо и для значений типа `short` или `byte`.)



НА ЗАМЕТКУ. Статический метод compare() существует для классов Double и Float, начиная с версии Java 1.2.

Требования к безопасности

После выпуска версии Java 1.0 в 1995 году воображение большинства программистов поразили, прежде всего, аплеты — удаленно обслуживавшийся прикладной код, выполнявшийся в веб-браузере пользователя. Разработчики Java вполне осознавали, что выполнение удаленного кода представляло известный риск нарушения безопасности, и поэтому они разработали модель в виде “песочницы”, препятствовавшей действию любых разрушительных инструкций.

Вскоре после этого одни академические исследователи обнаружили в реализации Java ряд изъянов, которые были немедленно устраниены, а другие исследователи высказывали недовольство тем, что модель безопасности в Java была слишком сложной, и не было никакой гарантии, что ее темные стороны надежно защищены от проникновения извне. В то время я лично не принимал все эти опасения всерьез, поскольку подавляющее большинство потребителей аплетов на Java пользовались операционной системой Windows, которая была намного менее безопасной и гораздо более сложной.

Действие аплетов ограничивалось визуальными эффектами и сетевыми соединениями с первоначальным хостом, что сдерживало многих разработчиков приложений. Им требовался локальный доступ к устройствам для хранения, вывода и прочих операций с данными. В 2001 году технология Java Web Start предоставила довольно эффективное расширение “песочницы”, сравнимое с современными расширениями HTML 5 для локального доступа к устройствам. К сожалению, технология Java Web Start оказалась плохо понятой, не интегрированной с аплетами и не нашедшей энергичной поддержки.

Вместо этого многие разработчики приложений просто подписывали сертификаты на свои распространяемые через Интернет программы, что давало им полное право делать все, что угодно, на пользовательском компьютере. Подписание сертификатов на прикладные программы в коммерческих организациях стало доступно всякому, желающему понести определенные затраты и хлопоты. Появилась также возможность самостоятельно, хотя и бесцельно подписывать сертификаты на свои прикладные программы или же предлагать пользователям соглашаться на выполнение аплетов без сертификатов. С выпуском каждой новой версии предупреждения о возможных нарушениях безопасности постепенно становились все менее резкими до тех пор, пока они вообще не смягчились до уровня малозаметных помех. И это было очень плохо.

Между тем корпорация Microsoft, проведя немалый объем технических работ, постаралась закрыть бреши в защите Windows, что заставило злоумышленников обратить основное внимание на скрытые уязвимые места в Java. В 2010 году, когда компания Oracle приобрела компанию Sun Microsystems, ее разработчики унаследовали весьма ограниченную инфраструктуру для защиты от атак злоумышленников без надежных средств для обновления клиентских виртуальных машин. Злоумышленникам все чаще удавалось пользоваться в своих злонамеренных целях программными ошибками в реализации Java. И в этом отношении полностью оправдались опасения первых исследователей модели защиты в Java, предупреждавших об обширном поле

деятельности для совершения атак. И только в 2013 году компании Oracle удалось найти эффективное противодействие подобным атакам. Тем не менее работа над совершенствованием управления клиентскими виртуальными машинами продолжается.

В настоящее время компания Oracle заявляет, что не собирается больше уделять основное внимание защите аплетов Java для бытовых пользователей и приложений Web Start, совместно называемых насыщенными интернет-приложениями (RIA). Компания Oracle продолжает устранять уязвимости в модели защиты Java и разрабатывает инструментальные средства, пригодные для корпоративного развертывания, чтобы с их помощью можно было надежно развертывать устаревшие насыщенные интернет-приложения. С коммерческой точки зрения в этом есть свой смысл. Ожидается, что бытовые пользователи постепенно станут переходить с домашних ПК на планшетные компьютеры и смартфоны. Ведь в браузерах этих мобильных устройств отсутствует поддержка виртуальной машины Java. Что же касается небытовых пользователей, то для них выгоднее по-прежнему сопровождать устаревшие приложения.

