



Джошуа Блох

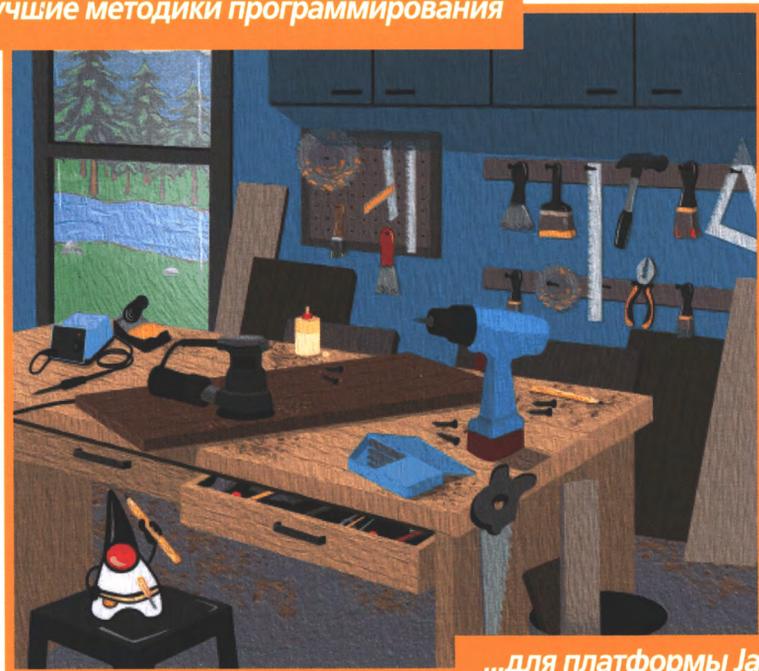
Рассмотрен
современный
Java!

Java

эффективное программирование

Третье издание

Лучшие методики программирования



...для платформы Java



Java

ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ

Третье издание

EFFECTIVE JAVA

Third Edition

Joshua Bloch

◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Java

ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ

Третье издание

Джошуа Блох



Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75

Б70

УДК 681.3.07

ООО “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

Рецензент канд. физ.-мат. наук *Д.Е. Намиот*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info@dialektika.com, http://www.dialektika.com

Блох, Джошуа

Б70 Java: эффективное программирование, 3-е изд. : Пер. с англ. — СПб. : ООО “Диалектика”, 2019. — 464 с. : ил. — Парал. тит. англ.

ISBN 978-5-6041394-4-8 (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.

Copyright © 2019 by Dialektika Computer Publishing Ltd.

Authorized translation from the English language edition of *Effective Java, 3rd Edition* (ISBN 978-0-13-468599-1) published by Addison-Wesley Publishing Company, Inc., Copyright © 2018 Pearson Education Inc.

Portions copyright © 2001–2008 Oracle and/or its affiliates.

This translation is published and sold by permission of Pearson Education, Inc., which owns or controls all rights to sell the same.

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 or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Джошуа Блох

Java: эффективное программирование, 3-е издание

Подписано в печать 31.10.2018.

Формат 70x100/16. Гарнитура Times.

Усл. печ. л. 29,0. Уч.-изд. л. 23,4.

Тираж 500 экз. Заказ № 11008.

Отпечатано в АО “Первая Образцовая типография”

Филиал “Чеховский Печатный Двор”

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО “Диалектика”, 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-6041394-4-8 (рус.)

ISBN 978-0-13-468599-1 (англ.)

© ООО “Диалектика”, 2019

© Pearson Education Inc., 2018

Оглавление

Вступительное слово	13
Предисловие	15
Благодарности	19
Глава 1. Введение	25
Глава 2. Создание и уничтожение объектов	29
Глава 3. Методы, общие для всех объектов	67
Глава 4. Классы и интерфейсы	109
Глава 5. Обобщенное программирование	159
Глава 6. Перечисления и аннотации	203
Глава 7. Лямбда-выражения и потоки	245
Глава 8. Методы	283
Глава 9. Общие вопросы программирования	321
Глава 10. Исключения	359
Глава 11. Параллельные вычисления	381
Глава 12. Сериализация	413
Приложение. Соответствие статей второго издания разделам третьего издания	445
Список литературы	449
Предметный указатель	453

Содержание

Вступительное слово	13
Предисловие	15
Предисловие к третьему изданию	15
Предисловие ко второму изданию	16
Предисловие к первому изданию	17
Благодарности	19
Благодарности к третьему изданию	19
Благодарности ко второму изданию	20
Благодарности к первому изданию	21
Ждем ваших отзывов!	22
Глава 1. Введение	25
Глава 2. Создание и уничтожение объектов	29
2.1. Рассмотрите применение статических фабричных методов вместо конструкторов	29
2.2. При большом количестве параметров конструктора подумайте о проектном шаблоне <i>Строитель</i>	34
2.3. Получайте синглтон с помощью закрытого конструктора или типа перечисления	43
2.4. Обеспечивайте неинстанцируемость с помощью закрытого конструктора	46
2.5. Предпочитайте внедрение зависимостей жестко прошитым ресурсам	47
2.6. Избегайте создания излишних объектов	50
2.7. Избегайте устаревших ссылок на объекты	53
2.8. Избегайте финализаторов и очистителей	57
2.9. Предпочитайте try-c-ресурсами использованию try-finally	63
Глава 3. Методы, общие для всех объектов	67
3.1. Перекрывая equals, соблюдайте общий контракт	67
3.2. Всегда при перекрытии equals перекрывайте hashCode	81

3.3. Всегда перекрывайте <code>toString</code>	87
3.4. Перекрывайте метод <code>clone</code> осторожно	90
3.5. Подумайте о реализации <code>Comparable</code>	100
Глава 4. Классы и интерфейсы	109
4.1. Минимизируйте доступность классов и членов	109
4.2. Используйте в открытых классах методы доступа, а не открытые поля	114
4.3. Минимизируйте изменяемость	117
4.4. Предпочитайте композицию наследованию	125
4.5. Проектируйте и документируйте наследование либо запрещайте его	131
4.6. Предпочитайте интерфейсы абстрактным классам	138
4.7. Проектируйте интерфейсы для потомков	144
4.8. Используйте интерфейсы только для определения типов	147
4.9. Предпочитайте иерархии классов дескрипторам классов	149
4.10. Предпочитайте статические классы-члены нестатическим	152
4.11. Ограничивайтесь одним классом верхнего уровня на исходный файл	156
Глава 5. Обобщенное программирование	159
5.1. Не используйте несформированные типы	159
5.2. Устраняйте предупреждения о непроверяемом коде	165
5.3. Предпочитайте списки массивам	168
5.4. Предпочитайте обобщенные типы	173
5.5. Предпочитайте обобщенные методы	178
5.6. Используйте ограниченные символы подстановки для повышения гибкости API	183
5.7. Аккуратно сочетайте обобщенные типы и переменное количество аргументов	190
5.8. Применяйте безопасные с точки зрения типов гетерогенные контейнеры	196
Глава 6. Перечисления и аннотации	203
6.1. Используйте перечисления вместо констант <code>int</code>	203
6.2. Используйте поля экземпляров вместо порядковых значений	216
6.3. Используйте <code>EnumSet</code> вместо битовых полей	217

6.4. Используйте EnumMap вместо индексирования порядковыми номерами	219
6.5. Имитируйте расширяемые перечисления с помощью интерфейсов	225
6.6. Предпочитайте аннотации схемам именования	229
6.7. Последовательно используйте аннотацию Override	239
6.8. Используйте интерфейсы-маркеры для определения типов	242
Глава 7. Лямбда-выражения и потоки	245
7.1. Предпочитайте лямбда-выражения анонимным классам	245
7.2. Предпочитайте ссылки на методы лямбда-выражениям	250
7.3. Предпочитайте использовать стандартные функциональные интерфейсы	252
7.4. Разумно используйте потоки	257
7.5. Предпочитайте в потоках функции без побочных эффектов	265
7.6. Предпочитайте коллекции потокам в качестве возвращаемых типов	271
7.7. Будьте внимательны при параллелизации потоков	277
Глава 8. Методы	283
8.1. Проверяйте корректность параметров	283
8.2. При необходимости создавайте защитные копии	287
8.3. Тщательно проектируйте сигнатуры методов	292
8.4. Перегружайте методы разумно	294
8.5. Используйте методы с переменным количеством аргументов с осторожностью	302
8.6. Возвращайте пустые массивы и коллекции, а не null	304
8.7. Возвращайте Optional с осторожностью	307
8.8. Пишите документирующие комментарии для всех открытых элементов API	312
Глава 9. Общие вопросы программирования	321
9.1. Минимизируйте область видимости локальных переменных	321
9.2. Предпочитайте циклы for для коллекции традиционным циклам for	324
9.3. Изучите и используйте возможности библиотек	328
9.4. Если вам нужны точные ответы, избегайте float и double	331

9.5. Предпочитайте примитивные типы упакованным примитивным типам	334
9.6. Избегайте применения строк там, где уместнее другой тип	338
9.7. Помните о проблемах производительности при конкатенации строк	341
9.8. Для ссылки на объекты используйте их интерфейсы	342
9.9. Предпочитайте интерфейсы рефлексии	344
9.10. Пользуйтесь машинно-зависимыми методами осторожно	348
9.11. Оптимизируйте осторожно	350
9.12. Придерживайтесь общепринятых соглашений по именованию	353
Глава 10. Исключения	359
10.1. Используйте исключения только в исключительных ситуациях	359
10.2. Используйте для восстановления проверяемые исключения, а для программных ошибок — исключения времени выполнения	362
10.3. Избегайте ненужных проверяемых исключений	365
10.4. Предпочитайте использовать стандартные исключения	367
10.5. Генерируйте исключения, соответствующие абстракции	370
10.6. Документируйте все исключения, которые может генерировать метод	372
10.7. Включайте в сообщения информацию о сбое	374
10.8. Добивайтесь атомарности сбоев	376
10.9. Не игнорируйте исключения	378
Глава 11. Параллельные вычисления	381
11.1. Синхронизируйте доступ к совместно используемым изменяемым данным	381
11.2. Избегайте излишней синхронизации	386
11.3. Предпочитайте исполнителей, задания и потоки данных потокам исполнения	394
11.4. Предпочитайте утилиты параллельности методам <code>wait</code> и <code>notify</code>	396
11.5. Документируйте безопасность с точки зрения потоков	402
11.6. Аккуратно применяйте отложенную инициализацию	406
11.7. Избегайте зависимости от планировщика потоков	409

Глава 12. Сериализация	413
12.1. Предпочитайте альтернативы сериализации Java	413
12.2. Реализуйте интерфейс <code>Serializable</code> крайне осторожно	418
12.3. Подумайте о применении пользовательской сериализованной формы	421
12.4. Создавайте защищенные методы <code>readObject</code>	429
12.5. Для управления экземпляром предпочитайте типы перечислений методу <code>readResolve</code>	435
12.6. Подумайте о применении прокси-агента сериализации вместо сериализованных экземпляров	440
Приложение. Соответствие статей второго издания разделам третьего издания	445
Список литературы	449
Предметный указатель	453

Моей семье: Синди, Тиму и Мэтту

Вступительное слово

Если ваш коллега скажет вам “Моя супруга сегодня перед ночью изготовит дома необычную еду. Не объединишься с нами в поедании?”, то вам в голову, вероятно, придут три мысли: вас пригласили на ужин; ваш коллега явно иностранец; ну, а первым вашим ощущением будет озадаченность.

Если вы когда-либо изучали иностранный язык, а затем пробовали пользоваться им за пределами учебной аудитории, то вы понимаете, что есть три вещи, которые необходимо знать: каким образом структурирован изучаемый язык (его грамматику), какими словами обозначаются вещи, о которых вы хотите рассказать (словарь), а также общепринятые и эффективные способы говорить о повседневных вещах (лексические обороты). На занятиях слишком часто уделяется внимание только первым двум темам, и позже вы обнаруживаете, что настоящие носители изучаемого вами языка прячут улыбку, пытаясь понять ваши обороты.

В случае языка программирования все обстоит почти так же. Вам необходимо понимать основы языка: является он алгоритмическим, функциональным или объектно-ориентированным. Вам нужно знать словарь языка: какие структуры данных, операции и возможности предоставляют язык и его стандартные библиотеки. Вам необходимо также ознакомиться с общепринятыми и эффективными способами структурирования вашего кода. В книгах, посвященных языкам программирования, часто освещаются лишь первые два вопроса, а эффективные приемы работы с языком если и обсуждаются, то лишь крайне бегло. Возможно, дело в том, что писать на первые две темы гораздо проще. Грамматика и словарь — это свойства самого языка, тогда как способ его использования характеризует скорее людей, которые этим языком пользуются.

Например, язык программирования Java — объектно-ориентированный язык с единичным наследованием, поддерживающий императивный (ориентированный на инструкции) стиль программирования каждого метода. Его библиотеки ориентированы на поддержку графического вывода, работы с сетью, распределенных вычислений и безопасности. Однако как использовать этот язык на практике наилучшим образом?

Имеется и другой аспект. Программы, в отличие от произнесенных предложений, а также большинства книг и журналов, со временем меняются. Обычно недостаточно создать код, который эффективно работает и легко может быть понят другими людьми. Нужно еще и организовать этот код таким образом, чтобы его можно было легко модифицировать. Практически для любой задачи T имеется с десятков вариантов написания решающего ее кода. Из этого десятка семь окажутся запутанными, неэффективными или непонятными для читающего их человека. Но какой из оставшихся трех вариантов будет более всего похож на исходный текст, который потребуется в следующем году при разработке новой версии программы, решающей задачу T' ?

Есть масса книг, по которым можно изучать грамматику языка программирования Java, включая *The Java™ Programming Language* Арнольда (Arnold), Гослинга (Gosling) и Холмса (Holmes) или *The Java™ Language Specification* Гослинга, Джоя (Joy), Стила (Steele), Брача (Bracha) и Бакли (Buckley) [25]. Имеется также множество книг, посвященных библиотекам и API, связанным с языком программирования Java.

Эта книга посвящена третьей теме: эффективному использованию языка Java. Джошуа Блох провел несколько лет в Sun Microsystems, занимаясь расширением, реализацией и применением языка программирования Java; он также прочел огромное количество исходных текстов, написанных многими программистами, в том числе мной. Приведя свои знания и опыт в систему, он дает отличные советы о том, каким образом структурировать код, чтобы он эффективно работал, был понятен другим программистам, чтобы последующие его изменения и усовершенствования доставляли как можно меньше хлопот и даже по возможности чтобы ваши программы были к тому же элегантными и красивыми.

*Гай Л. Стил-младший (Guy L. Steele Jr.)
Берлингтон, Массачусетс
Апрель 2001*

Предисловие

Предисловие к третьему изданию

В 1997 году, когда язык Java был еще новинкой, отец Java, Джеймс Гослинг (James Gosling), описал его как “весьма простой язык для синих воротничков” [14]. Примерно в то же время отец C++, Бьярне Страуструп (Bjarne Stroustrup), описал C++ как “мультипарадигменный язык”, который “сознательно сделан отличным от языков, созданных для поддержки единого способа написания программ” [47]. Касаясь языка программирования Java, Страуструп предупредил:

Большая часть относительной простоты языка программирования Java — как в большинстве новых языков — отчасти иллюзия, а отчасти результат его незавершенности. Со временем размеры и сложность Java значительно вырастут. Его размер удвоится или утроится, как увеличится и количество зависящих от реализации расширений или библиотек [46].

Сейчас, двадцать лет спустя, можно сказать, что и Гослинг, и Страуструп были правы. Java теперь большой и сложный язык программирования с множеством абстракций для множества вещей — от параллельного выполнения до итераций и представления даты и времени.

Мне все еще нравится Java, хотя мой пыл несколько остыл с его ростом. Учитывая увеличение размера и сложности языка, все более важной является необходимость руководства по этому языку программирования, охватывающего наилучшие современные практики его применения. В третьем издании данной книги я сделал все, чтобы предоставить вам именно такое руководство. Я надеюсь, что это издание, оставаясь верным духу первых двух изданий, удовлетворит всем вашим потребностям.

*Сан-Хосе, Калифорния
Ноябрь 2017*

P.S. Было бы упущением не упомянуть о передовой практике, которая ныне отнимает изрядное количество моего времени. С момента рождения нашей области в 1950-е годы мы, программисты, свободно обменивались и перепроектировали API друг друга. Эта практика имела и имеет решающее значение

для быстрых успехов компьютерных технологий. Я активно работаю для того, чтобы сохранить эту свободу и далее [9], и призываю вас присоединиться ко мне. Если мы сохраним право повторной реализации API друг друга, это будет иметь решающее значение для дальнейшего здоровья всей нашей отрасли.

Предисловие ко второму изданию

С тех пор как я написал первое издание этой книги в 2001 году, в платформе Java произошло много изменений, и я решил, что настала пора написать второе издание. Наиболее значимыми изменениями стали добавление обобщенных типов, типов перечислений, комментариев, автоматической инкапсуляции, циклов по диапазону в Java 5. Новшеством стало добавление новой библиотеки `java.util.concurrent`, также появившейся в Java 5. Мне повезло, что вместе с Гиладом Брачей я смог возглавить команду, разрабатывавшую новые возможности языка. Мне также повезло работать в команде, разрабатывавшей библиотеку параллельных вычислений и возглавляемой Дугом Ли (Doug Lea).

Другим значительным изменением платформы стало ее оснащение современными интегрированными средами разработки, такими как Eclipse, IntelliJ IDEA и NetBeans, а также инструментами статического анализа, как, например, FindBugs. Хотя я и не принимал в этом участия, но смог извлечь для себя огромную выгоду и узнал, как все эти инструменты влияют на опыт разработки на языке программирования Java.

В 2004 году я перешел из Sun в Google, однако продолжал принимать участие в разработке платформы Java в течение последних четырех лет, работая в области API для параллельных вычислений и коллекций, используя офисы Google и Java Community Process. Я также имел удовольствие использовать платформу Java для разработки библиотек для внутреннего использования в Google. Теперь я знаю, что чувствуют пользователи.

Когда в 2001 году я писал первое издание книги, моей основной целью было поделиться моим опытом с читателями, чтобы они могли повторить мои успехи и избежать моих неудач. Новый материал продолжает традицию использования реальных примеров из библиотек платформы Java.

Успех первой редакции превзошел все мои ожидания, и, освещая новый материал, я сделал все возможное, чтобы сохранить дух предыдущего издания. Избежать увеличения книги было невозможно — и теперь в ней вместо 57 уже 78 разделов. И я не просто добавил 23 новых раздела, а тщательно переработал исходное издание, в том числе удалив некоторые ставшие неактуальными разделы. В приложении вы можете увидеть, как соотносятся материалы данного издания с материалом первого издания.

В предисловии к первому изданию я писал, что язык программирования Java и его библиотеки очень способствуют качеству и производительности труда и что работать с ними — одно удовольствие. Изменения в версиях 5 и 6 сделали их еще лучше. Сейчас платформа Java гораздо больше и сложнее, чем в 2001 году, но, познакомившись с ее проектными шаблонами и идиомами, использующими новые возможности, вы улучшите свои программы и упростите свою жизнь. Надеюсь, что это издание передаст вам мой энтузиазм и поможет более эффективно и с большим удовольствием использовать платформу и ее новые возможности.

*Сан-Хосе, Калифорния
Апрель 2008*

Предисловие к первому изданию

В 1996 году я направился на запад работать в компании JavaSoft (как она тогда называлась), поскольку для меня было очевидно, что именно там происходят главные события. На протяжении пяти лет я работал архитектором библиотек для платформы Java. Я проектировал, реализовывал и сопровождал множество библиотек, а также давал консультации по многим другим библиотекам. Руководить этими библиотеками в ходе становления платформы языка Java — такая возможность выпадает только раз в жизни. Не будет преувеличением сказать, что я имел честь работать с некоторыми великими программистами нашего поколения. В процессе работы я узнал о языке программирования Java очень многое: что хорошо работает, а что — нет и как пользоваться языком и его библиотеками для получения наиболее эффективного результата.

Эта книга является попыткой поделиться с вами моим опытом, чтобы вы могли повторить мои успехи и избежать моих ошибок. Формат книги я позаимствовал из книги Скотта Мейерса (Scott Meyers) *Effective C++* [33], состоящей из разделов, каждый из которых посвящен одному конкретному правилу, позволяющему улучшить программы и проекты. Такой формат кажется мне необычайно эффективным, и, надеюсь, вам он также понравится.

Во многих случаях я иллюстрирую разделы реальными примерами из библиотек платформы Java. Говоря, что нечто можно сделать лучше, я стараюсь привести исходный текст, который я писал сам, однако иногда я брал разработки моих коллег. Приношу мои искренние извинения, если, несмотря на все старания, я кого-либо при этом обидел. Отрицательные примеры приведены не для того, чтобы кого-то опорочить, а в соответствии с духом сотрудничества, чтобы все мы могли извлечь пользу из опыта тех, кто прошел этот путь ранее.

Хотя эта книга предназначена не только для людей, занимающихся разработкой повторно используемых компонентов, она неизбежно отражает мой опыт в написании таковых, накопленный за последние два десятилетия. Я привык думать в терминах интерфейсов прикладного программирования (API) и предлагаю вам поступать так же. Даже если вы не занимаетесь разработкой повторно используемых компонентов, мышление в соответствующих терминах может повысить качество разрабатываемых вами программ. Более того, нередко случается писать многократно используемые компоненты, даже не подозревая об этом: вы написали что-то полезное, поделились своим результатом с приятелем, и вскоре оказывается, что у вашего кода уже с полдюжины пользователей. С этого момента вы лишаетесь возможности свободно менять этот API и благодарите сами себя за усилия, которые потратили на его первоначальную разработку и которые позволили получить качественный продукт.

Мое особое внимание к разработке API может показаться несколько противостественным для ярых приверженцев новых облегченных методик разработки программного обеспечения, таких как “*Экстремальное программирование*” [2]. В этих методиках особое значение придается написанию самой простой программы, которая только сможет работать. Если вы пользуетесь одной из таких методик, то вскоре обнаружите, что особое внимание к разработке API вернется сторицей при последующем *рефакторинге* программы. Основной задачей рефакторинга является усовершенствование системной структуры, а также исключение дублирования кода. Этой цели невозможно достичь, если у системных компонентов нет хорошо продуманного и спроектированного API.

Ни один язык не идеален, но некоторые из них великолепны. Я обнаружил, что язык программирования Java и его библиотеки в огромной степени способствуют повышению качества и производительности труда и что работать с ними — одно удовольствие. Надеюсь, эта книга сможет передать вам мой энтузиазм и сделает вашу работу с языком Java более эффективной и приятной.

Купертино, Калифорния
Апрель 2001

Благодарности

Благодарности к третьему изданию

Я благодарю читателей первых двух изданий за их восторженный прием этой книги и принятие ее идей в свои сердца, а также за отзывы о том, какое положительное влияние она оказала на них и их работу. Я благодарю тех преподавателей, которые использовали эту книгу в своих учебных курсах, а также те команды программистов, которые приняли ее как руководство в повседневной работе.

Я благодарю всю команду Addison-Wesley и Pearson за их отзывчивость, профессионализм и терпение. В любых условиях мой редактор Грег Денч (Greg Doench) оставался идеальным редактором и истинным джентльменом. Боюсь, что у него прибавилось седин в результате работы над этой книгой, и смиренно прошу у него за это прощения. Менеджер проекта Джули Наиль (Julie Nahil) и редактор проекта Дана Вилсон (Dana Wilson) превзошли все мои ожидания и надежды: всегда прилежны, оперативны, организованны и дружелюбны — как и редактор Ким Вимпсетт (Kim Wimpsett).

Я вновь получил лучшую команду рецензентов, какую только можно вообразить, и я искренне благодарен каждому из них. Вот состав основной группы, которая подробно проанализировала каждую главу: Синди Блох (Cindy Bloch), Брайан Керниган (Brian Kernighan), Кевин Бурриллион (Kevin Bourrillion), Джо Боубир (Joe Bowbeer), Уильям Чарджин (William Chargin), Джо Дарси (Joe Darcy), Брайан Гётц (Brian Goetz), Тим Хеллоран (Tim Halloran), Стюарт Маркс (Stuart Marks), Тим Пейерлс (Tim Peierls) и Йошики Шибата (Yoshiki Shibata). Кроме того, свой вклад внесли такие рецензенты, как Маркус Биль (Marcus Biel), Дэн Блох (Dan Bloch), Бет Боттос (Beth Bottos), Мартин Буххольц (Martin Buchholz), Майкл Дайамонд (Michael Diamond), Чарли Гэррод (Charlie Garrod), Том Хоутин (Tom Hawtin), Дуг Ли (Doug Lea), Алексей Шипилёв (Aleksey Shipilëv), Лу Вассерман (Lou Wasserman) и Питер Вайнбергер (Peter Weinberger). Их отзывы содержали многочисленные предложения по исправлению материала книги, что привело к существенному улучшению книги и спасло меня от множества затруднений.

Особую благодарность заслужили Уильям Чарджин, Дуг Ли и Тим Пейерлс, которые помогли в “обкатке” многих идей в этой книге. Уильям, Дуг и Тим неизменно щедро делились своими временем и знаниями.

Наконец, я благодарю мою жену, Синди Блох, которая не только с пониманием относилась к моей занятости, но и помогала во всем — от чтения черновиков до подготовки предметного указателя; словом, во всех тех неприятных мелочах, которые неизбежно сопутствуют любому большому проекту.

Благодарности ко второму изданию

Я благодарю читателей первого издания за их восторженный прием этой книги и принятие ее идей в свои сердца, а также за отзывы о том, какое положительное влияние она оказала на них и их работу. Я благодарю тех преподавателей, которые использовали эту книгу в своих учебных курсах, а также те команды программистов, которые приняли ее как руководство в повседневной работе.

Я благодарю всю команду Addison-Wesley за их отзывчивость, профессионализм и терпение. В любых условиях мой редактор Грег Денч (Greg Doench) оставался идеальным редактором и истинным джентльменом. Менеджер проекта Джули Наиль (Julie Nahil) всегда была такой, какой должен быть менеджер проекта: прилежной, оперативной, организованной и дружески настроенной — как и редактор Барбара Вуд (Barbara Wood).

Я вновь получил лучшую команду рецензентов, какую только можно вообразить, и я искренне благодарен каждому из них. Вот состав основной группы, которая подробно проанализировала каждую главу: Лекси Бугер (Lexi Vaugher), Синди Блох (Cindy Bloch), Бет Боттос (Beth Bottos), Джо Боубир (Joe Bowbeer), Брайан Гётц (Brian Goetz), Тим Хеллоран (Tim Halloran), Брайан Керниган (Brian Kernighan), Роб Кёнигсберг (Rob Konigsberg), Тим Пейерлс (Tim Peierls), Билл Паф (Bill Pugh), Йошики Шибата (Yoshiki Shibata), Питер Стаут (Peter Stout), Питер Вайнбергер (Peter Weinberger) и Франк Еллин (Frank Yellin). Кроме того, свой вклад внесли такие рецензенты, как Пабло Беллвер (Pablo Bellver), Дэн Блох (Dan Bloch), Дэн Борнштейн (Dan Bornstein), Кевин Бурриллион (Kevin Bourrillion), Мартин Бухгольц (Martin Buchholz), Джо Дарси (Joe Darcy), Нил Гафтер (Neal Gafter), Лоренс Гонсальвес (Laurence Gonsalves), Аарон Гринхаус (Aaron Greenhouse), Барри Хайес (Barry Hayes), Питер Джонс (Peter Jones), Анжелика Ланжер (Angelika Langer), Дуг Ли (Doug Lea), Боб Ли (Bob Lee), Джереми Мэнсон (Jeremy Manson), Том Мэй (Tom May), Майк Мак-Клоски (Mike McCloskey), Андрей Терещенко (Andriy Tereshchenko) и Пол Тима (Paul Tuma). Их отзывы содержали многочисленные

предложения по исправлению материала книги, что привело к существенному улучшению книги и спасло меня от множества затруднений. Все оставшиеся в книге огрехи — на моей совести.

Особую благодарность заслужили Дуг Ли и Тим Пейерлс, которые помогали в “обкатке” многих идей в этой книге. Дуг и Тим неизменно щедро делились своими временем и знаниями.

Я благодарю моего менеджера в Google, Прабху Кришну (Prabha Krishna), за ее неизменную поддержку и содействие.

Наконец, я благодарю мою жену, Синди Блох, которая не только с пониманием относилась к моей занятости, но и помогала во всем — от чтения черновиков до работы с Framemaker и подготовки предметного указателя.

Благодарности к первому изданию

Я благодарю Патрика Чана (Patrick Chan) за предложение написать эту книгу и за то, что он увлек этой идеей управляющего редактора серии Лизу Френдли (Lisa Friendly); Тима Линдхольма (Tim Lindholm), технического редактора серии; а также Майка Хендриксона (Mike Hendrickson), исполнительного редактора Addison-Wesley. Я благодарю вас, Лиза, Тим и Майк, за то, что вы помогли мне довести этот проект до конца, за ваши сверхчеловеческое терпение и непоколебимую веру в то, что однажды я напишу эту книгу.

Я благодарю Джеймса Гослинга (James Gosling) и его команду за создание того, о чем я смог написать, а также многих разработчиков платформы Java, шедших по стопам Джеймса. В частности, я благодарю моих коллег по Sun Java Platform Tools and Libraries Group за идеи и поддержку. Эта команда включает Эндрю Беннетта (Andrew Bennett), Джо Дарси (Joe Darcy), Нила Гафтера (Neal Gafter), Ирис Гарсию (Iris Garcia), Константина Кладько (Konstantin Kladko), Яна Литтла (Ian Little), Майка Мак-Клоски (Mike McCloskey) и Марка Рейнхольда (Mark Reinhold). Должен упомянуть также бывших членов команды — Дзенхуа Ли (Zhenghua Li), Билла Мэддокса (Bill Maddox) и Навиина Сандживу (Naveen Sanjeeva).

Я благодарю моего менеджера Эндрю Беннета (Andrew Bennett) и директора Ларри Абрамса (Larry Abrahams) за их полную и активную поддержку этого проекта. Я благодарю Рича Грина (Rich Green), вице-президента Java Software, за создание условий, в которых инженеры могут мыслить творчески и публиковать свои работы.

Судьба подарила мне лучшую команду рецензентов, о какой только можно мечтать, и я искренне благодарен всем ее членам. Вот состав этой команды: Эндрю Беннетт (Andrew Bennett), Синди Блох (Cindy Bloch), Дэн Блох (Dan

Bloch), Бет Боттос (Beth Bottos), Джо Боубир (Joe Bowbeer), Гилад Брача (Gilad Bracha), Мэри Кампьон (Mary Campione), Джо Дарси (Joe Darcy), Дэвид Экхардт (David Eckhardt), Джо Фиалли (Joe Fialli), Лиза Френдли (Lisa Friendly), Джеймс Гослинг (James Gosling), Питер Хаггар (Peter Hagggar), Дэвид Холмс (David Holmes), Брайан Керниган (Brian Kernighan), Константин Кладько (Konstantin Kladko), Дуг Ли (Doug Lea), Дзенхуа Ли (Zhenghua Li), Тим Линдхольм (Tim Lindholm), Майк Мак-Клоски (Mike McCloskey), Тим Пейерлс (Tim Peierls), Марк Рейнхольд (Mark Reinhold), Кен Расселл (Ken Russell), Билл Шеннон (Bill Shannon), Питер Стаут (Peter Stout), Фил Уадлер (Phil Wadler) и два рецензента, оставшихся анонимными. Они сделали многочисленные предложения, которые привели к существенным улучшениям книги и спасли меня от множества затруднений. Все оставшиеся в книге огрехи — на моей совести.

Многочисленные коллеги как из Sun, так и из других компаний, принимали участие в технических обсуждениях, улучшивших качество книги. Среди прочих полезные идеи внесли Бен Гомес (Ben Gomes), Стеффен Граруп (Steffen Grarup), Питер Кесслер (Peter Kessler), Ричард Рода (Richard Roda), Джон Роуз (John Rose) и Дэвид Стутамир (David Stoutamire). Особую благодарность заслужил Дуг Ли (Doug Lea), который помогал в “обкатке” многих идей в этой книге. Дуг неизменно щедро делился своими временем и знаниями.

Я благодарю Джули Диниколу (Julie Dinicola), Джеки Дусетт (Jacqui Doucette), Майка Хендриксона (Mike Hendrickson), Хизер Ольшик (Heather Olszyk), Трейси Расс (Tracy Russ) и всю команду Addison-Wesley за их поддержку и профессионализм. Даже при невероятно плотном графике они всегда оставались дружелюбны и гостеприимны.

Я благодарю Гая Стила (Guy Steele) за вступительное слово к книге. Я польщен, что он решил принять участие в этом проекте.

Наконец, я благодарю мою жену, Синди Блох, которая не только с пониманием относилась к моей занятости, но и помогала во всем — от чтения черновиков до работы с Framemaker и подготовки предметного указателя.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. 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 интересны любые ваши замечания 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 ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

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

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: <http://www.dialektika.com>

Введение

Эта книга разработана с тем, чтобы помочь вам максимально эффективно использовать возможности языка программирования Java и его основных библиотек `java.lang`, `java.util` и `java.io`, а также подпакетов наподобие `java.util.concurrent` и `java.util.function`. Прочие библиотеки рассматриваются эпизодически.

Эта книга состоит из 90 разделов, каждый из которых посвящен одному правилу. В этих правилах собран опыт, который лучшие, наиболее опытные программисты считают весьма полезным. Эти разделы сгруппированы в одиннадцать глав, каждая из которых охватывает один из аспектов проектирования программного обеспечения. Книга не предназначена для чтения от корки до корки: каждый раздел более или менее самодостаточен. Разделы снабжены перекрестными ссылками, позволяющими пройти собственный путь через книгу.

Со времени публикации предыдущего издания книги к платформе Java было добавлено немало новых функциональных возможностей. Большинство разделов этой книги тем или иным способом используют эти возможности. В приведенной ниже таблице показано, где именно в первую очередь освещены те или иные ключевые особенности языка.

Функциональная возможность	Разделы	Версия Java
Лямбда-выражения	7.1–7.3	8
Потоки	7.4–7.7	8
Использование класса <code>Optional</code>	8.7	8
Методы по умолчанию в интерфейсах	4.7	8
try-c-ресурсами	2.9	7
<code>@SafeVarargs</code>	5.7	7
Модули	4.1	9

Большинство разделов проиллюстрированы примерами программ. Ключевой особенностью этой книги является то, что она содержит примеры кода,

иллюстрирующие многие проектные шаблоны и идиомы. Там, где это уместно, представлены перекрестные ссылки на стандартный справочник в этой области [12].

Многие разделы содержат один или несколько примеров программ, иллюстрирующих некоторые практики, которых следует избегать. Такие примеры, иногда известные как “*антишаблоны*” (антипаттерны), ясно указываются с помощью комментариев наподобие

```
// Никогда этого не делайте!
```

В каждом случае поясняется, почему этот пример плох, и предлагается альтернативный подход.

Эта книга не для начинающих: предполагается, что вы уже знакомы с Java. Если это не так, обратитесь к одной из множества книг для новичков, таких как *Java Precisely* Питера Сестофта (Peter Sestoft) [41]. Хотя данная книга должна быть доступна любому обладающему рабочим знанием языка, она должна дать пищу для размышлений даже самым “продвинутым” программистам.

Большинство правил этой книги вытекают из нескольких основополагающих принципов. Ясность и простота имеют первостепенное значение. Пользователь компонента никогда не должен удивляться его поведению. Компоненты должны быть как можно меньшими, но не меньше, чем нужно. (В этой книге термин *компонент* относится к любому повторно используемому элементу программы, от отдельных методов до сложных каркасов, состоящих из нескольких пакетов.) Код должен повторно использоваться, а не копироваться. Зависимости между компонентами должны быть сведены к минимуму. Ошибки должны обнаруживаться немедленно после того, как они сделаны, в идеале — во время компиляции.

Хотя правила из этой книги и не применяются постоянно во время работы, в подавляющем большинстве случаев они характеризуют лучшие практики программирования. Вы не должны следовать этим правилам рабски и без размышлений, но нарушать их следует лишь изредка и по уважительной причине. Обучение искусству программирования, как и большинству других дисциплин, состоит из, во-первых, обучения правилам и, во-вторых, обучения, когда эти правила нарушать.

По большей части эта книга не посвящена вопросам производительности. Речь идет о написании ясных, правильных, полезных, надежных, гибких и легких в обслуживании и поддержке программ. Если вы можете написать такую программу, то добиться требуемой производительности — обычно относительно простой вопрос (раздел 9.11). В некоторых разделах обсуждаются проблемы производительности и в некоторых из них приводятся цифры, эту производительность характеризующие. Эти цифры, которые обычно дополнены словами

“на моей машине”, следует рассматривать как в лучшем случае очень приближенные.

Для тех, кому это важно: у меня старенькая домашняя машина с четырехъядерным 3.5 ГГц процессором Intel Core i7-4770K с 16 Гбайтами DDR3-1866 CL9 RAM, работающая с Azul’s Zulu 9.0.0.15-версией OpenJDK, с операционной системой Microsoft Windows 7 Professional SP1 (64-bit).

При обсуждении возможностей языка программирования Java и его библиотек иногда необходимо сослаться на определенные версии. Для удобства в этой книге используются краткие названия вместо официальных. В приведенной далее таблице показано соответствие официальных названий используемым в книге кратким названиям.

Официальное название выпуска	Краткое название
JDK 1.0.x	Java 1.0
JDK 1.1.x	Java 1.1
Java 2 Platform, Standard Edition, v1.2	Java 2
Java 2 Platform, Standard Edition, v1.3	Java 3
Java 2 Platform, Standard Edition, v1.4	Java 4
Java 2 Platform, Standard Edition, v5.0	Java 5
Java Platform, Standard Edition 6	Java 6
Java Platform, Standard Edition 7	Java 7
Java Platform, Standard Edition 8	Java 8
Java Platform, Standard Edition 9	Java 9

Примеры в книге достаточно полные, но удобочитаемость предпочтительнее полноты. В них используются классы из пакетов `java.util` и `java.io`. Для компиляции примеров, возможно, придется добавить одно или несколько объявлений импорта или иной подобный шаблон. Веб-сайт книги по адресу <http://joshbloch.com/effectivejava> содержит расширенную версию каждого примера, который можно скомпилировать и запустить.

По большей части в этой книге используются технические термины, определенные в книге *The Java Language Specification, Java SE 8 Edition* [25]. Несколько терминов заслуживают отдельного упоминания. Язык поддерживает четыре разновидности типов: *интерфейсы* (включая *аннотации*), *классы* (включая *перечисления*), *массивы* и *примитивы*. Первые три называются *ссылочными типами*. Экземпляры класса и массивы являются *объектами*; примитивные значения таковыми не являются. *Члены* класса включают его *поля*, *методы*, *классы-члены* и *интерфейсы-члены*. *Сигнатура* метода состоит из его имени и типов его формальных параметров; сигнатура *не* содержит тип возвращаемого методом значения.

В этой книге используется несколько терминов, отличных от используемых в упомянутой книге. Здесь термин *наследование* (inheritance) используется как синоним для создания подклассов (subclassing). Вместо использования термина *наследования* для интерфейсов книга просто говорит, что класс *реализует* интерфейс или что один интерфейс *расширяет* другой. Чтобы описать уровень доступа, применяемый при отсутствии явного указания, в книге используется традиционный доступ *закрытый на уровне пакета* (package-private) вместо технически правильного *доступа пакета* (package access) [25, 6.6.1].

Также здесь используется несколько технических терминов, которые не определены в упомянутой книге. Термин *экспортируемый API*, или просто *API*, ссылается на классы, интерфейсы, конструкторы, члены и сериализованные формы, с помощью которых программист обращается к классу, интерфейсу или пакету. (Аббревиатуре “API”, означающей *интерфейс прикладного программирования*, отдается предпочтение перед термином *интерфейс*, чтобы избежать путаницы с конструкцией языка с этим названием.) Программист, который пишет программу, которая использует API, именуется *пользователем API*. Класс, реализация которого использует API, является *клиентом API*.

Классы, интерфейсы, конструкторы, члены и сериализованные формы вместе называются *элементами API*. Экспортированный API состоит из элементов API, которые доступны за пределами пакета, в котором определен API. Это те элементы API, которые может использовать любой клиент и которые автор API обязуется поддерживать. Не случайно они также являются элементами, для которых утилита Javadoc генерирует документацию в режиме работы по умолчанию. Грубо говоря, экспортируемый API пакета состоит из открытых и защищенных членов и конструкторов каждого открытого класса или интерфейса в пакете.

В Java 9 в платформу была добавлена *модульная система* (module system). Если библиотека использует модульную систему, ее экспортированный API представляет собой объединение экспортированных API всех пакетов, экспортируемых объявлением модуля библиотеки.