Успешный выпуск версии Java 7 свидетельствует о том, что компания Oracle со-средоточила главное внимание на соблюдении правил защиты. Начиная с января 2014 года, сертификаты на насыщенные Интернет-приложения, выполняющиеся за пределами "песочницы", должны подписываться в коммерческих центрах и органах сертификации. Еще одно требование направлено на то, чтобы предотвратить атаки, изменяющие назначение прикладной программы. Ведь в настоящее время злоумышленник может воспользоваться архивным JAR-файлом приложения, сертификат на которое законно подписан сторонней организацией, разместив его для свободного доступа на своем веб-сайте и эксплуатируя некоторые уязвимости в этом стороннем приложении. Поэтому, начиная с января 2014 года, в манифестах всех архивных JAR-файлов должна присутствовать одна из следующих записей:

Permissions: sandbox

или

Permissions: all-permissions

А поскольку запись манифеста находится в архивном JAR-файле, то она подписана и не может быть видоизменена. Клиентская программа на Java не разрешит выполнение клиента с полномочиями all-permission в "песочнице", предотвращая атаки так называемой "попутной" загрузки там, где аплет выполняется без согласия пользователя. Безусловно, по-прежнему существует угроза атак для тех пользователей, которые привыкли соглашаться на любой диалог, чреватый нарушением безопасности. Для того чтобы затруднить такой диалог, в манифест введена еще одна, хотя и необязательная запись, разрешающая загрузку прикладной программы только по одному из заданных URL:

Codebase: <https://www.mycompany.com> www.mycompany.com:8080



НА ЗАМЕТКУ. Аплеты можно было всегда вызывать из сценариев JavaScript, что считается еще одним сомнительным решением с точки зрения безопасности. Если вы пользуетесь такой возможностью в своей прикладной программе, то можете свести к минимуму риск изменить ее назначение, введя в манифест запись Caller-Allowable-Codebase: <https://www.mycompany.com> перед тем, как подписать сертификат.

В целом итоги разработок модели безопасности в Java оказались весьма неутешительными: большие надежды сделать Java универсальной платформой для

выполнения удаленного кода так и не оправдались. Язык Java стал бы такой платформой, если бы была предложена более убедительная "песочница", контроль кода вне "песочницы" — более жестким, реагирование на бреши в защите — более согласованным, а виртуальные клиентские машины — надежно обновлялись. Но вряд ли стоит останавливаться на том, что могло бы быть. В настоящий момент Java уже не является платформой, пригодной для широкого распространения клиентских приложений через Интернет.

Если вам приходится сопровождать аплет или приложение Java Web Start для бытовых пользователей, постарайтесь отказаться от этого. Если же ваша прикладная программа предназначена для особых целей (например, разработки программного обеспечения, редактирования изображений или обработки документов), предложите пользователям своей программы установить Java или же включить виртуальную машину Java в комплект установки. А если ваша прикладная программа имеет общее назначение (например, предназначена для любителей компьютерных игр), то рассмотрите возможность применения другой технологии — возможно, HTML 5.

 **НА ЗАМЕТКУ.** Если вы решите предложить пользователям своей программы установить Java в Windows, направив их на веб-страницу по адресу <http://java.com>, перед вами встанет еще одно препятствие: им будет предложено установить пользующееся дурной славой дополнение от компании Ask.com (рис. 9.1). В таком случае у вас имеются две возможности. Во-первых, предложить пользователям установить JDK, направив их на веб-страницу по адресу www.oracle.com/technetwork/java/javase/downloads и предоставив им инструкции по обращению с этой постоянно меняющейся страницей. И во-вторых, включить виртуальную машину Java в комплект установки своей прикладной программы, но тогда вы берете на себя обязательство выполнять обновления, поскольку компания Oracle не предоставляет для этого эффективный механизм.



Рис. 9.1. По умолчанию установщик Windows JRE устанавливает дополнение от компании Ask.com в виде панели инструментов Ask

Насыщенные интернет-приложения можно эффективно защитить в корпоративной среде при наличии контроля над этими приложениями и клиентскими компьютерами. Кроме того, придется в плотную заняться упаковкой приложений и подготовиться к обновлению клиентских виртуальных машин, как только станут доступными обновления системы безопасности.



НА ЗАМЕТКУ. Для более полного управления корпоративными насыщенными интернет-приложениями можно предложить свод правил их развертывания на машинах конечных пользователей. Этот непростой процесс подробнее объясняется в документации по адресу http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/security/deployment_rules.html.

Прочие изменения

Как и в аналогичном разделе предыдущей главы, в этом разделе описывается целый ряд менее значительных языковых средств, которые появились в версии Java 7 и могут оказаться полезными и интересными.

Преобразование символьных строк в числа

Какой результат получался при выполнении приведенного ниже фрагмента кода до версии JDK 1.7? Можете погладить себя по голове, если знаете ответ на этот вопрос. Выполнение первой строки из данного фрагмента кода всегда приводило к действительному числовому значению **+1,0** с плавающей точкой, тогда как целочисленное значение **+1**, предполагавшееся во второй строке кода, не было действительным до появления версии Java 7.

```
double x = Double.parseDouble("+1.0");
int n = Integer.parseInt("+1");
```

Этот недостаток был исправлен во всех методах, где числовые значения типа `int`, `long`, `short`, `byte` и `BigInteger` составляются из символьных строк. Но таких методов намного больше, чем кажется на первый взгляд. Помимо методов `parse(Int|Long|Short|Byte)`, имеются методы типа `decode`, обрабатывающие шестнадцатеричные и восьмеричные входные значения, а также методы типа `valueOf`, выдающие объекты-оболочки. Аналогичным образом обновлен и конструктор `BigInteger(String)`.

Глобальный регистратор

Для того чтобы стимулировать протоколирование даже в простейших случаях, в классе `Logger` появилась возможность получать глобальный регистратор. Он был реализован таким образом, чтобы максимально упростить его применение с целью вводить трассировочные операторы, например:

```
Logger.global.finest("x=" + x);
```

вместо

```
System.out.println("x=" + x);
```

К сожалению, такую переменную экземпляра придется инициализировать где-нибудь в другом месте, и если в статическом коде инициализации произойдет протоколирование, то могут возникнуть взаимоблокировки. Поэтому, начиная с версии Java 6, метод `Logger.global()` перестал быть рекомендованным к употреблению. Вместо этого предполагалось вызывать метод `Logger.getLogger(Logger.GLOBAL_LOGGER_NAME)`, что вряд ли можно было счесть простым и быстрым решением проблемы протоколирования. А в версии Java 7 появилась возможность вызывать метод `Logger.getGlobal()`, что не так уж плохо.

Проверки на пустые значения

В классе `Objects` имеются методы типа `requireNonNull` для организации удобных проверок параметров на пустые значения. Ниже приведен простейший пример организации такой проверки.

```
public void process(String directions) {  
    this.directions = Objects.requireNonNull(directions);  
    ...  
}
```

Если параметр `directions` имеет пустое значение `null`, то генерируется исключение типа `NullPointerException`, что на первый взгляд не кажется таким уж значительным достижением. Но если обратиться к трассировке стека, то, обнаружив причину ошибки в вызове метода `requireNonNull()`, можно сразу же выяснить, что именно было сделано неверно. Генерируемое исключение можно также снабдить соответствующим сообщением, как показано ниже.

```
this.directions = Objects.requireNonNull(directions,  
    "directions must not be null");
```

Класс ProcessBuilder

До версии Java 5 вызов метода `Runtime.exec()` был единственным способом выполнить внешнюю команду из приложения на Java. В версии Java 5 был внедрен класс `ProcessBuilder`, предоставляющий больше возможностей для управления процессом, запускаемым на уровне операционной системы. В частности, средствами класса `ProcessBuilder` можно внести изменения в рабочий каталог.

В версии Java 7 в классе `ProcessBuilder` появились служебные методы для связывания файлов со стандартными потоками ввода-вывода из процесса данных и сообщений об ошибках. Ниже приведен характерный тому пример.

```
ProcessBuilder builder = new ProcessBuilder(  
    "grep", "-o", "[A-Za-z_][A-Za-z_0-9]*");  
builder.redirectInput(Paths.get("Hello.java").toFile());  
builder.redirectOutput(Paths.get("identifiers.txt").toFile());  
Process process = builder.start();  
process.waitFor();
```