Создание и уничтожение объектов

В этой главе рассматриваются создание и уничтожение объектов: когда и как их создавать, когда и как избегать их создания, как обеспечить их своевременное уничтожение и как управлять любыми действиями по очистке, которые должны предшествовать уничтожению.

2.1. Рассмотрите применение статических фабричных методов вместо конструкторов

Традиционный способ, которым класс позволяет клиенту получить экземпляр, — предоставление открытого (`public`) конструктора. Существует еще один метод, который должен быть частью инструментария каждого программиста. Класс может предоставить открытый *статический фабричный метод*¹. Вот простой пример из `Boolean` (*упакованный примитивный класс* `boolean`). Этот метод преобразует значение примитива типа `boolean` в ссылку на объект `Boolean`:

```
public static Boolean valueOf(boolean b)
{
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Обратите внимание, что статический фабричный метод — это не то же самое, что проектный шаблон *Фабричный Метод* (Factory Method) из [12]. Статический фабричный метод, описанный в этом разделе, не имеет прямого эквивалента в [12].

Класс может предоставить своим клиентам статические фабричные методы вместо открытых (`public`) конструкторов (или в дополнение к ним). Такое

¹ Статический фабричный метод — это статический метод, который возвращает экземпляр класса. — *Примеч. ред.*

предоставление статического фабричного метода вместо открытого конструктора имеет как преимущества, так и недостатки.

Одним из преимуществ статических фабричных методов является то, что, в отличие от конструкторов, они имеют имена. Если параметры конструктора не описывают возвращаемые объекты (и сами по себе не являются ими), то хорошо подобранное имя статического фабричного метода легче в использовании, а получающийся в результате код оказывается более удобочитаемым. Например, вместо конструктора `BigInteger(int, int, Random)`, который возвращает объект `BigInteger`, который, вероятно, представляет собой простое число, было бы лучше использовать статический фабричный метод с именем `BigInteger.probablePrime` (этот метод был добавлен в Java 4).

Класс может иметь только один конструктор с заданной сигнатурой. Программисты, как известно, обходят это ограничение путем предоставления конструкторов, списки параметров которых отличаются только порядком типов их параметров. Это плохая идея. Пользователь такого API не сможет вспомнить, какой конструктор ему нужен, и в конечном итоге будет ошибаться и вызывать неверный конструктор. Программисты, читающие код с такими конструкторами, без документации не будут знать, что делает этот код.

Благодаря наличию имен на статические фабричные методы не накладывается ограничение из предыдущего абзаца. В тех случаях, когда классу, как представляется, требуется несколько конструкторов с одинаковой сигнатурой, замените конструкторы статическими фабричными методами с тщательно подобранными именами, чтобы подчеркнуть их различия.

Вторым преимуществом статических фабричных методов является то, что, в отличие от конструкторов, они не обязаны создавать новые объекты при каждом вызове. Это позволяет неизменяемым классам (раздел 4.3) использовать предварительно сконструированные экземпляры или кешировать экземпляры при их создании, чтобы избежать создания ненужных дубликатов объектов. Метод `Boolean.valueOf(boolean)` иллюстрирует этот метод: он *никогда* не создает объект. Этот метод аналогичен проектному шаблону *Птица-полетунец* (*Flyweight*) [12]. Он может значительно улучшить производительность, если часто запрашиваются эквивалентные объекты, особенно если их создание является дорогостоящим.

Возможность статических фабричных методов возвращать один и тот же объект при повторных вызовах позволяет классам строго контролировать, какие экземпляры существуют в любой момент времени. Классы, которые работают таким образом, называются классами с *управлением экземплярами* (*instance-controlled*). Существует несколько причин для написания таких классов. Управление экземплярами позволяет классу гарантировать, что он является синглтоном (раздел 2.3) или неинстанцируемым (раздел 2.4). Кроме того, это позволяет неизменяемому классу значения (раздел 4.3) гарантировать, что

не существует двух одинаковых экземпляров: `a.equals(b)` истинно тогда и только тогда, когда `a==b`. Это основа проектного шаблона *Приспособленец* [12]. Такую гарантию предоставляют типы перечислений (раздел 6.1).

Третье преимущество статических фабричных методов заключается в том, что, в отличие от конструкторов, они могут возвращать объект любого подтипа их возвращаемого типа. Это дает вам большую гибкость в выборе класса возвращаемого объекта.

Одним из применений этой гибкости является то, что API может возвращать объекты, не делая их классы открытыми. Соккрытие классов реализации таким способом приводит к очень компактному API. Эта техника ведет к *каркасам на основе интерфейсов* (раздел 4.6), в которых интерфейсы предоставляют естественные возвращаемые типы для статических фабричных методов.

До Java 8 интерфейсы не могли иметь статических методов. По соглашению статические фабричные методы для интерфейса с именем `Type` размещались в сопутствующем неинстанцируемом классе (раздел 2.4) с именем `Types`. Например, Java Collections Framework содержит 45 реализаций интерфейсов, предоставляя немодифицируемые коллекции, синхронизированные коллекции и т.п. Почти все эти реализации экспортируются с помощью статических фабричных методов в одном неинстанцируемом классе (`java.util.Collections`). Все классы возвращаемых объектов являются закрытыми.

Collections Framework API гораздо меньше, чем потребовалось бы в случае экспорта 45 отдельных открытых классов, по одному для каждой реализации. Это не только уменьшенный *размер* API, но и меньший *концептуальный вес*: количество и сложность концепций, которые программисты должны освоить для того, чтобы использовать API. Программист знает, что возвращаемый объект имеет API, в точности предусмотренный его интерфейсом, так что нет необходимости читать дополнительную документацию для класса реализации. Кроме того, использование такого статического фабричного метода требует от клиента обращения к возвращаемому объекту через интерфейс, а не через класс реализации, что в общем случае является хорошей практикой (раздел 9.8).

В Java 8 было ликвидировано ограничение, что интерфейсы не могут содержать статические методы, так что теперь мало причин для предоставления неинстанцируемого сопутствующего класса для интерфейса. Многие открытые статические члены, которые ранее располагались в таком классе, теперь размещаются в самом интерфейсе. Обратите, однако, внимание, что по-прежнему может оставаться необходимым поместить основную часть кода реализации этих статических методов в отдельный класс, закрытый на уровне пакета. Дело в том, что Java 8 требует, чтобы все статические члены интерфейса были открытыми. Java 9 разрешает иметь закрытые статические методы, но статические поля и статические классы-члены по-прежнему обязаны быть открытыми.

Четвертым преимуществом статических фабричных методов является то, что класс возвращенного объекта может варьироваться от вызова к вызову в зависимости от входных параметров. Допускается любой подтип объявленного типа возвращаемого значения. Класс возвращенного объекта может также изменяться от выпуска к выпуску.

Класс EnumSet (раздел 6.3) не имеет открытых конструкторов, а только статические фабрики. В реализации OpenJDK они возвращают экземпляр одного из двух подклассов в зависимости от размера базового типа перечисления: если в нем не более 64 элементов (как в большинстве перечислений), то статические фабрики возвращают экземпляр RegularEnumSet, который реализуется как один long; если же перечисление содержит больше 64 элементов, фабрики возвращают экземпляр JumboEnumSet с массивом long.

Существование этих двух реализаций классов является невидимым для клиентов. Если RegularEnumSet перестанет давать преимущества в производительности для малых перечислений, он может быть устранен из будущих версий без каких бы то ни было последствий. Аналогично в будущую версию можно добавить третью или четвертую реализацию EnumSet, если она окажется полезной для производительности. Клиенты не знают и не должны беспокоиться о классе объекта, который они получают от фабрики; для них важно только, что это некоторый подкласс EnumSet.

Пятое преимущество статических фабрик заключается в том, что класс возвращаемого объекта не обязан существовать во время разработки класса, содержащего метод. Такие гибкие статические фабричные методы образуют основу *каркасов провайдеров служб* (service provider frameworks) наподобие Java Database Connectivity API (JDBC). Каркас провайдера службы представляет собой систему, в которой провайдер реализует службу, а система делает реализацию доступной для клиентов, отделяя клиентов от реализаций.

Имеется три основных компонента каркаса провайдера службы: *интерфейс службы*, который представляет реализацию; *API регистрации провайдера*, который провайдеры используют для регистрации реализации; и *API доступа к службе*, который клиенты используют для получения экземпляров службы. API доступа к службе может позволить клиентам указать критерии выбора реализации. В отсутствие таких критериев API возвращает экземпляр реализации по умолчанию или позволяет клиенту циклически обойти все имеющиеся реализации. API доступа к службе представляет собой гибкую статическую фабрику, которая лежит в основе каркаса провайдера службы.

Необязательный четвертый компонент каркаса провайдера службы представляет собой *интерфейс провайдера службы*, который описывает объект фабрики, производящий экземпляры интерфейса службы. В отсутствие интерфейса провайдера службы реализации должны инстанцироваться рефлексивно

(раздел 9.9). В случае JDBC Connection играет роль части интерфейса службы, DriverManager.registerDriver представляет собой API регистрации провайдера, DriverManager.getConnection — API доступа к службе, а Driver — интерфейс провайдера службы.

Имеется множество вариантов шаблонов каркасов провайдеров служб. Например, API доступа к службе может возвращать более богатый интерфейс службы клиентам, чем представленной провайдерами. Это проектный шаблон *Мост* (Bridge) [12]. Каркасы (фреймворки) внедрения зависимостей (раздел 2.5) можно рассматривать как мощные провайдеры служб. Начиная с Java 6 платформа включает каркас провайдера служб общего назначения, java.util.ServiceLoader, так что вам не нужно (а в общем случае и не стоит) писать собственный каркас (раздел 9.3). JDBC не использует ServiceLoader, так как предшествует ему.

Основное ограничение предоставления только статических фабричных методов заключается в том, что классы без открытых или защищенных конструкторов не могут порождать подклассы. Например, невозможно создать подкласс любого из классов реализации в Collections Framework. Пожалуй, это может быть благом, потому что требует от программистов использовать композицию вместо наследования (раздел 4.4), и необходимо для неизменяемых типов (раздел 4.3).

Вторым недостатком статических фабричных методов является то, что их трудно отличить от других статических методов. Они не выделены в документации API так же, как конструкторы, поэтому может быть трудно понять, как создать экземпляр класса, предоставляемый статическим фабричным методом вместо конструкторов. Инструментарий Javadoc, возможно, когда-нибудь обратит внимание на статические фабричные методы. Указанный недостаток может быть смягчен путем привлечения внимания к статическим фабричным методам в документации класса или интерфейса, а также путем применения соглашений по именованию. Ниже приведены некоторые распространенные имена для статических фабричных методов, и этот список является далеко не исчерпывающим.

- **from** — *метод преобразования типа*, который получает один параметр и возвращает соответствующий экземпляр требуемого типа, например:

```
Date d = Date.from(instant);
```

- **of** — *метод агрегации*, который получает несколько параметров и возвращает соответствующий экземпляр требуемого типа, объединяющий их, например:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf** — более многословная альтернатива `from` и `of`, например:


```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```
- **instance** или **getInstance** — возвращает экземпляр, описываемый параметрами (если таковые имеются), но о котором нельзя сказать, что он имеет то же значение, например:


```
StackWalker luke = StackWalker.getInstance(options);
```
- **create** или **newInstance** — подобен `instance` или `getInstance`, но отличается тем, что гарантирует, что каждый вызов дает новый экземпляр, например:


```
Object newArray = Array.newInstance(classObject, arrayLen);
```
- **getType** — подобен `getInstance`, но используется, если фабричный метод находится в другом классе. *Type* представляет собой тип объекта, возвращаемого фабричным методом, например:


```
FileStore fs = Files.getFileStore(path);
```
- **newType** — подобен `newInstance`, но используется, если фабричный метод находится в другом классе. *Type* представляет собой тип объекта, возвращаемого фабричным методом, например:


```
BufferedReader br = Files.newBufferedReader(path);
```
- **type** — краткая альтернатива для `getType` и `newType`, например:


```
List<Complaint> litany = Collections.list(legacyLitany);
```

Итак, могут использоваться как статические фабричные методы, так и открытые конструкторы, и следует уделить внимание преимуществам одних перед другими. Часто статические фабрики являются предпочтительнее, так что гасите свой первый порыв предоставления открытого конструктора классу, не рассмотрев вначале возможность использования статических фабрик.

2.2. При большом количестве параметров конструктора подумайте о проектном шаблоне *Строитель*

Статические фабрики и конструкторы имеют общее ограничение: они не масштабируются для большого количества необязательных параметров. Рассмотрим случай класса, представляющего этикетку Nutrition Facts, которая имеется на упакованных пищевых продуктах. Эти этикетки имеют несколько обязательных полей — размер порции, число порций в упаковке, калорийность порции, а также более двадцати необязательных полей — количество жира,

содержание насыщенных жиров, трансжиров, холестерина, натрия и т.д. Большинство продуктов имеют ненулевые значения только для нескольких из этих необязательных полей.

Какие конструкторы или статические фабрики следует написать для такого класса? Традиционно программисты используют шаблон *телескопического конструктора*, когда предоставляется конструктор только с необходимыми параметрами, другой — с одним необязательным параметром, третий — с двумя необязательными параметрами и так далее до конструктора со всеми необязательными параметрами. Вот как это выглядит на практике (для краткости показаны только четыре необязательных поля).

// Шаблон телескопического конструктора - не масштабируется!

```
public class NutritionFacts
{
    private final int servingSize; // (мл в порции)      Необходим
    private final int servings;    // (количество порций) Необходим
    private final int calories;    // (калорий в порции)  Необязателен
    private final int fat;        // (жиров в порции)   Необязателен
    private final int sodium;     // (Na в порции)     Необязателен
    private final int carbohydrate; // (углеводы в порции) Необязателен
    public NutritionFacts(int servingSize, int servings)
    {
        this(servingSize, servings, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories)
    {
        this(servingSize, servings, calories, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat)
    {
        this(servingSize, servings, calories, fat, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium)
    {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium,
                          int carbohydrate)
    {
        this.servingSize = servingSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
    }
}
```

```

        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}

```

Когда вы хотите создать экземпляр этого класса, вы используете конструктор с наиболее коротким списком параметров, который содержит все параметры, которые вы хотите установить:

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

Обычно такой вызов конструктора требует множества параметров, которые вы не хотите устанавливать, но вынуждены передавать для них значения так или иначе. В данном случае мы передали значение 0 для жиров. Когда есть “всего лишь” шесть параметров, это может показаться не таким уж страшным, но по мере увеличения количества параметров ситуация быстро выходит из-под контроля.

Короче говоря, **шаблон телескопического конструктора работает, но очень трудно написать код клиента, у которого есть много параметров, и еще труднее его читать.** Читатель исходного текста должен гадать, что означают все эти значения, и тщательно рассчитывать позицию интересующего параметра. Длинные последовательности одинаково типизированных параметров могут вести к трудно обнаруживаемым ошибкам. Если клиент случайно поменяет местами два таких параметра, компилятор не будет жаловаться, но во время выполнения программа будет вести себя неверно (раздел 8.3).

Еще одной альтернативой при наличии большого количества обязательных параметров в конструкторе является шаблон *JavaBeans*, в котором для создания объекта вызывается конструктор без параметров, а затем вызываются методы для задания каждого обязательного параметра и всех необязательных параметров, требуемых в конкретной ситуации.

// Шаблон JavaBeans – обеспечивает изменяемость

```

public class NutritionFacts
{
    // Параметры инициализируются значениями
    // по умолчанию (если таковые имеются)
    // Необходим; значения по умолчанию нет:
    private int servingSize = -1;
    // Необходим; значения по умолчанию нет:
    private int servings = -1;

    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }
}

```

```
// Методы установки значений
public void setServingSize(int val)
{
    servingSize = val;
}
public void setServings(int val)
{
    servings = val;
}
public void setCalories(int val)
{
    calories = val;
}
public void setFat(int val)
{
    fat = val;
}
public void setSodium(int val)
{
    sodium = val;
}
public void setCarbohydrate(int val)
{
    carbohydrate = val;
}
}
```

У этого шаблона нет ни одного из недостатков шаблона телескопического конструктора. Создание экземпляров оказывается немного многословным, но легким и для написания, и для чтения:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

К сожалению, шаблон **JavaBeans** имеет собственные серьезные недостатки. Поскольку создание экземпляра распределено между несколькими вызовами, **JavaBean в процессе построения может оказаться в частично несогласованном состоянии**. Класс не имеет возможности обеспечить согласованность просто путем проверки корректности параметров конструктора. Попытка использовать объект, находящийся в несогласованном состоянии, может привести к ошибкам, которые находятся далеко от кода, содержащего ошибку, а потому трудно отлаживаются. Еще одним связанным недостатком является то, что шаблон **JavaBeans** исключает возможность сделать класс неизменяемым

(раздел 4.3) и требует дополнительных усилий со стороны программиста для обеспечения безопасности с точки зрения потоков.

Эти недостатки можно уменьшить, вручную “замораживая” объект после завершения его строительства и не позволяя использовать его до тех пор, пока он не будет разморожен, — но этот вариант громоздкий и редко используется на практике. Кроме того, он может привести к ошибкам времени выполнения, поскольку компилятор не может гарантировать, что программист вызвал метод заморозки объекта перед его использованием.

К счастью, существует третий вариант, который сочетает в себе безопасность шаблона телескопического конструктора и удобочитаемость шаблона JavaBeans. Это разновидность проектного шаблона *Строитель* (Builder) [12]. Вместо того чтобы создавать объект непосредственно, клиент вызывает конструктор (или статическую фабрику) со всеми необходимыми параметрами и получает *объект строителя*. Затем клиент вызывает методы установки полей объекта строителя для задания каждого необязательного параметра, представляющего интерес. Наконец, клиент вызывает метод `build` параметров для создания объекта (обычно неизменяемого). Строитель обычно представляет собой статический класс-член (раздел 4.10) класса, который он строит. Вот как это выглядит на практике.

// Шаблон Builder

```
public class NutritionFacts
{
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
    public static class Builder
    {
        // Необходимые параметры
        private final int servingSize;
        private final int servings;

        // Необязательные параметры — инициализированы
        // значениями по умолчанию
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;
        private int carbohydrate = 0;

        public Builder(int servingSize, int servings)
        {
            this.servingSize = servingSize;
            this.servings = servings;
        }
    }
}
```

```
public Builder calories(int val)
{
    calories = val;
    return this;
}
public Builder fat(int val)
{
    fat = val;
    return this;
}
public Builder sodium(int val)
{
    sodium = val;
    return this;
}
public Builder carbohydrate(int val)
{
    carbohydrate = val;
    return this;
}
public NutritionFacts build()
{
    return new NutritionFacts(this);
}
}
private NutritionFacts(Builder builder)
{
    servingSize = builder.servingSize;
    servings = builder.servings;
    calories = builder.calories;
    fat = builder.fat;
    sodium = builder.sodium;
    carbohydrate = builder.carbohydrate;
}
}
```

Класс `NutritionFacts` неизменяемый, и все значения параметров по умолчанию находятся в одном месте. Методы установки полей строителя возвращают сам строитель, так что эти вызовы можно объединять в цепочки, получая *поточковый* (fluent) API. Вот как выглядит код клиента:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
    .calories(100).sodium(35).carbohydrate(27).build();
```

Такой клиентский код легко писать, а главное — легко читать. **Шаблон Строитель имитирует именованные необязательные параметры**, как в языках программирования Python или Scala.

Проверка корректности параметров для краткости опущена. Чтобы как можно скорее обнаруживать недопустимые параметры, правильность параметров

проверяется в конструкторе и методах строителя. Проверка инвариантов включает несколько параметров в конструкторе, вызываемом методом `build`. Чтобы защитить эти инварианты, проверка полей объекта выполняется после копирования параметров из строителя (раздел 8.2). Если проверка не пройдена, генерируется исключение `IllegalArgumentException` (раздел 10.4), в котором подробно указывается, какие параметры оказались недопустимыми (раздел 10.7).

Шаблон *Строитель* хорошо подходит для иерархий классов. Используйте параллельные иерархии строителей, в которых каждый вложен в соответствующий класс. Абстрактные классы имеют абстрактных строителей; конкретные классы имеют конкретных строителей. Например, рассмотрим абстрактный класс в корне иерархии, представляющей различные виды пиццы:

```
// Шаблон Строитель для иерархий классов
public abstract class Pizza
{
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;
    abstract static class Builder<T extends Builder<T>>
    {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping)
        {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }
        abstract Pizza build();

        // Подклассы должны перекрывать этот метод, возвращая "себя"
        protected abstract T self();
    }
    Pizza(Builder<?> builder)
    {
        toppings = builder.toppings.clone(); // См. раздел 8.2
    }
}
```

Обратите внимание, что `Pizza.Builder` является *обобщенным типом с рекурсивным параметром типа* (раздел 5.5). Это, наряду с абстрактным методом `self`, обеспечивает корректную работу цепочек методов в подклассах без необходимости приведения типов. Этот обходной путь для того факта, что в Java нет “типа самого себя” (или “собственного типа”), известен как идиома *имитации собственного типа*.

Вот два конкретных подкласса класса `Pizza`, один из которых представляет стандартную Нью-Йоркскую пиццу, другой — Кальцоне. Первый имеет

необходимый параметр размера, а второй позволяет указать, где должен находиться соус — внутри или снаружи:

```
public class NyPizza extends Pizza
{
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;
    public static class Builder extends Pizza.Builder<Builder>
    {
        private final Size size;
        public Builder(Size size)
        {
            this.size = Objects.requireNonNull(size);
        }
        @Override public NyPizza build()
        {
            return new NyPizza(this);
        }
        @Override protected Builder self()
        {
            return this;
        }
    }
    private NyPizza(Builder builder)
    {
        super(builder);
        size = builder.size;
    }
}

public class Calzone extends Pizza
{
    private final boolean sauceInside;
    public static class Builder extends Pizza.Builder<Builder>
    {
        private boolean sauceInside = false; // По умолчанию
        public Builder sauceInside()
        {
            sauceInside = true;
            return this;
        }
        @Override public Calzone build()
        {
            return new Calzone(this);
        }
        @Override protected Builder self()
        {
            return this;
        }
    }
}
```

```

private Calzone(Builder builder)
{
    super(builder);
    sauceInside = builder.sauceInside;
}
}

```

Обратите внимание, что метод `build` в строителе каждого подкласса объявляется как возвращающий корректный подкласс: метод `build` класса `NyPizza.Builder` возвращает `NyPizza`, в то время как в `Calzone.Builder` возвращается `Calzone`. Эта методика, в которой метод подкласса объявляется как возвращающий подтип возвращаемого типа, объявленного в суперклассе, известна как *ковариантное типизирование возврата*. Она позволяет клиентам использовать эти строители без необходимости приведения типов.

Клиентский код этих “иерархических строителей”, по сути, идентичен коду простого строителя `NutritionFacts`. В примере кода клиента, показанном далее, для краткости предполагается статический импорт констант перечисления:

```

NyPizza pizza = new NyPizza.Builder(SMALL)
    .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
    .addTopping(HAM).sauceInside().build();

```

Незначительное преимущество строителей над конструкторами заключается в том, что строители могут иметь несколько переменных (необязательных) параметров, поскольку каждый параметр указан в собственном методе. Кроме того, строители могут собирать параметры, передаваемые в нескольких вызовах метода, в едином поле, как было показано в методе `addTopping` выше.

Проектный шаблон *Строитель* весьма гибкий. Один строитель может использоваться многократно для создания нескольких объектов. Параметры строителя могут корректироваться между вызовами метода `build` для небольших изменений создаваемых объектов. Строитель может автоматически заполнять некоторые поля при создании объекта, например серийный номер, который увеличивается каждый раз при создании нового объекта.

Конечно же, проектный шаблон *Строитель* имеет и недостатки. Чтобы создать объект, необходимо сначала создать его строитель. На практике обычно стоимость создания строителя будет незаметной, но может оказаться проблемой в смысле производительности в критических ситуациях. Кроме того, шаблон *Строитель* является более многословным, чем шаблон телескопического конструктора, поэтому его следует использовать только при наличии достаточно большого количества параметров, по крайней мере не менее четырех. Но не забывайте, что в будущем могут добавиться и другие параметры. Но если

вы начинаете с конструкторов или статических фабрик и переключаетесь на строитель, когда класс уже достаточно активно используется, то устаревшие конструкторы или статические фабрики будут мешать, как больной зуб. Таким образом, зачастую лучше сразу начинать работать со строителем.

Итак, шаблон проектирования *Строитель* является хорошим выбором при проектировании классов, конструкторы или статические фабрики которых будут иметь большое количество параметров, особенно если многие из этих параметров оказываются необязательными или имеют одинаковый тип. Код клиента с использованием строителей гораздо проще для чтения и записи, чем код с телескопическими конструкторами, а применение строителей гораздо безопаснее, чем использование JavaBeans.

2.3. Получайте синглтон с помощью закрытого конструктора или типа перечисления

Синглтон — это просто класс, который инстанцируется только один раз [12]. Синглтоны обычно представляют собой либо объект без состояния, такой как функция (раздел 4.10), либо системный компонент, который уникален по своей природе. **Превращение класса в синглтон может затруднить тестирование его клиентов**, потому что невозможно заменить ложную реализацию синглтоном, если только он не реализует интерфейс, который служит в качестве его типа.

Имеется два распространенных способа реализации классов синглтонов. Оба они основаны на создании закрытого конструктора и экспорте открытого статического элемента для предоставления доступа к единственному экземпляру. В первом подходе член представляет собой поле `final`:

```
// Синглтон с полем public final
public class Elvis
{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis()
    {
        ...
    }
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Закрытый конструктор вызывается только один раз для инициализации открытого статического `final`-поля `Elvis.INSTANCE`. Отсутствие открытого или защищенного конструктора *гарантирует* “одноэльвисность”: после инициализации класса `Elvis` будет существовать ровно один экземпляр класса `Elvis` — ни больше, ни меньше. Что бы ни делал клиент, это ничего не изменит (с одной оговоркой: привилегированный клиент может вызвать закрытый конструктор рефлексивно (раздел 9.9) с помощью метода `AccessibleObject.setAccessible`). Если вам нужно защититься от такого нападения, измените конструктор так, чтобы он генерировал исключение при попытках создания второго экземпляра.

Во втором подходе к реализации классов-синглтонов открытый член представляет собой статический фабричный метод:

```
// Синглтон со статической фабрикой
public class Elvis
{
    private static final Elvis INSTANCE = new Elvis();
    private Elvis()
    {
        ...
    }
    public static Elvis getInstance()
    {
        return INSTANCE;
    }
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Все вызовы `Elvis.getInstance` возвращают ссылку на один и тот же объект, и никакой другой экземпляр `Elvis` никогда не будет создан (с той же ранее упомянутой оговоркой).

Главным преимуществом подхода с открытым полем является то, что API делает очевидным, что класс является синглтоном: открытое статическое поле объявлено как `final`, так что оно всегда будет содержать ссылку на один и тот же объект. Вторым преимуществом является то, что он проще.

Одним из преимуществ подхода со статической фабрикой является то, что он обеспечивает достаточную гибкость для того, чтобы изменить синглтон на класс, не являющийся таковым, без изменения его API. Фабричный метод возвращает единственный экземпляр, но может быть изменен таким образом, чтобы возвращать, скажем, отдельный экземпляр для каждого вызывающего его потока. Вторым преимуществом является то, что можно написать *обобщенную*

фабрику синглтонов, если таковая требуется вашему приложению (раздел 5.5). Последнее преимущество использования статической фабрики состоит в том, что *ссылка на метод* может использоваться в качестве поставщика, например `Elvis::instance` является `Supplier<Elvis>`. Если ни одно из этих преимуществ не является значимым, предпочтительнее оказывается подход с открытым полем.

Чтобы сделать класс синглтона, который использует один из этих подходов, *сериализуемым* (глава 12, “Сериализация”), недостаточно просто добавить в его объявление `implements Serializable`. Для гарантии сохранения свойства синглтона объявите все поля экземпляра как `transient` и предоставьте метод `readResolve` (раздел 12.5). В противном случае при каждой десериализации сериализованного экземпляра будет создаваться новый экземпляр, что в нашем примере приведет к появлению ложных объектов `Elvis`. Чтобы предотвратить этот эффект, добавьте в класс `spurious` такой метод `readResolve`:

```
// Метод readResolve для сохранения свойства синглтона
private Object readResolve() {
    // Возвращает истинный объект Elvis и позволяет
    // сборщику мусора позаботиться о самозванце.
    return INSTANCE;
}
```

Третий способ реализации синглтона состоит в объявлении одноэлементного перечисления:

```
// Синглтон-перечисление – предпочтительный подход
public enum Elvis
{
    INSTANCE;
    public void leaveTheBuilding()
    {
        ...
    }
}
```

Этот подход аналогичен подходу с открытым полем, но является более компактным, с бесплатным предоставлением механизма сериализации, и обеспечивает твердую гарантию отсутствия нескольких экземпляров, даже перед лицом сложной сериализации или атаки через рефлексиию. Этот подход может казаться немного неестественным, но **одноэлементное перечисление зачастую представляет собой наилучший способ реализации синглтона**. Обратите внимание, что этот подход нельзя использовать, если ваш синглтон должен расширять суперкласс, отличный от `Enum` (хотя *можно* объявить перечисление для реализации интерфейсов).

2.4. Обеспечивайте неизinstancируемость с помощью закрытого конструктора

Иногда требуется написать класс, который представляет собой просто сгруппированные статические методы и статические поля. Такие классы приобрели плохую репутацию, потому что некоторые программисты злоупотребляют ими, чтобы превратить объектно-ориентированное программирование в процедурное, но они имеют и вполне законные применения. Они могут использоваться для группирования связанных методов над примитивными значениями или массивами, как `java.lang.Math` или `java.util.Arrays`. Они могут также использоваться для группирования статических методов, включая фабрики (раздел 2.1), для объектов, реализующих некоторый интерфейс, наподобие `java.util.Collections`. (Начиная с Java 8 такие методы можно поместить в интерфейс, в предположении, что вы можете его изменять.) Наконец, такие классы могут использоваться для группирования методов в `final`-классе, поскольку их нельзя поместить в подкласс.

Такие *служебные классы* не предназначены для создания экземпляров: их экземпляры не имеют смысла. Однако в отсутствие явных конструкторов компилятор предоставляет открытый *конструктор по умолчанию* без параметров. Для пользователя этот конструктор неотличим от любого другого. Не редкость встретить непреднамеренное инстанцирование пользовательских классов в опубликованных API.

Попытки запретить инстанцирование, делая класс абстрактным, неработоспособны. Кроме того, этот способ вводит в заблуждение пользователей, считающих, что абстрактный класс предназначен для наследования (раздел 4.5). Однако имеется очень простая идиома обеспечения неизinstancируемости. Конструктор по умолчанию создается, только если класс не содержит явных конструкторов, так что **класс можно сделать неизinstancируемым, добавляя в него закрытый конструктор:**

```
// Неинстанцируемый служебный класс
public class UtilityClass
{
    // Подавление создания конструктора по умолчанию
    // для достижения неизinstancируемости
    private UtilityClass()
    {
        throw new AssertionError();
    }
    ... // Остальной код опущен
}
```

Поскольку явный конструктор является закрытым, он недоступен вне класса. Исключение `AssertionError` не является строго обязательным, но обеспечивает страховку при случайном вызове конструктора внутри класса. Это гарантирует, что класс не будет создаваться никогда, ни при каких обстоятельствах. Эта идиома несколько контринтуитивна, потому что конструктор предоставляется, но так, что его нельзя вызвать. Поэтому целесообразно включать в код соответствующий комментарий, как показано выше.

В качестве побочного эффекта эта идиома предотвращает также наследование такого класса. Все конструкторы подкласса должны прямо или косвенно вызывать конструктор суперкласса, но подкласс не будет иметь возможности вызова конструктора суперкласса.

2.5. Предпочитайте внедрение зависимостей жестко прошитым ресурсам

Многие классы зависят от одного или нескольких базовых ресурсов. Например, средство проверки орфографии зависит от словаря. Не редкость — увидеть реализацию таких классов как статических (раздел 2.4):

```
// Ненадлежащее использование статического служебного
// класса — негибкое и не тестируемое!
public class SpellChecker
{
    private static final Lexicon dictionary = ...;
    private SpellChecker() {} // Неинстанцируемый
    public static boolean isValid(String word)
    {
        ...
    }
    public static List<String> suggestions(String typo)
    {
        ...
    }
}
```

Аналогично не редкость и реализация таких классов как синглтонов (раздел 2.3):

```
// Ненадлежащее использование синглтона — негибкое и не
// тестируемое!
public class SpellChecker
{
    private final Lexicon dictionary = ...;
    private SpellChecker(...) {}
}
```

```

public static INSTANCE = new SpellChecker(...);
public boolean isValid(String word)
{
    ...
}
public List<String> suggestions(String typo)
{
    ...
}
}

```

Ни один из этих подходов не является удовлетворительным, поскольку они предполагают, что существует только один словарь, который стоит использовать. На практике каждый язык имеет собственный словарь, а для специальных целей могут использоваться специальные словари. Кроме того, может быть желательным использовать специальный словарь для тестирования. Глупо считать, что одного словаря будет достаточно всегда и для всех целей.

Вы могли бы попытаться обеспечить поддержку классом `SpellChecker` нескольких словарей, сделав поле `dictionary` не окончательным и добавив метод изменения словаря в существующем классе проверки орфографии, но этот выход некрасивый, подверженный ошибкам и непригодный при использовании параллелизма. **Статические служебные классы и синглтоны непригодны для классов, поведение которых параметризовано лежащим в их основе ресурсом.**

Что на самом деле требуется — это возможность поддержки нескольких экземпляров класса (в нашем примере — класса `SpellChecker`), каждый из которых использует необходимый клиенту ресурс (в нашем примере — словарь). Простая схема, удовлетворяющая этому требованию — **передача ресурса конструктору при создании нового экземпляра**. Это одна из форм *внедрения зависимостей* (*dependency injection*): словарь является *зависимостью* класса проверки орфографии, которая *внедряется* в класс при его создании.

// Внедрение зависимостей обеспечивает гибкость и тестируемость

```

public class SpellChecker
{
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary)
    {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word)
    {
        ...
    }
}

```

```
public List<String> suggestions(String typo)
{
    ...
}
```

Схема внедрения зависимостей настолько проста, что многие программисты годами используют ее, даже не подозревая, что она имеет собственное имя. Хотя наш пример с проверкой орфографии использует только один ресурс (словарь), внедрение зависимостей работает с произвольным количеством ресурсов и произвольными графами зависимостей. Он сохраняет неизменность классов (раздел 4.3), так что несколько клиентов могут совместно использовать зависимые объекты (в предположении, что клиентам нужны одни и те же базовые ресурсы). Внедрение зависимостей в равной степени применимо к конструкторам, статическим фабрикам (раздел 2.1) и строителям (раздел 2.2).

Полезная разновидность схемы состоит в передаче конструктору *фабрики* ресурсов. Фабрика — это объект, который может многократно вызываться для создания экземпляров типа. Такие фабрики воплощают проектный шаблон *Фабричный метод* [12]. Интерфейс `Supplier<T>`, введенный в Java 8, идеально подходит для представления фабрики. Методы, получающие `Supplier<T>` в качестве входных данных, обычно должны ограничивать параметр типа фабрики с помощью *ограниченного подстановочного типа* (bounded wildcard type) (раздел 5.6), чтобы позволить клиенту передать фабрику, которая создает любой подтип указанного типа. Например, вот метод, который создает мозаику с использованием клиентской фабрики для производства каждой плитки мозаики:

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

Хотя внедрение зависимостей значительно повышает гибкость и тестируемость, оно может засорять крупные проекты, которые обычно содержат тысячи зависимостей. Такое засорение можно устранить с помощью *каркаса внедрения зависимостей* (dependency injection framework), такого как Dagger [10], Guice [16] или Spring [45]. Использование этих каркасов выходит за рамки данной книги, однако обратите внимание, что API, разработанные для внедрения зависимостей вручную, тривиально адаптируются для использования этими каркасами.

Резюме: не используйте синглтон или статический служебный класс для реализации класса, который зависит от одного или нескольких базовых ресурсов, поведение которых влияет на этот класс, и не давайте классу непосредственно создавать эти ресурсы. Вместо этого передавайте ресурсы, или фабрики для их создания, конструктору (или статической фабрике или строителю). Эта практика, известная как внедрение зависимостей, значительно повышает гибкость, степень повторного использования и возможности тестирования класса.

2.6. Избегайте создания излишних объектов

Зачастую целесообразно повторно использовать один объект вместо создания нового, функционально эквивалентного объекта всякий раз, когда он становится необходимым. Повторное использование может быть как более быстрым, так и более элегантным. Неизменяемый объект всегда может использоваться повторно (раздел 4.3).

Вот (экстремальный) пример того, как не надо поступать:

```
String s = new String("bikini"); // Не делайте так!
```

Инструкция создает новый экземпляр `String` при каждом выполнении, и при этом ни одно из этих созданий не является необходимым. Аргументом конструктора `String("bikini")` является сам экземпляр `String`, функционально идентичный всем объектам, создаваемым конструктором. Если это происходит в цикле или в часто вызываемом методе, миллионы экземпляров `String` могут быть созданы напрасно.

Усовершенствованная версия имеет следующий вид:

```
String s = "bikini";
```

Эта версия использует единственный экземпляр `String` вместо создания нового при каждом выполнении. Кроме того, гарантируется, что этот объект будет повторно использоваться любым другим кодом, выполняемым той же виртуальной машиной и содержащим ту же литеральную строку [25, 3.10.5].

Часто можно избежать создания ненужных объектов, если в неизменяемом классе, имеющем и конструкторы, и *статические фабричные методы* (раздел 2.1), предпочесть последние первым. Например, фабричный метод `Boolean.valueOf(String)` предпочтительнее конструктора `Boolean(String)` (который упразднен в Java 9). Конструктор *обязан* создавать новый объект при каждом вызове, в то время как фабричный метод не обязан поступать таким образом (и на практике так и не поступает). В дополнение к повторному использованию неизменяемых объектов можно также повторно использовать изменяемые объекты, если вы знаете, что они не будут изменены.

Создание одних объектов оказывается гораздо более дорогим, чем других. Если вам многократно требуется такой “дорогой объект”, возможно, целесообразно кешировать его для повторного использования. К сожалению, не всегда очевидно, что вы создаете такой объект. Предположим, что вы хотите написать метод для определения, является ли строка корректным римским числом. Вот самый простой способ сделать это с помощью регулярного выражения:

```
// Производительность можно существенно повысить!  
static boolean isRomanNumeral(String s) {  
    return s.matches("(?=.)M*(C[MD]|D?C{0,3}) "  
        + "(X[CL]|L?X{0,3}) (I[XV]|V?I{0,3})$");  
}
```

Проблема этой реализации в том, что она опирается на метод `String.matches`. Хотя **`String.matches`** — простейший способ проверки, соответствует ли строка регулярному выражению, он не подходит для многократного использования в ситуациях, критичных в смысле производительности. Беда в том, что он внутренне создает экземпляр `Pattern` для регулярного выражения и использует его только один раз, после чего он становится добычей сборщика мусора. Создание экземпляра `Pattern` достаточно дорогостоящее, поскольку требует компиляции регулярного выражения в конечный автомат.

Для повышения производительности, как часть инициализации класса, можно явно компилировать регулярное выражение в экземпляр `Pattern` (который является неизменяемым) и повторно использовать тот же экземпляр для каждого вызова метода `isRomanNumeral`:

```
// Повторное использование дорогостоящего объекта  
// для повышения производительности  
public class RomanNumerals  
{  
    private static final Pattern ROMAN =  
        Pattern.compile("(?=.)M*(C[MD]|D?C{0,3}) "  
            + "(X[CL]|L?X{0,3}) (I[XV]|V?I{0,3})$");  
  
    static boolean isRomanNumeral(String s)  
    {  
        return ROMAN.matcher(s).matches();  
    }  
}
```

Улучшенная версия `isRomanNumeral` обеспечивает значительное повышение производительности при частых вызовах. На моей машине исходная версия требует 1.1 мкс на 8-символьной входной строке, а улучшенная версия — 0.17 мкс, что в 6,5 раза быстрее. При этом повышается не только производительность, но, возможно, и ясность кода. Статическое поле `final` для в противном случае невидимого экземпляра `Pattern` позволяет дать ему имя, которое гораздо более понятно, чем само регулярное выражение.

Если класс, содержащий улучшенную версию метода `isRomanNumeral`, инициализируется, но рассматриваемый метод никогда не вызывается, поле `ROMAN` будет инициализировано напрасно. Такую ненужную инициализацию можно было бы устранить с помощью *ленивой (отложенной) инициализации* поля (раздел 11.6) при первом вызове метода `isRomanNumeral`, но это не рекомендуется. Как это часто бывает с отложенной инициализацией, она

усложняет реализацию, не приводя к измеримым улучшениям производительности (раздел 9.11).

Когда объект является неизменяемым, очевидно, что он может безопасно использоваться повторно; но бывают другие ситуации, в которых это гораздо менее очевидно и даже противоречит здравому смыслу. Рассмотрим случай *адаптеров* [12]. Адаптер представляет собой объект, который выполняет делегирование другому объекту, обеспечивая альтернативный интерфейс. Поскольку адаптер не имеет состояния, помимо состояния объекта, которому выполняется делегирование, нет и необходимости создавать более одного экземпляра данного адаптера для заданного объекта.

Например, метод `keySet` интерфейса `Map` возвращает представление `Set` объекта `Map`, состоящего из всех ключей отображения. Представление, что каждый вызов `keySet` должен создавать новый экземпляр `Set`, наивное, поскольку каждый вызов `keySet` данного объекта `Map` может возвращать один и тот же экземпляр `Set`. Хотя возвращаемый экземпляр `Set` обычно изменяемый, все возвращаемые объекты функционально идентичны: когда один из возвращаемых объектов изменяется, то же самое происходит и с остальными объектами, потому что все они основаны на одном и том же экземпляре `Map`. Хотя создание нескольких экземпляров объекта `keySet` в основном безвредно, оно излишне и не имеет никаких преимуществ.

Еще одним путем создания ненужных объектов является *автоматическая упаковка* (`autoboxing`), которая позволяет программисту смешивать примитивные и упакованные примитивные типы, упаковывая и распаковывая их при необходимости автоматически. **Автоматическая упаковка размывает, но не стирает различие между примитивными и упакованными примитивными типами.** Имеются тонкие семантические различия и не столь тонкие различия в производительности (раздел 9.5). Рассмотрим следующий метод, который вычисляет сумму всех положительных значений типа `int`. Для этого программа должна использовать арифметику `long`, поскольку тип `int` недостаточно большой, чтобы содержать сумму положительных значений `int`:

// Ужасно медленно! Вы можете найти создание объекта?