НА ЗАМЕТКУ. В версии Java 8 в классе `Process` появился метод `waitFor()` с параметром блокировки по времени. Ниже показано, каким образом вызывается этот метод.

```
boolean completed = process.waitFor(1, TimeUnit.MINUTES);
```

Кроме того, в версии Java 7 в классе ProcessBuilder появился новый метод `inheritIO()`, который задает стандартные потоки ввода-вывода данных и сообщений об ошибках из процесса в соответствующие потоки ввода-вывода из программы на Java. Например, при выполнении приведенного ниже фрагмента кода вывод из внешней команды `ls` направляется в поток `System.out`.

```
ProcessBuilder builder = new ProcessBuilder("ls", "-al");
builder.inheritIO();
builder.start().waitFor();
```

Класс URLClassLoader

Допустим, что требуется написать на Java программу, автоматизирующую выполнение блочных тестов в среде JUnit. Для загрузки класса `JUnitCore` потребуется загрузчик классов, читающий архивные JAR-файлы JUnit:

```
URL[] urls = {
    new URL("file:junit-4.11.jar"),
    new URL("file:hamcrest-core-1.3.jar")
};

URLClassLoader loader = new URLClassLoader(urls);
Class<?> klass = loader.loadClass("org.junit.runner.JUnitCore");
```

До версии Java 7 выполнение такого кода могло привести к утечкам ресурсов. Поэтому в версии Java 7 был просто внедрен метод `close()` для закрытия загрузчика классов. Теперь класс `URLClassLoader` реализует интерфейс `AutoCloseable`, а следовательно, можно воспользоваться оператором `try` с ресурсами следующим образом:

```
try (URLClassLoader loader = new URLClassLoader(urls)) {
    Class<?> klass = loader.loadClass("org.junit.runner.JUnitCore");
    ...
}
```



ВНИМАНИЕ. Пользоваться классами после закрытия загрузчика классов вообще нельзя. В противном случае эти классы не смогут функционировать нормально, если в них потребуется загрузить другие классы.

Класс BitSet

Класс `BitSet` предоставляет множество целочисленных значений в виде последовательности битов, где *i*-й бит устанавливается, если множество содержит целочисленное значение *i*. Благодаря этому повышается эффективность операций над множеством. В частности, операции объединения, пересечения, дополнения становятся простыми логическими операциями И, ИЛИ, НЕ. Для построения множеств битов в версии Java 7 внедрены соответствующие методы. Ниже приведены некоторые примеры их применения.

```
byte[] bytes = { (byte) 0b10101100, (byte) 0b00101000 };
BitSet primes = BitSet.valueOf(bytes);
// {2, 3, 5, 7, 11, 13}
```

```
long[] longs = { 0x100010116L, 0x1L, 0x1L, 0L, 0x1L };
BitSet powersOfTwo = BitSet.valueOf(longs);
// {1, 2, 4, 8, 16, 32, 64, 128, 256}
```

Обратные операции выполняют методы `toByteArray()` и `toLongArray()`, как показано ниже.

```
byte[] bytes = powersOfTwo.toByteArray();
// [0b00010110, 1, 1, 0, 1, 0, 0, 0, 1, ...]
```



НА ЗАМЕТКУ. В версии Java 8 в классе `BitSet` появился метод `stream()`, выдающий поток ввода-вывода типа `IntStream`.

Упражнения

1. Реализуйте фрагмент кода для построения потока ввода в виде объекта типа `Scanner` и потока чтения в виде объекта типа `PrintWriter`, как было показано в конце раздела “Оператор `try` с ресурсами” ранее в этой главе, но на этот раз без оператора `try` с ресурсами. Непременно закройте оба объекта, при условии, что они правильно построены. Для этого вам придется принять во внимание следующие условия.
 - Конструктор класса `Scanner` генерирует исключение.
 - Конструктор класса `PrintWriter` генерирует исключение.
 - Метод `hasNext()`, `next()` или `println()` генерирует исключение.
 - Метод `in.close()` генерирует исключение.
 - Метод `out.close()` генерирует исключение.
2. Усовершенствуйте код из предыдущего упражнения, добавив любые исключения, генерируемые методом `n.close()` или `out.close()`, как подавляемые по отношению к первоначальному исключению, если таковое возникло.
3. Если вы генерируете исключение, перехватываемое в блоке оператора `catch`, перехватывающего несколько исключений, то как указать тип этого исключения в объявлении `throws` внешнего метода? Рассмотрите в качестве примера следующий фрагмент кода:

```
public void process() throws ... {
    try {
        ...
        catch (FileNotFoundException | UnknownHostException ex) {
            logger.log(Level.SEVERE, "...", ex);
            throw ex;
        }
    }
```