```
private static long sum()
{
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;
    return sum;
}
```

Эта программа дает правильный ответ, но она *гораздо* медленнее, чем должна бы быть, — из-за опечатки в одном символе. Переменная `sum` объявлена как

Long вместо long, а это означает, что программа создает около 2^{31} ненужных экземпляров Long (примерно по одному для каждого добавления long i к Long sum). Изменение объявления sum из Long в long уменьшает время выполнения от 6,3 до 0,59 с на моей машине. Урок ясен: **предпочитайте примитивы упакованным примитивам и следите за непреднамеренной автоматической упаковкой.**

В этом разделе не утверждается, что создание объектов является дорогостоящим и его следует избегать. Напротив, создание и уничтожение небольших объектов, конструкторы которых выполняют мало действий, оказывается дешевым, в особенности в современных реализациях JVM. Создание дополнительных объектов для повышения ясности, простоты или мощи программы, как правило, оказывается хорошим решением.

И наоборот, избегание создания объектов путем поддержки собственного пула объектов оказывается плохой идеей, если объекты в пуле чрезвычайно тяжеловесны. Классическим примером объекта, который *оправдывает* существование пула объектов, является подключение к базе данных. Стоимость подключения достаточно высока, что придает смысл повторному использованию этих объектов. Однако, вообще говоря, поддержка собственных пулов объектов запутывает код, увеличивает объем используемой памяти и наносит ущерб производительности. Современные реализации JVM имеют хорошо оптимизированные сборщики мусора, которые легко превосходят такие пулы небольших объектов.

“Противовесом” к этому разделу является раздел 8.2, посвященный *защитному копированию* (defensive copying). В текущем разделе говорится “не создавайте новый объект тогда, когда вы должны повторно использовать уже существующий”, раздел же 8.2 гласит “не используйте повторно существующий объект, когда вы должны создать новый”. Обратите внимание, что ущерб от повторного использования объекта при вызове защитного копирования гораздо больший, чем ущерб от напрасного создания дубликата объекта. Неспособность сделать при необходимости защитные копии может привести к коварным ошибкам и брешам в безопасности; создание излишних объектов влияет только на стиль и производительность.

2.7. Избегайте устаревших ссылок на объекты

Когда вы переходите с языка с ручным управлением памятью, как, например, С или С++, на язык с автоматической сборкой мусора наподобие Java, ваша работа как программиста упрощается, потому что ваши объекты автоматически утилизируются, когда вы перестаете с ними работать. Когда вы

впервые сталкиваетесь с этим, кажется, что все выполняется как по волшебству². Сборка мусора может легко ввести вас в заблуждение, что вы не должны думать об управлении памятью, но это не совсем верно.

Рассмотрим следующую простую реализацию стека:

// Вы можете найти утечку памяти?

```
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();

        return elements[--size];
    }
    /**
     * Убеждаемся, что в стеке есть место хотя бы для одного
     * элемента; если нет - удваиваем емкость массива.
     */
    private void ensureCapacity()
    {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

В этой программе нет ничего неверного (но читайте обобщенную версию в разделе 5.4). Вы можете провести исчерпывающее тестирование — и программа пройдет любой тест, но все равно в ней есть скрытая проблема. Грубо говоря, программа имеет “утечку памяти”, которая может проявляться как снижение производительности из-за усиленной работы сборщика мусора или увеличения используемой памяти. В предельных случаях такая утечка памяти

² Или, наоборот, вы в растерянности, потому что теряете контроль над происходящим. — *Примеч. ред.*

может вызвать свопинг на диск и даже сбой программы с ошибкой `OutOfMemoryError`, но такие ситуации сравнительно редки.

Так где же прячется эта утечка? Если стек растет, а затем уменьшается, то объекты, которые были сняты со стека, не могут быть удалены сборщиком мусора, даже если программа, пользующаяся стеком, уже не обращается к ним. Вся проблема в том, что стек хранит *устаревшие ссылки* (*obsolete reference*) на эти объекты. Устаревшая ссылка — это ссылка, которая уже никогда не будет разыменована. В данном случае устаревшими являются любые ссылки, оказавшиеся за пределами “активной части” массива элементов стека. Активная часть стека включает только элементы, индексы которых меньше значения `size`.

Утечки памяти в языках со сборкой мусора (точнее, их следует называть *непреднамеренным удержанием объектов* (*unintentional object retention*)) очень коварны. Если ссылка на объект случайно сохранена, сборщик мусора не может удалить не только этот объект, но и все объекты, на которые он ссылается, и далее по цепочке. Если даже непреднамеренно было сохранено всего несколько объектов, недоступными сборщику мусора могут стать многие и многие объекты, что может существенно повлиять на производительность программы.

Решение проблем такого рода очень простое: как только ссылки устаревают, их нужно обнулять. В нашем классе `Stack` ссылка на элемент становится устаревшей, как только ее объект снимается со стека. Исправленный вариант метода `pop` выглядит следующим образом:

```
public Object pop()
{
    if (size == 0)
        throw new EmptyStackException();

    Object result = elements[--size];
    elements[size] = null; // Устранение устаревшей ссылки
    return result;
}
```

Дополнительным преимуществом обнуления устаревших ссылок является то, что, если они впоследствии по ошибке разыменовываются, программа немедленно генерирует исключение `NullPointerException` вместо того, чтобы молча делать неправильные вещи. Всегда выгодно обнаруживать ошибки программирования как можно раньше.

Когда программисты впервые сталкиваются с этой проблемой, они могут перестраховываться и обнулять каждую ссылку на объект, как только программа больше его не использует. Эта практика не является ни необходимой, ни желательной, и только излишне загромождает программу. **Обнуление ссылки на объект должно быть скорее исключением, чем нормой.** Лучший способ

устранить устаревшие ссылки — выход переменной, содержащей ссылку, из области видимости. Это происходит естественным путем, если вы определяете каждую переменную в наиболее узкой области видимости из возможных (раздел 9.1).

Так когда следует обнулять ссылки? Какой аспект класса `Stack` делает его восприимчивым к утечке памяти? Попросту говоря, он *управляет собственной памятью*. Пул хранения состоит из элементов массива `elements` (ссылок на объекты, но не самих объектов). Элементы в активной части массива (определенной выше) *распределены* (*allocated*), а в оставшейся части массива — *свободны* (*free*). Сборщик мусора ничего об этом не знает; для него все ссылки на объекты в массиве `elements` являются одинаково корректными. Только программист знает, что неактивная часть массива не имеет значения. Программист сообщает об этом факте сборщику мусора, вручную обнуляя элементы массива, как только они оказываются в неактивной части.

Вообще говоря, **как только какой-либо класс начинает управлять своей памятью, программист должен озаботиться вопросами утечки памяти**. Когда элемент массива освобождается, необходимо обнулять любые ссылки на объекты, имевшиеся в этом элементе.

Еще одним распространенным источником утечек памяти являются кеши. Поместив в кеш ссылку на некий объект, можно легко забыть о том, что она там есть, и хранить ссылку в кеше еще долгое время после того, как она стала ненужной. Возможны несколько решений этой проблемы. Если вам повезло реализовать кеш, запись в котором остается значимой ровно до тех пор, пока за пределами кеша имеются ссылки на ее ключ вне кеша, представляя кеш наподобие `WeakHashMap`, то когда записи устареют, они будут удалены автоматически. Помните, что использовать `WeakHashMap` имеет смысл, только если желаемое время жизни записей кеша определяется внешними ссылками на ключ, а не значением.

В более общем случае время жизни записи в кеше менее хорошо определено. Записи с течением времени просто становятся менее значимыми. В такой ситуации кеш следует время от времени очищать от записей, которые уже не используются. Подобная очистка может выполняться фоновым потоком (например `ScheduledThreadPoolExecutor`) либо быть побочным эффектом добавления в кеш новых записей. Класс `LinkedHashMap` облегчает этот подход с помощью своего метода `removeEldestEntry`. Для более сложных кешей может потребоваться использовать `java.lang.ref` непосредственно.

Третий распространенный источник утечки памяти — приложения в режиме ожидания и другие обратные вызовы. Если вы реализуете API, в котором клиенты регистрируют обратные вызовы, но позже не отменяют эту регистрацию, то, если вы ничего не предпринимаете, они накапливаются. Один

из способов гарантировать, что обратные вызовы доступны сборщику мусора — это хранить на них только *слабые ссылки* (weak references), например, сохраняя их лишь в качестве ключей в WeakHashMap.

Поскольку утечка памяти обычно не обнаруживает себя в виде очевидного сбоя, она может оставаться в системе годами. Как правило, обнаруживается она лишь в результате тщательной инспекции программного кода или с помощью инструмента отладки, известного как *профайлер памяти* (heap profiler). Поэтому очень важно научиться предвидеть проблемы, похожие на описанную, еще до того, как они возникнут, и предупредить их появление.

2.8. Избегайте финализаторов и очистителей

Финализаторы непредсказуемы, часто опасны и в общем случае не нужны. Их использование может вызвать ошибочное поведение, снижение производительности и проблемы переносимости. Финализаторы имеют несколько корректных применений, которые мы рассмотрим позже в этом разделе, но, как правило, их нужно избегать. В Java 9 финализаторы являются устаревшими и не рекомендуемыми к применению, но они по-прежнему используются библиотеками Java. Java 9 заменяет финализаторы *очистителями* (cleaners). **Очистители менее опасны, чем финализаторы, но столь же непредсказуемые, медленные и, в общем случае, ненужные.**

Программистов на C++ следует предостеречь воспринимать финализаторы или очистители как аналог деструкторов C++ в Java. В C++ деструкторы являются обычным способом освобождения связанных с объектом ресурсов, необходимым дополнением к конструкторам. В Java сборщик мусора освобождает связанную с объектом память, когда объект становится недоступным, не требуя никаких усилий со стороны программиста. Деструкторы C++ используются также для освобождения других ресурсов, не связанных с памятью. В Java для этого используется блок try-c-ресурсами или try-finally (раздел 2.9).

Один из недостатков финализаторов и очистителей является то, что нет никакой гарантии их своевременного выполнения [25, 12.6].

С момента, когда объект становится недоступным, и до момента выполнения финализатора или очистителя может пройти сколь угодно длительное время. Это означает, что **с помощью финализатора или очистителя нельзя выполнять никакие операции, критичные по времени.** Например, будет серьезной ошибкой ставить процедуру закрытия открытых файлов в зависимость от финализатора или очистителя, поскольку дескрипторы открытых файлов — ограниченный ресурс. Если из-за того, что JVM задерживается с выполнением финализаторов или очистителей, будут оставаться открытыми много файлов,

программа может аварийно завершиться из-за того, что не сможет открывать новые файлы.

Частота, с которой запускаются финализаторы или очистители, в первую очередь, определяется алгоритмом сборки мусора, который существенно меняется от одной реализации JVM к другой. Точно так же может меняться и поведение программы, работа которой зависит от частоты вызова финализаторов или очистителей. Вполне возможно, что такая программа будет превосходно работать с одной JVM, на которой вы выполняете ее тестирование, а затем перестанет работать на другой JVM, которую предпочитает ваш самый важный клиент.

Запоздавая финализация — проблема не просто теоретическая. Создание финализатора для какого-либо класса может привести к задержке произвольной длины при удалении его экземпляров. Один мой коллега отлаживал приложение GUI, которое было рассчитано на длительное функционирование, но таинственно умирало с ошибкой `OutOfMemoryError`. Анализ показал, что в момент смерти у этого приложения в очереди на удаление стояли тысячи графических объектов, ждавших вызова финализатора и освобождения памяти. К несчастью, поток финализации выполнялся с меньшим приоритетом, чем другой поток того же приложения, так что удаление объектов не могло выполняться в том же темпе, в каком они становились доступными для финализации. Спецификация языка Java не дает никаких гарантий относительно выполнения финализаторов, так что нет никакого переносимого способа предотвратить проблемы такого вида, кроме как просто воздерживаться от использования финализаторов. Очистители в этом отношении оказываются немного лучше финализаторов, так как авторы классов могут управлять их потоками очистки, но очистители по-прежнему работают в фоновом режиме, под управлением сборщика мусора, так что никакой гарантии своевременной очистки не может быть.

Спецификация языка программирования Java не только не дает гарантии своевременного вызова финализаторов или очистителей, но и не дает гарантии, что они вообще будут вызваны. Вполне возможно (и даже вероятно), что программа завершится, так и не вызвав их для некоторых объектов, ставших недоступными. Как следствие **вы никогда не должны ставить обновление сохраняемого (persistent) состояния в зависимость от финализатора или очистителя.** Например, зависимость от финализатора или очистителя освобождения сохраняемой блокировки совместно используемого ресурса, такого как база данных, — верный способ привести всю вашу распределенную систему к краху.

Не поддавайтесь соблазнам методов `System.gc` и `System.runFinalization`. Они могут увеличить вероятность выполнения финализаторов и очистителей, но не гарантируют его. Единственные методы, которые, как заявлялось, гарантируют удаление, — это `System.runFinalizersOnExit` и его близнец `Runtime.runFinalizersOnExit`. Эти методы фатально ошибочны

и много лет как признаны устаревшими и не рекомендованными к употреблению [50].

Еще одна проблема, связанная с финализаторами, состоит в том, что непревзвученное исключение в ходе финализации игнорируется, а финализация этого объекта прекращается [25, 12.6]. Необработанное исключение может оставить объект в поврежденном состоянии. И если другой поток попытается воспользоваться таким “испорченным” объектом, результат может быть непредсказуем. Обычно необработанное исключение завершает поток и выдает распечатку стека, однако в случае финализатора этого не происходит — не выдается даже предупреждение. Очистители этой проблемы не имеют, поскольку библиотека, использующая очиститель, управляет его потоком.

Есть и серьезные проблемы производительности при использовании финализаторов или очистителей. На моей машине время создания простого объекта `AutoCloseable`, его закрытия с помощью `try-c`-ресурсами и очистки с помощью сборщика мусора занимает около 12 нс. Применение вместо этого финализатора увеличивает время до 550 нс. Другими словами, создание и уничтожение объектов с помощью финализаторов примерно в 50 раз медленнее. Главным образом это связано с тем, что финализаторы подавляют эффективный сбор мусора. Очистители сопоставимы по скорости с финализаторами, если вы используете их для очистки всех экземпляров класса (около 500 нс на экземпляр на моей машине), но оказываются намного быстрее, если вы используете их только как подстраховку, о чем будет сказано несколько позже. В этих условиях создание, очистка и уничтожение объекта занимает около 66 нс на моей машине, что означает, что вы платите в пять (а не в пятьдесят) раз больше за страховку, если *не* используете ее.

Финализаторы являются серьезной проблемой безопасности: они открывают ваш класс для атак финализаторов. Идея, лежащая в основе атаки финализатора, проста: если в конструкторе или его эквивалентах при сериализации — методах `readObject` и `readResolve` (глава 12, “Сериализация”) — генерируется исключение, то финализатор вредоносного подкласса может быть запущен для частично построенного объекта. Этот метод завершения может записать ссылку на объект в статическое поле, предотвращая тем самым его утилизацию сборщиком мусора. После того как этот объект (который вообще не должен был существовать) оказывается записанным, очень просто вызывать его произвольные методы. **Генерации исключения в конструкторе должно быть достаточно для предотвращения существования объекта; однако при наличии финализатора это не так.** Такие атаки могут иметь пагубные последствия. Классы `final` имеют иммунитет к атакам финализаторов, поскольку никто не может написать вредоносный подкласс для такого класса. **Для защиты классов, не являющихся финальными, от атак**

финализаторов напишите метод `finalize`, который не выполняет никаких действий.

Так что же вам делать вместо написания финализатора или очистителя для класса, объекты которого инкапсулируют ресурсы, требующие освобождения, например файлы или потоки? Просто **сделайте ваш класс реализующим `AutoCloseable`** и потребуйте от его клиентов вызова метода `close` для каждого экземпляра, когда он больше не нужен (обычно с помощью try-c-ресурсами для гарантии освобождения даже при исключениях (раздел 2.9)). Стоит упомянуть одну деталь: следует отслеживать, не был ли закрыт экземпляр — метод `close` должен записать в некоторое поле, что объект больше не является корректным, а другие методы должны проверять это поле и генерировать исключение `IllegalStateException`, если они вызываются после того, как объект был закрыт.

Так для чего же годятся финализаторы и очистители (если они вообще для чего-то нужны)? У них есть два корректных предназначения. Одно из них — служить в качестве подстраховки на случай, если владелец ресурса не вызовет его метод `close`. Хотя нет никакой гарантии, что очиститель или финализатор будет выполняться незамедлительно (если вообще будет выполнен), все же лучше освободить ресурс позже, чем никогда, если клиент не сделал этого. Если вы планируете писать такой финализатор-подстраховку, хорошо подумайте, стоит ли такая подстраховка цены, которую за нее придется платить. Некоторые классы библиотек Java, например `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor` и `java.sql.Connection`, содержат финализаторы, служащие в качестве подстраховки.

Второе обоснованное применение очистителей касается объектов с *платформозависимыми узлами* (`native peers`). Такой узел — это платформозависимый (не являющийся объектом Java) объект, к которому обычный объект обращается через машинные команды. Поскольку такой узел не является обычным объектом, сборщик мусора о нем не знает, и, соответственно, при утилизации обычного объекта утилизировать платформозависимый объект он не может. Очиститель или финализатор является подходящим механизмом для решения этой задачи при условии, что узел не содержит критических ресурсов. Если же снижение производительности неприемлемо или узел содержит ресурсы, которые необходимо освободить немедленно, класс должен содержать метод `close`, описанный ранее.

Очистители немного сложнее в использовании. Ниже это продемонстрировано на примере простого класса `Room`. Давайте предположим, что объекты `Room` должны быть очищены перед тем как они будут удалены. Класс `Room` реализует `AutoCloseable`; тот факт, что его подстраховка в виде автоматической очистки использует очиститель, — просто деталь реализации. В отличие от финализаторов, очистители не загрязняют открытый API класса:

```
// Класс, реализующий AutoCloseable, с использованием
// очистителя в качестве подстраховки
public class Room implements AutoCloseable
{
    private static final Cleaner cleaner = Cleaner.create();
    // Ресурс, требующий очистки. Не должен ссылаться на Room!
    private static class State implements Runnable
    {
        int numJunkPiles;    // Количество мусора в комнате
        State(int numJunkPiles)
        {
            this.numJunkPiles = numJunkPiles;
        }

        // Вызывается методом close или cleaner
        @Override public void run()
        {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // Состояние комнаты, используется совместно с cleanable
    private final State state;

    // Очистка комнаты, когда она готова для сборщика мусора
    private final Cleaner.Cleanable cleanable;
    public Room(int numJunkPiles)
    {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }
    @Override public void close()
    {
        cleanable.clean();
    }
}
```

Статический вложенный класс `State` содержит ресурсы, которые требуются очистителю для очистки комнаты. В данном случае это просто поле `numJunkPiles`, которое представляет собой количество мусора в комнате. В более реалистичном случае это может быть `final long`, содержащий указатель на платформозависимый объект. `State` реализует `Runnable`, и его метод `run` вызывается не более одного раза, с помощью `Cleanable`, который мы получаем при регистрации экземпляра нашего `State` с нашим очистителем в конструкторе `Room`. Вызов метода `run` будет запущен одним из двух событий: обычно он запускается вызовом метода `close` класса `Room`, вызывающим метод

очистки `Cleanable`. Если клиент не вызывал метод `close` до момента, когда экземпляр `Room` становится пригодным для сборщика мусора, очиститель (надемся) вызовет метод `run` класса `State`.

Важно, чтобы экземпляр `State` не ссылался на его экземпляр `Room`. Если это произойдет, возникнет циклическая ссылка, которая не позволит сборщику мусора освободить экземпляр `Room` (и не допустит автоматической очистки). Таким образом, `State` должен быть *статическим* вложенным классом, потому что нестатические вложенные классы содержат ссылки на охватывающие их экземпляры (раздел 4.10). Аналогично нецелесообразно использовать лямбда-выражения, потому что они могут легко захватить ссылки на охватывающие объекты.

Как мы уже говорили, очиститель `Room` используется только как подстраховка. Если клиенты поместят все экземпляры `Room` в блоки `try-c`-ресурсами, автоматическая очистка никогда не потребуется. Это поведение демонстрирует следующий правильно написанный клиент:

```
public class Adult
{
    public static void main(String[] args)
    {
        try (Room myRoom = new Room(7))
        {
            System.out.println("Goodbye");
        }
    }
}
```

Как и ожидается, запущенная программа `Adult` выводит `Goodbye`, за которым следует `Cleaning room`. Но что если программа не столь хороша и никогда не убирает комнату?

```
public class Teenager
{
    public static void main(String[] args)
    {
        new Room(99);
        System.out.println("Peace out");
    }
}
```

Можно ожидать, что будет выведено `Peace out`, а затем `Cleaning room`, но на моей машине `Cleaning room` никогда не выводится — программа просто завершается. Это непредсказуемость, о которой говорилось выше. Спецификация `Cleaner` гласит: “поведение очистителей во время `System.exit` зависит от конкретной реализации. Нет никакой гарантии относительно того, будут

ли выполнены действия очистителя”. Хотя спецификация этого и не говорит, то же самое верно и для обычного завершения программы. На моей машине было достаточно добавить строку `System.gc()` в метод `main` класса `Teenager`, чтобы заставить его вывести перед выходом из программы `Cleaning room`, но нет никакой гарантии, что вы получите то же поведение на вашей машине.

Итак, не используйте очистители или, в версиях до Java 9, финализаторы, за исключением применения в качестве подстраховки или освобождения некритических машинных ресурсов. Но даже в этом случае остерегайтесь последствий, связанных с неопределенным поведением и производительностью.

2.9. Предпочитайте try-c-ресурсами использованию try-finally

Java-библиотеки включают множество ресурсов, которые должны быть закрыты вручную с помощью вызова метода `close`. Примеры включают `InputStream`, `OutputStream` и `java.sql.Connection`. Закрытие ресурсов часто забывается клиентами, с предсказуемо неприятными последствиями. Хотя многие из этих ресурсов используют финализаторы в качестве подстраховки, на самом деле финализаторы работают не очень хорошо (раздел 2.8).

Исторически инструкция `try-finally` была наилучшим средством гарантии корректного закрытия ресурса даже при генерации исключения или возврате значения:

```
// try-finally - теперь это не наилучшее средство закрытия ресурсов!  
static String firstLineOfFile(String path) throws IOException  
{  
    BufferedReader br = new BufferedReader(new FileReader(path));  
  
    try {  
        return br.readLine();  
    } finally {  
        br.close();  
    }  
}
```

Это может выглядеть неплохо, но при добавлении второго ресурса все становится куда хуже:

```
// try-finally с несколькими ресурсами выглядит уродливо!  
static void copy(String src, String dst) throws IOException  
{
```

```

InputStream in = new FileInputStream(src);

try {
    OutputStream out = new FileOutputStream(dst);

    try {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;

        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    } finally {
        out.close();
    }
} finally {
    in.close();
}
}

```

В это трудно поверить, но даже хорошие программисты в большинстве случаев используют этот жуткий способ. Для начала я ошибся на с. 88 книги *Java Puzzlers* [4], и годами никто этого не замечал. Фактически в 2007 году две трети использований метода `close` в Java-библиотеках были неверными.

Даже корректный код закрытия ресурсов с помощью `try-finally`, как показано в предыдущих двух примерах кода, имеет свои тонкие неприятности. Код в блоке `try` и в блоке `finally` может генерировать исключения. Например, в методе `firstLineOfFile` вызов `readLine` может вызвать исключение из-за сбоя в физическом устройстве и вызов `close` может в результате сбиться по той же причине. В этих обстоятельствах второе исключение полностью уничтожает первое. Первое исключение не записывается в трассировку стека исключения, что может значительно усложнить отладку в реальных системах — ведь для того, чтобы диагностировать проблему, обычно желательно увидеть именно первое исключение. Хотя можно написать код для подавления второго исключения в пользу первого, практически этого никто не делал, потому что для этого требуется слишком много исходного текста.

Все эти проблемы были решены одним махом, когда в Java 7 была введена инструкция `try-c-ресурсами` [25, 14.20.3]. Для использования этой конструкции ресурс должен реализовывать интерфейс `AutoCloseable`, который состоит из единственного ничего не возвращающего метода `close`. Многие классы и интерфейсы в библиотеках Java и библиотеках сторонних производителей теперь реализуют или расширяют `AutoCloseable`. Если вы пишете класс, представляющий ресурс, который должен быть закрыт, класс должен реализовывать `AutoCloseable`.

Вот как выглядит наш первый пример при использовании `try-c-ресурсами`:

```
// try-c-ресурсами - наилучшее средство закрытия ресурсов!  
static String firstLineOfFile(String path) throws IOException  
{  
    try (BufferedReader br = new BufferedReader(  
        new FileReader(path)))  
    {  
        return br.readLine();  
    }  
}
```

А вот как при использовании try-c-ресурсами выглядит наш второй пример:

```
// try-c-ресурсами элегантен и для нескольких ресурсов  
static void copy(String src, String dst) throws IOException  
{  
    try (InputStream in = new FileInputStream(src);  
        OutputStream out = new FileOutputStream(dst))  
    {  
        byte[] buf = new byte[BUFFER_SIZE];  
        int n;  
  
        while ((n = in.read(buf)) >= 0)  
            out.write(buf, 0, n);  
    }  
}
```

Версия с использованием try-c-ресурсами короче и более удобочитаема, чем оригиналы, но они обеспечивают лучшую диагностику. Рассмотрим метод `firstLineOfFile`. Если исключения генерируются как при вызове `readLine`, так и при (невидимом) вызове `close`, то последнее исключение подается в пользу первого. Фактически, для того чтобы сохранить исключение, которое вы хотите видеть, могут быть подавлены несколько других исключений. Эти подавляемые исключения не просто отбрасываются; они выводятся в трассировке стека с указанием, что они были подавлены. Вы можете также получить доступ к ним программно, с помощью метода `getSuppressed`, который был добавлен в `Throwable` в Java 7.

Вы можете добавить в конструкцию try-c-ресурсами часть `catch` так же, как и для обычной конструкции try-finally. Это позволяет обрабатывать исключения без засорения вашего кода еще одним уровнем вложенности. В качестве несколько надуманного примера рассмотрим версию нашего метода `firstLineOfFile`, которая не генерирует исключения, но получает возвращаемое значение по умолчанию для случая, если не удастся открыть файл или выполнить чтение из него:

```
// try-with-resources с конструкцией catch  
static String firstLineOfFile(String path, String defaultVal)
```

```
{
    try (BufferedReader br = new BufferedReader(
        new FileReader(path)))
    {
        return br.readLine();
    }
    catch (IOException e)
    {
        return defaultVal;
    }
}
```

Урок ясен: всегда предпочитайте `try-c-ресурсами` применению `try-finally` при работе с ресурсами, которые должны быть закрыты. Результирующий код получается короче и понятнее, а исключения, которые он генерирует, — более полезными. Оператор `try-c-ресурсами` облегчает написание корректного кода с использованием ресурсов, которые должны быть закрыты, что практически невозможно с помощью `try-finally`.

Методы, общие для всех объектов

Хотя `Object` является конкретным классом, он главным образом предназначен для расширения. Все его `не-final`-методы (`equals`, `hashCode`, `toString`, `clone` и `finalize`) имеют явные *общие контракты*, поскольку предназначены для перекрытия. В ответственность любого класса входит перекрытие этих методов таким образом, чтобы они подчинялись их общим контрактам. Если это будет сделано некорректно, это будет препятствовать другим классам, зависящим от контрактов (например, `HashMap` и `HashSet`), корректно работать с таким классом. В этой главе рассказывается, когда и как следует перекрывать `не-final`-методы `Object`. Метод `finalize` в этой главе опущен, потому что он был рассмотрен в разделе 2.8. Хотя метод `Comparable.compareTo` и не является методом `Object`, он рассматривается в этой главе, потому что имеет аналогичный характер.

3.1. Перекрывая `equals`, соблюдайте общий контракт

Перекрытие метода `equals` кажется простым, но есть много способов сделать это неправильно, а последствия могут оказаться самыми плачевными. Самый простой способ избежать проблем — не перекрывать метод `equals`; в этом случае каждый экземпляр класса равен только самому себе. Это именно то, что нужно делать при выполнении любого из следующих условий.

- **Каждый экземпляр класса уникален по своей природе.** Это верно для классов, таких как `Thread`, которые представляют активные сущности, а не значения. Реализация `equals`, предоставляемая классом `Object`, правильно ведет себя для таких классов.
- **У класса нет необходимости в проверке “логической эквивалентности”.** Например, `java.util.regex.Pattern` может иметь перекрытый метод `equals` для проверки, представляют ли шаблоны в точности одно и

то же регулярное выражение, но проектировщики считают, что клиенты не нуждаются в такой функциональности. В этих условиях идеально подойдет реализация `equals`, унаследованная от `Object`.

- **Суперкласс уже переопределяет `equals`, и поведение суперкласса подходит для данного класса.** Например, большинство реализаций `Set` наследует методы `equals` от `AbstractSet`, реализации `List` — от `AbstractList`, а реализации `Map` — от `AbstractMap`.
- **Класс является закрытым или закрытым на уровне пакета, и вы уверены, что его метод `equals` никогда не будет вызываться.** Если вы не хотите рисковать, можете переопределить метод `equals` для гарантии того, что он не будет случайно вызван:

```
@Override public boolean equals(Object o)
{
    throw new AssertionError(); // Метод никогда не вызывается
}
```

Так когда же имеет смысл перекрывать `equals`? Когда для класса определено понятие *логической эквивалентности* (logical equality), которая не совпадает с тождественностью объектов, а метод `equals` в суперклассе не перекрыт. В общем случае это происходит с *классами значений*. Класс значений (value class) — это класс, который представляет значение, такой как `Integer` или `String`. Программист, сравнивающий ссылки на объекты значений с помощью метода `equals`, скорее всего, желает выяснить, являются ли они логически эквивалентными, а не просто узнать, указывают ли эти ссылки на один и тот же объект. Перекрытие метода `equals` необходимо не только для того, чтобы удовлетворить ожидания программистов, но и позволяет использовать экземпляры класса в качестве ключей отображения или элементов в некотором множестве с предсказуемым, желательным поведением.

Одна из разновидностей классов значений, которым *не* нужно перекрытие метода `equals`, — это класс, использующий управление экземплярами (раздел 2.1), чтобы гарантировать наличие не более одного объекта с каждым значением. Типы перечислений (раздел 6.1) также попадают в эту категорию. Для этих классов логическая эквивалентность представляет собой то же самое, что и тождественность объектов, так что метод `equals` класса `Object` для этих классов функционирует так же, как и логический метод `equals`.

Перекрывая метод `equals`, твердо придерживайтесь его общего контракта. Вот как выглядит этот контракт в спецификации `Object`.

Метод `equals` реализует *отношение эквивалентности*, которое обладает следующими свойствами.

- **Рефлексивность:** для любой ненулевой ссылки на значение x выражение `x.equals(x)` должно возвращать `true`.
- **Симметричность:** для любых ненулевых ссылок на значения x и y выражение `x.equals(y)` должно возвращать `true` тогда и только тогда, когда `y.equals(x)` возвращает `true`.
- **Транзитивность:** для любых ненулевых ссылок на значения x , y и z , если `x.equals(y)` возвращает `true` и `y.equals(z)` возвращает `true`, `x.equals(z)` должно возвращать `true`.
- **Непротиворечивость:** для любых ненулевых ссылок на значения x и y многократные вызовы `x.equals(y)` должны либо постоянно возвращать `true`, либо постоянно возвращать `false` при условии, что никакая информация, используемая в сравнениях `equals`, не изменяется.
- Для любой ненулевой ссылки на значение x выражение `x.equals(null)` должно возвращать `false`.

Если у вас нет склонности к математике, все это может выглядеть страшным, однако игнорировать это нельзя! Если вы нарушите эти условия, то рискуете обнаружить, что ваша программа работает с ошибками или вообще аварийно завершается, а найти источник ошибок при этом крайне сложно. Перефразируя Джона Донна (John Donne), можно сказать: ни один класс — не остров¹. Экземпляры одного класса часто передаются другому классу. Работа многих классов, включая все классы коллекций, зависит от того, подчиняются ли передаваемые им объекты контракту метода `equals`.

Теперь, когда вы знаете, насколько опасно нарушение контракта для метода `equals`, давайте рассмотрим его детальнее. Хорошая новость заключается в том, что вопреки внешнему виду контракт не такой уж и сложный. Как только вы поймете его, будет совсем не сложно его придерживаться.

Что же такое отношение эквивалентности? Грубо говоря, это оператор, который разбивает набор элементов на подмножества, элементы которых считаются равными один другому. Такие подмножества называются *классами эквивалентности*. Чтобы метод `equals` был полезным, все элементы в каждом классе эквивалентности должны быть взаимозаменяемыми с точки зрения пользователя. Теперь давайте рассмотрим поочередно все пять требований.

Рефлексивность. Первое требование просто утверждает, что объект должен быть равен самому себе. Трудно представить себе непреднамеренное нарушение этого требования. Если вы сделаете это, а затем добавите экземпляр вашего класса в коллекцию, то метод `contains` вполне может сообщить вам, что в коллекции нет экземпляра, которой вы только что в нее добавили.

¹ “Нет человека, что был бы сам по себе, как остров...” — Джон Донн. *Взывая на краю.* — *Примеч. пер.*

Симметричность. Второе требование гласит, что любые два объекта должны иметь одно и то же мнение относительно своего равенства или неравенства. В отличие от первого требования представить непреднамеренное нарушение этого требования достаточно легко. Рассмотрим, например, следующий класс, который реализует строку, нечувствительную к регистру букв. Регистр строки сохраняется методом `toString`, но игнорируется сравнением `equals`:

```
// Нарушение симметричности!
public final class CaseInsensitiveString
{
    private final String s;
    public CaseInsensitiveString(String s)
    {
        this.s = Objects.requireNonNull(s);
    }
    // Нарушение симметричности!
    @Override public boolean equals(Object o)
    {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString) o).s);

        if (o instanceof String) // Одностороннее взаимодействие!
            return s.equalsIgnoreCase((String) o);

        return false;
    }
    ... // Остальная часть кода опущена
}
```

Переполненный благими намерениями метод `equals` в этом классе наивно пытается взаимодействовать с обычными строками. Давайте предположим, что у нас есть одна строка без учета регистра букв и одна обычная:

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

Как и ожидалось, `cis.equals(s)` возвращает значение `true`. Проблема заключается в том, что, хотя метод `equals` в `CaseInsensitiveString` знает об обычных строках, метод `equals` в `String` забывает о строках, нечувствительных к регистру. Таким образом, `s.equals(cis)` возвращает значение `false`, что является явным нарушением симметрии. Предположим, что вы поместили в коллекцию строки, нечувствительные к регистру:

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);
```

Что теперь вернет вызов `list.contains(s)`? В текущей реализации OpenJDK он возвращает значение `false`, но это просто специфика данной реализации. В другой реализации точно так же может быть возвращено значение `true` или сгенерировано исключение времени выполнения. **После того как вы нарушили контракт `equals`, вы просто не знаете, как другие объекты будут вести себя при столкновении с вашим.**

Чтобы устранить эту проблему, просто удалите неудачную попытку взаимодействия с классом `String` из метода `equals`. После этого можете рефакторизовать метод в один оператор `return`:

```
@Override public boolean equals(Object o)
{
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

Транзитивность. Третье требование контракта метода `equals` гласит, что если один объект равен второму, а второй объект равен третьему, то и первый объект должен быть равен третьему. Вновь несложно представить себе непреднамеренное нарушение этого требования. Рассмотрим случай подкласса, который добавляет к своему суперклассу новый *компонент-значение*. Другими словами, подкласс добавляет немного информации, оказывающей влияние на процедуру сравнения. Начнем с простого неизменяемого класса, представляющего точку в двумерном пространстве:

```
public class Point
{
    private final int x;
    private final int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    @Override public boolean equals(Object o)
    {
        if (!(o instanceof Point))
            return false;

        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ... // Остальная часть кода опущена
}
```

Предположим, что вы хотите расширить этот класс, добавляя к точке понятие цвета:

```
public class ColorPoint extends Point
{
    private final Color color;
    public ColorPoint(int x, int y, Color color)
    {
        super(x, y);
        this.color = color;
    }
    ... // Остальная часть кода опущена
}
```

Как должен выглядеть метод `equals`? Если вы не будете его трогать, реализация наследуется от `Point` и информация о цвете в сравнении `equals` будет игнорироваться. Хотя это не нарушает контракт `equals`, такое поведение явно неприемлемо. Предположим, что вы пишете метод `equals`, который возвращает значение `true`, только если его аргументом является другая цветная точка того же цвета с тем же положением:

```
// Нарушение симметричности!
@Override public boolean equals(Object o)
{
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Проблема этого метода заключается в том, что можно получить разные результаты, сравнивая обычную точку с цветной и наоборот. Прежняя процедура сравнения игнорирует цвет, а новая всегда возвращает `false` из-за неправильного типа аргумента. Для ясности давайте создадим одну обычную точку и одну цветную:

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

Тогда `p.equals(cp)` возвращает `true`, в то время как `cp.equals(p)` возвращает `false`. Можно попытаться исправить ситуацию, сделав так, чтобы метод `ColorPoint.equals` игнорировал цвет при “смешанном сравнении”:

```
// Нарушение транзитивности!
@Override public boolean equals(Object o)
{
    if (!(o instanceof Point))
        return false;
```

```
// Если o — обычная точка, сравнение не учитывает цвет
if (!(o instanceof ColorPoint))
    return o.equals(this);

// o — объект ColorPoint; выполняется полное сравнение
return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Этот подход обеспечивает симметричность — ценой транзитивности:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Теперь `p1.equals(p2)` и `p2.equals(p3)` возвращают `true`, в то время как `p1.equals(p3)` возвращает `false`, нарушая принцип транзитивности. Два первых сравнения не учитывают цвет, в то время как третье учитывает.

Кроме того, этот подход может привести к бесконечной рекурсии. Предположим, что есть два подкласса `Point`, скажем — `ColorPoint` и `SmellPoint`, и каждый обладает подобным методом `equals`. Тогда вызов `myColorPoint.equals(mySmellPoint)` приведет к генерации исключения `StackOverflowError`.

Так как же решить эту проблему? Оказывается, что это фундаментальная проблема отношения эквивалентности в объектно-ориентированных языках программирования. **Не существует способа расширения инстанцируемого класса с добавлением компонента-значения с сохранением контракта `equals`**, если только вы не готовы отказаться от преимуществ объектно-ориентированной абстракции.

Вы могли слышать, что можно расширить инстанцируемый класс с добавлением компонента-значения с сохранением контракта `equals`, используя тест `getClass` вместо теста `instanceof` в методе `equals`:

```
// Нарушение принципа подстановки Лисков
@Override public boolean equals(Object o)
{
    if (o == null || o.getClass() != getClass())
        return false;

    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

Мы получаем сравнение объектов, только если они имеют один и тот же класс реализации. Это может показаться не так уж плохо, но последствия оказываются неприемлемыми: экземпляр подкласса `Point` все еще является `Point` и должен функционировать в качестве таковой точки, но при принятии

описываемого подхода это не удастся сделать! Давайте предположим, что мы хотим написать метод, который проверяет, находится ли точка на единичной окружности. Вот один из способов, как это можно сделать:

```
// Инициализация множества unitCircle как содержащего
// все Point на единичной окружности
private static final Set<Point> unitCircle = Set.of(
    new Point(1, 0), new Point(0, 1),
    new Point(-1, 0), new Point(0, -1));
public static boolean onUnitCircle(Point p)
{
    return unitCircle.contains(p);
}
```

Хотя это может быть и не самым быстрым способом реализации данной функциональности, он отлично работает. Предположим, что вы расширяете Point некоторым тривиальным способом, который не добавляет компонент-значение, скажем, меняя конструктор для отслеживания того, сколько экземпляров класса было создано:

```
public class CounterPoint extends Point
{
    private static final AtomicInteger counter =
        new AtomicInteger();
    public CounterPoint(int x, int y)
    {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated()
    {
        return counter.get();
    }
}
```

Принцип подстановки Лисков гласит, что любое важное свойство типа должно выполняться и для всех его подтипов, так что любой метод, написанный для типа, должен одинаково хорошо работать и для его подтипов [32]. Это формальное утверждение в применении к нашему коду утверждает, что подкласс Point (такой, как CounterPoint) по-прежнему представляет собой Point и должен действовать, как он. Но предположим, что мы передаем CounterPoint методу onUnitCircle. Если класс Point использует метод equals на основе getClass, метод onUnitCircle возвратит значение false независимо от координат x и y экземпляра CounterPoint. Дело в том, что большинство коллекций, включая HashSet, используемую в методе onUnitCircle, используют для проверки на включение метод equals, а ни

один экземпляр `CounterPoint` не равен ни одному экземпляру `Point`. Если, однако, использовать корректный метод `equals` класса `Point` на основе `instanceof`, то тот же метод `onUnitCircle` отлично работает с экземпляром `CounterPoint`.

Хотя удовлетворительного способа расширения инстанцируемого класса с добавлением компонента-значения нет, есть обходной путь: следовать советам раздела 4.4, “Предпочитайте использовать стандартные функциональные интерфейсы”. Вместо того чтобы `ColorPoint` расширял `Point`, дадим `ColorPoint` закрытое поле `Point` и открытый метод для *представления* (раздел 2.6), который возвращает точку в той же позиции, что и цветная точка:

```
// Добавление компонента-значения без нарушения контракта equals
```

```
public class ColorPoint
{
    private final Point point;
    private final Color color;
    public ColorPoint(int x, int y, Color color)
    {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }
    /**
     * Возвращает представление цветной точки в виде обычной.
     */
    public Point asPoint()
    {
        return point;
    }
    @Override public boolean equals(Object o)
    {
        if (!(o instanceof ColorPoint))
            return false;

        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color);
    }
    ... // Остальная часть кода опущена
}
```

В библиотеках платформы Java имеются некоторые классы, которые расширяют инстанцируемые классы с добавлением компонента-значения. Например, `java.sql.Timestamp` является подклассом класса `java.util.Date` и добавляет поле `nanoseconds`. Реализация метода `equals` в `Timestamp` нарушает симметричность, что может привести к некорректному поведению программы, если объекты `Timestamp` и `Date` использовать в одной коллекции или смешивать как-нибудь иначе. В документации к классу `Timestamp` есть

предупреждение, предохраняющее программиста от смешивания объектов Date и Timestamp. Пока вы не смешиваете их, проблем не возникнет; однако ничто не может помешать вам сделать это, и получающиеся в результате ошибки будут очень трудно отладить. Такое поведение класса Timestamp является некорректным, и имитировать его не следует.

Заметим, что *можно* добавить компонент-значение в подкласс *абстрактного* класса, не нарушая при этом контракта метода equals. Это важно для тех разновидностей иерархий классов, которые получаются при следовании совету из раздела 4.9, “Предпочитайте иерархии классов дескрипторам классов”. Например, у вас могут быть простой абстрактный класс Shape без компонент-значений, а также подклассы Circle, добавляющий поле radius, и Rectangle, добавляющий поля length и width. Описанные проблемы при этом не будут возникать до тех пор, пока не будет возможности создавать экземпляры суперкласса.

Непротиворечивость. Четвертое требование контракта метода equals гласит, что если два объекта эквивалентны, они должны быть эквивалентны всегда, пока один из них (или оба они) не будет изменен. Другими словами, изменяемые объекты могут быть равны различным объектам в различные моменты времени, а неизменяемые объекты — не могут. Когда вы пишете класс, хорошо подумайте, не следует ли сделать его неизменяемым (раздел 4.3). Если вы решите, что так и нужно поступить, позаботьтесь о том, чтобы ваш метод equals обеспечивал это ограничение: одинаковые объекты должны все время оставаться одинаковыми, а разные — соответственно, разными.

Вне зависимости от того, является ли класс неизменяемым, **не делайте метод equals зависимым от ненадежных ресурсов.** Очень трудно соблюдать требование непротиворечивости при нарушении этого запрета. Например, метод equals из java.net.URL основан на сравнении IP-адресов узлов, связанных с этими URL. Перевод имени хоста в IP-адрес может потребовать доступа к сети, и нет гарантии, что с течением времени результат не изменится. Это может привести к тому, что указанный метод equals нарушит контракт equals и на практике возникнут проблемы. К сожалению, такое поведение невозможно изменить из-за требований совместимости. За очень небольшим исключением, методы equals должны выполнять детерминированные расчеты над находящимися в памяти объектами.

Отличие от null. Последнее требование, которое ввиду отсутствия названия я позволил себе назвать “отличие от null” (non-nullity), гласит, что все объекты должны отличаться от null. Хотя трудно себе представить, чтобы вызов o.equals(null) случайно вернул значение true, не так уж трудно представить случайную генерацию исключения NullPointerException. Общий контракт этого не допускает. Во многих классах методы equals включают защиту в виде явной проверки аргумента на равенство null:

```
@Override public boolean equals(Object o)
{
    if (o == null)
        return false;

    ...
}
```

Этот тест является ненужным. Чтобы проверить аргумент на равенство, метод `equals` должен сначала преобразовать аргумент в соответствующий тип так, чтобы можно было вызывать его методы доступа или обращаться к его полям. Прежде чем выполнить преобразование типа, метод должен использовать оператор `instanceof` для проверки, что его аргумент имеет корректный тип:

```
@Override public boolean equals(Object o)
{
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;

    ...
}
```

Если бы эта проверка отсутствовала, а метод `equals` получил аргумент неправильного типа, то в результате было бы сгенерировано исключение `ClassCastException`, нарушающее контракт метода `equals`. Однако оператор `instanceof` возвращает `false`, если его первый операнд равен `null`, независимо от второго операнда [25, 15.20.2]. Таким образом, если был передан `null`, проверка типа возвратит `false`, и, соответственно, нет никакой необходимости делать отдельную проверку на равенство `null`.

Собрав все сказанное вместе, мы получаем рецепт для создания высококачественного метода `equals`.

- 1. Используйте оператор `==` для проверки того, что аргумент является ссылкой на данный объект.** Если это так, возвращайте `true`. Это просто оптимизация производительности, которая может иметь смысл при потенциально дорогостоящем сравнении.
- 2. Используйте оператор `instanceof` для проверки того, что аргумент имеет корректный тип.** Если это не так, возвращайте `false`. Обычно корректный тип — это тип класса, которому принадлежит данный метод. В некоторых случаях это может быть некоторый интерфейс, реализованный этим классом. Если класс реализует интерфейс, который уточняет контракт метода `equals`, то в качестве типа указывайте этот интерфейс: это позволит выполнять сравнение классов, реализующих этот интерфейс. Подобным свойством обладают интерфейсы коллекций, таких как `Set`, `List`, `Map` и `Map.Entry`.

3. **Приводите аргумент к корректному типу.** Поскольку эта операция следует за проверкой `instanceof`, она гарантированно успешна.
4. **Для каждого “важного” поля класса убедитесь, что значение этого поля в аргументе соответствует полю данного объекта.** Если все тесты успешны, возвращайте `true`; в противном случае возвращайте `false`. Если в п. 2 тип определен как интерфейс, вы должны получить доступ к полям аргумента через методы интерфейса; если тип представляет собой класс, вы можете обращаться к его полям непосредственно, в зависимости от их доступности.

Для примитивных полей, тип которых — не `float` и не `double`, для сравнения используйте оператор `==`; для полей, которые представляют собой ссылки на объекты, рекурсивно вызывайте метод `equals`; для полей `float` воспользуйтесь статическим методом `Float.compare(float, float)`, а для полей `double` — `Double.compare(double, double)`. Отдельное рассмотрение полей `float` и `double` необходимо из-за существования `Float.NaN`, `-0.0f` и аналогичных значений типа `double` (см. подробную документацию по `Float.equals` в [25, 15.21.1]). При сравнении полей `float` и `double` с помощью статических методов `Float.equals` и `Double.equals` для каждого сравнения выполняется автоматическая упаковка, что отрицательно сказывается на производительности. В случае полей-массивов применяйте эти рекомендации к каждому элементу. Если каждый элемент в поле-массиве имеет значение, воспользуйтесь одним из методов `Arrays.equals`.

Некоторые ссылки на объекты могут оправданно содержать значение `null`. Чтобы избежать возможности генерации исключения `NullPointerException`, проверяйте такие поля на равенство с использованием статического метода `Objects.equals(Object, Object)`.

Для некоторых классов, таких как рассматривавшийся выше `CaseInsensitiveString`, сравнение полей оказывается более сложным, чем простая проверка равенства. В таком случае вы можете захотеть сохранить поле в некотором *каноническом виде* так, чтобы метод `equals` мог выполнять дешевое точное сравнение канонических значений вместо применения более дорогостоящего нестандартного сравнения. Эта методика лучше всего подходит для неизменяемых классов (раздел 4.3); если объект может изменяться, требуется поддерживать актуальность канонической формы.

На производительность метода `equals` может оказывать влияние порядок сравнения полей. Чтобы добиться наилучшей производительности,

в первую очередь, следует сравнивать те поля, которые будут различны с большей вероятностью, либо те, сравнение которых дешевле, либо, в идеале, и те, и другие. Не следует сравнивать поля, которые не являются частью логического состояния объекта, например такие, как поля блокировок, используемые для синхронизации операций. Не нужно сравнивать *производные поля* (derived fields), значение которых вычисляется на основе “значащих полей” объекта; однако такое сравнение может повысить производительность метода equals. Если значение производного поля равнозначно суммарному описанию объекта в целом, то сравнение подобных полей позволит сэкономить на сравнении фактических данных, если будет выявлено расхождение. Например, предположим, что есть класс Polygon, площадь которого кешируется. Если два многоугольника имеют разные площади, нет смысла сравнивать их ребра и вершины.

Когда вы закончите написание вашего метода equals, задайте себе три вопроса: “Симметричен ли он?”, “Транзитивен?”, “Непротиворечив?” И не просто спросите себя — лучше писать модульные тесты, чтобы убедиться в том, что ответы на эти вопросы положительны (если только вы не использовали AutoValue для генерации метода equals — в этом случае тесты можно безопасно опустить). Если указанные свойства не выполняются, выясните, почему, и соответствующим образом исправьте метод equals. Конечно, ваш метод equals должен также удовлетворять двум другим свойствам (рефлексивности и “не нулевости”), но они обычно заботятся о себе сами.

Ниже показан метод equals упрощенного класса PhoneNumber, построенный согласно приведенным указаниям.

```
// Класс с типичным методом equals
public final class PhoneNumber
{
    private final short areaCode, prefix, lineNum;
    public PhoneNumber(int areaCode, int prefix, int lineNum)
    {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix = rangeCheck(prefix, 999, "prefix");
        this.lineNum = rangeCheck(lineNum, 9999, "line num");
    }
    private static short rangeCheck(int val, int max, String arg)
    {
        if (val < 0 || val > max)
            throw new IllegalArgumentException(arg + ": " + val);

        return (short) val;
    }
}
```

```

@Override public boolean equals(Object o)
{
    if (o == this)
        return true;

    if (!(o instanceof PhoneNumber))
        return false;

    PhoneNumber pn = (PhoneNumber)o;
    return pn.lineNum == lineNum && pn.prefix == prefix
        && pn.areaCode == areaCode;
}
... // Остальная часть кода опущена
}

```

Вот несколько завершающих предостережений.

- **Всегда перекрывайте hashCode при перекрытии equals** (раздел 3.2).
- **Не пытайтесь быть слишком умным.** Если вы просто тестируете поля на равенство, нетрудно придерживаться контракта equals. Если же вы чрезмерно агрессивны в поисках эквивалентности, то легко попасть в беду. Обычно плохая идея — учитывать ту или иную разновидность псевдонимов. Например, класс File не должен пытаться приравнивать символические ссылки, ссылающиеся на тот же самый файл. К счастью, это не так.
- **Не подставляйте другой тип вместо Object в объявлении equals.** Зачастую программисты пишут метод equals, который выглядит, как показано ниже, а затем часами разбираются, почему он не работает должным образом:

```

// Тип параметра должен быть Object!
public boolean equals(MyClass o)
{
    ...
}

```

Проблема заключается в том, что этот метод *не перекрывает* (override) Object.equals, аргумент которого имеет тип Object, но вместо этого *перегружает* (overload) его (раздел 8.4). Неприемлемо предоставлять такой “строго типизированный” метод equals даже в качестве дополнения к обычному, поскольку это может привести к аннотациям Override в подклассах для ложных срабатываний и создать ложное чувство безопасности.

Последовательное использование аннотации Override, показанное на протяжении всего этого раздела, защитит вас от этой ошибки (раздел 6.7).

Данный метод `equals` не скомпилируется, а сообщение об ошибке сообщит вам, в чем именно ошибка:

```
// Не работает и не компилируется
@Override public boolean equals(MyClass o)
{
    ...
}
```

Написание и тестирование методов `equals` и `hashCode` утомительно, а в получающемся коде нет ничего из ряда вон выходящего. Отличной альтернативой написанию и тестированию этих методов вручную является использование каркаса Google с открытым исходным кодом `AutoValue`, который автоматически сгенерирует эти методы для вас на основе всего лишь одной аннотации для класса. В большинстве случаев методы, сгенерированные `AutoValue`, по существу, идентичны тем, которые вы пишете сами.

Интегрированные среды разработки также имеют средства для создания методов `equals` и `hashCode`, но получающийся код оказывается более подробным и менее понятным, чем код, использующий `AutoValue`. Кроме того, изменения в классе автоматически не отслеживаются, а потому код требует тестирования. То есть, имея средство интегрированной среды разработки, предпочтительнее использовать его для генерации методов `equals` и `hashCode`, а не реализовывать их вручную, потому что в отличие от программистов компьютеры не делают ошибок по небрежности.

Таким образом, не перекрывайте метод `equals`, если только вы не вынуждены это делать: в большинстве случаев реализация, унаследованная от `Object`, делает именно то, что вам нужно. Выполняя перекрытие `equals`, убедитесь, что вы сравниваете все значащие поля класса, причем так, что выполняются все пять положений контракта `equals`.

3.2. Всегда при перекрытии `equals` перекрывайте `hashCode`

Вы обязаны перекрывать `hashCode` в каждом классе, перекрывающем `equals`. Если это не сделать, ваш класс будет нарушать общий контракт `hashCode`, что не позволит ему корректно работать с коллекциями, такими как `HashMap` и `HashSet`. Вот как выглядит этот контракт, взятый из спецификации `Object`.

- Во время выполнения приложения при многократном вызове для одного и того же объекта метод `hashCode` должен всегда возвращать одно и то

же целое число при условии, что никакая информация, используемая при сравнении этого объекта с другими методом `equals`, не изменилась. Однако не требуется, чтобы это же значение оставалось тем же при другом выполнении приложения.

- Если два объекта равны согласно результату работы `equals(Object)`, то при вызове для каждого из них метода `hashCode` должны получиться одинаковые целочисленные значения.
- Если метод `equals(Object)` утверждает, что два объекта не равны один другому, это *не* означает, что метод `hashCode` возвратит для них разные числа. Однако программист должен понимать, что генерация разных чисел для неравных объектов может повредить производительность хеш-таблиц.

Главным условием при перекрытии метода `hashCode` является второе: равные объекты должны давать одинаковый хеш-код. Два различных экземпляра с точки зрения метода `equals` могут быть логически эквивалентными, однако для метода `hashCode` класса `Object` оказаться всего лишь двумя объектами, не имеющими между собой ничего общего. Поэтому метод `hashCode`, скорее всего, возвратит для этих объектов два кажущихся случайными числа, а не два одинаковых, как того требует контракт.

В качестве примера рассмотрим попытку использования экземпляров класса `PhoneNumber` из раздела 3.1 в качестве ключей `HashMap`:

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

Здесь вы могли бы ожидать, что вызов `m.get(new PhoneNumber(707, 867, 5309))` вернет "Jenny", но вместо этого он возвращает значение `null`. Обратите внимание, что здесь используются два экземпляра `PhoneNumber`: один — для вставки в `HashMap`, а второй, равный, — для (попытки) выборки. Класс `PhoneNumber`, не перекрывая `hashCode`, приводит к тому, что два равных экземпляра имеют разные хеш-коды, нарушая контракт `hashCode`. Таким образом, метод `get`, скорее всего, будет искать номер телефона в другом блоке, а не в том, в котором он был сохранен методом `put`. Даже если два экземпляра хешированы в один блок, метод `get` почти наверняка будет возвращать значение `null`, потому что `HashMap` оптимизируется с использованием кеширования хеш-кодов, связанных с каждой записью, и не пытается проверять равенство объектов, хеш-коды которых не совпадают.

Решить эту проблему просто — написать корректный метод `hashCode` для `PhoneNumber`. Как такой метод должен выглядеть? Написать плохой метод — тривиально. Например, показанный далее метод является законным, но никогда не должен использоваться:

```
// Наихудшая допустимая реализация
// hashCode — никогда так не делайте!
@Override public int hashCode()
{
    return 42;
}
```

Это допустимый метод, поскольку он гарантирует, что одинаковые объекты имеют *один и тот же* хеш-код. Но он ужасно плохой, потому что каждый объект имеет один и тот же хеш-код. Таким образом, все объекты хранятся в одном и том же блоке, и хеш-таблица превращается в связанный список. Программа, которая должна была бы выполняться за линейное время, вместо этого будет работать квадратичное время. Для больших хеш-таблиц это фактически разница между работоспособностью и неработоспособностью.

Хорошая хеш-функция, как правило, для неравных экземпляров дает неравные хеш-коды. Именно это и подразумевается третьей частью контракта `hashCode`. В идеале хеш-функции следует разбрасывать любые разумные коллекции неравных экземпляров равномерно по всем значениям `int`. Достижение этой цели может быть трудным; к счастью, не слишком трудно достичь неплохого приближения. Вот простой рецепт.

1. Объявите переменную типа `int` с именем `result` и инициализируйте ее хеш-кодом `s` для первого значащего поля вашего объекта, как показано в п. 2, а. (Вспомните из раздела 3.1, что значащее поле означает поле, значение которого влияет на сравнение объектов на равенство.)
2. Для каждого из остальных значащих полей выполняйте следующее.
 - а. Вычислите хеш-код `s` типа `int` для такого поля.
 - Если поле примитивного типа, вычислите `Type.hashCode(f)`, где `Type` — упакованный примитивный класс, соответствующий типу `f`.
 - Если поле представляет собой ссылку на объект, и метод `equals` этого класса сравнивает поля путем рекурсивных вызовов `equals`, рекурсивно вызывайте `hashCode` для поля. Если требуется более сложное сравнение, вычислите “каноническое представление” этого поля и вызовите для него `hashCode`. Если значение поля — `null`, используйте `0` (или некоторую иную константу, но `0` — более традиционное значение).
 - Если поле представляет собой массив, рассматривайте его, как если бы каждый значащий элемент был отдельным полем. То есть вычислите хеш-код для каждого значащего элемента путем рекурсивного применения этих правил и объедините эти значения так, как показано в п. 2, б. Если в массиве нет значащих элементов, используйте

константу, предпочтительнее — не 0. Если все элементы являются значащими, воспользуйтесь `Arrays.hashCode`.

б. Объедините хеш-код `c`, вычисленный в п. 2, а, с `result` следующим образом:

```
result = 31 * result + c;
```

3. Верните `result`.

Закончив написание метода `hashCode`, спросите себя, имеют ли равные экземпляры одинаковые хеш-коды? Напишите модульные тесты для проверки вашей интуиции (если вы использовали `AutoValue` для генерации своих методов `equals` и `hashCode`, можете спокойно опустить эти тесты). Если равные экземпляры имеют разные хеш-коды, выясните, почему это происходит, и исправьте проблему.

Производные поля можно из вычисления хеш-кода исключить. Другими словами, вы можете игнорировать любое поле, значение которого может быть вычислено из полей, включаемых в вычисления. *Необходимо* исключить любые поля, которые не используются в сравнении методом `equals`, иначе вы рискуете нарушить второе положение контракта `hashCode`.

Умножение в п. 2, б делает результат зависящим от порядка полей и дает гораздо лучшую хеш-функцию, если класс имеет несколько аналогичных полей. Например, если опустить умножение из хеш-функции для `String`, все анаграммы будут иметь одинаковые хеш-коды. Значение 31 выбрано потому, что оно является нечетным простым числом. Если бы оно было четным и умножение приводило к переполнению, то происходила бы потеря информации, потому что умножение на 2 эквивалентно сдвигу. Преимущество использования простых чисел менее понятно, но это традиционная практика. Приятным свойством 31 является то, что умножение можно заменить сдвигом и вычитанием для повышения производительности на некоторых архитектурах: $31 * i == (i << 5) - i$. Современные виртуальные машины выполняют оптимизацию такого вида автоматически.

Применим предыдущий рецепт к классу `PhoneNumber`:

```
// Типичный метод hashCode
@Override public int hashCode()
{
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

Поскольку этот метод возвращает результат простого детерминированного вычисления только для трех значимых полей экземпляра `PhoneNumber`, очевидно, что равные экземпляры `PhoneNumber` имеют одинаковые хеш-коды. Этот метод является, по сути, очень хорошей реализацией метода `hashCode` для класса `PhoneNumber` наравне с реализациями в библиотеках платформы Java. Она простая, достаточно быстрая и делает разумную работу по “рассеиванию” различных телефонных номеров в разные хеш-блоки.

Хотя описанный способ дает достаточно хорошие хеш-функции, они не являются идеальными. По качеству они сравнимы с хеш-функциями для типов значений в библиотеках платформы Java и достаточно адекватны для большинства применений. Если у вас есть настоящая потребность в хеш-функции с меньшим количеством коллизий, обратитесь к `com.google.common.hash.Hashing` [15].

Класс `Object` имеет статический метод, который принимает произвольное количество объектов и возвращает для них хеш-код. Этот метод с именем `hash` позволяет писать однострочные методы `hashCode`, качество которых сравнимо с количеством методов, написанных в соответствии с рекомендациями из данного раздела. К сожалению, они работают медленнее, потому что влекут за собой создание массива для передачи переменного количества аргументов, а также упаковку и распаковку, если любой из аргументов имеет примитивный тип. Этот стиль хеш-функции рекомендуется использовать только в ситуациях, когда производительность не является критической. Вот хеш-функция для `PhoneNumber`, написанная с использованием этой техники:

```
// Однострочный метод hashCode с посредственной производительностью
@Override public int hashCode()
{
    return Objects.hash(lineNum, prefix, areaCode);
}
```

Если класс является неизменяемым, а стоимость вычисления хеш-функции имеет значение, вы можете подумать о хранении хеш-кода в самом объекте вместо того, чтобы вычислять его заново каждый раз, когда в нем появится необходимость. Если вы полагаете, что большинство объектов данного типа будут использоваться в качестве ключей хеш-таблицы, то вы должны вычислять соответствующий хеш-код в момент создания соответствующего экземпляра. В противном случае вы можете выбрать *отложенную инициализацию* хеш-кода, выполняющуюся при первом вызове метода `hashCode` (раздел 11.6). Наш класс `PhoneNumber` в таком подходе не нуждается, но давайте просто покажем, как это делается. Обратите внимание, что начальное значение поля `hashCode` (в данном случае — 0) не должно быть хеш-кодом обычного экземпляра:

```

// Метод hashCode с отложенной инициализацией
// и кешированием хеш-кода
private int hashCode; // Автоматически инициализируется
                       // значением 0
@Override public int hashCode()
{
    int result = hashCode;

    if (result == 0)
    {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }

    return result;
}

```

Не пытайтесь исключить значимые поля из вычисления хеш-кода для повышения производительности. В то время как результирующая хеш-функция может работать быстрее, ее низкое качество может существенно ухудшить производительность хеш-таблицы вплоть до ее полной непригодности для использования. В частности, хеш-функция может столкнуться с большой коллекцией экземпляров, которые отличаются в основном именно теми полями, которые вы решили игнорировать. Если такое случится, хеш-функция будет отображать все экземпляры на несколько хеш-кодов, и программа, которая должна выполняться за линейное время, будет в действительности выполняться за квадратичное время.

И это не просто теоретические рассуждения. До Java 2 хеш-функция `String` использовала не более 16 символов, равномерно разбросанных по всей строке, начиная с первого символа. Для больших коллекций иерархических имен, например URL-адресов, эта функция демонстрировала описанное патологическое поведение.

Не предоставляйте подробную спецификацию значения, возвращаемого `hashCode`, так, чтобы клиенты не могли от него зависеть; это обеспечит возможность при необходимости его изменять. Многие классы в Java-библиотеках, такие как `String` и `Integer`, указывают точное значение, возвращаемое их методом `hashCode` как функцией значения экземпляра. Это не хорошая идея, а ошибка, с которой мы теперь вынуждены жить: она затрудняет возможность улучшения хеш-функций в будущих версиях. Если вы оставите детали неописанными, а в хеш-функции будет обнаружен дефект (или найдется лучшая хеш-функция), ее можно будет изменить в следующей версии.

Итак, *необходимо* перекрывать `hashCode` каждый раз, когда выполняется перекрытие `equals`, иначе ваша программа может работать неправильно. Ваш метод `hashCode` должен подчиняться общему контракту, определенному в `Object`, и выполнять разумную работу по назначению неравным экземплярам разных значений хеш-кодов. Достичь этого просто (пусть и слегка утомительно), используя приведенный в данном разделе рецепт. Как уже упоминалось в разделе 3.1, каркас `AutoValue` является прекрасной альтернативой написанию методов `equals` и `hashCode` вручную; кроме того, такая функциональность предоставляется различными интегрированными средами разработки.

3.3. Всегда перекрывайте `toString`

Хотя `Object` и предоставляет реализацию метода `toString`, строка, которую он возвращает, как правило, существенно отличается от той, которую хотел бы видеть пользователь вашего класса. Она состоит из названия класса, за которым следует символ “коммерческое at” (“собака” @) и хеш-код в форме беззнакового шестнадцатеричного числа, например `PhoneNumber@163b91`. Общий контракт метода `toString` гласит, что возвращаемая строка должна быть “лаконичным, но информативным и удобочитаемым представлением объекта”. И пусть даже в принципе можно утверждать, что строка `PhoneNumber@163b91` является лаконичной и легко читаемой, она явно не столь информативна, как, например, `"(707) 867-5309"`. Далее в контракте метода `toString` говорится: “Рекомендуется перекрывать этот метод во всех подклассах”. Действительно хороший совет!

Хотя выполнение данного контракта не столь критично, как контрактов `equals` и `hashCode` (разделы 3.1 и 3.2), **предоставление хорошей реализации метода `toString` делает ваш класс гораздо более удобным в использовании, а использующую его систему — более простой в отладке.** Метод `toString` автоматически вызывается при передаче объекта методам `println`, `printf`, оператору конкатенации строк или в `assert`, или при выводе отладчиком. Даже если вы никогда не вызываете `toString` для объекта, это еще не значит, что этого не могут делать другие. Например, компонент, имеющий ссылку на ваш объект, может использовать строковое представление объекта в журнале сообщений об ошибках. Если вы не перекроете `toString`, сообщение может оказаться бесполезным.

Если вы предоставляете хороший метод `toString` для класса `PhoneNumber`, сгенерировать информативное диагностическое сообщение будет очень просто:

```
System.out.println("Ошибка соединения с " + phoneNumber);
```

Программисты будут создавать такие диагностические сообщения независимо от того, переопределите вы метод `toString` или нет, и, если этого не сделать, понятнее эти сообщения не станут. Преимущества удачной реализации метода `toString` играют роль не только в пределах экземпляров данного класса, но и распространяются на объекты, содержащие ссылки на эти экземпляры (в особенности это касается коллекций: что бы вы предпочли увидеть при выводе отображения — `{Jenny=PhoneNumber@163b91}` или `{Jenny=707-867-5309}`?).

Чтобы представлять интерес на практике, метод `toString` должен возвращать всю полезную информацию, которая содержится в объекте, как это было только что показано в примере с телефонными номерами. Однако такой подход неприемлем для больших объектов или объектов, состояние которых трудно представить в виде строки. В этом случае метод `toString` должен возвращать резюме наподобие "Телефонная книга Урюпинска" или `"Thread[main, 5, main]"`. В идеале полученная строка не должна требовать разъяснений. (Последний пример с `Thread` этому требованию не удовлетворяет.) Особенно раздражает, когда в строку включена не вся значимая информация об объекте, и получают сообщения об ошибках наподобие

```
Assertion failure: ожидалось {abc, 123}, получено {abc, 123}.
```

При реализации метода `toString` следует принять одно важное решение — следует ли указывать формат возвращаемого значения в документации. Рекомендуется делать это для *классов-значений*, таких как номер телефона или матрица. Преимуществом определения формата является то, что он служит в качестве стандартного, недвусмысленного, удобочитаемого представления объекта. Это представление может использоваться для ввода и вывода, а также в хранилищах данных, предназначенных для чтения как машиной, так и человеком, таких как CSV-файлы. Если вы указываете формат, то при этом хорошо бы обеспечить соответствующую статическую фабрику или конструктор, чтобы программисты могли легко выполнять трансляцию объекта в строковое представление и обратно. Этот подход используется во многих классах значений в библиотеках платформы Java, включая `BigInteger`, `BigDecimal` и большинство упакованных примитивных классов.

Недостатком спецификации формата возвращаемого значения `toString` является то, что после его определения вы остаетесь с ним на всю оставшуюся жизнь (в предположении, что ваш класс активно используется). Программисты будут писать код для анализа вашего представления, для его генерации и встраивания в данные длительного хранения. Изменив это представление в будущем, вы нарушите работоспособность их кода и ценность данных, доведя их до слез. Не указывая формат, вы сохраняете гибкость, которая позволит вам

в будущем добавлять информацию или совершенствовать формат в последующих версиях.

Независимо от того, решите ли вы специфицировать формат, следует четко документировать свои намерения. Если вы специфицируете формат, то делайте это очень точно. Например, вот как выглядит метод `toString`, предоставляемый с классом `PhoneNumber` из раздела 3.2:

```
/**
 * Возвращает строковое представление телефонного номера.
 * Строка содержит 12 символов в формате "XXX-YYY-ZZZZ",
 * где XXX – код области, YYY – префикс, а ZZZZ – номер.
 * Каждая из заглавных букв представляет отдельную
 * десятичную цифру.
 *
 * Если любая из трех частей слишком мала, она дополняется
 * ведущими нулями. Например, если значение номера – 123,
 * то последние четыре символа строкового
 * представления имеют вид "0123".
 */
@Override public String toString()
{
    return String.format("%03d-%03d-%04d",
                        areaCode, prefix, lineNum);
}
```

Если вы решите не специфицировать формат, то документирующий комментарий будет иметь вид наподобие следующего:

```
/**
 * Возвращает краткое описание зелья. Точные детали
 * не определены и могут меняться, но следующее
 * описание можно рассматривать как типичное:
 *
 * "[Зелье #9: тип=приворотное, запах=скипидар,
 *   вид=густая темная жидкость]"
 */
@Override public String toString()
{
    ...
}
```

После прочтения этого комментария программисты, пишущие код, зависящий от формата описания, будут сами виноваты, если после изменения формата их код перестанет работать.

Вне зависимости от того, специфицируете вы формат или нет, предоставляйте программный доступ ко всей информации в значении, возвращаемом методом `toString`. Например, класс `PhoneNumber` должен содержать

методы доступа к коду области, префиксу и номеру. Если это не сделано, вы вынуждаете программистов, которым нужна эта информация, делать анализ возвращаемой строки. Помимо того, что вы снижаете производительность приложения и заставляете программистов делать ненужную работу, это чревато ошибками и ведет к созданию ненадежной системы, которая перестанет работать, как только вы поменяете формат. Не предоставив альтернативных методов доступа, вы превращаете формат строки в API de facto, даже если и указываете в документации, что он может быть изменен.

Не имеет смысла писать метод `toString` в статическом вспомогательном классе (раздел 2.4). Не должны вы писать метод `toString` и в большинстве типов перечислений (раздел 6.1), потому что Java предоставляет идеальный метод `toString` для этого случая. Следует, однако, писать метод `toString` для любого абстрактного класса, подклассы которого совместно используют общее строковое представление. Например, методы `toString` большинства реализаций коллекций наследуются от абстрактных классов коллекций.

Средство Google с открытым исходным кодом `AutoValue`, рассматриваемое в разделе 3.1, генерирует метод `toString` вместо вас, как и большинство интегрированных сред разработки. Эти методы очень хорошо рассказывают о содержимом каждого поля, но они не специализированы для *смысла* класса. Так, например, было бы нецелесообразно использовать метод `toString`, автоматически сгенерированный для нашего класса `PhoneNumber` (так как телефонные номера имеют стандартное строковое представление), но для класса зелья он был бы вполне пригоден. С учетом вышесказанного автоматически сгенерированный метод `toString` гораздо предпочтительнее метода, унаследованного от `Object`, который *ничего* не говорит о значения объекта.

Резюме: перекрывайте реализацию `toString` класса `Object` в каждом инстанцируемом классе, который вы пишете, если только суперкласс уже не сделал это вместо вас. Это делает использование классов гораздо более приятным и помогает в отладке. Метод `toString` должен возвращать сжатое, полезное описание объекта в эстетически приятном виде.

3.4. Перекрывайте метод `clone` осторожно

Интерфейс `Cloneable` был задуман как *интерфейс миксина* (`mixin interface`, раздел 4.6) для классов для объявления, что они могут быть клонированы. К сожалению, он не сможет использоваться для этой цели. Его основной недостаток в том, что в нем отсутствует метод `clone`, а метод `clone` класса `Object` является защищенным. Невозможно, не прибегая к рефлексии (раздел 9.9), вызвать `clone` объекта просто потому, что он реализует `Cloneable`.

Даже рефлексивный вызов может завершиться неудачно, потому что нет никакой гарантии, что объект имеет доступный метод `clone`. Несмотря на этот и многие другие недостатки, данный механизм достаточно широко используется, поэтому стоит его понять. В этом разделе рассказывается о том, как реализовать метод `clone` с корректным поведением, обсуждается, когда эта реализация уместна, и представляются альтернативные варианты.

Что же *делает* интерфейс `Cloneable`, который, как оказалось, не имеет методов? Он определяет поведение реализации закрытого метода `clone` в классе `Object`: если класс реализует интерфейс `Cloneable`, то метод `clone` класса `Object` возвратит копию объекта с поочередным копированием всех полей; в противном случае будет сгенерировано исключение `CloneNotSupportedException`. Это совершенно нетипичный способ использования интерфейсов — не из тех, которым следует подражать. Обычно реализация некоторого интерфейса говорит что-то о том, что этот класс может делать для своих клиентов. В случае же с интерфейсом `Cloneable` он просто меняет поведение защищенного метода суперкласса.

Хотя в спецификации об этом и не говорится, **на практике ожидается, что класс, реализующий `Cloneable`, предоставляет надлежащим образом функционирующий открытый метод `clone`**. Для того, чтобы этого добиться, класс и все его суперклассы должны подчиняться сложному, трудно обеспечиваемому и слабо документированному протоколу. Получающийся в результате механизм оказывается хрупким, опасным и *не укладывающимся в рамки языка*: он создает объекты без вызова конструктора.

Общий контракт метода `clone` достаточно свободен. Вот о чем говорится в спецификации `Object`.

Метод создает и возвращает копию данного объекта. Точный смысл слова “копия” может зависеть от класса объекта. Общее намерение таково, чтобы для любого объекта `x` были истинны выражения

```
x.clone() != x
```

и

```
x.clone().getClass() == x.getClass()
```

Однако это требование не является абсолютным. Типичным условием является требование, чтобы

```
x.clone().equals(x)
```

было равно `true`, но и это требование не является безусловным.

По соглашению объект, возвращаемый этим методом, должен быть получен путем вызова `super.clone`. Если класс и все его суперклассы

(за исключением `Object`) подчиняются этому соглашению, то будет выполняться условие

```
x.clone().getClass() == x.getClass().
```

По соглашению объект, возвращаемый этим методом, должен быть независимым от клонируемого объекта. Для достижения этого может быть необходимо модифицировать одно или несколько полей объекта, возвращаемого `super.clone`, перед тем как вернуть его.

Этот механизм отдаленно похож на цепочку конструкторов, с тем отличием, что она не обязательна: если метод `clone` класса возвращает экземпляр, который получается путем вызова *не* `super.clone`, а путем вызова конструктора, то компилятор не будет жаловаться; но если подкласс этого класса вызывает `super.clone`, то результирующий объект будет иметь неверный класс, не давая методу `clone` подкласса работать должным образом. Если класс, который перекрывает `clone`, является окончательным (`final`), это соглашение можно безопасно игнорировать, поскольку подклассов, о которых нужно беспокоиться, нет. Но если окончательный класс имеет метод `clone`, который не вызывает `super.clone`, то для класса нет причин реализовывать `Cloneable`, так как он не полагается на поведение реализации `clone` класса `Object`.

Предположим, вы хотите реализовать `Cloneable` в классе, суперкласс которого предоставляет корректный метод `clone`. Сначала вызывается `super.clone`. Объект, который вы получите, будет полностью функциональной копией оригинала. Любые поля, объявленные в вашем классе, будут иметь значения, идентичные значениям полей оригинала. Если каждое поле содержит примитивное значение или ссылку на неизменяемый объект, возвращенный объект может быть именно тем, что вам нужно, и в этом случае дальнейшая обработка не требуется. Это относится, например, к классу `PhoneNumber` из раздела 3.2, но обратите внимание, что **неизменяемые классы никогда не должны предоставлять метод `clone`**, потому что это будет просто поощрением излишнего копирования. С этой оговоркой вот как будет выглядеть метод `clone` для `PhoneNumber`:

```
// Метод clone для класса без ссылок на изменяемые состояния
@Override public PhoneNumber clone()
{
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError(); // Can't happen
    }
}
```

Для работы этого метода объявление класса `PhoneNumber` придется изменить, чтобы указать, что он реализует `Cloneable`. Хотя метод `clone` класса `Object` возвращает `Object`, данный метод `clone` возвращает объект `PhoneNumber`. Это законно и желательно, потому что Java поддерживает *ковариантные типы возвращаемых значений*. Другими словами, возвращаемый тип перекрывающего метода может быть подклассом возвращаемого типа перекрытого метода. Это устраняет необходимость преобразования типа в клиенте. Мы должны преобразовать перед возвращением результат `super.clone` из `Object` в `PhoneNumber`, но такое преобразование гарантированно завершится успешно.

Вызов `super.clone` содержится в блоке `try-catch`, потому что `Object` объявляет свой метод `clone` как генерирующий исключение `CloneNotSupportedException`, которое является *проверяемым исключением* (`checked exception`). Поскольку `PhoneNumber` реализует `Cloneable`, мы знаем, что вызов `super.clone` будет успешным. Необходимость в этом шаблоне указывает, что исключение `CloneNotSupportedException` должно быть непроверяемым (раздел 10.3).