4. В каком из других мест библиотеки Java вам встречались ситуации, где можно было извлечь выгоду из блока оператора `catch`, перехватывающего несколько исключений, а еще лучше — из общих для исключений суперклассов? (Подсказка: синтаксический анализ XML-разметки.)

5. Напишите программу, читающую все символы из файла и записывающую их в обратном порядке. Воспользуйтесь для этого методами `Files.readAllBytes()` и `Files.write()`.
6. Напишите программу, читающую все строки из файла и записывающую их в обратном порядке. Воспользуйтесь для этого методами `Files.readAllLines()` и `Files.write()`.
7. Напишите программу, читающую содержимое веб-страницы и сохраняющую его в файле. Воспользуйтесь для этого методами `URL.openStream()` и `Files.copy()`.
8. Реализуйте метод `compareTo()` в классе `Point` из раздела “Сравнение числовых типов” ранее в этой главе, не пользуясь методом `Integer.compareTo()`.
9. Реализуйте методы `equals()` и `hashCode()` в следующем классе:

```
public class LabeledPoint {  
    private String label;  
    private int x;  
    private int y;  
    ...  
}
```

10. Реализуйте метод `compareTo()` для класса `LabeledPoint` из предыдущего упражнения.
11. Используя класс `ProcessBuilder`, напишите прикладную программу, выполняющую внешнюю команду `grep -r` для поиска номеров кредитных карточек во всех файлах любого подкаталога из начального каталога пользователя. Организуйте накопление всех номеров, обнаруживаемых в файлах.
12. Преобразуйте прикладную программу из предыдущего упражнения в аплет или приложение Java Web Start. Допустим, что эту программу требуется предоставить пользователям для проверки безопасности. Упакуйте ее таким образом, чтобы ее можно было установить с помощью установщика Windows JRE. Что нужно сделать вам и что сделать пользователям для установки программы с вашего веб-сайта?

Преодметныи цказатель

А

Аннотации

- к использованию типов, 175
- к объявлению, 175
- назначение, 173
- обработка, особенности, 174
- повторяющиеся, 173

Б

Библиотеки

- AWT, назначение, 83
- Swing, назначение, 83

Будущие действия

- завершаемые, 139
- конвейер составления, 139
- назначение, 138
- составление, 139

В

Временная шкала

- отчет времени, 113
- условия применения, 113

Время

- абсолютное, точный отчет, 113
- всемирное скоординированное, обозначение, 120
- местное
 - методы расчета, 118
 - обозначение, 115
- поясное
 - методы расчета, 120
 - обозначение, 115
- синтаксический анализ, 124
- форматирование, средства, 122

Г

Глобальный регистратор, получение, 198

Д

Даты

- коррекция, методы, 117
- локальные
 - методы расчета, 115
 - обозначение, 115
- поясные, обозначение, 115
- синтаксический анализ, 124
- форматирование, средства, 122

Дисперсия

- используемая по месту, понятие, 75
- типов, понятие, 74

З

Закрытие загрузчика классов, особенности, 200

И

Изменения в технологии JDBC, 180

- Интерпретатор Nashorn
 - вызов методов, 149
 - выполнение
 - из кода Java, 148
 - из командной строки, 146
 - и JavaFX, 160
 - назначение, 146
 - обработка
 - исключений, 156
 - лямбда-выражений, 154
 - массивов, 152
 - символьных строк, 151
 - списков и отображений, 153
 - чисел, 151
 - построение объектов, 150
 - расширение классов и реализация
 - интерфейсов Java, 154
- цикла REPL, назначение, 147

Исключения

- в лямбда-выражениях, обработка, 72
- в рефлексивных методах, обработка, 187
- многие, перехват, 187
- подавляемые, обработка, 186
- проверяемые, в лямбда-выражениях, 24