Если объект содержит поля, которые ссылаются на изменяемые объекты, простая реализация `clone`, показанная ранее, может оказаться неудачной. Например, рассмотрим класс `Stack` из раздела 2.7:

```
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();

        Object result = elements[--size];
        elements[size] = null; // Удаление устаревшей ссылки
        return result;
    }
}
```

```
// Гарантируем место как минимум для еще одного элемента.
private void ensureCapacity()
{
    if (elements.length == size)
        elements = Arrays.copyOf(elements, 2 * size + 1);
}
}
```

Предположим, что вы хотите сделать этот класс клонируемым. Если метод `clone` просто возвращает `super.clone()`, полученный экземпляр `Stack` будет иметь правильное значение в поле `size`, но его поле `elements` будет ссылаться на тот же массив, что и исходный экземпляр `Stack`. Изменение исходного экземпляра уничтожит инварианты клона и наоборот. Вы быстро обнаружите, что ваша программа производит бессмысленные результаты или генерирует исключение `NullPointerException`.

Эта ситуация никогда не может произойти в результате вызова единственного конструктора в классе `Stack`. По сути, метод `clone` функционирует как конструктор; необходимо гарантировать, что он не наносит никакого вреда исходному объекту и что он должным образом устанавливает инварианты клона. Чтобы метод `clone` класса `Stack` работал корректно, он должен копировать внутреннее содержимое стека. Самый простой способ сделать это — рекурсивно вызвать метод `clone` для массива `elements`:

```
// Метод clone для класса со ссылками на изменяемое состояние
@Override public Stack clone()
{
    try
    {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    }
    catch (CloneNotSupportedException e)
    {
        throw new AssertionError();
    }
}
```

Обратите внимание, что мы не должны преобразовывать результат `elements.clone` в `Object[]`. Вызов `clone` для массива возвращает массив, типы времени выполнения и времени компиляции которого идентичны таковым у клонируемого массива. Это предпочтительная идиома дублирования массива. Фактически массивы являются единственным интересным применением `clone`.

Обратите также внимание, что раннее решение не будет работать, если поле `elements` объявить как `final`, поскольку методу `clone` будет запрещено присваивать новое значение полю. Это фундаментальная проблема: подобно сериализации **архитектура Cloneable несовместима с нормальным использованием final-полей, ссылающихся на изменяемые объекты**, за исключением случаев, когда изменяемые объекты могут безопасно совместно использоваться объектом и его клоном. Чтобы сделать класс клонируемым, может потребоваться удалить модификаторы `final` из некоторых полей.

Не всегда достаточно просто рекурсивно вызывать `clone`. Например, предположим, что вы пишете метод `clone` для хеш-таблицы, внутреннее представление которой состоит из массива блоков, каждый из которых ссылается на первую запись связанного списка пар “ключ/значение”. Для повышения производительности класс реализует собственный упрощенный однонаправленный список вместо `java.util.LinkedList`:

```
public class HashTable implements Cloneable
{
    private Entry[] buckets = ...;
    private static class Entry
    {
        final Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next)
        {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
    ... // Остальная часть кода опущена
}
```

Предположим, что вы просто рекурсивно клонируете массив блоков, как мы делали это для `Stack`:

```
// Неверный метод clone - совместное
// использование изменяемого состояния!
@Override public HashTable clone()
{
    try
    {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    }
}
```

```

catch (CloneNotSupportedException e)
{
    throw new AssertionError();
}
}

```

Хотя клон и имеет собственный массив блоков, этот массив ссылается на те же связанные списки, что и оригинал, что может легко привести к недетерминированному поведению и копии, и оригинал. Чтобы устранить эту проблему, вам нужно скопировать связанный список каждого блока. Вот как выглядит один распространенный подход:

```

// Рекурсивный метод clone для класса
// со сложным изменяемым состоянием
public class HashTable implements Cloneable
{
    private Entry[] buckets = ...;
    private static class Entry
    {
        final Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next)
        {
            this.key = key;
            this.value = value;
            this.next = next;
        }
        // Рекурсивное копирование связанного списка данного Entry
        Entry deepCopy()
        {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
        }
    }
    @Override public HashTable clone()
    {
        try
        {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];

            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();

            return result;
        }
    }
}

```

```

    catch (CloneNotSupportedException e)
    {
        throw new AssertionError();
    }
}
... // Остальная часть кода опущена
}

```

Закрытый класс `HashTable.Entry` был расширен для поддержки глубокого копирования. Метод `clone` в `HashTable` выделяет новый массив `buckets` нужного размера и проходит по исходному массиву `buckets`, выполняя глубокое копирование каждого непустого блока. Метод `deepCopy` у `Entry` рекурсивно вызывает сам себя, чтобы скопировать весь связанный список. Хотя эта техника отлично работает для не слишком больших блоков, этот способ не очень хорош для клонирования связанных списков, потому что требует по одному кадру стека для каждого элемента списка. Если список слишком длинный, это может легко привести к переполнению стека. Чтобы предотвратить такую ситуацию, можно заменить рекурсию в `deepCopy` итерацией:

```

// Итеративное копирование связанного списка для данного Entry
Entry deepCopy()
{
    Entry result = new Entry(key, value, next);

    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);

    return result;
}

```

Последний подход к клонированию сложных изменяемых объектов состоит в вызове `super.clone`, установке всех полей получающегося в результате объекта в их первоначальное состояние и в вызове высокоуровневых методов для восстановления состояния исходного объекта. В случае нашего примера с `HashTable` поле `buckets` будет инициализировано новым массивом, и (не показанный) метод `put(key, value)` будет вызван для каждой пары “ключ/значение” в копируемой хеш-таблице. Этот подход обычно дает простой, достаточно элегантный метод `clone`, который работает не так быстро, как метод, непосредственно работающий с внутренним представлением клона. Хотя такой подход очень ясен, он противоречит всей архитектуре `Cloneable`, потому что слепо перезаписывает копирование из поля в поле, которое лежит в основе архитектуры.

Подобно конструктору, метод `clone` никогда не должен вызывать перекрываемый метод для создаваемого клона (раздел 4.5). Если `clone` вызывает

метод, который перекрыт в подклассе, этот метод будет выполняться то того, как подкласс получит возможность исправить свое состояние в клоне, что вполне может привести к повреждению как копии, так и оригинала. Таким образом, метод `put(key, value)` из предыдущего абзаца должен быть объявлен либо как `final`, либо как `private`. (Если это метод `private`, то, предположительно это “вспомогательный метод” для открытого метода, не являющегося `final`.)

Метод `clone` класса `Object` объявлен как генерирующий исключение `CloneNotSupportedException`, но перекрывающие методы не обязаны быть таковыми. **Открытые методы `clone` должны опускать конструкцию `throws`**, поскольку методы, не генерирующие проверяемые исключения, более просты в использовании (раздел 10.3).

У вас есть два варианта при проектировании класса для наследования (раздел 4.5), но какой бы из них вы ни выбрали, класс *не* должен реализовывать `Cloneable`. Вы можете выбрать имитацию поведения `Object` путем реализации корректно функционирующего защищенного метода `clone`, объявленного как генерирующий исключение `CloneNotSupportedException`. Это дает подклассам возможность как реализовывать `Cloneable`, так и не делать этого, как если бы они расширяли `Object` непосредственно. Кроме того, вы можете предпочесть *не* реализовывать работающий метод `clone` и воспрепятствовать его реализации в подклассах, предоставляя следующую вырожденную реализацию `clone`:

```
// Метод clone для расширяемого класса, не поддерживающего
Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException
{
    throw new CloneNotSupportedException();
}
```

Следует отметить еще одну деталь. Если вы пишете класс, безопасный с точки зрения потоков, который реализует `Cloneable`, помните, что его метод `clone` должен быть соответствующим образом синхронизирован, как и любой другой метод (раздел 11.1). Метод `clone` класса `Object` не синхронизирован, так что, даже если его реализация является удовлетворительной, может потребоваться написать синхронизированный метод `clone`, который возвращает `super.clone()`.

Итак, все классы, реализующие `Cloneable`, должны перекрывать метод `clone` как открытый метод, возвращаемым типом которого является сам класс. Этот метод должен сначала вызвать метод `super.clone`, а затем привести в порядок все поля, подлежащие исправлению. Обычно это означает

копирование всех изменяемых объектов, составляющих внутреннюю “глубокую структуру” объекта и замену всех ссылок клона на эти объекты ссылками на их копии. Хотя обычно эти внутренние копии можно получить путем рекурсивного вызова `clone`, такой подход не всегда является наилучшим. Если класс содержит только примитивные поля или ссылки на неизменяемые объекты, то в таком случае, по-видимому, нет полей, нуждающихся в исправлении. Но и из этого правила есть исключения. Например, поле, предоставляющее серийный номер или иной уникальный идентификатор, должно быть исправлено, даже если оно представляет собой примитивное или неизменяемое значение.

Действительно ли нужны все эти сложности? Редко. Если вы расширяете класс, который уже реализует интерфейс `Cloneable`, у вас практически не остается иного выбора, кроме как реализовать правильно работающий метод `clone`. В противном случае обычно лучше воспользоваться некоторыми альтернативными способами копирования объектов. **Лучший подход к копированию объекта состоит в предоставлении копирующего конструктора или фабрики копий.** Копирующий конструктор представляет собой просто конструктор, который получает единственный аргумент, типом которого является класс, содержащий этот конструктор, например:

```
// Копирующий конструктор
public Yum(Yum yum)
{
    ...
};
```

Фабрика копий представляет собой статическую фабрику (раздел 2.1), аналогичную копирующему конструктору:

```
// Фабрика копий
public static Yum newInstance(Yum yum)
{
    ...
};
```

Подход с использованием копирующего конструктора и его варианта со статической фабрикой имеет много преимуществ над `Cloneable/clone`: не полагается на рискованный внеязыковый механизм создания объектов; не требует обеспечения средствами языка соблюдения слабо документированных соглашений; не конфликтует с корректным использованием `final`-полей; не генерирует ненужные проверяемые исключения и не требует преобразования типов.

Кроме того, копирующий конструктор или фабрика может иметь аргумент, тип которого представляет собой интерфейс, реализуемый этим классом. Например, все реализации коллекций общего назначения по соглашению имеют копирующий конструктор с аргументом типа `Collection` или `Map`.

Копирующие конструкторы и фабрики на основе интерфейсов, известные под более точным названием *конструкторы или фабрики преобразований*, позволяют клиенту выбирать вариант реализации копирования вместо того, чтобы принуждать его принять реализацию исходного класса. Например, предположим, что у нас есть объект `HashSet s` и мы хотим скопировать его как экземпляр `TreeSet`. Метод `clone` такой возможности не предоставляет, хотя это легко делается с помощью конструктора преобразования: `new TreeSet<>(s)`.

Учитывая все проблемы, связанные с `Cloneable`, новые интерфейсы не должны его расширять, а новые расширяемые классы не должны его реализовывать. Хотя для окончательных классов реализация `Cloneable` и менее опасна, ее следует рассматривать как средство оптимизации производительности, зарезервированное для тех редких случаев, когда ее применение оправдано (раздел 9.11). Как правило, функциональность копирования лучше всего обеспечивают конструкторы и фабрики. Важным исключением из этого правила являются массивы, которые действительно лучше всего копировать с помощью метода `clone`.

3.5. Подумайте о реализации `Comparable`

В отличие от других рассматривавшихся в этой главе методов метод `compareTo` в классе `Object` не объявлен. Вместо этого он является единственным методом интерфейса `Comparable`. По своим свойствам он похож на метод `equals` класса `Object`, но вместо простой проверки на равенство позволяет выполнять упорядочивающее сравнение, а кроме того, является обобщенным. Реализуя интерфейс `Comparable`, класс указывает, что его экземпляры обладают свойством *естественного упорядочения* (*natural ordering*). Сортировка массива объектов, реализующих интерфейс `Comparable`, очень проста:

```
Arrays.sort(a);
```

Для объектов `Comparable` столь же просто выполняется поиск, вычисляются предельные значения и обеспечивается поддержка автоматически сортируемых коллекций. Например, приведенная далее программа, использующая тот факт, что класс `String` реализует интерфейс `Comparable`, выводит список аргументов, указанных в командной строке, в алфавитном порядке, удаляя при этом дубликаты:

```
public class WordList
{
    public static void main(String[] args)
    {
```

```

Set<String> s = new TreeSet<>();
Collections.addAll(s, args);
System.out.println(s);
}
}

```

Реализуя интерфейс `Comparable`, вы разрешаете вашему классу взаимодействовать со всем обширным набором обобщенных алгоритмов и реализаций коллекций, которые зависят от этого интерфейса. Вы получаете огромное множество возможностей ценой скромных усилий. Практически все классы значений в библиотеках платформы Java реализуют интерфейс `Comparable`. Если вы пишете класс значения с очевидным естественным упорядочением, таким как алфавитное, числовое или хронологическое, вы должны реализовать интерфейс `Comparable`:

```

public interface Comparable<T> {
    int compareTo(T t);
}

```

Общий контракт метода `compareTo` похож на контракт метода `equals`.

Метод выполняет сравнение объекта с указанным и определяет их порядок. Возвращает отрицательное целое число, нуль или положительное целое число, если данный объект меньше, равен или больше указанного объекта. Генерирует исключение `ClassCastException`, если тип указанного объекта не позволяет сравнивать его с текущим.

В последующем описании запись `sgn(expression)` означает функцию, которая возвращает -1 , 0 или 1 , если значение `expression` отрицательное, нулевое или положительное соответственно.

- Разработчик должен гарантировать, что $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ для всех x и y . (Отсюда вытекает, что $x.\text{compareTo}(y)$ должно генерировать исключение тогда и только тогда, когда $y.\text{compareTo}(x)$ генерирует исключение.)
- Разработчик должен также гарантировать, что отношение транзитивно: из $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$ следует $x.\text{compareTo}(z) > 0$.
- Наконец, разработчик должен гарантировать, что из $x.\text{compareTo}(y) == 0$ следует, что $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$ для всех z .
- Крайне рекомендуется (хотя и не требуется), чтобы было справедливым соотношение $(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$. Вообще говоря, любой класс, который реализует интерфейс `Comparable` и

нарушает данное условие, должен явно указать этот факт (рекомендуется использовать текст “примечание: этот класс имеет естественное упорядочение, не согласующееся с equals”).

Не отбрасывайте математическую природу этого контракта. Как и контракт `equals` (раздел 3.1), этот контракт не так сложен, как выглядит. В отличие от метода `equals`, который устанавливает глобальное отношение эквивалентности для всех объектов, `compareTo` не может работать с объектами разных типов. Методу `compareTo`, когда он сталкивается с объектами разных типов, разрешается сгенерировать исключение `ClassCastException`; обычно именно так он и поступает. Контракт *разрешает* сравнения объектов разных типов, что обычно определяется в интерфейсе, реализованном сравниваемыми объектами.

Так же, как класс, нарушающий контракт `hashCode`, может нарушить работу других классов, которые зависят от хеширования, класс, нарушающий контракт `compareTo`, может нарушить работу других классов, которые зависят от сравнения. Классы, которые зависят от сравнения, включают отсортированные коллекции `TreeSet` и `TreeMap` и вспомогательные классы `Collections` и `Arrays`, которые содержат алгоритмы поиска и сортировки.

Рассмотрим условия контракта `compareTo`. Первое условие гласит, что, если вы измените порядок сравнения для двух ссылок на объекты, произойдет вполне ожидаемая вещь: если первый объект меньше второго, то второй должен быть больше первого, если первый объект равен второму, то и второй должен быть равен первому, и наконец, если первый объект больше второго, то второй должен быть меньше первого. Второе условие гласит, что если первый объект больше второго, а второй объект больше третьего, то тогда первый объект должен быть больше третьего. Последнее условие гласит, что объекты, сравнение которых дает равенство, при сравнении с любым третьим объектом должны демонстрировать одинаковый результат.

Из этих трех условий следует, что проверка равенства, осуществляемая с помощью метода `compareTo`, должна подчиняться тем же ограничениям, которые диктуются контрактом метода `equals`: рефлексивность, симметричность и транзитивность. Следовательно, здесь уместно такое же предупреждение: невозможно расширить инстанцируемый класс, вводя новый компонент-значение и сохраняя при этом контракт метода `compareTo`, если только вы не готовы отказаться от преимуществ объектно-ориентированной абстракции (раздел 3.1). Применим и тот же обходной путь. Если вы хотите добавить компонент-значение к классу, реализующему интерфейс `Comparable`, не расширяйте его, а напишите новый независимый класс, содержащий экземпляр первого класса. Затем добавьте метод “представления”, возвращающий содержащийся в нем

экземпляр. Это даст вам возможность реализовать в классе любой метод `compareTo`, который вам нужен, позволяя клиенту рассматривать при необходимости экземпляр содержащего класса как экземпляр содержащегося класса.

Последний абзац контракта `compareTo`, являющийся скорее настоящим предложением, чем настоящим требованием, просто указывает, что проверка равенства, осуществляемая с помощью метода `compareTo`, в общем случае должна давать те же результаты, что и метод `equals`. Если это условие выполняется, считается, что упорядочение, задаваемое методом `compareTo`, *согласуется с equals* (*consistent with equals*). Если же оно нарушается, то упорядочение называется *не согласующимся с equals* (*inconsistent with equals*). Класс, метод `compareTo` которого устанавливает порядок, не согласующийся с `equals`, будет работоспособен, однако отсортированные коллекции, содержащие элементы этого класса, могут не подчиняться общим контрактам соответствующих интерфейсов коллекций (`Collection`, `Set` или `Map`). Дело в том, что общие контракты для этих интерфейсов определяются в терминах метода `equals`, тогда как в отсортированных коллекциях используется проверка равенства, которая основана на методе `compareTo`, а не `equals`. Если это произойдет, катастрофы не случится, но об этом следует помнить.

Например, рассмотрим класс `BigDecimal`, метод `compareTo` которого не согласуется с `equals`. Если вы создадите пустой экземпляр `HashSet` и добавите в него `new BigDecimal("1.0")`, а затем `new BigDecimal("1.00")`, то это множество будет содержать два элемента, поскольку два добавленных в этот набор экземпляра класса `BigDecimal` будут не равны, если сравнивать их с помощью метода `equals`. Однако если выполнить ту же процедуру с `TreeSet`, а не `HashSet`, то полученный набор будет содержать только один элемент, поскольку два упомянутых экземпляра `BigDecimal` будут равны, если сравнивать их с помощью метода `compareTo`. (Подробности — в документации `BigDecimal`.)

Написание метода `compareTo` похоже на написание метода `equals`, но есть и несколько ключевых различий. Поскольку интерфейс `Comparable` параметризован, метод `compareTo` статически типизирован, так что нет необходимости в проверке типа или преобразовании типа его аргумента. Если аргумент имеет неправильный тип, вызов не должен даже компилироваться. Если же аргумент имеет значение `null`, метод `compareTo` должен сгенерировать исключение `NullPointerException` — и именно так и происходит, как только метод пытается обратиться к членам аргумента.

Поля в методе `compareTo` сравниваются для упорядочения, а не для выяснения равенства. Для сравнения полей, представляющих собой ссылки на объекты, метод `compareTo` следует вызывать рекурсивно. Если поле не реализует интерфейс `Comparable` или вам необходимо нестандартное упорядочение,

используйте вместо него `Comparator`. Вы можете написать собственный метод либо воспользоваться уже имеющимся, как это было в методе `compareTo` класса `CaseInsensitiveString` из раздела 3.1:

```
// Comparable с единственным полем, которое
// представляет собой ссылку на объект
public final class CaseInsensitiveString
    implements Comparable<CaseInsensitiveString>
{
    public int compareTo(CaseInsensitiveString cis)
    {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Остальная часть кода опущена
}
```

Обратите внимание, что класс `CaseInsensitiveString` реализует интерфейс `Comparable<CaseInsensitiveString>`. Это означает, что ссылка на `CaseInsensitiveString` может сравниваться только с другой ссылкой на `CaseInsensitiveString`. Это нормальная схема, которой нужно следовать, когда класс объявлен как реализующий интерфейс `Comparable`.

В предыдущих изданиях этой книги рекомендовалось в методе `compareTo` сравнить целочисленные примитивные поля с использованием операторов отношений `<` и `>`, а примитивные поля с плавающей точкой — используя статические методы `Double.compare` и `Float.compare`. В Java 7 ко всем упакованным примитивным классам Java были добавлены методы `compare`. **Применение операторов отношений `<` и `>` в методе `compareTo` рассматривается как многословное и ведущее к ошибкам и больше не рекомендуется.**

Если класс имеет несколько значимых полей, критичным является порядок их сравнения. Вы должны начинать с наиболее значимого поля и затем следовать в порядке убывания значимости. Если сравнение дает ненулевой результат (нулевой результат означает равенство), то все, что вам нужно сделать, — это просто вернуть полученный результат. Если наиболее значимые поля равны, продолжайте сравнивать следующие по значимости поля, и т.д. Если все поля равны, равны и объекты, и в таком случае возвращайте нуль. Этот прием демонстрирует метод `compareTo` класса `PhoneNumber` из раздела 3.2:

```
// Comparable с несколькими примитивными полями
public int compareTo(PhoneNumber pn)
{
    int result = Short.compare(areaCode, pn.areaCode);

    if (result == 0)
    {
        result = Short.compare(prefix, pn.prefix);
    }
}
```

```

        if (result == 0)
            result = Short.compare(lineNum, pn.lineNum);
    }

    return result;
}

```

В Java 8 интерфейс `Comparator` был оснащен набором *методов конструирования компаратора* (`comparator construction methods`), которые позволяют легко создавать компараторы. Эти компараторы могут затем использоваться для реализации метода `compareTo`, как того требует интерфейс `Comparable`. Многие программисты предпочитают лаконичность такого подхода, хотя он дает скромную производительность: сортировка массивов экземпляров `PhoneNumber` на моей машине становится примерно на 10% медленнее. При использовании этого подхода рассмотрите возможность использования *статического импорта* Java, чтобы иметь возможность для ясности и краткости ссылаться на статические методы конструирования компараторов по их простым именам. Вот как выглядит метод `compareTo` для `PhoneNumber` с использованием этого подхода:

```

// Comparable с методами конструирования компаратора
private static final Comparator<PhoneNumber> COMPARATOR =
    comparingInt((PhoneNumber pn) -> pn.areaCode)
    .thenComparingInt(pn -> pn.prefix)
    .thenComparingInt(pn -> pn.lineNum);
public int compareTo(PhoneNumber pn)
{
    return COMPARATOR.compare(this, pn);
}

```

Эта реализация создает компаратор во время инициализации класса с использованием двух методов конструирования компаратора. Первым из них является `comparingInt`. Это статический метод, который принимает *функцию извлечения ключа* (`key extractor function`), которая отображает ссылку на объект на ключ типа `int` и возвращает компаратор, который упорядочивает экземпляры согласно этому ключу. В предыдущем примере `comparingInt` принимает *лямбда-выражение*, которое извлекает код области из `PhoneNumber` и возвращает `Comparator<PhoneNumber>`, который упорядочивает номера телефонов в соответствии с их кодами областей. Обратите внимание, что лямбда-выражение явно указывает свой тип входного параметра (`PhoneNumber pn`). Дело в том, что в этой ситуации вывод типов Java оказывается недостаточно мощным, чтобы вывести тип, так что мы вынуждены помочь ему, чтобы программа скомпилировалась.

Если два телефонных номера имеют один и тот же код области, нам нужно дальнейшее уточнение сравнения, и это именно то, что делает второй метод конструирования компаратора, `thenComparingInt`. Это метод экземпляра `Comparator`, который принимает функцию извлечения ключа типа `int` и возвращает компаратор, который сначала применяет исходный компаратор, а затем использует извлеченный ключ для уточнения в случае возврата равенства первым компаратором. Вы можете собрать воедино столько вызовов `thenComparingInt`, сколько вам нужно, получая в результате *лексикографическое упорядочение*. В приведенном выше примере мы собрали два вызова `thenComparingInt`, что приводит к упорядочению, у которого вторичным ключом является префикс, а третичным — номер. Обратите внимание, что нам *не* нужно указывать тип параметра функции извлечения ключа, передаваемой в `thenComparingInt`: вывод типов Java в этом случае достаточно умен, чтобы разобраться в типах.

Класс `Comparator` имеет полный набор методов конструирования. Есть аналоги для `comparingInt` и `thenComparingInt` для примитивных типов `long` и `double`. Версии для `int` могут также использоваться для более узких целочисленных типов, таких как `short`, как в нашем примере с `PhoneNumber`. Версии для `double` могут также использоваться и для `float`. Это обеспечивает охват всех числовых примитивных типов Java.

Имеются также методы конструирования компараторов для объектов ссылочных типов. Статический метод `comparing` имеет две перегрузки. Одна принимает функцию извлечения ключа и использует естественное упорядочение ключей. Вторая принимает как функцию извлечения ключа, так и компаратор для использования с извлеченными ключами. Существует три перегрузки метода экземпляра `thenComparing`. Одна перегрузка принимает только компаратор и использует его для обеспечения вторичного упорядочения. Вторая перегрузка принимает только функцию извлечения ключа и использует естественное упорядочение ключей в качестве вторичного упорядочения. Последняя перегрузка принимает как функцию извлечения ключа, так и компаратор, используемый для сравнения извлеченных ключей.

Иногда вы можете встретить методы `compareTo` и `compare`, которые опираются на тот факт, что разность между двумя значениями является отрицательной, если первое значение меньше второго, нулю, если два значения равны, и положительной, если первое значение больше. Вот пример:

```
// НЕВЕРНЫЙ компаратор на основе разности: нарушает транзитивность!
static Comparator<Object> hashCodeOrder = new Comparator<>()
{
    public int compare(Object o1, Object o2)
    {
```

```

        return o1.hashCode() - o2.hashCode();
    }
};

```

Не используйте эту методику. Ей грозит опасность целочисленного переполнения и артефактов арифметики с плавающей точкой IEEE 754 [25 — 15.20.1, 15.21.1]. Кроме того, получающиеся в результате методы вряд ли будут значительно быстрее написанных с использованием описанных в этом разделе методик. Используйте либо статический метод `compare`

```

// Компаратор, основанный на методе статического сравнения
static Comparator<Object> hashCodeOrder = new Comparator<>()
{
    public int compare(Object o1, Object o2)
    {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};

```

либо метод конструирования компаратора

```

// Компаратор, основанный на методе конструирования компаратора
static Comparator<Object> hashCodeOrder =
    Comparator.comparingInt(o -> o.hashCode());

```

Реализуя класс значения, которое имеет разумное упорядочение, обеспечьте реализацию классом интерфейса `Comparable`, чтобы его экземпляры можно было легко сортировать, искать и использовать в коллекциях, основанных на сравнениях. При сравнении значений полей в реализациях методов `compareTo` избегайте операторов `<` и `>`. Вместо этого используйте статические методы `compare` упакованных примитивных классов или методы конструирования компараторов в интерфейсе `Comparator`.

Классы и интерфейсы

Классы и интерфейсы занимают в языке программирования Java центральное положение. Они являются фундаментальными элементами абстракции. Язык Java предоставляет множество мощных элементов, которые можно использовать для построения классов и интерфейсов. В данной главе даются рекомендации, которые помогут вам наилучшим образом использовать эти элементы, чтобы ваши классы и интерфейсы были удобными, надежными и гибкими.

4.1. Минимизируйте доступность классов и членов

Единственный чрезвычайно важный фактор, отличающий хорошо спроектированный модуль от неудачного, — степень сокрытия его внутренних данных и других деталей реализации от других модулей. Хорошо спроектированный модуль скрывает все детали реализации, четко отделяя API от реализации. Затем компоненты взаимодействуют один с другим только через свои API и ни один из них не знает, какая именно работа выполняется внутри другого модуля. Представленная концепция, именуемая *сокрытием информации* (information hiding) или *инкапсуляцией* (encapsulation), представляет собой один из фундаментальных принципов разработки программного обеспечения [35].

Сокрытие информации важно по многим причинам, большинство которых связано с тем обстоятельством, что этот механизм эффективно *разделяет* составляющие систему компоненты, позволяя разрабатывать, тестировать, оптимизировать, использовать, изучать и модифицировать их по отдельности. Благодаря этому ускоряется разработка системы, поскольку различные модули могут разрабатываться параллельно. Кроме того, уменьшаются расходы на сопровождение приложения, поскольку каждый модуль можно быстро изучить и отладить или вообще заменить, минимально рискуя навредить другим модулям. Хотя само по себе сокрытие информации не может обеспечить повышения производительности, оно создает условия для эффективного управления

ею: когда разработка системы завершена и ее профилирование показало, работа каких модулей приводит к проблемам производительности (раздел 9.11), можно заняться их оптимизацией, не нарушая правильной работы остальных модулей. Соккрытие информации повышает степень повторного использования программного обеспечения, поскольку каждый отдельно взятый модуль не связан с другими модулями и часто оказывается полезным в иных контекстах, отличных от того, для которого он разрабатывался. Наконец, соккрытие информации уменьшает риски при построении больших систем, так как отдельные модули могут оказаться удачными, даже если в целом система не будет пользоваться успехом.

Язык программирования Java имеет множество функциональных возможностей, способствующих соккрытию информации. Одна из них — механизм *управления доступом* (access control) [25, 6.6], задающий степень *доступности* (accessibility) классов, интерфейсов и членов. Доступность любой сущности определяется тем, в каком месте она была объявлена и какие модификаторы доступа (если таковые имеются) присутствуют в ее объявлении (`private`, `protected` или `public`). Правильное использование этих модификаторов имеет большое значение для соккрытия информации.

Эмпирическое правило выглядит просто: **делайте каждый класс или член как можно более недоступным**. Другими словами, используйте наинизший возможный уровень доступа, при котором обеспечивается корректное функционирование разрабатываемого вами программного обеспечения.

Для классов и интерфейсов верхнего уровня (не являющихся вложенными) имеется лишь два возможных уровня доступа: *доступность в пределах пакета* (`package-private`) и *открытость* (`public`). Если вы объявляете класс или интерфейс верхнего уровня с модификатором `public`, он будет открытым; в противном случае он будет доступен только в пределах пакета. Если класс или интерфейс верхнего уровня можно сделать доступным только в пакете, он должен быть таковым. При этом класс или интерфейс становится частью реализации этого пакета, но не частью его экспортируемого API. Вы можете модифицировать его, заменить или исключить из пакета, не опасаясь нанести вред имеющимся клиентам. Если же вы делаете класс или интерфейс открытым, на вас возлагается обязанность всегда поддерживать его во имя сохранения совместимости.

Если класс или интерфейс верхнего уровня, доступный лишь в пределах пакета, используется только в одном классе, вы должны рассмотреть возможность его превращения в закрытый статический класс, вложенный только в тот класс, в котором он используется (раздел 4.10). Тем самым вы еще сильнее уменьшите его доступность. Однако гораздо важнее уменьшить доступность необоснованно открытого класса, чем класса верхнего уровня, доступного в

пределах пакета: открытый класс является частью API пакета, в то время как класс верхнего уровня, доступный лишь в пакете, уже является частью реализации этого пакета.

Для членов (полей, методов, вложенных классов и вложенных интерфейсов) имеется четыре возможных уровня доступа, перечисленных ниже в порядке увеличения доступности.

- **Закрытый** (`private`) — член доступен лишь в пределах того класса верхнего уровня, в котором он объявлен.
- **Доступный в пределах пакета** (`package-private`) — член доступен из любого класса пакета, в котором он объявлен. Технически это доступ *по умолчанию*, действующий, если не указан никакой модификатор доступа (за исключением членов интерфейсов, открытых по умолчанию).
- **Защищенный** (`protected`) — член доступен из подклассов класса, в котором он объявлен (с небольшими ограничениями [25, 6.6.2]) и из любого класса в пакете, где он был объявлен.
- **Открытый** (`public`) — член доступен отовсюду.

После тщательного проектирования открытого API класса вам следует сделать все остальные члены класса закрытыми. И только если другому классу из того же пакета действительно необходим доступ к такому члену, вы можете убрать модификатор `private` и сделать этот член доступным в пределах пакета. Если вы обнаружите, что таких членов слишком много, пересмотрите дизайн своей системы и попытайтесь найти иной вариант разбиения на классы, при котором они были бы лучше изолированы один от другого. Как уже было сказано, и закрытые члены, и члены, доступные в пределах пакета, являются частью реализации класса и обычно не оказывают воздействия на его экспортируемый API. Тем не менее они могут “просочиться” во внешний API, если класс реализует интерфейс `Serializable` (разделы 12.2 и 12.3).

Если уровень доступа для члена открытого класса меняется с доступного на уровне пакета на защищенный, уровень доступности этого члена резко возрастает. Защищенный член является частью экспортируемого API класса, а потому всегда должен поддерживаться. Кроме того, наличие защищенного члена экспортируемого класса представляет собой открытую передачу деталей реализации (раздел 4.5). Потребность в использовании защищенных членов должна возникать относительно редко.

Имеется ключевое правило, ограничивающее возможности по уменьшению доступности методов. Если какой-либо метод перекрывает метод суперкласса, то в подклассе он не может иметь более ограниченный уровень доступа, чем был в суперклассе [25, 8.4.8.3]. Это необходимо для того, чтобы гарантировать,

что экземпляр подкласса может быть использован везде, где можно использовать экземпляр суперкласса (*принцип подстановки Лисков*, см. раздел 3.1). Если нарушить это правило, то при попытке компиляции подкласса компилятор будет выводить сообщение об ошибке. Частным случаем этого правила является то, что если класс реализует некий интерфейс, то все методы в этом интерфейсе должны быть объявлены в классе как открытые.

Можно соблазниться сделать класс, интерфейс или член более доступным, чем необходимо, чтобы облегчить тестирование кода. Это приемлемо только до определенной степени. Можно сделать закрытый член открытого класса доступным в пределах пакета, чтобы протестировать его, но дальнейшее повышение доступности неприемлемо. Другими словами, нельзя делать класс, интерфейс или член частью экспортируемого API пакета для упрощения тестирования. К счастью, это необязательно в любом случае, поскольку тесты могут запускаться как часть тестируемого пакета, что предоставит доступ к его элементам, доступным на уровне пакета.

Поля экземпляров открытых классов должны быть открытыми очень редко (раздел 4.2). Если поле экземпляра не является `final` или представляет собой ссылку на изменяемый объект, то, делая его открытым, вы упускаете возможность ограничения значений, которые могут храниться в этом поле. Это означает, что вы теряете возможность обеспечить инварианты, включающие это поле. Вы также теряете возможность предпринимать какие-либо действия при изменении этого поля, так что **классы с открытыми изменяемыми полями в общем случае не являются безопасными с точки зрения потоков**. Даже если поле объявлено как `final` и ссылается на неизменяемый объект, делая его `public`, также теряется гибкость, позволяющая переходить к новому внутреннему представлению данных, в котором это поле не существует.

Тот же самый совет применим и к статическим полям — с одним исключением. Вы можете предоставлять константы с помощью полей `public static final` в предположении, что константы образуют неотъемлемую часть абстракции, предоставляемой классом. По соглашению названия таких полей состоят из прописных букв, слова в названии разделены символами подчеркивания (раздел 9.12). Крайне важно, чтобы эти поля содержали либо примитивные значения, либо ссылки на неизменяемые объекты (раздел 4.3). Поле, содержащее ссылку на изменяемый объект, обладает всеми недостатками поля без модификатора `final`: хотя саму ссылку нельзя изменить, объект, на который она ссылается, может быть изменен — с нежелательными последствиями.

Обратите внимание, что массив ненулевой длины всегда является изменяемым, так что **является ошибкой иметь в классе массив, объявленный как `public static final`, или метод доступа, возвращающий такое поле**. Если класс имеет такое поле или метод доступа, клиенты смогут

модифицировать содержимое массива. Это часто становится источником брешей в системе безопасности:

```
// Потенциальная брешь безопасности!
public static final Thing[] VALUES = { ... };
```

Учтите, что многие интегрированные среды разработки генерируют методы доступа, возвращающие ссылки на закрытые поля массивов, что приводит в точности к указанным неприятностям. Есть два способа решения проблемы. Можно сделать открытый массив закрытым и добавить открытый неизменяемый список:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

В качестве альтернативы можно сделать массив закрытым и добавить метод, который возвращает копию закрытого массива:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final Thing[] values()
{
    return PRIVATE_VALUES.clone();
}
```

При выборе метода для использования подумайте, что клиент собирается делать с результатом. Какой тип возвращаемого значения будет для него более удобным? Какое решение приведет к более высокой производительности?

Начиная с Java 9 есть два дополнительных, неявных уровня доступа, введенных как часть *модульной системы*. Модуль представляет собой группу пакетов, как пакет представляет собой группу классов. Модуль может явно экспортировать некоторые из своих пакетов через *объявления экспорта* в своем *объявлении модуля* (которое по соглашению содержится в файле исходного текста с именем `module-info.java`). Открытые и защищенные члены неэкспортируемых пакетов в модуле недоступны извне модуля; объявления экспорта на доступность внутри модуля влияния не оказывают. Модульная система позволяет совместно использовать классы пакетами внутри модуля, не делая их видимыми для всего мира. Открытые и защищенные члены открытых классов в неэкспортируемых пакетах дают два неявных уровня доступа, которые являются внутримодульными аналогами обычных открытых и защищенных уровней. Необходимость такого совместного использования относительно редка и зачастую может быть устранена путем переупорядочения классов внутри пакетов.

В отличие от четырех основных уровней доступа, два модульных уровня в основном носят консультативный характер. Если вы поместите JAR-файл модуля в пути классов (CLASSPATH) вашего приложения вместо пути к модулю,

пакеты в модуле вернуться к своему “немодульному” поведению: все открытые и защищенные члены открытых классов пакетов будут иметь свою нормальную доступность независимо от экспорта пакетов модулем. Единственным местом, где строго обеспечивается работоспособность нововведенных уровней доступа, является сам JDK: неэкспортируемые пакеты Java-библиотек действительно недоступны за пределами их модулей.

Ограниченная полезность модульной защиты доступа для типичного программиста на Java и ее консультативный характер — это еще не все; чтобы воспользоваться ее преимуществами, необходимо сгруппировать свои пакеты в модули, сделать все их зависимости явными в объявлениях модуля, переупорядочить дерево исходных текстов и предпринять специальные действия для обеспечения любого доступа к немодульным пакетам из ваших модулей. Пока слишком рано говорить, достигнут ли модули широкого применения сами по себе, вне JDK. В то же время, похоже, лучше их избегать, если только у вас нет в них острой необходимости.

Итак, следует всегда снижать уровень доступа настолько, насколько это возможно (в пределах разумного). Тщательно разработав минимальный открытый API, вы не должны позволить каким-либо случайным классам, интерфейсам и членам стать частью этого API. За исключением полей `public static final`, выступающих в качестве констант, других открытых полей в открытых классах быть не должно. Убедитесь в том, что объекты, на которые есть ссылки в полях `public static final`, не являются изменяемыми.

4.2. Используйте в открытых классах методы доступа, а не открытые поля

Иногда трудно сопротивляться искушению написать вырожденный класс, служащий единственной цели — сгруппировать поля экземпляров:

```
// Вырожденные классы наподобие данного не должны быть открытыми!
class Point
{
    public double x;
    public double y;
}
```

Поскольку доступ к полям данных таких классов осуществляется непосредственно, они лишены преимуществ *инкапсуляции* (раздел 4.1). Вы не можете изменить представление такого класса, не изменив его API, не можете обеспечить выполнение инвариантов и предпринимать какие-либо дополнительные действия при обращении к полю. С точки зрения программистов, строго

придерживающихся объектно-ориентированного подхода, такой класс в любом случае следует заменить классом с закрытыми полями и открытыми *методами доступа* (getter), а для изменяемых классов — еще и с *методами установки* (setter):

```
// Инкапсуляция данных с помощью методов доступа и установки
class Point
{
    private double x;
    private double y;
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
}
```

В отношении открытых классов борцы за чистоту языка программирования, определенно, правы: **если класс доступен за пределами пакета, следует обеспечить методы доступа**, позволяющие изменять внутреннее представление этого класса. Если открытый класс демонстрирует клиенту свои поля данных, все дальнейшие надежды на изменение этого представления потеряны, поскольку код клиентов открытого класса может быть слишком широко распространенным.

Однако **если класс доступен только в пределах пакета или является закрытым вложенным классом, то никакого ущерба от предоставления доступа к его полям данных не будет** — при условии, что эти поля действительно описывают предоставляемую этим классом абстракцию. По сравнению с методами доступа такой подход создает меньше визуального беспорядка и в объявлении класса, и у клиентов, пользующихся этим классом. И хотя программный код клиента зависит от внутреннего представления класса, он может располагаться лишь в том же пакете, где находится сам этот класс. Когда необходимо поменять внутреннее представление класса, изменения можно произвести так, чтобы за пределами пакета они никого не коснулись. В случае же с закрытым вложенным классом область изменений ограничена еще сильнее — только охватывающим классом.

Некоторые классы в библиотеках платформы Java нарушают данный совет не предоставлять непосредственного доступа к полям открытого класса. Яркими примерами являются классы `Point` и `Dimension` из пакета `java.awt`. Следует не использовать эти классы как пример, а рассматривать их как

предупреждение. Как описано в разделе 9.11, решение раскрыть внутреннее содержание класса `Dimension` стало результатом серьезных проблем с производительностью, которые актуальны и сегодня.

Идею предоставить непосредственный доступ к полям открытого класса нельзя считать хорошей, но она становится менее опасной, если поля являются неизменяемыми. Представление такого класса нельзя изменить без изменения его API, нельзя также выполнять вспомогательные действия при чтении такого поля, но зато можно обеспечить выполнение инвариантов. Например, показанный далее класс гарантирует, что каждый экземпляр представляет корректное значение времени:

```
// Открытый класс с предоставлением доступа
// к неизменяемым полям — под вопросом.
public final class Time
{
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOUR = 60;

    public final int hour;
    public final int minute;

    public Time(int hour, int minute)
    {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);

        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Min: " + minute);

        this.hour = hour;
        this.minute = minute;
    }
    ... // Остальная часть кода опущена
}
```

Итак, открытые классы никогда не должны открывать изменяемые поля. Менее опасно (хотя и спорно) решение открывать неизменяемые поля. Однако иногда бывают ситуации, когда классам, доступным в пределах пакета (или закрытым вложенным классам), желательно раскрытие полей — как изменяемых, так и неизменяемых.

4.3. Минимизируйте изменяемость

Неизменяемый класс — это просто класс, экземпляры которого нельзя изменять. Вся информация, содержащаяся в любом его экземпляре, записывается в момент его создания и остается неизменной в течение всего времени существования объекта. В библиотеках платформы Java имеется целый ряд неизменяемых классов, в том числе `String`, упакованные примитивные классы, а также `BigInteger` и `BigDecimal`. Для этого имеется множество веских причин: по сравнению с изменяемыми классами их проще проектировать, реализовывать и использовать. Они менее подвержены ошибкам и более безопасны.

Чтобы сделать класс неизменяемым, следуйте таким пяти правилам.

1. **Не предоставляйте методы, которые изменяют состояние объекта** (известные как *методы установки* (`mutator`)).
2. **Гарантируйте невозможность расширения класса.** Это предотвратит злоумышленную или по небрежности потерю неизменяемого поведения в подклассе. Предотвращение создания подклассов в общем случае выполняется путем объявления класса как `final`, но имеются и другие способы, которые мы обсудим позже.
3. **Объявите все поля `final`.** Это выразит ваши намерения очевидным способом, поддерживаемым самой системой. Необходимо также гарантировать корректное поведение, когда ссылка на вновь создаваемый экземпляр передается из одного потока в другой без синхронизации, как изложено в обсуждении *модели памяти* [25, 17.5; 13, 16].
4. **Объявите все поля `private`.** Это предупредит обращение клиентов к изменяемым объектам, на которые ссылаются эти поля, и их непосредственное изменение. Хотя технически неизменяемые классы могут иметь поля `public final`, содержащие примитивные значения или ссылки на неизменяемые объекты, это не рекомендуется, поскольку это будет препятствовать изменению внутреннего представления в последующих версиях (разделы 4.1 и 4.2).
5. **Обеспечьте монополярный доступ ко всем изменяемым компонентам.** Если класс имеет любые поля, ссылающиеся на изменяемые объекты, убедитесь, что клиенты класса не смогут получить ссылки на эти объекты. Никогда не инициализируйте такие поля предоставляемыми клиентами ссылками на объекты и не возвращайте эти поля из методов доступа. Делайте *защитные копии* (`defensive copies`) (раздел 8.2) в конструкторах, методах доступа и методах `readObject` (раздел 12.4).

Многие из примеров классов в предыдущих разделах являются неизменяемыми. Одним таким классом является `PhoneNumber` из раздела 3.2, который имеет методы доступа для каждого атрибута, но не имеет соответствующих методов установки. Вот немного более сложный пример:

```
// Неизменяемый класс комплексного числа
public final class Complex
{
    private final double re;
    private final double im;
    public Complex(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public double realPart()
    {
        return re;
    }
    public double imaginaryPart()
    {
        return im;
    }
    public Complex plus(Complex c)
    {
        return new Complex(re + c.re, im + c.im);
    }
    public Complex minus(Complex c)
    {
        return new Complex(re - c.re, im - c.im);
    }
    public Complex times(Complex c)
    {
        return new Complex(re * c.re - im * c.im,
                           re * c.im + im * c.re);
    }
    public Complex dividedBy(Complex c)
    {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                           (im * c.re - re * c.im) / tmp);
    }
    @Override public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof Complex))
            return false;
    }
}
```

```

Complex c = (Complex) o;
// См. в разделе 3.1, почему мы используем compare вместо ==
return Double.compare(c.re, re) == 0
        && Double.compare(c.im, im) == 0;
}
@Override public int hashCode()
{
    return 31 * Double.hashCode(re) + Double.hashCode(im);
}
@Override public String toString()
{
    return "(" + re + " + " + im + "i";
}
}

```

Данный класс представляет *комплексное число* (число с действительной и мнимой частями). Помимо стандартных методов класса `Object`, он реализует методы доступа к действительной и мнимой частям числа, а также четыре основные арифметические операции: сложение, вычитание, умножение и деление. Обратите внимание, что представленные арифметические операции вместо того, чтобы менять данный экземпляр, создают и возвращают новый экземпляр класса `Complex`. Такой подход известен как *функциональный*, потому что метод возвращает результат применения функции к операнду, не изменяя сам операнд. Альтернативой является более распространенный *процедурный*, или *императивный*, подход, при котором метод выполняет для своего операнда некую процедуру, которая меняет его состояние. Обратите внимание, что имена методов являются предложениями (такими, как `plus` (плюс)), а не командами (такими, как `add` (сложить)). Это подчеркивает тот факт, что методы не изменяют значения объектов. Классы `BigInteger` и `BigDecimal` не подчиняются этому соглашению именования, и это привело ко множеству ошибок при их использовании.

Если вы с ним еще не знакомы, функциональный подход может показаться неестественным, однако он создает условия для неизменяемости объектов, что имеет множество преимуществ. **Неизменяемые объекты просты.** Неизменяемый объект может находиться только в одном состоянии — с которым он был создан. Если вы гарантируете, что все конструкторы класса устанавливают инварианты класса, то функциональный подход гарантирует, что данные инварианты будут оставаться действительными всегда, без каких-либо дополнительных усилий как с вашей стороны, так и со стороны программиста, использующего этот класс. Что же касается изменяемого объекта, то он может иметь пространство состояний произвольной сложности. Если в документации не представлено точное описание переходов между состояниями, выполняемыми методами установки, то надежное использование изменяемого класса может оказаться сложным или даже невыполнимым.

Неизменяемые объекты по своей природе безопасны с точки зрения потоков; им не нужна синхронизация. Они не могут быть испорчены только из-за того, что одновременно к ним обращается несколько потоков. Несомненно, это самый простой способ добиться безопасности при работе с потоками. В самом деле, ни один поток не может наблюдать воздействие со стороны другого потока через неизменяемый объект. По этой причине **неизменяемые объекты можно безопасно использовать совместно.** Таким образом, неизменяемые классы должны поощрять клиентов везде, где можно, использовать уже существующие экземпляры. Один из простых приемов, позволяющих достичь этого, — предоставление `public static final` констант для часто используемых значений. Например, в классе `Complex` можно представлять следующие константы:

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

Можно сделать еще один шаг в этом направлении. Неизменяемый класс может предоставлять статические фабрики (раздел 2.1), кеширующие часто запрашиваемые экземпляры, чтобы избежать создания новых экземпляров, если таковые уже имеются. Так поступают все упакованные примитивные классы и `BigInteger`. Такие статические фабрики заставляют клиентов совместно использовать экземпляры вместо создания новых, снижая расходы памяти и расходы по сборке мусора. Выбор статических фабрик вместо открытых конструкторов при проектировании нового класса дает гибкость, необходимую для дальнейшего добавления кеширования без изменения клиентов.

Следствием того факта, что неизменяемые объекты можно свободно предоставлять для совместного доступа, является то, что для них не требуется создавать *защитные копии* (*defensive copies*, раздел 8.2). Фактически вам вообще не надо делать никаких копий, поскольку они всегда будут идентичны оригиналу. Соответственно, для неизменяемого класса вам не нужно предоставлять метод `clone` или *копирующий конструктор* (*copy constructor*, раздел 3.4). Когда платформа Java только появилась, еще не было ясного понимания этого обстоятельства, и потому класс `String` в настоящее время имеет копирующий конструктор (который используется очень редко, если используется вообще (раздел 2.6)).

Можно совместно использовать не только неизменяемые объекты, но и их внутреннее представление. Например, класс `BigInteger` использует внутреннее представление “знак/величина”. Знак числа представлен полем типа `int`, его абсолютная величина — массивом `int`. Метод `negate` создает новый экземпляр `BigInteger` с той же величиной и с противоположным знаком. При этом нет необходимости копировать массив, поскольку вновь

созданный экземпляр `BigInteger` содержит внутри ссылку на тот же самый массив, что и исходный экземпляр.

Неизменяемые объекты образуют крупные строительные блоки для прочих объектов, как изменяемых, так и неизменяемых. Гораздо легче обеспечивать поддержку инвариантов сложного объекта, если вы знаете, что составляющие его объекты не будут внезапно изменены. Частный случай данного принципа состоит в том, что неизменяемые объекты формируют большую схему ключей отображений и элементов множества: вы не должны беспокоиться о том, что значения, однажды записанные в это отображение или множество, вдруг изменятся и это приведет к разрушению инвариантов отображения или множества.

Неизменяемые объекты бесплатно обеспечивают атомарность (раздел 10.8). Их состояние никогда не изменяется, так что нет никакой возможности получить временную несогласованность.

Основным недостатком неизменяемых классов является то, что они требуют отдельный объект для каждого уникального значения. Создание этих объектов может быть дорогостоящим, особенно если они большие. Например, предположим, что у вас значение `BigInteger` из миллиона битов и вы хотите изменить его младший бит:

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

Метод `flipBit` создает новый экземпляр `BigInteger`, также длиной миллион бит, который отличается от исходного только одним битом. Эта операция требует времени и памяти, пропорциональных размеру `BigInteger`. Противоположностью является класс `java.util.BitSet`. Подобно `BigInteger`, `BitSet` представляет последовательности битов произвольной длины, но `BitSet`, в отличие от `BigInteger`, — изменяемый класс. Класс `BitSet` предоставляет метод, который позволяет вам изменить состояние отдельного бита в экземпляре с миллионом битов за константное время:

```
BitSet moby = ...;
moby.flip(0);
```

Проблема производительности усугубляется, когда вы выполняете многошаговую операцию, генерируя на каждом этапе новый объект, а в конце выбрасываете все эти объекты, оставляя только конечный результат. Справиться с этой проблемой можно двумя способами. Прежде всего, можно догадаться, какие многошаговые операции будут требоваться чаще всего, и предоставить их как примитивы. Если многошаговая операция реализована как примитивная, неизменяемый класс уже не обязан на каждом шаге создавать отдельный объект. Внутри неизменяемый класс может быть сколь угодно хитроумно устроен.

Например, у класса `BigInteger` есть изменяемый “класс-компаньон”, доступный только в пределах пакета и используемый для ускорения многошаговых операций, таких как возведение в степень по модулю. По изложенным выше причинам использовать какой изменяемый класс-компаньон гораздо сложнее, чем `BigInteger`. Однако, к счастью, вам это делать не надо. Разработчики класса `BigInteger` уже выполнили вместо вас всю тяжелую работу.

Подход с доступным на уровне пакета изменяемым классом-компаньоном прекрасно работает, если можно заранее предсказать, какие сложные операции клиенты захотят выполнять с вашим неизменяемым классом. Если же нет, то лучший выбор состоит в предоставлении *открытого* изменяемого класса-компаньона. Основной пример этого подхода в библиотеках платформы Java — это класс `String`, изменяемый компаньон которого — класс `StringBuilder` (и его устаревший предшественник `StringBuffer`).

Теперь, когда вы знаете, как сделать неизменяемый класс, и понимаете плюсы и минусы неизменяемости, давайте обсудим несколько альтернативных проектов. Напомним, что для гарантии неизменности класс не должен позволить себе иметь подклассы. Этого можно достичь путем финализации класса, но есть еще одна, более гибкая, альтернатива. Вместо того чтобы делать неизменяемый класс `final`, можно сделать все его конструкторы закрытыми или доступными на уровне пакета и добавить открытую статическую фабрику вместо открытых конструкторов (раздел 2.1). Чтобы не говорить отвлеченно, вот как будет выглядеть класс `Complex` при принятии этого подхода:

```
// Неизменяемый класс со статическими фабриками вместо конструкторов
public class Complex
{
    private final double re;
    private final double im;

    private Complex(double re, double im)
    {
        this.re = re;
        this.im = im;
    }
    public static Complex valueOf(double re, double im)
    {
        return new Complex(re, im);
    }
    ... // Остальная часть кода опущена
}
```

Зачастую этот подход оказывается наилучшей альтернативой. Он наиболее гибкий, потому что позволяет использовать несколько классов реализации, доступных в пределах пакета. Для клиентов, которые находятся за пределами

пакета, неизменяемый класс является, по сути, финальным, потому что невозможно расширить класс, взятый из другого пакета, у которого отсутствует открытый или защищенный конструктор. Помимо того, что такой подход позволяет гибко использовать несколько классов реализации, он обеспечивает возможность настройки производительности класса в последующих выпусках путем совершенствования возможностей статических фабрик по кешированию объектов.

Когда разрабатывались классы `BigInteger` и `BigDecimal`, еще не было общепринято, что неизменяемые классы должны быть, по сути, `final`, так что все их методы могут быть перекрыты. К сожалению, эта ситуация из-за обратной совместимости не подлежит исправлению. Если вы пишете класс, безопасность которого зависит от неизменности аргумента `BigInteger` или `BigDecimal`, получаемого от ненадежного клиента, необходимо убедиться, что аргумент является “реальным” `BigInteger` или `BigDecimal`, а не экземпляром ненадежного подкласса. В последнем случае его следует на всякий случай скопировать в предположении, что он может быть изменяемым (раздел 8.2):

```
public static BigInteger safeInstance(BigInteger val)
{
    return val.getClass() == BigInteger.class ?
        val : new BigInteger(val.toByteArray());
}
```

Список правил для неизменяемых классов в начале этого раздела гласит, что методы не могут изменять объект и что все его поля должны быть финальными. На самом деле эти правила немного сильнее, чем необходимо, и могут быть смягчены для повышения производительности. На самом деле метод не может производить *видимого* *извне* изменения состояния объекта. Однако некоторые неизменяемые классы имеют одно или несколько нефинальных полей, в которых они кешируют результаты дорогостоящих вычислений при первом их вычислении. Если значение запрашивается вновь, возвращается кешированное значение, экономя на стоимости пересчета. Этот трюк работает именно потому, что объект является неизменяемым, что гарантирует повторяемость результата при повторении вычислений.

Например, метод `hashCode` класса `PhoneNumber` (раздел 3.2) вычисляет хеш-код при первом вызове и кеширует его на случай повторных вызовов. Эта методика, известная как *отложенная инициализация* (*lazy initialization*, раздел 11.6), используется также классом `String`.

Подведем итоги сказанному в разделе. Постарайтесь устоять перед желанием написать метод установки для каждого метода доступа. **Классы должны быть неизменяемыми, если только нет очень важной причины, чтобы**

сделать их изменяемыми. Неизменяемые классы обладают множеством преимуществ, а их единственным недостатком являются потенциальные проблемы производительности при определенных обстоятельствах. Вы всегда должны делать объекты с небольшими значениями, такими как `PhoneNumber` или `Complex`, неизменяемыми. (В библиотеке платформы Java есть несколько классов, например `java.util.Date` и `java.awt.Point`, которые должны были бы быть неизменяемыми, но таковыми не являются.) Вы должны серьезно подумать и о том, чтобы сделать неизменяемыми и объекты с большими значениями, такие как `String` и `BigInteger`. Предоставлять открытый изменяемый класс-компаньон для вашего неизменяемого класса можно *только* после того, как вы убедитесь, что это необходимо для достижения удовлетворительной производительности (раздел 9.11).

Существуют классы, для которых неизменяемость является нецелесообразной. Если класс невозможно сделать неизменяемым, ограничьте его изменяемость как можно сильнее. Сокращение числа состояний, в которых может находиться объект, уменьшает вероятность ошибок. Делайте каждое поле окончательным, если только у вас нет веских оснований, чтобы сделать его не таковым. Сочетание совета из этого раздела с советами из раздела 4.1 приводят к тому, что **каждое поле следует объявлять как `private final`**, если только нет веских оснований делать иначе.

Конструкторы должны создавать полностью инициализированные объекты, все инварианты которых должны быть установлены. Не следует предоставлять открытый метод инициализации отдельно от конструктора или статической фабрики, если только у вас нет *очень веских* причин так поступить. Аналогично не следует предоставлять метод “повторной инициализации”, который позволяет повторно использовать объект, как если бы он был построен с иным начальным состоянием. Такие методы, как правило, обеспечивают только небольшое повышение производительности (если вообще его обеспечивают) за счет повышения сложности.

Примером этих принципов служит класс `CountDownLatch`. Он является изменяемым, но его пространство состояний преднамеренно сделано небольшим. Вы создаете экземпляр, однократно его используете, и на этом работа с ним завершается: после того как счетчик достигает нуля, вы не можете использовать объект повторно.

Последнее замечание, которое следует добавить в этом разделе, — о классе `Complex`. Приведенный пример был призван лишь проиллюстрировать неизменяемость. Это не промышленная реализация класса комплексного числа. Она использует стандартные формулы для комплексных умножения и деления без корректного округления и со скудной семантикой для комплексных значений `NaN` и бесконечностей [28, 43, 49].

4.4. Предпочитайте композицию наследованию

Наследование (inheritance) — мощное средство добиться повторного использования кода, но не всегда наилучший инструмент для этой работы. При неправильном применении наследование приводит к появлению ненадежных программ. Наследование можно безопасно использовать внутри пакета, в котором реализация и подкласса, и суперкласса находится под контролем одних и тех же программистов. Столь же безопасно пользоваться наследованием, когда расширяемые классы специально созданы и документированы для последующего расширения (раздел 4.5). Однако наследование обычных конкретных классов за пределы пакета сопряжено с опасностями. Напомним, что в этой книге термин “наследование” (inheritance) используется для обозначения *наследования реализации* (implementation inheritance), когда один класс расширяет другой. Проблемы, обсуждаемые в этом разделе, не касаются *наследования интерфейса* (interface inheritance), когда класс реализует интерфейс или когда один интерфейс расширяет другой.

В отличие от вызова метода, наследование нарушает инкапсуляцию [44]. Иными словами, правильное функционирование подкласса зависит от деталей реализации его суперкласса. Реализация суперкласса может меняться от версии к версии, и, если это происходит, подкласс может перестать корректно работать, даже если его код остается нетронутым. Как следствие подкласс должен изменяться и развиваться вместе со своим суперклассом, если только авторы суперкласса не спроектировали и не документировали его специально для последующего расширения.

Перейдем к конкретному примеру и предположим, что у нас есть программа, использующая класс `HashSet`. Для повышения производительности нам необходимо запрашивать у `HashSet`, сколько элементов было добавлено с момента его создания (не путать с текущим размером, который при удалении элемента уменьшается!). Чтобы обеспечить такую возможность, мы пишем вариант класса `HashSet`, который содержит счетчик количества попыток добавления элемента и предоставляет метод доступа к этому счетчику. В классе `HashSet` есть два метода, с помощью которых можно добавлять элементы: `add` и `addAll`, так что перекроем оба эти метода:

```
// Плохо — ненадлежащее использование наследования!
public class InstrumentedHashSet<E> extends HashSet<E>
{
    // Количество попыток вставки элементов
    private int addCount = 0;

    public InstrumentedHashSet()
    {
```

```

    }
    public InstrumentedHashSet(int initCap, float loadFactor)
    {
        super(initCap, loadFactor);
    }
    @Override public boolean add(E e)
    {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection<? extends E> c)
    {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount()
    {
        return addCount;
    }
}

```

Этот класс выглядит вполне разумно, но не работает. Предположим, мы создаем экземпляр и добавляем три элемента с помощью метода `addAll`. Кстати, обратите внимание, что мы создаем список с помощью статического фабричного метода `List.of`, который был добавлен в Java 9; если вы используете более ранние версии, используйте вместо него `Arrays.asList`:

```

InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("Snap", "Crackle", "Pop"));

```

Можно ожидать, что метод `getAddCount` в этот момент должен вернуть значение 3, но на самом деле он возвращает 6. Что же не так? Внутри класса `HashSet` метод `addAll` реализован через метод `add`, хотя в документации вполне оправданно эта деталь реализации не отражена. Метод `addAll` в классе `InstrumentedHashSet` добавляет к значению поля `addCount` значение 3. Затем с помощью `super.addAll` вызывается реализация `addAll` класса `HashSet`. В свою очередь, она вызывает метод `add`, перекрытый в классе `InstrumentedHashSet`, по одному разу для каждого элемента. Каждый из этих трех вызовов добавляет к значению `addCount` еще единицу, так что в итоге общее увеличение счетчика равно 6: добавление каждого элемента с помощью метода `addAll` засчитывалось два раза.

Можно “исправить” подкласс, отказавшись от перекрытия метода `addAll`. Хотя полученный класс будет работать, правильность его работы зависит от того, что метод `addAll` в классе `HashSet` реализуется через метод `add`. Такое “использование собственного метода” является деталью реализации, и нет

никакой гарантии, что она будет сохранена во всех реализациях платформы Java и не поменяется при переходе от одной версии к другой. Соответственно, полученный класс `InstrumentedHashSet` оказывается ненадежным.

Ненамного лучшим решением будет перекрытие `addAll` как метода, который итерирует переданную коллекцию, вызывая для каждого элемента метод `add`. Да, это может гарантировать правильность результата независимо от того, реализован ли метод `addAll` в классе `HashSet` с использованием метода `add`, поскольку реализация `addAll` в классе `HashSet` больше не применяется. Однако такой прием не решает всех наших проблем. Он подразумевает повторную реализацию методов суперкласса заново. Этот вариант сложен, трудоемок, подвержен ошибкам и может приводить к снижению производительности. К тому же это не всегда возможно, поскольку некоторые методы нельзя реализовать без доступа к закрытым полям, которые недоступны для подкласса.

Еще одна причина ненадежности таких подклассов связана с тем, что в новых версиях суперкласс может обзавестись новыми методами. Предположим, безопасность программы требует, чтобы все элементы, помещаемые в некоторую коллекцию, удовлетворяли определенному предикату. Это можно гарантировать, создав для коллекции подкласс и перекрыв в нем все методы добавления элементов таким образом, чтобы перед добавлением элемента проверялось его соответствие упомянутому предикату. Такая схема работает замечательно — до тех пор, пока в следующей версии суперкласса не появится новый метод, который также может добавлять элемент в коллекцию. Как только это произойдет, станет возможным добавление “незаконных” элементов в экземпляр подкласса простым вызовом нового метода, который не был перекрыт в подклассе. И эта проблема не является чисто теоретической. Когда классы `Hashtable` и `Vector` пересматривались для включения в `Collections Framework`, пришлось заделывать несколько дыр безопасности такого вида.

Обе проблемы связаны с перекрытием методов. Вы можете решить, что расширение класса окажется безопасным, если ограничиться добавлением в класс новых методов и воздержаться от перекрытия уже имеющихся. Хотя расширение такого рода гораздо безопаснее, оно также не исключает риска. Если в очередной версии суперкласс получит новый метод, но вдруг окажется, что в подклассе уже имеется метод с той же сигнатурой, но с другим типом возвращаемого значения, то ваш подкласс перестанет компилироваться [25, 8.4.8.3]. Если же вы создали в подклассе метод с точно такой же сигнатурой, как и у нового метода в суперклассе, то получается, что вы его перекрыли — и вы опять сталкиваетесь с описанными ранее проблемами. Более того, вряд ли ваш метод будет отвечать требованиям, предъявляемым к новому методу в суперклассе, ведь когда вы писали этот метод в подклассе, эти требования даже не были сформулированы.

К счастью, есть способ устранить все описанные проблемы. Вместо того чтобы расширять имеющийся класс, создайте в своем новом классе закрытое поле, которое будет содержать ссылку на экземпляр существующего класса. Такая схема называется *композицией* (composition), поскольку имеющийся класс становится компонентом нового класса. Каждый метод экземпляра в новом классе вызывает соответствующий метод содержащегося в классе экземпляра существующего класса, а затем возвращает полученный результат. Такая технология известна как *передача* (forwarding), а соответствующие методы нового класса носят название *методов передачи* (forwarding methods). Полученный класс будет прочен, как скала: он не будет зависеть от деталей реализации существующего класса. Даже если к имевшемуся ранее классу будут добавлены новые методы, на новый класс это никак не повлияет. В качестве конкретного примера использования технологии композиции и передачи рассмотрим соответствующую замену класса InstrumentedHashSet. Обратите внимание, что реализация разделена на две части: сам класс и повторно используемый *класс передачи* (forwarding class), который содержит только методы передачи и ничего более.

// Класс-оболочка: использует композицию вместо наследования

```
public class InstrumentedSet<E> extends ForwardingSet<E>
{
    private int addCount = 0;
    public InstrumentedSet(Set<E> s)
    {
        super(s);
    }
    @Override public boolean add(E e)
    {
        addCount++;
        return super.add(e);
    }
    @Override public boolean addAll(Collection <? extends E > c)
    {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount()
    {
        return addCount;
    }
}
```

// Повторно используемый класс передачи

```
public class ForwardingSet<E> implements Set<E>
{
    private final Set<E> s;
    public ForwardingSet(Set<E> s)    { this.s = s;          }
```

```

public void clear()                { s.clear();                }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty()          { return s.isEmpty();   }
public int size()                 { return s.size();      }
public Iterator<E> iterator()     { return s.iterator(); }
public boolean add(E e)           { return s.add(e);           }
public boolean remove(Object o)   { return s.remove(o);      }
public boolean containsAll(Collection<?> c)
                                { return s.containsAll(c); }
public boolean addAll(Collection <? extends E > c)
                                { return s.addAll(c);         }
public boolean removeAll(Collection<?> c)
                                { return s.removeAll(c);     }
public boolean retainAll(Collection<?> c)
                                { return s.retainAll(c);     }
public Object[] toArray()         { return s.toArray();     }
public <T> T[] toArray(T[] a)    { return s.toArray(a);     }
@Override public boolean equals(Object o)
                                { return s.equals(o);       }
@Override public int hashCode()   { return s.hashCode();     }
@Override public String toString(){ return s.toString();     }
}

```

Дизайн класса `InstrumentedSet` обеспечен наличием интерфейса `Set`, охватывающим функциональность класса `HashSet`. Помимо надежности, данная конструкция характеризуется чрезвычайной гибкостью. Класс `InstrumentedSet` реализует интерфейс `Set` и имеет единственный конструктор, аргументом которого также имеет тип `Set`. По сути, класс преобразует один `Set` в другой, добавляя необходимую функциональность. В отличие от подхода на основе наследования, который работает только для одного конкретного класса и требует отдельного конструктора для каждого поддерживаемого конструктора суперкласса, класс-оболочка может использоваться для реализации любого `Set` и будет работать с любым ранее существовавшим конструктором:

```

Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));

```

Класс `InstrumentedSet` может использоваться даже для временного оснащения экземпляра, который до этого момента использовался без дополнительной функциональности:

```

static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // В этом методе вместо dogs используется iDogs
}

```

Класс `InstrumentedSet` известен как класс *оболочки* (*wrapper*), поскольку каждый экземпляр `InstrumentedSet` является оболочкой для другого

экземпляра `Set`. Эта технология известна также как проектный шаблон *Декоратор* (Decorator) [12], поскольку класс `InstrumentedSet` “декорирует” множество, добавляя ему новые функциональные возможности. Иногда сочетание композиции и передачи ошибочно называют *делегированием* (delegation). Технически это не делегирование, если только объект-оболочка не передает себя “обернутому” объекту [31, 12].

Недостатков у классов-оболочек немного. Один из них заключается в том, что классы-оболочки не приспособлены для использования в *каркасах с обратным вызовом* (callback framework), где один объект передает другому объекту ссылку на самого себя для последующего вызова. Поскольку обернутый объект не знает о своей оболочке, он передает ссылку на самого себя (`this`), и в результате обратные вызовы минуют оболочку. Эта проблема известна как *проблема самоидентификации* (SELF problem) [31]. Некоторых разработчиков беспокоит влияние методов передачи на производительность системы, а также влияние объектов-оболочек на расход памяти. На практике же ни один из этих факторов не имеет существенного влияния. Писать методы передачи несколько утомительно, однако такой класс пишется для каждого интерфейса только один раз, и такие классы уже могут быть вам предоставлены. Например, `Guava` предоставляет передающие классы для всех интерфейсов коллекций [15].

Наследование уместно только в ситуациях, когда подкласс действительно является *подтипом* (subtype) суперкласса. Иными словами, класс *B* должен расширять класс *A* только тогда, когда между двумя этими классами есть отношение типа “является”. Если вы хотите сделать класс *B* расширением класса *A*, задайте себе вопрос “Действительно ли каждый *B* является *A*?” Если вы не можете с уверенностью ответить на этот вопрос утвердительно, то *B* не должен расширять *A*. Если же ответ отрицательный, то зачастую это означает, что *B* должен просто иметь закрытый от всех экземпляр *A* и предоставлять при этом отличный от него API: *A* не является необходимой частью *B*, это просто деталь его реализации.

В библиотеках платформы `Java` имеется множество очевидных нарушений этого принципа. Например, стек не является вектором, так что класс `Stack` не должен быть расширением класса `Vector`. Точно так же список свойств не является хеш-таблицей, а потому класс `Properties` не должен расширять класс `Hashtable`. В обоих случаях более уместной была бы композиция.

Используя наследование там, где подошла бы композиция, вы без всякой необходимости раскрываете детали реализации. Полученный при этом API привязывает вас к первоначальной реализации, навсегда ограничивая производительность вашего класса. Еще более серьезно то, что, раскрывая внутренние элементы класса, вы позволяете клиентам обращаться к ним непосредственно. Самое меньшее, к чему это может привести — к запутанной

семантике. Например, если `p` ссылается на экземпляр класса `Properties`, то `p.getProperty(key)` может давать результаты, отличные от результатов `p.get(key)`: старый метод учитывает значения по умолчанию, тогда как второй метод, унаследованный от класса `Hashtable`, этого не делает. Еще более серьезная неприятность: непосредственно модифицируя суперкласс, клиент получает возможность разрушать инварианты подкласса. В случае с классом `Properties` разработчики рассчитывали, что в качестве ключей и значений можно будет использовать только строки, однако прямой доступ к базовому классу `Hashtable` позволяет нарушать этот инвариант. Как только указанный инвариант нарушается, пользоваться другими элементами API для класса `Properties` (методами `load` и `store`) становится невозможно. Когда эта проблема была обнаружена, исправлять что-либо было слишком поздно, поскольку появились клиенты, работа которых зависела от возможности применения ключей и значений, не являющихся строками.

Последняя группа вопросов, которые вы должны рассмотреть, прежде чем решиться использовать наследование вместо композиции, — нет ли в API того класса, который вы намереваетесь расширять, каких-либо изъянов? Если они есть, то не волнует ли вас то, что эти изъяны перейдут в API вашего класса? Наследование копирует любые дефекты API суперкласса, тогда как композиция позволяет вам разработать новый API, который эти недостатки скрывает.

Итак, наследование является мощным, но проблематичным инструментом, поскольку нарушает принцип инкапсуляции. Пользоваться им можно лишь в том случае, когда между суперклассом и подклассом есть реальная связь типа и подтипа. Но даже в этом случае применение наследования может сделать программу ненадежной, особенно если подкласс и суперкласс принадлежат к разным пакетам, а сам суперкласс не был изначально предназначен для расширения. Для устранения этой ненадежности вместо наследования используйте композицию и передачу, особенно когда для реализации класса-оболочки есть подходящий интерфейс. Классы-оболочки не только надежнее подклассов, но и обладают большими возможностями.

4.5. Проектируйте и документируйте наследование либо запрещайте его

Раздел 4.4 предупреждает вас об опасности создания подклассов от “чужих” классов, которые не были специально разработаны для наследования и соответствующим образом задокументированы. Так что же означает для класса разработка с целью наследования и документирование как такового?

Требуется точно документировать последствия перекрытия любого метода класса. Иными словами, **класс должен документировать, какие из могущих быть перекрытыми методов он использует сам (self-use)**. Для каждого открытого или защищенного метода документация должна указывать, какие могущие быть перекрытыми методы он вызывает, в какой последовательности, а также каким образом результаты их вызова влияют на дальнейшую работу. (Под *могущими быть перекрытыми* (overridable) методами мы подразумеваем открытые либо защищенные методы, не объявленные как `final`.) В общем случае класс должен документировать все условия, при которых он может вызвать метод, могущий быть перекрытым. Например, вызов может поступить из фоновых потоков или от статических инициализаторов.

Метод, который вызывает методы, могущие быть перекрытыми, содержит описание этих вызовов в конце документирующего комментария. Это описание является отдельным разделом спецификации, именуемым “Требования реализации” и генерируемым дескриптором `@implSpec` в Javadoc. В этом разделе описана внутренняя работа метода. Вот пример, скопированный из спецификации `java.util.AbstractCollection`:

```
public boolean remove(Object o)
```

Удаляет единственный экземпляр указанного элемента из коллекции, если таковой имеется (необязательная операция). Более формально — удаляет элемент `e`, такой, что `Objects.equals(o, e)`, если эта коллекция содержит один или несколько таких элементов. Возвращает `true`, если эта коллекция содержала указанный элемент (или, что эквивалентно, если коллекция изменилась в результате вызова).

Требования реализации. Данная реализация проходит по коллекции в поисках указанного элемента. Если она находит элемент, он удаляется из коллекции с помощью метода итератора `remove`. Обратите внимание, что данная реализация генерирует исключение `UnsupportedOperationException`, если итератор, возвращенный методом `iterator` этой коллекции, не реализует метод `remove`, а коллекция содержит указанный объект.

Эта документация не оставляет никаких сомнений в том, что перекрытие метода `iterator` будет влиять на поведение метода `remove`. Она также описывает точное влияние поведения `Iterator`, возвращенного методом `iterator`, на поведение метода `remove`. Сравните это с ситуацией в разделе 4.4, где программист подкласса `HashSet` просто ничего не мог сказать о влиянии перекрытия метода `add` на поведение метода `addAll`.

Но разве это не нарушает требование, что хорошая документация API должна описать то, *что* делает данный метод, а не то, *как* он это делает? Да, нарушает! Это является печальным следствием того факта, что *наследование нарушает принцип инкапсуляции*. Чтобы документировать класс так, чтобы он мог быть безопасно наследован, необходимо описать детали реализации, которые в противном случае должны были быть оставлены неуказанными.

Дескриптор `@implSpec` был добавлен в Java 8 и активно используется в Java 9. Этот дескриптор должен быть включен по умолчанию, но по состоянию на Java 9 утилита Javadoc по-прежнему игнорирует его, если только вы не передадите в командной строке ключ `-tag "apiNote:a:API Note:"`.

Проектирование для наследования включает в себя больше, чем просто документирование схемы “самоиспользования”. Чтобы позволить программистам писать эффективные подклассы без головной боли, **класс может предоставлять точки входа при внутренней обработке в виде разумно выбранных защищенных методов** (или, в редких случаях, защищенных полей). Например, рассмотрим метод `removeRange` из `java.util.AbstractList`:

```
protected void removeRange(int fromIndex, int toIndex)
```

Удаляет из данного списка все элементы, индексы которых находятся между `fromIndex` (включительно) и `toIndex` (исключительно). Переносит все последующие элементы влево (уменьшая их индексы). Этот вызов сокращает список на $(toIndex - fromIndex)$ элементов. (Если `toIndex == fromIndex`, операция не выполняет никаких действий.)

Этот метод вызывается операцией `clear` данного списка и его подсписков. Перекрытие данного метода может воспользоваться преимуществами внутренностей реализации списка и существенно улучшить производительность операции `clear` для данного списка и его подсписков.

Требования реализации. Данная реализация получает итератор списка, позиционированный перед `fromIndex`, и многократно вызывает `ListIterator.next`, за которым следует `ListIterator.remove`, до тех пор, пока весь диапазон не будет удален. **Примечание: если `ListIterator.remove` требует линейного времени работы, эта реализация требует квадратичного времени работы.**

Параметры

`fromIndex`: Индекс первого удаляемого элемента

`toIndex`: Индекс после последнего удаляемого элемента

Этот метод не представляет интереса для конечных пользователей реализации `List`. Он предоставляется исключительно для того, чтобы облегчить

подклассам предоставление быстрого метода `clear` для подписков. В отсутствие метода `removeRange` подклассам пришлось бы иметь дело с квадратичной производительностью при вызове метода `clear` для подписков или переписывать весь механизм `subList` “с нуля” — задача не из легких!

Так как же решить, какие защищенные члены раскрывать при проектировании класса для наследования? К сожалению, единого рецепта нет. Лучшее, что вы можете сделать, — это подумать, принять лучшее из придуманных решений, а затем протестировать его, написав несколько подклассов. Вы должны раскрывать как можно меньше защищенных членов, поскольку каждый из них представляет собой обязательство по реализации. С другой стороны, вы не должны раскрывать слишком мало, потому что отсутствующий защищенный член может сделать класс практически непригодным для наследования.

Единственный способ протестировать класс, предназначенный для наследования — написать подклассы. Если при написании подкласса вы пропустите критический защищенный член, то попытки написания подкласса сделают это упущение болезненно очевидным. И наоборот — если написано несколько подклассов, и ни один из них не использует некоторый защищенный член, то, вероятнее всего, его следует сделать закрытым. Один или несколько таких подклассов должны быть написаны кем-либо, не являющимся автором суперкласса.

Проектируя для наследования класс, который, вероятно, получит широкое распространение, учтите, что вы *навсегда* задаете схему использования классом самого себя, а также способ реализации, неявно представленный защищенными методами и полями. Такие обязательства могут усложнять или даже делать невозможным дальнейшее улучшение производительности и функциональных возможностей в будущих версиях класса. Следовательно, **необходимо обязательно протестировать класс путем написания подклассов до того, как он будет выпущен.**

Заметим также, что специальная документация, требуемая для организации наследования, засоряет и усложняет обычную документацию, которая предназначена для программистов, создающих экземпляры класса и использующих их методы. Что же касается собственно документации, то лишь немногие инструменты и правила комментирования способны отделить документацию обычного API от информации, которая представляет интерес лишь для программистов, создающих подклассы.

Есть несколько ограничений, которым обязан соответствовать класс, чтобы его наследование стало возможным. **Конструкторы класса не должны вызывать методы, которые могут быть перекрыты** (не важно, непосредственно или косвенно). Нарушение этого правила может привести к некорректной работе программы. Конструктор суперкласса выполняется до конструктора

подкласса, а потому перекрывающий метод из подкласса будет вызываться до выполнения конструктора этого подкласса. И если переопределенный метод зависит от инициализации, которую осуществляет конструктор подкласса, то этот метод будет вести себя совсем не так, как ожидалось. Для ясности приведем пример класса, который нарушает это правило:

```
public class Super
{
    // Ошибка - конструктор вызывает метод,
    // который может быть переопределен
    public Super()
    {
        overrideMe();
    }
    public void overrideMe()
    {
    }
}
```

Вот подкласс, который перекрывает метод `overrideMe`, ошибочно вызываемый единственным конструктором `Super`:

```
public final class Sub extends Super
{
    // Пустое final-поле, устанавливаемое конструктором
    private final Instant instant;
    Sub()
    {
        instant = Instant.now();
    }
    // Перекрывающий метод вызывается конструктором суперкласса
    @Override public void overrideMe()
    {
        System.out.println(instant);
    }
    public static void main(String[] args)
    {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

Можно было бы ожидать, что эта программа выведет `instant` дважды, однако в первый раз она выводит `null`, поскольку метод `overrideMe` вызывается конструктором `Super` до того, как конструктор `Sub` получает возможность инициализировать поле `instant`. Заметим, что данная программа видит поле, объявленное как `final`, в двух разных состояниях! Заметим также, что если бы `overrideMe` вызывал любой метод `instant`, то это приводило бы

к генерации исключения `NullPointerException`, когда конструктор `Super` вызывал бы `overrideMe`. Единственная причина, по которой программа не генерирует исключение `NullPointerException`, как это должно быть, заключается в том, что метод `println` умеет работать с нулевыми аргументами.