К

Кодировка Base64

- назначение, 172
- операции кодирования и декодирования, 172
- принцип действия, 172

Коллекции

- класс Collections и его методы, 169
- компараторы, назначение, 167
- новые методы, назначение, 167

Л**Лямбда-выражения**

- вывод типа результата из контекста, 22
- и обобщения, 74
- как замыкания, 27
- область действия
 - переменных, 27
 - пределы, 29
- определение, 18
- отложенное выполнение, 62
- параметры, 64
- преобразования в функциональные
 - интерфейсы, 23
- происхождение названия, 21
- синтаксис, 21
- составляющие, 27

М**Методы**

- автоматического обновления,
 - применение, 133
- атомарной установки переменных, 129
- для завершаемых будущих действий, 140
- обработки
 - пустых значений, назначение, 193
 - регулярных выражений, назначение, 178
- по умолчанию
 - неоднозначность определения, правила разрешения, 30
 - определение, 30
- проверки на пустые значения,
 - назначение, 199
- рефлексия параметров, 176
- сведения
 - применение, 46
 - разновидности, 42
- статические
 - в интерфейсах, назначение, 32
 - в классах, назначение, 32
- управление внешними процессами,
 - назначение, 199

Н

- Номинальная типизация, определение, 64
- Оператор try с ресурсами
 - назначение, 185
 - применение, 185
- Операции
 - арифметические, точные, 166
 - групповые
 - назначение, 134
 - порог параллелизма, 134
 - разновидности, 134

- над множествами, выполнение, 200
- над моментами и промежутками времени,
 - методы, 114
- над файлами и каталогами,
 - выполнение, 190
- одноместные, назначение, 76
- оконечные
 - выполнение, 42
 - определение, 42
- отложенные, выполнение по требованию, 70
- параллельные, с массивами, 137
- произвольного накопления данных,
 - принцип действия, 130
- распараллеливание, 71
- сведения
 - ассоциативные, 46
 - определение, 42
 - разновидности, 42
- составные, выполнение, 69

О

- Отложенные сообщения, составление по требованию, 177
- Отсчет времени
 - начиная с эпохи, 113
 - особенности, 113
 - синхронизация, 113

П

- Передача кодовых блоков на исполнение, примеры, 18

- Переменные
 - атомарные, установка, 129
 - действительно конечные, определение, 28
 - свободные
 - определение, 27
 - фиксация лямбда-выражением, 27
- Переход на летнее и зимнее время, учет, 122
- Потоки ввода-вывода
 - методы, назначение, 39; 55
 - назначение, 36
- накопление
 - данных в отображении, 49
 - результатов, особенности, 47
- объединение, 41
- объектов, 54
- операции
 - группирования и разделения, 50
 - сведения, 42; 46

- организация конвейера операций, 37
- отличия от коллекций, 37
- параллельные и последовательные, 55
- преобразования

без сохранения состояния, 41
 с сохранением состояния, 41
 примитивных типов, 53
 проверка пустых значений, 177
 содержимого каталогов, методы, 171
 создание, 38
 строк, методы, 170
 упорядоченные и неупорядоченные, 56
 Примеры программ
 загрузка, 13

Пути
 абсолютные и относительные,
 составление, 188
 корневая составляющая, 188
 нормализация, 189
 правила разрешения, 189
 разделение и объединение, 189

P

Региональные настройки
 назначение, 51
 основные составляющие, 179
 форматирование даты и времени, 123
 языковые пределы, 179

C

Секунды
 потерянные, ввод, 113
 точное определение, 112
 Символьные строки
 интерполяция, 158
 преобразование в числа, 198
 соединение и разделение, 164
 Ссылки
 на конструкторы
 назначение, 25
 применение, 26
 формирование, 26
 на методы
 механизм действия, 25
 определение, 24
 передача, 24

Сценарии командного процессора
 ввод данных, 158
 выполнение команд, 157
 назначение, 156
 написание, 157
 недостатки, 157

T

Технология JavaFX
 анимация и спецэффекты, 103
 внедрение, 83

декоративные элементы управления, 105
 компоновка ППИ
 визуальная, 92
 программная, 93
 разновидности, 96
 обработка событий, 85
 основные понятия, 84
 панели, назначение, 93
 построители, назначение, 97
 применение, примеры, 84
 свойства
 назначение, 86
 получение и установка значений, 86
 привязка, 88
 реализация, 86
 таблицы стилей CSS, применение, 101
 языки разметки FXML, назначение, 98

Типы данных

Optional
 назначение, 43
 обработка значений, 43
 примитивные, числовые классы, 165

Требования к безопасности, 195**Ф**

Функции
 классификаторов, назначение, 51
 накопителя
 нисходящего, назначение, 51
 обычного, назначение, 47; 135
 объединителя, назначение, 47
 порядок возврата, 67
 поставщика, назначение, 47
 потребителя, назначение, 135
 преобразователя, назначение, 135
 составляемые, назначение, 76
 структурные и обобщенные типы, 64
 Функциональные интерфейсы
 аннотирование, 23
 для потоков ввода-вывода, 57
 для примитивных типов, 66
 наиболее распространенные,
 применение, 64
 обобщенные, 23
 определение, 22
 преобразование лямбда-выражений, 23

Ч**Часовые пояса, обозначение**, 120

Java SE 8 Вводный курс



Версия Java SE 8, с нетерпением ожидаемая миллионами программистов, включает в себя самое важное обновление за многие прошедшие годы. Появление в этой версии лямбда-выражений и новых потоков ввода-вывода знаменует собой главное изменение в программировании на Java с момента внедрения обобщений и аннотаций.

В своей книге Кей С. Хорстманн, маститый автор и знаток Java, предлагает вниманию читателей наиболее ценные из новых языковых средств в версии Java 8, а также рассматривает те средства, которые были внедрены в версии Java 7, но не удостоились должного внимания программистов. Те, у кого имеется опыт программирования на Java, найдут в этой книге практические рекомендации и примеры кода, демонстрирующие нововведения в версии Java 8, чтобы как можно быстрее воспользоваться этими и другими усовершенствованиями языка и платформы Java. В этом незаменимом руководстве рассматриваются следующие важные темы.

- Применение лямбда-выражений для написания вычисляемых фрагментов кода, которые могут быть переданы служебным функциям.
- Новые потоки ввода-вывода, организованные в отдельный прикладной программный интерфейс API, который позволяет значительно повысить эффективность коллекций и удобство обращения с ними.
- Существенное обновление средств параллельного программирования, где применяются лямбда-выражения для выполнения операций фильтрации, отображения, сведения и достигается значительное повышение производительности при обращении с общими счетчиками и хеш-таблицами.
- Полезные рекомендации по практическому применению лямбда-выражений в прикладных программах.
- Описание долгожданной качественно разработанной библиотеки для даты, времени и календаря (JSR 310).
- Прикладной программный интерфейс JavaFX, предназначенный на замену библиотеки Swing, для построения графических пользовательских интерфейсов, а также интерпретатор Nashorn языка JavaScript.
- Многочисленные мелкие изменения в библиотеке, позволяющие сделать программирование на Java более продуктивным и приятным занятием.

Это первая книга, в которой освещаются упомянутые выше нововведения в версии Java 8, поэтому она служит ценным источником информации для тех, кто стремится писать в недалекой перспективе самый надежный, эффективный и безопасный код на Java.

Кей С. Хорстманн – автор книги *Scala for the Impatient* (издательство Addison-Wesley, 2012 г.), а также основной автор двухтомного издания *Core Java™, Volumes I and II, Ninth Edition* (издательство Prentice Hall, 2013 г.; в русском переводе это издание вышло в двух томах под общим названием *Java. Библиотека профессионала, 9-е изд в ИД "Вильямс", 2014 г.*). Он также написал десяток других книг для профессиональных программистов и изучающих вычислительную технику. Кей служит профессором на кафедре вычислительной техники при университете штата Калифорния в Сан-Хосе, а также является обладателем почетного звания “Чемпион по Java”.

Категория: программирование

ISBN 978-5-8459-1900-7

Предмет рассмотрения: язык Java, версия SE 8

14025

Уровень: промежуточный/продвинутый



www.williamspublishing.com

horstmann.com/java8

9 785845 919007

 Addison-Wesley

ALWAYS LEARNING

PEARSON