Обратите внимание, что в конструкторе *можно* безопасно вызывать закрытые, окончательные и статические методы, ни один из которых не может быть перекрыт.

Интерфейсы `Cloneable` и `Serializable` при проектировании для наследования создают особые трудности. В общем случае реализация любого из этих интерфейсов в классах, предназначенных для наследования, — не очень хорошая идея, потому что они создают большие сложности для программистов, расширяющих этот класс. Однако имеются специальные действия, которые можно предпринять для того, чтобы позволить подклассам реализовать эти интерфейсы без обязанности делать это. Эти действия описаны в разделах 3.4 и 12.2.

Если вы решите реализовать интерфейс `Cloneable` или `Serializable` в классе, предназначенном для наследования, то учтите, что, поскольку методы `clone` и `readObject` ведут себя почти так же, как конструкторы, к ним применимо то же самое ограничение: **ни методу `clone`, ни методу `readObject` не разрешается вызывать методы, которые могут быть перекрыты, ни непосредственно, ни косвенно.** В случае метода `readObject` перекрытый метод будет выполняться до десериализации состояния подкласса. Что касается метода `clone`, то перекрытый метод будет выполняться до того, как метод `clone` подкласса получит возможность исправить состояние клона. В любом случае, скорее всего, последует сбой программы. При работе с методом `clone` такой сбой может нанести повреждения и клонируемому объекту, и клону. Это может случиться, например, если перекрывающий метод предполагает, что он изменяет копию глубокой структуры объекта в клоне, но само копирование еще не было выполнено.

Наконец, если вы решили реализовать интерфейс `Serializable` в классе, предназначенном для наследования, и у этого класса есть метод `readResolve` или `writeReplace`, то вы должны сделать эти методы не закрытыми, а защищенными. Если эти методы будут закрытыми, то подклассы будут молча их игнорировать. Это еще один случай, когда для обеспечения наследования детали реализации класса становятся частью его API.

Теперь должно быть очевидно, что **проектирование класса для наследования требует больших усилий и накладывает на класс существенные ограничения.** Это не то решение, которое может быть с легкостью принято. Есть ряд ситуаций, в которых это очевидная необходимость, например когда речь идет об абстрактных классах, содержащих *скелетную реализацию*

интерфейсов (раздел 4.6). В других ситуациях очевидно, что этого делать нельзя, например в случае с неизменяемыми классами (раздел 4.3).

А что можно сказать об обычных неабстрактных классах? Традиционно они не являются ни окончательными, ни предназначенными и документированными для наследования, но подобное положение дел опасно. Каждый раз, когда в такой класс вносится изменение, существует вероятность того, что перестанут работать подклассы, которые расширяют этот класс. Это не просто теоретическая проблема. Нередко сообщения об ошибках в подклассах возникают после того, как в конкретном классе, не являющемся *final*, но не разработанном и документированном для наследования, поменялось внутреннее содержимое.

Лучшим решением этой проблемы является запрет создания подклассов для тех классов, которые не были специально разработаны и документированы для безопасного выполнения этой операции. Запретить создание подклассов можно двумя способами. Более простой заключается в объявлении класса как окончательного (*final*). Другой подход заключается в том, чтобы сделать все конструкторы класса закрытыми или доступными лишь в пределах пакета, а вместо них создать открытые статические фабрики. Такая альтернатива, дающая возможность гибкого внутреннего использования подклассов, рассмотрена в разделе 4.3. Приемлем любой из этих подходов.

Возможно, этот совет несколько спорен, так как многие программисты выросли с привычкой создавать подклассы обычного конкретного класса просто для добавления новых возможностей, таких как средства контроля, оповещения и синхронизации, либо для ограничения функциональных возможностей. Если класс реализует некий интерфейс, в котором отражена его сущность, например *Set*, *List* или *Map*, то у вас не должно быть сомнений по поводу запрета создания подклассов. Проектный шаблон *класса-оболочки* (*wrapper class*), описанный в разделе 4.4, предлагает превосходную альтернативу наследованию, используемому лишь для изменения функциональности.

Если конкретный класс не реализует стандартный интерфейс, то, запретив наследование, вы можете создать неудобство для некоторых программистов. Если вы чувствуете, что должны разрешить наследование для этого класса, то один из возможных подходов заключается в следующем: необходимо убедиться, что этот класс не использует каких-либо собственных методов, которые могут быть перекрыты, и отразить этот факт в документации. Иначе говоря, полностью исключите использование перекрываемых методов самим классом. Сделав это, вы создадите класс, достаточно безопасный для создания подклассов, поскольку перекрытие метода не будет влиять на работу других методов класса.

Вы можете механически исключить использование классом собственных методов, которые могут быть перекрыты, без изменения его поведения. Переместите тело каждого метода, который может быть перекрыт, в закрытый

“вспомогательный метод”, а затем каждый перекрываемый метод должен вызывать собственный закрытый вспомогательный метод. Наконец, каждый вызов перекрываемого метода внутри класса замените прямым вызовом соответствующего закрытого вспомогательного метода.

Из всего сказанного можно сделать вывод: разработка класса для наследования — это тяжелый труд. Вы должны документировать все схемы применения его собственных методов в рамках класса, а как только вы их документируете, вы должны следовать им все время существования класса. Если вам не удастся сделать это, подклассы могут стать зависимыми от деталей реализации суперкласса, и это может нарушить их работоспособность при изменении реализации суперкласса. Чтобы разрешить другим пользователям писать *эффективные* подклассы, возможно, придется также экспортировать один или несколько защищенных методов. Если только вы не знаете совершенно точно, что существует реальная потребность в подклассах, вероятно, лучше всего будет запретить наследование, объявив ваш класс как `final` или обеспечив отсутствие доступных конструкторов.

4.6. Предпочитайте интерфейсы абстрактным классам

В Java имеется два механизма для определения типа, допускающих несколько реализаций: интерфейсы и абстрактные классы. С момента введения *методов по умолчанию* для интерфейсов в Java 8 [25, 9.4.3] оба механизма позволяют вам предоставлять реализации для некоторых методов экземпляра. Основное различие заключается в том, что для реализации типа, определенного абстрактным классом, класс должен являться подклассом абстрактного класса. Поскольку Java разрешает только единичное наследование, это ограничение на абстрактные классы серьезно сдерживает их использование в качестве определений типов. Любому классу, который определяет все необходимые методы и подчиняется общему контракту, разрешено реализовывать интерфейс независимо от того, где в иерархии классов располагается данный класс.

Существующие классы можно легко приспособить для реализации нового интерфейса. Все, что для этого нужно, — добавить в класс необходимые методы, если их там еще нет, и внести в объявление класса конструкцию `implements`. Например, многие существующие классы были переделаны для реализации интерфейсов `Comparable`, `Iterable` и `Autocloseable`, когда они были добавлены в платформу Java. Но уже существующие классы в общем случае не могут быть переделаны для расширения нового абстрактного класса. Если вы хотите, чтобы два класса расширяли один и тот же абстрактный класс, вам придется поднять этот абстрактный класс в иерархии типов настолько

высоко, чтобы он стал предком обоих этих классов. К сожалению, это может вызвать значительное нарушение в иерархии типов, заставляя всех потомков нового абстрактного класса расширять его независимо от того, насколько это целесообразно.

Интерфейсы идеально подходят для создания миксинов. Грубо говоря, *миксин* (mixin) — это тип, который класс может реализовать в дополнение к своему “первичному типу”, объявляя о том, что этот класс предоставляет некоторое необязательное поведение. Например, Comparable представляет собой интерфейс-миксин, который дает классу возможность объявить, что его экземпляры могут быть упорядочены по отношению к другим взаимно сравнимым с ними объектам. Такой интерфейс называется миксином, поскольку позволяет “примешивать” (mixed in) к первоначальной функциональности некоторого типа необязательные функциональные возможности. Использовать абстрактные классы для создания миксинов нельзя по той же причине, по которой их невозможно приспособить к уже имеющимся классам: класс не может иметь больше одного родителя, и в иерархии классов нет подходящего места, куда можно поместить миксин.

Интерфейсы позволяют создавать неиерархические каркасы типов. Иерархии типов прекрасно подходят для организации некоторых сущностей, но зато сущности других типов невозможно аккуратно уложить в строгую иерархию. Например, предположим, что у нас есть один интерфейс, представляющий певца, а другой — автора песен:

```
public interface Singer
{
    AudioClip sing(Song s);
}
public interface Songwriter
{
    Song compose(int chartPosition);
}
```

В реальности некоторые певцы пишут песни. Поскольку для определения этих типов мы использовали интерфейсы, а не абстрактные классы, вполне допустимо, чтобы один класса реализовывал и певца, и автора песни. Фактически мы можем определить третий интерфейс, который расширяет и Singer, и Songwriter и добавляет новые методы, имеющие смысл для данной комбинации:

```
public interface SingerSongwriter extends Singer, Songwriter
{
    AudioClip strum();
    void actSensitive();
}
```

Такой уровень гибкости нужен не всегда, но когда он необходим, интерфейсы становятся спасительным средством. Альтернативой им является раздутая иерархия классов, которая содержит отдельный класс для каждой поддерживаемой комбинации атрибутов. Если в системе имеется n атрибутов, то имеется 2^n возможных их сочетаний, которые, возможно, придется поддерживать. Это то, что называется *комбинаторным взрывом*. Раздутые иерархии классов могут вести к созданию раздутых классов с множеством методов, отличающихся один от другого лишь типом аргументов, поскольку в такой иерархии классов не будет типов, охватывающих общее поведение.

Интерфейсы обеспечивают безопасное и мощное развитие функциональности с использованием идиомы *класса-оболочки*, описанной в разделе 4.4. Если для определения типов вы прибегаете к абстрактному классу, то не оставляете программисту, желающему добавить новые функциональные возможности, иного выбора, кроме использования наследования. Получающиеся в результате классы будут менее мощными и более хрупкими по сравнению с классами-оболочками.

Когда имеется очевидная реализация метода интерфейса в терминах другого метода интерфейса, рассмотрите возможность предоставления помощи по реализации в виде метода по умолчанию. В качестве примера этой техники рассмотрите метод `removeIf` в разделе 4.7. Если вы предоставляете методы по умолчанию, убедитесь, что они документированы для наследования с использованием дескриптора Javadoc `@implSpec` (раздел 4.5).

Существуют ограничения на то, какую помощь можно предоставить с использованием методов по умолчанию. Хотя многие интерфейсы специфицируют поведение методов `Object`, таких как `equals` и `hashCode`, вам не разрешается предоставлять для них методы по умолчанию. Кроме того, интерфейсам не разрешается содержать поля экземпляров или не открытых статических членов (за исключением закрытых статических методов). Наконец, нельзя добавлять методы по умолчанию в интерфейс, которым вы не управляете.

Однако можно объединить преимущества интерфейсов и абстрактных классов, предоставляя абстрактный *класс скелетной реализации* (*skeletal implementation class*), сопутствующий интерфейсу. Интерфейс определяет тип, возможно, предоставляющий некоторые методы по умолчанию, в то время как класс скелетной реализации реализует остальные непримитивные методы интерфейса поверх примитивных методов. Расширение скелетной реализации занимает большую часть работы по реализации интерфейса. Это проектный шаблон *Шаблонный метод* (Template Method) [12].

По соглашению классы скелетной реализации называются `AbstractInterface`, где `Interface` представляет собой имя интерфейса, который такой класс реализует. Например, `Collections Framework` предоставляет

скелетную реализацию для каждого из основных интерфейсов коллекций: `AbstractCollection`, `AbstractSet`, `AbstractList` и `AbstractMap`. Возможно, имело бы смысл назвать их `SkeletalCollection`, `SkeletalSet`, `SkeletalList` и `SkeletalMap`, но соглашение `Abstract` уже прочно устоялось. При правильном проектировании скелетные реализации (будь то отдельный абстрактный класс или состоящий исключительно из методов по умолчанию интерфейс) могут *очень* упростить для программистов предоставление их собственных реализаций интерфейса. Например, далее показан статический фабричный метод, содержащий полную, полностью функциональную реализацию `List` поверх `AbstractList`:

```
// Конкретная реализация, построенная поверх скелетной реализации
static List<Integer> intArrayAsList(int[] a)
{
    Objects.requireNonNull(a);
    // Оператор "ромб" <> корректен здесь только в Java 9
    // и более поздних версиях. При использовании более
    // ранних версий указывайте <Integer>
    return new AbstractList<>()
    {
        @Override public Integer get(int i)
        {
            return a[i]; // Автоупаковка (раздел 2.6)
        }
        @Override public Integer set(int i, Integer val)
        {
            int oldVal = a[i];
            a[i] = val; // Автораспаковка
            return oldVal; // Автоупаковка
        }
        @Override public int size()
        {
            return a.length;
        }
    };
}
```

Если рассмотреть все, что делает реализация `List`, этот пример окажется впечатляющей демонстрацией мощи скелетных реализаций. Кстати, этот представляет собой проектный шаблон *Адаптер* (Adapter) [12], который позволяет рассматривать массив `int` как список экземпляров `Integer`. Из-за преобразований из значений `int` в экземпляры `Integer` и обратно (упаковки и распаковки) производительность метода не очень высока. Заметим, что реализация принимает форму анонимного класса (раздел 4.10).

Красота классов скелетных реализаций заключается в том, что они предоставляют всю помощь в реализации абстрактных классов, не налагая при этом

строгих ограничений, которые демонстрировали бы абстрактные классы, если бы они служили определениями типов. Для большинства программистов, реализующих интерфейс с помощью класса скелетной реализации, очевидным выбором оказывается расширение этого класса (хотя это и необязательный выбор). Если класс нельзя сделать расширяющим скелетную реализацию, он всегда может реализовать непосредственно интерфейс. Более того, скелетная реализация может помочь в решении стоящей перед разработчиком задачи. Класс, реализующий интерфейс, может передавать вызовы методов интерфейса содержащемуся в нем экземпляру закрытого внутреннего класса, расширяющего скелетную реализацию. Такой прием, известный как *имитация множественного наследования* (simulated multiple inheritance), тесно связан с идиомой класса-оболочки, рассматривавшейся в разделе 4.4. Он обладает большинством преимуществ множественного наследования и при этом избегает его ловушек.

Написание скелетной реализации оказывается относительно простым, хотя иногда и утомительным процессом. Сначала следует изучить интерфейс и принять решение о том, какие из методов являются примитивами, в терминах которых можно было бы реализовать остальные методы интерфейса. Эти примитивы будут абстрактными методами вашей скелетной реализации. После этого в интерфейсе следует предоставить методы по умолчанию для всех методов, которые могут быть реализованы непосредственно поверх примитивов; следует только не забывать, что нельзя предоставлять методы по умолчанию для таких методов `Object`, как `equals` и `hashCode`. Если примитивы и методы по умолчанию охватывают интерфейс, работа завершена и нет никакой необходимости в классе скелетной реализации. В противном случае следует написать класс, объявленный как реализующий интерфейс, с реализациями всех остальных методов интерфейса. Этот класс может содержать любые не открытые поля и методы, соответствующие стоящей перед ним задаче.

В качестве простого примера рассмотрим интерфейс `Map.Entry`. Очевидными примитивами являются `getKey`, `getValue` и (необязательно) `setValue`. Интерфейс определяет поведение `equals` и `hashCode`, и имеется очевидная реализация `toString` в терминах примитивов. Поскольку для методов `Object` вы не можете предоставлять реализации по умолчанию, все реализации помещаются в класс скелетной реализации:

```
// Класс скелетной реализации
public abstract class AbstractMapEntry<K, V>
    implements Map.Entry<K, V>
{
    // Записи в изменяемом отображении должны перекрывать этот метод
    @Override public V setValue(V value)
    {
```

```
        throw new UnsupportedOperationException();
    }
    // Реализует общий контракт Map.Entry.equals
    @Override public boolean equals(Object o)
    {
        if (o == this)
            return true;

        if (!(o instanceof Map.Entry))
            return false;

        Map.Entry <?, ?> e = (Map.Entry) o;
        return Objects.equals(e.getKey(), getKey())
            && Objects.equals(e.getValue(), getValue());
    }
    // Реализует общий контракт Map.Entry.hashCode
    @Override public int hashCode()
    {
        return Objects.hashCode(getKey())
            ^ Objects.hashCode(getValue());
    }
    @Override public String toString()
    {
        return getKey() + "=" + getValue();
    }
}
```

Обратите внимание, что эта скелетная реализация не может быть реализована в интерфейсе `Map.Entry` или в качестве подынтерфейса, поскольку методы по умолчанию не могут перекрывать такие методы `Object`, как `equals`, `hashCode` и `toString`.

Поскольку скелетная реализация предназначена для наследования, вы должны следовать всем указаниям по разработке и документированию, представленным в разделе 4.5. Для краткости в предыдущем примере опущены документирующие комментарии, однако **хорошая документация скелетных реализаций абсолютно необходима**, состоят ли они из методов по умолчанию интерфейса или являются отдельными абстрактными классами.

Уменьшенным вариантом скелетной реализации является *простая реализация*, показанная в `AbstractMap.SimpleEntry`. Простая реализация подобна скелетной реализации тем, что она реализует интерфейс и предназначена для наследования, но отличается тем, что не является абстрактной: это простейшая возможная работающая реализация. Вы можете использовать ее как есть или создать из нее подкласс.

Подытожим сказанное в данном разделе. Интерфейс в общем случае является лучшим средством определения типа, который допускает несколько

реализаций. Если вы экспортируете нетривиальный интерфейс, следует хорошо подумать над созданием сопутствующей скелетной реализации. Насколько это возможно, следует предоставлять скелетную реализацию с помощью методов по умолчанию интерфейса таким образом, чтобы все программисты, реализующие интерфейс, могли их использовать. С учетом сказанного, ограничения на интерфейсы обычно приводят к тому, что скелетная реализация принимает форму абстрактного класса.

4.7. Проектируйте интерфейсы для потомков

До Java 8 было невозможно добавлять методы в интерфейсы без нарушения существующих реализаций. Если вы добавляли новый метод в интерфейс, то в общем случае существующие реализации, в которых отсутствовал этот метод, приводили к ошибке времени компиляции. В Java 8 была добавлена конструкция *метод по умолчанию* [25, 9.4] для возможности предоставлять дополнительные методы к существующим интерфейсам. Но добавление новых методов к существующим интерфейсам сопряжено с риском.

Объявление метода по умолчанию включает *реализацию по умолчанию*, которая используется всеми классами, реализующими интерфейс, но не реализующими метод по умолчанию. В то время как добавление методов по умолчанию в Java делает возможным добавление методов к существующему интерфейсу, нет никакой гарантии, что эти методы будут работать во всех ранее существовавших реализациях. Методы по умолчанию “вводятся” в существующие реализации без ведома или согласия их создателей. До Java 8 эти реализации писались с умалчиваемым соглашением о том, что их интерфейсы никогда не будут приобретать никакие новые методы.

В базовую коллекцию интерфейсов Java 8 было добавлено много новых методов по умолчанию, главным образом для облегчения использования лямбда-выражений (глава 6, “Перечисления и аннотации”). Методы по умолчанию библиотек Java представляют собой высококачественные реализации общего назначения и в большинстве случаев работают нормально. Но **не всегда возможно написать метод по умолчанию, который поддерживает все инварианты всех мыслимых реализаций.**

Рассмотрим, например, метод `removeIf`, который был добавлен к интерфейсу `Collection` в Java 8. Этот метод удаляет все элементы, для которых данная функция с возвращаемым типом `boolean` (*предикат*) возвращает значение `true`. Реализация по умолчанию определяется как обходящая коллекцию с использованием итератора и вызывающая предикат для каждого элемента; для удаления элементов, для которых предикат возвращает значение `true`,

используется метод итератора `remove`. Предположительно объявление выглядит примерно следующим образом:

```
// Метод по умолчанию добавлен в интерфейс Collection в Java 8
default boolean removeIf(Predicate<? super E> filter)
{
    Objects.requireNonNull(filter);
    boolean result = false;

    for (Iterator<E> it = iterator(); it.hasNext();)
    {
        if (filter.test(it.next()))
        {
            it.remove();
            result = true;
        }
    }

    return result;
}
```

Это лучшая реализация общего назначения, которую только можно было написать для метода `removeIf`, но, к сожалению, он не годится для некоторых реальных реализаций `Collection`. Например, рассмотрим `org.apache.commons.collections4.collection.SynchronizedCollection`. Этот класс из библиотеки Apache Commons аналогичен возвращаемому статической фабрикой `Collections.synchronizedCollection` в `java.util`. Версия Apache дополнительно предоставляет возможность использовать для блокировки вместо коллекции объект, предоставляемый клиентом. Другими словами, это класс-оболочка (раздел 4.4), все методы которого синхронизируются блокирующим объектом перед делегированием в “обернутую” коллекцию.

Класс Apache `SynchronizedCollection` все еще активно поддерживается, но на момент написания книги он не перекрывает метод `removeIf`. Если этот класс используется в сочетании с Java 8, он наследует реализацию `removeIf` по умолчанию, которая не поддерживает (фактически *не может* поддерживать) фундаментальные обещания класса: автоматически синхронизировать каждый вызов метода. Реализация по умолчанию ничего не знает о синхронизации и не имеет доступа к полю, которое содержит блокирующий объект. Если клиент вызывает метод `removeIf` для экземпляра `SynchronizedCollection` во время параллельного изменения коллекции другим потоком, результатом может стать генерация исключения `ConcurrentModificationException` или иное неопределенное поведение.

Для предотвращения таких ситуаций в аналогичных реализациях библиотек платформы Java, таких как доступный на уровне пакета класс, возвращаемый

`Collections.synchronizedCollection`, сопровождение JDK вынуждено было перекрыть реализацию по умолчанию `removeIf` и другие подобные методы, чтобы выполнять необходимую синхронизацию до вызова реализации по умолчанию. Существующие реализации коллекций, которые не были частью платформы Java, не имели возможности внести аналогичные изменения одновременно с изменением интерфейса, а некоторым это еще предстоит сделать.

В присутствии методов по умолчанию существующие реализации интерфейсов могут компилироваться без ошибок или предупреждений, но сбоят во время выполнения. Будучи не очень распространенной, эта проблема, тем не менее, не является и отдельным инцидентом. Известно, что ряд методов среди добавленных в интерфейсы коллекций Java 8 подвержены этим эффектам и влияют на работоспособность некоторых существующих реализаций.

Следует избегать методов по умолчанию для добавления новых методов к существующим интерфейсам, если только необходимость в таком действии не является критической; в этом случае следует хорошо подумать о том, не могут ли существующие реализации интерфейса быть повреждены этой реализацией метода по умолчанию. Однако методы по умолчанию чрезвычайно полезны для обеспечения реализаций стандартных методов при создании интерфейса, чтобы облегчить задачу его реализации (раздел 4.6).

Стоит также отметить, что методы по умолчанию не предназначались для поддержки удаления методов из интерфейсов или изменения сигнатур существующих методов. Ни одно из этих изменений интерфейса не может быть внесено без нарушения работы существующих клиентов.

Мораль ясна. Даже несмотря на то, что методы по умолчанию теперь являются частью платформы Java, **первостепенное значение при разработке интерфейсов по-прежнему имеет предельная аккуратность.** Хотя методы по умолчанию *позволяют* добавлять методы к существующим интерфейсам, это связано с большим риском. Если интерфейс содержит незначительные недостатки, это может постоянно раздражать пользователей; если интерфейс крайне недостаточен — он может обречь на неудачу весь содержащий его API.

Таким образом, критически важно тестировать каждый новый интерфейс перед его выпуском. Несколько программистов должны реализовывать каждый интерфейс различными способами. Как минимум следует стремиться к трем различным реализациям. Не менее важно написать несколько клиентских программ, которые используют экземпляры каждого нового интерфейса для выполнения различных задач. Путь к тому, что каждый интерфейс удовлетворяет всем его предполагаемым использованиям, долог, но позволяет обнаружить недостатки в интерфейсах до того, как они будут выпущены для широкого применения, когда их еще можно легко исправить. **Хотя может оказаться возможным исправить некоторые недостатки интерфейса и после его выпуска, рассчитывать на это не следует.**

4.8. Используйте интерфейсы только для определения типов

Если класс реализует интерфейс, то этот интерфейс служит в качестве *типа*, который может быть использован для ссылки на экземпляры класса. То, что класс реализует некий интерфейс, должно, таким образом, говорить о том, что именно клиент может делать с экземплярами этого класса. Создавать интерфейс для каких-либо иных целей не следует.

Среди интерфейсов, которые не отвечают этому критерию, имеется так называемый *интерфейс констант* (constant interface). Он не имеет методов и содержит исключительно поля `static final`, экспортирующие константы. Классы, использующие эти константы, реализуют данный интерфейс для того, чтобы избежать необходимости квалифицировать имена констант именем класса. Вот небольшой пример:

```
// Антишаблон интерфейса констант — не используйте его!
public interface PhysicalConstants
{
    // Число Авогадро (1/mol)
    static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    // Постоянная Больцмана (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;
    // Масса электрона (kg)
    static final double ELECTRON_MASS = 9.109_383_56e-31;
}
```

Проектный шаблон интерфейса констант представляет собой пример плохого использования интерфейсов. То, что класс использует некоторые константы внутренне, является деталью реализации. Реализация интерфейса констант приводит к тому, что эта деталь реализации перетекает в экспортируемый API класса. Для пользователей класса не имеет значения, что класс реализует такой интерфейс констант. На самом деле это может даже запутывать их. Что еще хуже, он представляет собой обязательство: даже если в будущих версиях класс изменится таким образом, что ему больше не потребуется использовать константы, он все равно должен будет реализовывать интерфейс для обеспечения бинарной совместимости. Если не окончательный класс реализует интерфейс констант, у всех его подклассов пространство имен будет замусорено этими константами.

В библиотеках платформы Java имеется несколько интерфейсов констант, например `java.io.ObjectStreamConstants`. Эти интерфейсы должны рассматриваться как аномалии, с которых не следует брать пример.

Если вы хотите экспортировать константы, используйте для этого несколько иных разумных способов. Если константы сильно привязаны к существующему классу или интерфейсу, необходимо добавить их к этому классу или интерфейсу. Например, все упакованные численные примитивные классы, такие как `Integer` и `Double`, экспортируют константы `MIN_VALUE` и `MAX_VALUE`. Если константы лучше рассматривать как члены перечислимого типа, их следует экспортировать как *тип перечисления* (раздел 6.1). В противном случае нужно экспортировать константы с помощью неинстанцируемого вспомогательного класса (раздел 2.4). Вот версия вспомогательного класса `PhysicalConstants` для рассмотренного выше примера:

```
// Вспомогательный класс констант
package com.effectivejava.science;

public class PhysicalConstants
{
    private PhysicalConstants() { } // Не допускает инстанцирование

    public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
    public static final double BOLTZMANN_CONST  = 1.380_648_52e-23;
    public static final double ELECTRON_MASS    = 9.109_383_56e-31;
}
```

Кстати, обратите внимание на использование символа подчеркивания (`_`) в числовых литералах. Подчеркивания, которые разрешены начиная с Java 7, не влияют на значения числовых литералов, но могут сделать их гораздо легче читаемыми, если использовать их с осторожностью. Рекомендуется добавлять подчеркивания в числовые литералы, как с фиксированной, так и с плавающей точкой, если они содержат пять или более цифр подряд. Для литералов в десятичной системе счисления, как для целых чисел, так и для чисел с плавающей точкой, следует использовать подчеркивания для разделения на группы по три цифры, указывающие положительные и отрицательные степени тысячи.

Обычно вспомогательный класс требует от клиентов квалифицировать имена констант именем класса, например `PhysicalConstants.AVOGADROS_NUMBER`. Если вы интенсивно используете константы, экспортируемые вспомогательным классом, можете избежать необходимости квалификации констант именем класса, прибегнув к *статическому импорту*:

```
// Использование статического импорта
// во избежание квалификации констант
import static com.effectivejava.science.PhysicalConstants.*;
public class Test
{
    double atoms(double mols)
    {
```

```

    return AVOGADROS_NUMBER * mols;
}
...
// Множество применений PhysicalConstants
// оправдывает статический импорт
}

```

Итак, главный вывод — интерфейсы нужно использовать только для определения типов. Их не следует использовать для простого экспорта констант.

4.9. Предпочитайте иерархии классов дескрипторам классов

Иногда можно встретить класс, экземпляры которого могут быть двух и более разновидностей и содержат поле *дескриптора* (tag), указывающего разновидность конкретного экземпляра. Рассмотрим, например, класс, который может представлять окружность либо прямоугольник:

// Класс с дескриптором: значительно уступает иерархии классов!

```

class Figure
{
    enum Shape { RECTANGLE, CIRCLE };
    // Поле дескриптора - форма данной фигуры
    final Shape shape;
    // Эти поля используются в RECTANGLE
    double length;
    double width;
    // Это поле используется в CIRCLE
    double radius;
    // Конструктор окружности
    Figure(double radius)
    {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }
    // Конструктор прямоугольника
    Figure(double length, double width)
    {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }
    double area()
    {
        switch (shape)
        {

```

```

    case RECTANGLE:
        return length * width;

    case CIRCLE:
        return Math.PI * (radius * radius);

    default:
        throw new AssertionError(shape);
    }
}
}
}

```

У таких классов с дескрипторами (tagged class) имеются многочисленные недостатки. Они загромождены излишним кодом, включая объявления перечислений, полями дескрипторов и инструкциями switch. Об удобочитаемости нечего и говорить, потому что в одном классе, по сути, перемешано несколько реализаций. Растет расход памяти, так как экземпляры переполнены ненужными полями, принадлежащими другим разновидностям. Поля нельзя сделать окончательными, если только конструкторы не инициализируют их значений, что приводит к более стереотипному коду. Конструкторы должны задавать поле дескриптора и инициализировать поля данных без помощи компилятора: если инициализировать поля неправильно, то программа завершится ошибкой во время выполнения. Нельзя добавить еще одну разновидность в класс, кроме как изменяя исходный файл. При добавлении разновидности следует не забывать о необходимости добавления соответствующих дополнений в каждую инструкцию switch, иначе работа класса завершится ошибкой времени выполнения. Наконец, тип данных экземпляра ничего не говорит о том, что же собой представляет этот экземпляр. Словом, такие классы с дескрипторами многословны, склонны к ошибкам и неэффективны.

К счастью, объектно-ориентированные языки программирования обладают намного лучшим механизмом определения типов, который можно использовать для представления объектов разных типов: создание подтипов. **Применение дескрипторов в действительности является лишь бледным подобием использования иерархии классов.**

Чтобы преобразовать класс с дескриптором в иерархию классов, сначала определите абстрактный класс, содержащий метод для каждого метода класса с дескриптором, работа которого зависит от значения дескриптора. В приведенном выше примере класса Figure единственным таким методом является area. Получившийся абстрактный класс будет корнем иерархии классов. Если имеются методы, поведение которых не зависит от значения дескриптора, представьте их как неабстрактные методы корневого класса. Аналогично, если имеются какие-либо поля данных, используемые всеми разновидностями

класса, перенесите их в корневой класс. В приведенном примере класса Figure подобных операций и полей данных, не зависящих от типа, нет.

Далее, для каждой разновидности исходного класса с дескриптором определите конкретный подкласс корневого класса. В нашем примере такими типами являются окружность и прямоугольник. В каждый подкласс поместите поля данных, специфичные для соответствующего типа. В нашем примере radius специфичен для окружности, а length и width характеризуют прямоугольник. Кроме того, в каждый подкласс поместите соответствующую реализацию всех абстрактных методов корневого класса. Вот как выглядит иерархия классов, соответствующая нашему исходному классу Figure:

// Иерархия классов взамен класса с дескриптором

```
abstract class Figure
{
    abstract double area();
}
class Circle extends Figure
{
    final double radius;
    Circle(double radius)
    {
        this.radius = radius;
    }
    @Override double area()
    {
        return Math.PI * (radius * radius);
    }
}
class Rectangle extends Figure
{
    final double length;
    final double width;
    Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }
    @Override double area()
    {
        return length * width;
    }
}
```

Эта иерархия классов исправляет все недостатки классов с дескрипторами, отмеченные ранее. Код прост и ясен, не содержит повторов. Реализация каждой разновидности выделена в собственный класс, и ни один из этих классов

не перегружен ненужными полями данных. Все поля окончательные. Компилятор гарантирует, что каждый конструктор класса инициализирует свои поля данных и что у каждого класса есть реализация каждого абстрактного метода, объявленного в корневом классе. Это помогает избежать возможных ошибок при выполнении благодаря отсутствию инструкций `switch`. Различные программисты могут расширять иерархию независимо друг от друга без необходимости доступа к исходному коду корневого класса. Для каждой разновидности класса имеется отдельный тип данных, позволяющий программистам указывать тип переменных и входные параметры для этого конкретного типа.

Еще одно преимущество иерархий классов заключается в том, что они отражают естественные иерархические отношения между типами, что обеспечивает повышенную гибкость и улучшает проверку типов во время компиляции. Предположим, что класс с дескриптором допускает также построение квадратов. В иерархии классов можно отразить тот факт, что квадрат — это частный случай прямоугольника (при условии, что оба они неизменяемы):

```
class Square extends Rectangle
{
    Square(double side)
    {
        super(side, side);
    }
}
```

Обратите внимание, что поля классов этой иерархии доступны непосредственно, а не через методы доступа. Это сделано для краткости и это было бы ошибкой, если бы классы были открытыми (раздел 4.2).

Резюмируя, можно сказать, что классы с дескрипторами редко оказываются приемлемыми. Если у вас появится искушение написать такой класс, подумайте, не стоит ли убрать дескриптор и заменить его иерархией классов. Если вам встретится уже существующий класс, подумайте над возможностью его рефакторинга в иерархию классов.

4.10. Предпочитайте статические классы-члены нестатическим

Вложенный класс (nested class) — это класс, определенный внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать охватывающий его класс. Если вложенный класс оказывается полезным в каком-либо ином контексте, он должен стать классом верхнего уровня. Существует четыре разновидности вложенных классов: *статический*

класс-член (static member class), *нестатический класс-член* (nonstatic member class), *анонимный класс* (anonymous class) и *локальный класс* (local class). За исключением первого, остальные разновидности классов называются *внутренними классами* (inner class). В этом разделе рассказывается о том, когда и какую разновидность вложенного класса из перечисленных нужно использовать и почему.

Статические классы-члены — это простейшая разновидность вложенных классов. Лучше всего рассматривать такой класс как обычный класс, который объявлен внутри другого класса и имеет доступ ко всем членам охватывающего класса, даже к закрытым. Статический класс-член представляет собой статический член своего охватывающего класса и подчиняется тем же правилам доступа, что и другие статические члены. Так, если он объявлен как `private`, то он доступен только в охватывающем классе, и т.д.

В одном из распространенных вариантов статический класс-член представляет собой открытый вспомогательный класс, который пригоден для применения только в сочетании с его внешним классом. Например, рассмотрим перечисление, описывающее операции, которые может выполнять калькулятор (раздел 6.1). Перечисление `Operation` должно быть открытым статическим классом-членом класса `Calculator`. Клиенты класса `Calculator` могут обращаться к этим операциям с использованием таких имен, как `Calculator.Operation.PLUS` или `Calculator.Operation.MINUS`.

С точки зрения синтаксиса единственное различие между статическими и нестатическими классами-членами заключается в том, что в объявлениях статических классов-членов присутствует модификатор `static`. Несмотря на свою синтаксическую схожесть, это совершенно разные категории вложенных классов. Каждый экземпляр нестатического члена-класса неявно связан с *охватывающим экземпляром* (enclosing instance) содержащего его класса. В методах экземпляра нестатического класса-члена можно вызывать методы содержащего его экземпляра или получать ссылку на охватывающий экземпляр с использованием конструкции *квалифицированного this* (qualified this) [25, 15.8.4]. Если экземпляр вложенного класса может существовать отдельно от экземпляра охватывающего класса, то вложенный класс *обязан* быть статическим членом-классом: нельзя создать экземпляр нестатического класса-члена, не создав включающий его экземпляр.

Связь между экземпляром нестатического класса-члена и включающим его экземпляром устанавливается при создании первого и не может быть изменена позднее. Обычно эта связь устанавливается автоматически путем вызова конструктора нестатического класса-члена из метода экземпляра охватывающего класса. Возможно, хотя и очень редко, установить связь вручную с использованием выражения `enclosingInstance.new MemberClass(args)`. Как

можно ожидать, эта связь занимает место в экземпляре нестатического класса-члена и увеличивает время его создания.

Нестатические классы-члены часто используются для определения *Адаптера* (Adapter) [12], который позволяет рассматривать экземпляр внешнего класса как экземпляр некоторого не связанного с ним класса. Например, в реализациях интерфейса Map нестатические классы-члены обычно применяются для реализации *представлений коллекций* (collection view), возвращаемых методами keySet, entrySet и values интерфейса Map. Аналогично реализации интерфейсов коллекций, таких как Set или List, обычно используют для реализации своих итераторов нестатические классы-члены:

```
// Типичное применение нестатического класса-члена
public class MySet<E> extends AbstractSet<E>
{
    ... // Основная часть класса опущена
    @Override public Iterator<E> iterator()
    {
        return new MyIterator();
    }
    private class MyIterator implements Iterator<E>
    {
        ...
    }
}
```

Если вы объявили класс-член, которому не нужен доступ к охватываемому экземпляру, *всегда* помещайте в его объявление модификатор **static**, делая этот класс-член статическим. Если вы пропустите этот модификатор, каждый экземпляр класса будет содержать ненужную ссылку на охватывающий экземпляр. Как уже упоминалось, такая связь требует времени и памяти, но не приносит никакой пользы. Более того, охватывающий экземпляр будет оставаться в памяти, в то время как он мог бы быть освобожден сборщиком мусора (раздел 2.7). Получающаяся в итоге утечка памяти может оказаться катастрофической. Ее часто трудно обнаружить, поскольку такая ссылка невидима.

Обычное применение закрытых статических классов-членов обычно состоит в представлении компонентов объекта, представленного охватывающим классом. Например, рассмотрим экземпляр класса Map, который связывает ключи и значения. Многие реализации Map содержат внутренний объект Entry для каждой пары “ключ/значение”. Хотя каждая такая запись связана с отображением, методам записи (getKey, getValue и setValue) доступ к отображению не требуется. Следовательно, использовать нестатические классы-члены для представления записей было бы расточительно. Лучшим

решением является закрытый статический класс-член. Если в объявлении этой записи вы случайно пропустите модификатор `static`, схема будет работать, но каждая запись будет содержать излишнюю ссылку на отображение, напрасно тратя время и память.

Вдвойне важно правильно сделать выбор между статическими и нестатическими классами-членами, если этот класс является открытым или защищенным членом экспортируемого класса. В этом случае класс-член является элементом экспортируемого API, и в последующих версиях нельзя будет сделать нестатический класс-член статическим, не нарушив обратную совместимость.

Как следует из названия, анонимный класс не имеет имени. Он не является членом охватывающего его класса. Вместо того чтобы быть объявленным с остальными членами класса, он одновременно и объявляется, и инстанцируется в точке использования. Анонимные классы могут быть разрешены в любом месте программы, где разрешается применять выражения. Анонимные классы имеют охватывающие экземпляры тогда и только тогда, когда они находятся в нестатическом контексте. Но даже если они находятся в статическом контексте, они не могут иметь никаких статических членов, отличных от *переменных констант* (*constant variables*), которые представляют собой `final`-поля примитивных типов или строк, инициализированные константными выражениями [25, 4.12.4].

Применение анонимных классов имеет ряд ограничений. Их нельзя инстанцировать, за исключением точки объявления. Нельзя применить тест `instanceof` или выполнить иные действия, требующие имени класса. Нельзя объявить анонимный класс для реализации нескольких интерфейсов или для расширения класса и реализации интерфейса одновременно. Клиенты анонимного класса не могут вызывать никакие члены за исключением унаследованных от супертипа. Поскольку анонимные классы встречаются среди выражений, они должны быть очень короткими, не более десятка строк, иначе читаемость программы существенно ухудшится.

До того, как в Java вошли лямбда-выражения (глава 6, “Перечисления и аннотации”), анонимные классы являлись предпочтительным средством создания небольших *функциональных объектов* (*function objects*) и *объектов процессов* (*process objects*) “на лету”, но в настоящее время предпочтительнее использовать лямбда-выражения (раздел 7.1). Еще одно распространенное применение анонимных классов — в реализации статических фабричных методов (см. `IntArrayAsList` в разделе 4.6).

Локальные классы, вероятно, относятся к наиболее редко используемой разновидности вложенных классов из четырех перечисленных выше. Локальный класс можно объявить практически везде, где разрешается объявить локальную переменную, и он подчиняется тем же правилам видимости. Локальные

классы имеют несколько общих признаков с каждой из трех других разновидностей вложенных классов. Как и классы-члены, локальные классы имеют имена и могут использоваться многократно. Подобно анонимным классам, они имеют охватывающий экземпляр только тогда, когда они определены в нестатическом контексте, и не могут содержать статические члены. И как и анонимные классы, они должны быть достаточно короткими, чтобы не мешать удобочитаемости исходного текста.

Подведем итоги. Существует четыре разновидности вложенных классов, каждая из которых занимает свою нишу. Если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах метода, используйте класс-член. Если каждому экземпляру класса-члена необходима ссылка на включающий его экземпляр, делайте его нестатическим; в остальных случаях он должен быть статическим. Далее предполагается, что класс находится внутри метода. Если вам нужно создавать экземпляры этого класса только в одном месте программы и уже имеется тип, который характеризует этот класс, сделайте данный класс анонимным. В противном случае делайте его локальным классом.

4.11. Ограничивайтесь одним классом верхнего уровня на исходный файл

Хотя компилятор Java и позволяет определить несколько классов верхнего уровня в одном исходном файле, в этом нет никаких выгод, зато есть значительные риски. Риски обусловлены тем фактом, что определение нескольких классов верхнего уровня в одном исходном файле делает возможным предоставление нескольких определений класса. Какое определение будет использовано, зависит от порядка, в котором исходные файлы передаются компилятору.

Для конкретности рассмотрим исходный файл, который содержит только класс `Main`, который ссылается на члены двух других классов верхнего уровня (`Utensil` (посуда) и `Dessert` (десерт)):

```
public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

Теперь предположим, что как `Utensil`, так и `Dessert` определены в одном исходном файле `Utensil.java`:

```
// Два класса определены в одном файле. Так делать не надо!
class Utensil
{
    static final String NAME = "pan";
}
class Dessert
{
    static final String NAME = "cake";
}
```

Конечно, программа выведет pancake.

Предположим теперь, что случайно создан *другой* исходный файл `Dessert.java`, который определяет те же два класса:

```
// Два класса определены в одном файле. Так делать не надо!
class Utensil
{
    static final String NAME = "pot";
}
class Dessert
{
    static final String NAME = "pie";
}
```

Если вы попытаетесь скомпилировать программу с помощью команды

```
javac Main.java Dessert.java
```

то компиляция завершится неудачно, и компилятор сообщит, что у вас имеются множественные определения классов `Utensil` и `Dessert`. Дело в том, что компилятор сначала компилирует `Main.java`. Когда он встретит ссылку на `Utensil` (которая предшествует ссылке на `Dessert`), он будет искать этот класс в `Utensil.java`, и найдет там оба класса — и `Utensil`, и `Dessert`. Когда затем компилятор встретит в командной строке `Dessert.java`, он будет компилировать и этот файл, что приведет к коллизии определений классов `Utensil` и `Dessert`.

Если вы компилируете программу с помощью команды

```
javac Main.java
```

или

```
javac Main.java Utensil.java
```

то программа будет вести себя так, как до написания файла `Dessert.java`, т.е. выводить слово pancake. Но если скомпилировать программу командой

```
javac Dessert.java Main.java
```

то она будет выводить слово `potpie`. Таким образом, поведение программы зависит от порядка, в котором исходные файлы передаются компилятору, что явно неприемлемо.

Исправить проблему просто — разделив классы верхнего уровня (в нашем примере — `Utensil` и `Dessert`) на отдельные исходные файлы. Если вас соблазняет возможность объединить несколько классов верхнего уровня в один исходный файл, то подумайте о возможности использования статических классов-членов (раздел 4.10) как об альтернативе разделению классов в отдельные исходные файлы. Если классы подчинены другому классу, то их превращение в статические классы-члены обычно является лучшим решением, потому что улучшает читаемость и делает возможным сократить доступность классов, объявив их закрытыми (раздел 4.1). Вот как выглядит наш пример со статическими классами-членами:

```
// Применение статических классов-членов вместо
// нескольких классов высокого уровня
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
    private static class Utensil
    {
        static final String NAME = "pan";
    }
    private static class Dessert
    {
        static final String NAME = "cake";
    }
}
```

Урок ясен: никогда не размещайте несколько классов верхнего уровня или интерфейсов в одном исходном файле. Следование этому правилу гарантирует, что у вас не будет нескольких определений одного класса во время компиляции. Это, в свою очередь, гарантирует, что файлы классов, генерируемые компилятором, и поведение результирующей программы не зависят от порядка, в котором исходные файлы передаются компилятору.

Обобщенное программирование

Начиная с Java 5 в языке программирования появились возможности обобщенного программирования. До этого программисту приходилось выполнять приведение каждого объекта, прочитанного из коллекции. Если кто-то случайно вставлял туда объект неправильного типа, приведение могло вызывать ошибку времени выполнения. С помощью обобщенных типов можно сообщить компилятору, какие типы объектов допускаются в каждой коллекции. Компилятор автоматически добавит приведение типов и *во время компиляции* обнаружит попытки вставить объект неправильного типа. Это делает программы более безопасными и ясными, но эти преимущества не ограничиваются только работой с коллекциями. В этой главе рассказывается о том, как получить максимальные выгоды от обобщенного программирования и свести к минимуму осложнения.

5.1. Не используйте несформированные типы

Для начала — несколько терминов. Класс или интерфейс, объявление которого содержит один или несколько *параметров типа* (type parameters), является *обобщенным* (generic) классом или интерфейсом [25, 8.1.2, 9.1.2]. Например, интерфейс `List` имеет единственный параметр типа, `E`, представляющий тип элементов списка. Полное имя интерфейса — `List<E>`, но его часто называют для краткости просто списком — `List`. Обобщенные классы и интерфейсы вместе известны как *обобщенные типы*.

Каждый обобщенный тип определяет множество *параметризованных типов* (parameterized types), которые состоят из имени класса или интерфейса, за которым в угловых скобках следует список *фактических параметров типа* (actual type parameters), соответствующих формальным параметрам типа обобщенного типа [25, 4.4, 4.5]. Например, список строк `List<String>` является параметризованным типом, представляющим список, элементы которого

имеют тип `String` (`String` является фактическим параметром типа, соответствующим формальному параметру типа `E`).

Наконец, каждый обобщенный тип определяет *несформированный тип* (raw type¹), который представляет собой имя обобщенного типа, использованное без сопутствующих параметров типа [25, 4.8]. Например, несформированный тип, соответствующий `List<E>`, — просто `List`. Несформированные типы ведут себя так, как если бы вся информация об обобщенном типе была стерта из объявления типа. Они существуют главным образом для обратной совместимости с кодом, написанным до появления возможностей обобщенного программирования.

До того, как обобщенные типы были введены в Java, приведенный ниже код был бы образцовым объявлением коллекции. По состоянию на Java 9 он продолжает быть вполне корректным, но очень далеким от образца:

```
// Несформированный тип коллекции - не делайте так!
// Моя коллекция марок. Содержит только экземпляры Stamp.
private final Collection stamps = ... ;
```

Если вы используете это объявление сегодня, а затем случайно добавите в свою коллекцию марок монету, такая ошибочная вставка скомпилируется и выполнится без ошибок (хотя компилятор выдаст расплывчатое предупреждение):

```
// Ошибочная вставка монеты в коллекцию марок
stamps.add(new Coin( ... )); // Предупреждение "непроверенный вызов"
```

Вы не получите сообщение об ошибке до тех пор, пока не попытаетесь получить монету из коллекции марок:

```
// Несформированный тип итератора - не делайте так!
for (Iterator i = stamps.iterator(); i.hasNext(); )
    Stamp stamp = (Stamp) i.next(); // Генерация ClassCastException
stamp.cancel();
```

Как уже упоминалось в книге, лучше всего обнаруживать ошибки сразу же, как только они сделаны, в идеале — во время компиляции. В нашем же случае ошибка не обнаружится до момента выполнения программы, гораздо позже, чем она была допущена, и совершенно не в том коде, в котором она была допущена. Когда вы увидите исключение `ClassCastException`, вам придется просматривать весь код и искать вызов метода, который помещает монету в коллекцию марок. Здесь компилятор вам не может помочь, потому что он не понимает комментарий, в котором сказано “Содержит только экземпляры `Stamp`”.

¹ Зачастую в русскоязычной литературе используется термин “сырой тип”. — Примеч. пер.

При наличии обобщенных типов объявление типа содержит необходимую информацию, а не комментарий:

```
// Параметризованный тип коллекции - безопасно
private final Collection<Stamp> stamps = ... ;
```

Из этого объявления компилятор знает, что коллекция `stamps` должна содержать только экземпляры `Stamp`, и *гарантирует*, что это условие будет соблюдаться, если, конечно, весь код компилируется без вывода каких-либо предупреждений (при условии отсутствия их подавления; см. раздел 5.2). Когда коллекция `stamps` объявляется с использованием параметризации, неверная вставка приводит на этапе компиляции к сообщению об ошибке, которое *точно* сообщает, где именно находится ошибка:

```
Test.java:9: error: incompatible types: Coin cannot be converted
to Stamp
    c.add(new Coin());
           ^
```

Компилятор вставляет невидимые приведения типов при извлечении элементов из коллекций и гарантирует, что они не потерпят неудачу (если, конечно, вы не получаете никаких предупреждений от компилятора (и при этом не подавляете их)). Хотя перспектива случайной вставки монеты в коллекцию марок может показаться надуманной, проблема оказывается реальной. Например, легко представить добавление `BigInteger` в коллекцию, которая должна содержать только экземпляры `BigDecimal`.

Как упоминалось ранее, использовать несформированные типы (обобщенные типы без параметров типа) разрешено, но не рекомендуется. **Используя несформированные типы, вы теряете все преимущества безопасности и выразительности обобщенного программирования.** Но если вы не должны использовать необработанные типы, то почему создатели языка разрешают это делать? Чтобы обеспечить совместимость. Платформа Java была близка ко входу в свое второе десятилетие в момент добавления в язык программирования средств обобщенного программирования, так что имеется немалое количество кода, который не использует обобщенные возможности. Критически важно, чтобы весь этот код оставался корректным и мог взаимодействовать с новым кодом, использующим средства обобщенного программирования. Необходимо также было разрешить передачу экземпляров параметризованных типов методам, разработанным для использования несформированных типов, и наоборот. Это требование, известное как *миграционная совместимость* (*migration compatibility*), привело к решению поддержки несформированных типов и реализации обобщенного программирования с использованием *затираний* (*erasure*) (раздел 5.3).

При том, что вы не должны использовать в новом коде несформированные типы, такие как `List`, совершенно нормально использовать параметризованные типы, разрешающие добавление произвольных объектов, такие как `List<Object>`. В чем же разница между несформированным типом `List` и параметризованным типом `List<Object>`? Грубо говоря, первый выпадает из системы обобщенных типов, в то время как последний явно говорит компилятору, что он может содержать объект любого типа. В то время как вы можете передать `List<String>` в качестве параметра типа `List`, вы не можете сделать этого для параметра типа `List<Object>`. Существуют правила образования подтипов для обобщенных типов, и `List<String>` является подтипом несформированного типа `List`, но не параметризованного типа `List<Object>` (раздел 5.3). Как следствие вы **теряете безопасность типов при использовании несформированных типов, таких как `List`, но не при использовании параметризованных типов наподобие `List<Object>`**.

В качестве конкретного примера рассмотрим следующую программу:

```
// Сбой во время выполнения – метод unsafeAdd
// использует несформированный тип (List)!
public static void main(String[] args)
{
    List<String> strings = new ArrayList<>();
    unsafeAdd(strings, Integer.valueOf(42));
    String s = strings.get(0); // Сгенерированное
                               // компилятором приведение
}

private static void unsafeAdd(List list, Object o)
{
    list.add(o);
}
```

Эта программа компилируется, но так как она использует несформированный тип `List`, вы получаете предупреждение:

```
Test.java:10: warning: [unchecked] unchecked call to add(E) as a
member of the raw type List
    list.add(o);
           ^
```

И действительно, если вы запустите эту программу, то получите исключение `ClassException`, когда программа попытается выполнить приведение результата вызова `strings.get(0)` (который представляет собой `Integer`) в `String`. Это приведение, сгенерированное компилятором, поэтому обычно гарантируется его успешное завершение; но в данном случае мы проигнорировали предупреждение компилятора и заплатили за это.

Если заменить в объявлении `unsafeAdd` несформированный тип `List` параметризованным типом `List<Object>` и попробовать снова скомпилировать программу, то можно обнаружить, что она больше не компилируется со следующим сообщением об ошибке:

```
Test.java:5: error: incompatible types: List<String> cannot be
converted to List<Object>
    unsafeAdd(strings, Integer.valueOf(42));
                ^
```

Возможно, у вас появится искушение использовать несформированный тип для коллекции, типы элементов которой неизвестны или не имеют значения. Например, предположим, что вы хотите написать метод, который получает два множества и возвращает количество элементов, которые входят в оба эти множества. Вот как будет выглядеть метод, написанный программистом, не знакомым со средствами обобщенного программирования:

```
// Использование несформированного типа для элементов
// неизвестного типа - не делайте так!
static int numElementsInCommon(Set s1, Set s2)
{
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

Этот метод работает, но использует несформированные типы, что опасно. Безопасная альтернатива состоит в применении *неограниченных типов с символами подстановки* (*unbound wildcard type*). Если вы хотите использовать обобщенные типы, но не знаете (или вам не важны) фактические параметры типов, можете использовать вместо них вопросительный знак. Например, неограниченный тип с символом подстановки для обобщенного типа `Set<E>` будет иметь вид `Set<?>` (“множество с элементами какого-то типа”). Это наиболее общий параметризованный тип `Set`, способный хранить *любое* множество. Вот как выглядит объявление `numElementsInCommon` с использованием символов подстановки:

```
// Неограниченные типы с символами подстановки - безопасно и гибко
static int numElementsInCommon(Set<?> s1, Set<?> s2) { ... }
```

В чем же разница между неограниченным типом с символом подстановки `Set<?>` и несформированным типом `Set`? Дает ли что-либо вопросительный знак реально? Главное — это то, что тип с символом подстановки безопасен, а

несформированный тип — нет. Вы можете поместить *любой* элемент в коллекцию с несформированным типом и при этом легко нарушить инварианты типа коллекции (как было показано на примере метода `unsafeAdd`), но **вы не можете поместить любой элемент (кроме `null`) в коллекцию `Collection<?>`**. Такая попытка приведет к сообщению об ошибке времени компиляции:

```
Wildcard.java:13: error: incompatible types: String cannot be
converted to CAP#1
```

```
    c.add("verboten");
```

```
    ^
```

where CAP#1 is a fresh type-variable:

```
    CAP#1 extends Object from capture of ?
```

Правда, такое сообщение об ошибке оставляет желать лучшего, но компилятор сделал свою работу, защищая вас от повреждения инварианта типа коллекции, независимо от того, каким может быть этот тип элемента. Вы не только не можете поместить в коллекцию `Collection<?>` любой элемент (отличный от `null`), но и не можете ничего предполагать о типе объектов, с которыми работаете. Если такие ограничения являются для вас неприемлемыми, можете использовать *обобщенные методы* (раздел 5.5) или *несвязанные типы с символами подстановки* (раздел 5.6).

Есть несколько небольших исключений из правила “не использовать несформированные типы”. **Необходимо использовать несформированные типы в литералах классов.** Спецификация не разрешает использование параметризованных типов (хотя и разрешает типы массивов и примитивные типы) [25, 15.8.2]. Иными словами, `List.class`, `String[].class` и `int.class` разрешены, а `List<String>.class` и `List<?>.class` — нет.

Вторым исключением из этого правила является оператор `instanceof`. Поскольку информация обобщенного типа во время выполнения затирается, нельзя использовать оператор `instanceof` для параметризованных типов, кроме как для типов с символами подстановки. Использование таких типов вместо несформированных никак не влияет на поведение оператора `instanceof`. В этом случае угловые скобки и вопросительные знаки представляют собой просто шум. **Это предпочтительный способ использования оператора `instanceof` с обобщенными типами:**

```
// Корректное применение несформированного
// типа - оператор instanceof
if (o instanceof Set)    // Raw type
{
    Set<?> s = (Set<?>) o; // Тип с символом подстановки
    ...
}
```

Обратите внимание, что как только мы определим `o` как `Set`, его нужно будет привести к типу с подстановочным знаком `Set<?>`, а не к несформированному типу `Set`. Это проверяемое приведение, так что оно не приведет к предупреждению со стороны компилятора.

Подводя итоги, можно сказать, что использование несформированных типов может привести к ошибке времени выполнения, так что не используйте их в своем коде. Они оставлены в языке лишь для обратной совместимости и взаимодействия с кодом, написанным до появления средств обобщенного программирования. Небольшое резюме: `Set<Object>` — это параметризованный тип, который может содержать объекты любого типа, `Set<?>` — тип с символами подстановки, который может содержать только объекты некоторого неизвестного типа, а `Set` — это несформированный тип, который лишает нас возможности использовать систему обобщенных типов. Первые два варианта безопасны, а последний — нет.

Термины, представленные в этом разделе (и в других местах этой главы), сведены в следующую таблицу.

Термин	Пример	Раздел
Параметризованный тип	<code>List<String></code>	5.1
Фактический параметр типа	<code>String</code>	5.1
Обобщенный тип	<code>List<E></code>	5.1, 5.4
Формальный параметр типа	<code>E</code>	5.1
Неограниченный тип с символом подстановки	<code>List<?></code>	5.1
Несформированный тип	<code>List</code>	5.1
Ограниченный параметр типа	<code><E extends Number></code>	5.4
Рекурсивно ограниченный тип	<code><T extends Comparable<T>></code>	5.5
Ограниченный тип с символом подстановки	<code>List<? extends Number></code>	5.6
Обобщенный метод	<code>static <E> List<E> asList(E[] a)</code>	5.5
Токен типа	<code>String.class</code>	5.8

5.2. Устраняйте предупреждения о непроверяемом коде

При программировании с использованием обобщенных средств часто приходится сталкиваться с множеством предупреждений компилятора: о непроверяемом приведении, о непроверяемом вызове метода, о непроверяемом параметризованном типе с переменным количеством аргументов или о непроверяемом преобразовании. По мере приобретения опыта работы с обобщенными

методами программирования таких предупреждений будет у вас все меньше и меньше, но не стоит ожидать, что вновь написанный код с использованием обобщенных средств будет компилироваться абсолютно без ошибок.

Многих предупреждений о непроверяемом коде можно легко избежать. Например, предположим, что вы случайно написали такое объявление:

```
Set<Lark> exaltation = new HashSet();
```

Компилятор осторожно наметнет вам, что вы поступили неправильно:

```
Venery.java:4: warning: [unchecked] unchecked conversion
    Set<Lark> exaltation = new HashSet();
                        ^
    required: Set<Lark>
    found:     HashSet
```

Вы можете внести указанные исправления, и предупреждение исчезнет. Обратите внимание, что на самом деле вам не нужно указывать параметр типа, надо просто указать, что он имеется, с помощью *оператора бубны*, или *ромба* (<>), введенного в Java 7. После этого компилятор *выведет* правильный фактический параметр типа (в данном случае — Lark):

```
Set<Lark> exaltation = new HashSet<>();
```

Устранить некоторые иные предупреждения *гораздо* сложнее. В этой главе полно примеров таких предупреждений. Когда вы получаете предупреждения, которые требуют размышлений, внимательно разбирайтесь, из-за чего они возникли. **Следует устранять все предупреждения о непроверяемом коде, какие вы только можете устранить!** Если удастся избежать всех предупреждений, можете быть уверены, что код безопасен с точки зрения типов, и это очень хорошо. Это означает, что вы не получите исключение `ClassCastException` при выполнении, и увеличит вашу уверенность в том, что программа ведет себя так, как вы и планировали.

Если вы не можете устранить предупреждения, но уверены, что код, о котором вас предупреждает компилятор, безопасен, то тогда (и только тогда!) можете скрыть предупреждения с помощью аннотации `@SuppressWarnings("unchecked")`. Подавив вывод предупреждений и не убедившись, что код безопасен, вы тем самым лишь создадите ложное ощущение безопасности. Такой код может скомпилироваться без предупреждений, но при выполнении все равно сгенерировать исключение `ClassCastException`. Игнорируя предупреждения о непроверяемом коде и зная при этом, что он безопасен (вместо того чтобы скрыть эти предупреждения), вы можете не заметить новое предупреждение, представляющее реальную проблему. Новое предупреждение потеряется среди ложных тревог, которые вы не скрыли.

Аннотация `SuppressWarnings` может использоваться с любым объявлением, от локальных переменных до целых классов. **Всегда используйте аннотацию `SuppressWarnings` в наименьшей возможной области видимости.** Обычно это объявление переменной, очень короткого метода или конструктора. Никогда не используйте `SuppressWarnings` для целого класса — это может замаскировать действительно критические предупреждения.

При использовании аннотации `SuppressWarnings` для метода или конструктора длиной более чем одна строка, возможно, ее следует перенести в объявление локальной переменной. Возможно также, вам придется объявить новую локальную переменную, но дело того стоит. Например, рассмотрим метод `toArray` из `ArrayList`:

```
public <T> T[] toArray(T[] a)
{
    if (a.length < size)
        return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

Если скомпилировать `ArrayList`, этот метод будет генерировать предупреждение:

```
ArrayList.java:305: warning: [unchecked] unchecked cast
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
                                   ^
required: T[]
found:    Object[]
```

Размещать аннотацию `SuppressWarnings` в инструкции возврата нельзя, потому что это не объявление [25, 9.7]. Вы можете захотеть аннотировать весь метод, но так поступать не нужно! Вместо этого объявите локальную переменную для хранения возвращаемого значения и аннотируйте ее объявление, примерно так:

```
// Добавление локальной переменной для
// уменьшения области видимости @SuppressWarnings
public <T> T[] toArray(T[] a)
{
    if (a.length < size)
    {
        // Это приведение корректно, поскольку создаваемый
        // массив того же типа, что и переданный, т.е. T[].
    }
}
```

```

    @SuppressWarnings("unchecked") T[] result =
        (T[]) Arrays.copyOf(elements, size, a.getClass());
    return result;
}

System.arraycopy(elements, 0, a, 0, size);

if (a.length > size)
    a[size] = null;

return a;
}

```

Получившийся в результате метод компилируется без ошибок и минимизирует область подавления предупреждений о непроверяемом коде.

Каждый раз при использовании аннотации `@SuppressWarnings("unchecked")` добавляйте комментарий с пояснением, почему в данном случае этот код безопасен. Это поможет понять код другим программистам и, что еще более важно, уменьшит вероятность того, что код будет кем-то изменен и вычисления станут небезопасными. Если вам сложно написать такой комментарий, задумайтесь — в результате вы можете прийти к выводу, что данная операция не столь безопасна, как вам казалось сначала.

Резюмируя, можно сказать, что предупреждения о непроверяемом коде важны, и игнорировать их нельзя. Каждое такое предупреждение представляет собой потенциал для генерации исключения `ClassCastException` во время выполнения. Сделайте все возможное, чтобы их устранить. Если же устранить предупреждение не удастся, но вы уверены, что код безопасен, подавите вывод предупреждения аннотацией `@SuppressWarnings("unchecked")` в наименьшем возможном диапазоне. Опишите в комментарии причину, по которой вы приняли решение скрыть предупреждение, и почему этот код, несмотря на предупреждение, безопасен.

5.3. Предпочитайте списки массивам

Массивы отличаются от обобщенных типов двумя важными аспектами. Прежде всего, массивы *ковариантны*. Это страшное слово просто означает, что если `Sub` является подтипом `Super`, то тип массива `Sub[]` является подтипом `Super[]`. Обобщенные типы, напротив, *инвариантны*: для любых двух различных типов `Type1` и `Type2` тип `List<Type1>` не является ни подтипом, ни супертипом `List<Type2>` [25, 4.10; 34, 2.5]. Вы можете решить, что это недостаток обобщенных типов, но, напротив, это недостаток массивов. Следующий фрагмент кода вполне корректен:

```
// Сбой времени выполнения!
```

```
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Исключение ArrayStoreException
```

А этот — нет:

```
// Не компилируется!
```

```
List<Object> ol = new ArrayList<Long>(); // Несовместимые типы
ol.add("I don't fit in");
```

В любом случае вы не можете вставить строку `String` в контейнер для `Long`, но, применяя массив, увидите, что сделали ошибку, только при выполнении программы, в то время как использование списка приведет к сообщению об ошибке во время компиляции.

Вторым важным отличием массивов от обобщенных типов является то, что массивы являются типами, *доступными при выполнении* (reified) [25, 4.7]. Это значит, что массивы знают тип элементов во время выполнения и обеспечивают его. Как было сказано ранее, попытавшись сохранить `String` в массив `Long`, вы получите исключение `ArrayStoreException`. Обобщенные типы, напротив, реализуются с использованием *затирания* (erasure) [25, 4.6]. Это значит, что они обеспечивают ограничения на типы только на этапе компиляции, а затем отбрасывают (или *затирают*) информацию о типах элементов при выполнении. Затирание позволяет обобщенным типам свободно взаимодействовать со старым кодом, который не использует обобщенные типы (раздел 5.1), обеспечивая плавный переход к обобщенным типам в Java 5.

Из-за этих фундаментальных различий массивы и обобщенные типы не удастся “перемешивать”. Например, нельзя создавать массив обобщенного типа, параметризованного типа или с параметрами типа. Ни одно из приведенных далее выражений создания массивов не является разрешенным: `new List<E>[], new List<String>[], new E[]`. Все эти выражения во время компиляции ведут к *ошибкам создания обобщенных массивов*.

Почему нельзя создавать обобщенные массивы? Потому что это небезопасно с точки зрения типов. Если бы это было разрешено, приведение типов, генерируемое компилятором в остальном корректной программе, могло бы приводить к исключению времени выполнения `ClassCastException`. Это нарушило бы фундаментальные гарантии, предоставляемые системой обобщенных типов.

Для большей конкретности рассмотрим следующий фрагмент кода:

```
// Почему создание обобщенных массивов
// некорректно - не компилируется!
List<String>[] stringLists = new List<String>[1]; // (1)
List<Integer> intList = List.of(42); // (2)
Object[] objects = stringLists; // (3)
```

```
objects[0] = intList; // (4)
String s = stringLists[0].get(0); // (5)
```

Представим, что строка 1, создающая обобщенный массив, разрешена. Строка 2 создает и инициализирует `List<Integer>`, содержащий единственный элемент. Строка 3 сохраняет массив `List<String>` в переменную массива `Object`, что разрешено, потому что массивы ковариантны. Строка 4 сохраняет `List<Integer>` в единственный элемент массива `Object`, что тоже закончится удачно, потому что обобщенные типы реализуются затиранием: типом времени выполнения экземпляра `List<Integer>` является просто `List`, а типом времени выполнения экземпляра `List<String>[]` является `List[]`, так что данное присваивание не приводит к генерации исключения `ArrayStoreException`. Далее мы получаем проблему. Мы сохранили экземпляр `List<Integer>` в массив, который объявлен как хранящий только экземпляры `List<String>`. В строке 5 мы выводим единственный элемент из единственного списка в этом массиве. Компилятор автоматически преобразует извлеченный элемент в `String`, но на самом деле это `Integer`, так что мы получим при выполнении исключение `ClassCastException`. Чтобы такого не случилось, строка 1 (создающая обобщенный массив) вызывает ошибку компиляции.

Типы, такие как `E`, `List<E>` и `List<String>`, технически известны как *недоступные во время выполнения* (*nonreifiable*) типы [25, 4.7]. Говоря интуитивно, недоступные во время выполнения типы — это типы, представление времени выполнения которых содержит меньше информации, чем представление времени компиляции. Из-за затирания единственными параметризованными типами, доступными во время выполнения, являются неограниченные типы с символами подстановки, такие как `List<?>` и `Map<?, ?>` (раздел 5.1). Разрешено, хотя и не слишком полезно, создание массивов неограниченных типов с символами подстановки.

Запрет создания обобщенных массивов может вызывать раздражение. Он означает, что в общем случае обобщенная коллекция не может вернуть массив с элементами этого типа (но в разделе 5.8 приведено частичное решение этой проблемы). Это также означает, что вы можете получить запутанные предупреждения при использовании методов с переменным количеством аргументов (раздел 8.5) в сочетании с обобщенными типами. Дело в том, что каждый раз при вызове метода с переменным количеством аргументов создается массив для хранения параметров метода. Если тип элемента этого массива не доступен при выполнении, вы получите предупреждение. Для решения этого вопроса можно воспользоваться аннотацией `SafeVarargs` (раздел 5.7).

Если вы получаете ошибку создания обобщенного массива или предупреждение о непроверяемом приведении к типу массива, наилучшим решением

зачастую оказывается использование типа коллекции `List<E>` вместо типа массива `E[]`. Вы можете частично пожертвовать производительностью или краткостью, но взамен получите лучшие безопасность типов и их взаимодействие.

Например, предположим, что вы хотите написать класс `Chooser` с конструктором, который принимает коллекцию, и с одним методом, который случайным образом возвращает элемент коллекции. В зависимости от того, какая коллекция передана конструктору, можно использовать объект класса в качестве, например, игровой матрицы или источника данных при моделировании методом Монте-Карло. Вот упрощенная реализация класса без обобщенных типов:

```
// Chooser – класс, нуждающийся в обобщенных типах!
public class Chooser
{
    private final Object[] choiceArray;
    public Chooser(Collection choices)
    {
        choiceArray = choices.toArray();
    }
    public Object choose()
    {
        Random rnd = ThreadLocalRandom.current();
        return choiceArray[rnd.nextInt(choiceArray.length)];
    }
}
```

Чтобы использовать этот класс, следует выполнять приведение возвращаемого методом `choose` значения из `Object` к требуемому типу каждый раз при вызове метода при получении неверного типа все закончится ошибкой времени выполнения. Прислушавшись к совету из раздела 5.4, мы пытаемся изменить класс `Chooser`, сделав его обобщенным. Внесенные изменения показаны полужирным шрифтом:

```
// Первая (некомпилируемая) попытка
// сделать Chooser обобщенным классом
public class Chooser<T>
{
    private final T[] choiceArray;
    public Chooser(Collection<T> choices)
    {
        choiceArray = choices.toArray();
    }
    // Метод choose не изменен
}
```

Если вы попытаетесь скомпилировать этот класс, то получите сообщение об ошибке:

```
Chooser.java:9: error: incompatible types: Object[] cannot be
converted to T[]
    choiceArray = choices.toArray();
                        ^
    where T is a type-variable:
      T extends Object declared in class Chooser
```

“Не страшно, — скажете вы. — Преобразуем массив `Object` в массив `T`”:

```
choiceArray = (T[]) choices.toArray();
```

Это избавляет от ошибки, но вместо этого вы получите предупреждение:

```
Chooser.java:9: warning: [unchecked] unchecked cast
    choiceArray = (T[]) choices.toArray();
                        ^
    required: T[], found: Object[]
    where T is a type-variable:
      T extends Object declared in class Chooser
```

Компилятор сообщает, что он не может проверить безопасность приведения во время выполнения, так как не знает, что собой будет представлять тип `T` — вспомните, что информация о типах элементов при выполнении затирается. Будет ли работать программа? Да, но компилятор не в состоянии это доказать. Вы можете доказать это для себя самостоятельно, поместить доказательство в комментарий и подавить вывод предупреждений с помощью аннотации, но лучше полностью устранить причину предупреждения (раздел 5.2).

Для устранения предупреждения о непроверяемом приведении используйте вместо массива список. Вот вариант класса `Chooser`, который будет компилироваться без ошибок и предупреждений:

```
// Chooser на основе списка: безопасен с точки зрения типов
public class Chooser<T>
{
    private final List<T> choiceList;
    public Chooser(Collection<T> choices)
    {
        choiceList = new ArrayList<>(choices);
    }
    public T choose()
    {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

Эта версия гораздо более многословна, чем версия с использованием массивов, но ваше спокойствие того стоит. Теперь мы точно знаем, что исключения `ClassCastException` при выполнении не будет.

Резюмируя, можно сказать, что массивы и обобщенные типы имеют различные правила. Массивы ковариантны и доступны во время выполнения, обобщенные типы — инвариантны и используют механизм затирания. Вследствие этого массивы обеспечивают безопасность времени выполнения с точки зрения типов, но не безопасность типов при компиляции, и наоборот. Как правило, массивы и обобщенные типы плохо “смешиваются” в одной программе. Если вам все же придется прибегнуть к такому смешению и получить при этом ошибки или предупреждения времени компиляции, то, в первую очередь, постарайтесь заменить массивы списками.

5.4. Предпочитайте обобщенные типы

Обычно не слишком трудно параметризовать объявления и использовать обобщенные типы и методы, предоставляемые JDK. Написание собственных обобщенных типов оказывается немного более сложным, но оно стоит усилий, затраченных на обучение.

Рассмотрим простую (игрушечную) реализацию стека из раздела 2.7:

```
// Коллекция на основе Object - основной кандидат на обобщение
public class Stack
{
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(Object e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop()
    {
        if (size == 0)
            throw new EmptyStackException();

        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
    }
}
```

```

        return result;
    }
    public boolean isEmpty()
    {
        return size == 0;
    }
    private void ensureCapacity()
    {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}

```

Этот класс должен был быть параметризован с самого начала, но так как это не было сделано, мы можем его *обобщить* постфактум. Другими словами, его можно параметризовать без вреда для клиентов, использующих исходную, не параметризованную версию. В имеющемся виде клиент должен выполнять приведение объектов, извлекаемых из стека, и это может привести к сбоям времени выполнения. Первый шаг в обобщении класса состоит в добавлении в объявление одного или нескольких параметров типа. В данном случае имеется единственный параметр типа, представляющий тип элемента стека, и обычно для такого параметра типа используется имя *E* (раздел 9.12).

Следующий шаг состоит в замене всех использований типа *Object* соответствующим параметром типа и попыткой скомпилировать получаемую в результате программу:

```

// Первоначальная (некомпилируемая) попытка обобщения Stack
public class Stack<E>
{
    private E[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    public Stack()
    {
        elements = new E[DEFAULT_INITIAL_CAPACITY];
    }
    public void push(E e)
    {
        ensureCapacity();
        elements[size++] = e;
    }
    public E pop()
    {
        if (size == 0)
            throw new EmptyStackException();
    }
}

```

```

    E result = elements[--size];
    elements[size] = null; // Удаление устаревшей ссылки
    return result;
}
... // никаких изменений в isEmpty или ensureCapacity
}

```

Обычно вы получаете как минимум одну ошибку или предупреждение, и этот класс — не исключение. К счастью, в данном случае ошибка только одна:

```

Stack.java:8: generic array creation
    elements = new E[DEFAULT_INITIAL_CAPACITY];
                ^

```

Как объясняется в разделе 5.3, вы не можете создавать массив недоступных при выполнении типов, таких как `E`. Эта проблема возникает всякий раз, когда вы пишете обобщенный тип на основе массивов. Есть два варианта решения проблемы. Первое решение непосредственно обходит запрет на создание обобщенных массивов: создайте массив для `Object` и передайте его обобщенному типу массива. Теперь вместо ошибки компилятор выдаст предупреждение. Такое использование разрешено, но (в общем случае) не является безопасным с точки зрения типов:

```

Stack.java:8: warning: [unchecked] unchecked cast
found: Object[], required: E[]
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
                ^

```

Компилятор может быть не способен доказать, что ваша программа безопасна, но это можете сделать вы. Убедитесь, что непроверяемое приведение не нарушит безопасность типов программы. Рассматриваемый массив (`elements`) хранится в закрытом поле, никогда не возвращается клиенту и не передается никакому методу. Единственные элементы, хранящиеся в массиве, — это элементы, передаваемые в метод `push` (которые имеют тип `E`), так что непроверяемое приведение не может причинить вреда.

После того как вы убедились, что непроверяемое приведение безопасно, подавите вывод предупреждений в самой узкой, насколько возможно, области видимости (раздел 5.2). В данном случае конструктор содержит только непроверяемое создание массива, так что вполне можно скрыть предупреждения во всем конструкторе. После добавления соответствующей аннотации `Stack` отлично компилируется, и вы можете использовать его без явных приведений и боязни исключений `ClassCastException`:

```

// Массив elements содержит только экземпляры E из push(E).
// Этого достаточно для гарантии безопасности типов, но тип

```

```
// времени выполнения массива не может быть E[]; он всегда
// должен быть Object[]!
@SuppressWarnings("unchecked")
public Stack()
{
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

Второй способ устранения ошибки создания обобщенного массива в Stack заключается изменении типа поля `elements` с `E[]` на `Object[]`. Если поступить таким образом, можно получить другую ошибку:

```
Stack.java:19: incompatible types
found: Object, required: E
    E result = elements[--size];
                ^
```

Эту ошибку можно изменить на предупреждение, если выполнить приведение полученного из массива элемента к `E`:

```
Stack.java:19: warning: [unchecked] unchecked cast
found: Object, required: E
    E result = (E) elements[--size];
                ^
```

Поскольку `E` является типом, недоступным при выполнении, компилятор не может выполнить проверку приведения при выполнении. Вы снова можете легко доказать, что непроверяемая передача в данном случае безопасна, поэтому можно просто подавить предупреждения. Следуя совету из раздела 5.2, мы убираем предупреждения только для присваивания, которое содержит непроверяемое приведение, а не для всего метода `pop`:

```
// Подавление предупреждения о непроверяемой операции
public E pop()
{
    if (size == 0)
        throw new EmptyStackException();

    // push требует, чтобы элементы были типа E,
    // поэтому приведение корректно
    @SuppressWarnings("unchecked") E result =
        (E) elements[--size];

    elements[size] = null; // Удаление устаревшей ссылки
    return result;
}
```

Оба варианта удаления создания обобщенных массивов имеют своих последователей. Первый способ более удобочитаем: массив объявляется как `E[]`, ясно указывая, что он содержит только экземпляры `E`. Он также более краток: в типичных обобщенных классах вы читаете массив во многих местах кода; первый метод требует только одного приведения (при создании массива), в то время как второй требует отдельного приведения каждый раз, когда выполняется чтение элемента массива. Таким образом, первый способ выглядит предпочтительным и наиболее часто используется на практике. Однако он приводит к *загрязнению кучи* (heap pollution, раздел 5.7): тип времени выполнения массива не совпадает с его типом времени компиляции (если только `E` не оказывается `Object`). Это приводит к тому, что некоторые достаточно привередливые программисты выбирают второй способ, хотя загрязнение кучи в этой ситуации безвредно.

Следующая программа демонстрирует использование обобщенного класса `Stack`. Программа выводит аргументы командной строки в обратном порядке и переводит их в верхний регистр. Для вызова метода `toUpperCase` класса `String` с элементами, снимаемыми со стека, явное приведение не требуется, а автоматически генерируемое приведение гарантированно успешное:

```
// Небольшая программа с использованием обобщенного Stack
public static void main(String[] args)
{
    Stack<String> stack = new Stack<>();

    for (String arg : args)
        stack.push(arg);

    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

Может показаться, что приведенный выше пример противоречит разделу 5.3, который поощряет использование списков вместо массивов. Но не всегда возможно или желательно использовать списки внутри обобщенных типов. Java не поддерживает списки как фундаментальные структуры данных, поэтому некоторые обобщенные типы, такие как `ArrayList`, *вынуждены* быть реализованными поверх массивов. Другие обобщенные типы, такие как `HashMap`, реализуются поверх массивов для достижения высокой производительности.

Подавляющее большинство обобщенных типов подобны нашему примеру `Stack` в том, что их параметры типа не имеют ограничений: можно создать `Stack<Object>`, `Stack<int[]>`, `Stack<List<String>>` или `Stack` любых других объектов ссылочного типа. Обратите внимание, что нельзя создать `Stack` примитивного типа: попытка создания `Stack<int>` или `Stack<double>` приведет к ошибке времени компиляции. Это

фундаментальное ограничение системы обобщенных типов Java. Его можно обойти с помощью упаковки примитивных типов (раздел 9.5).

Есть некоторые обобщенные типы, которые ограничивают допустимые значения своих параметров типа. Например, рассмотрим `java.util.concurrent.DelayQueue`, объявление которого выглядит следующим образом:

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>
```

Список параметров типа (`<E extends Delayed>`) требует, чтобы фактический параметр типа `E` был подтипом типа `java.util.concurrent.Delayed`. Это позволяет реализации `DelayQueue` и ее клиентам воспользоваться преимуществами методов `Delayed` для элементов `DelayQueue` без необходимости явного приведения или риска `ClassCastException`. Параметр типа `E` известен как *ограниченный параметр типа* (`bounded type parameter`). Обратите внимание, что отношение подтипа определяется таким образом, что каждый тип является подтипом самого себя [25, 4.10], поэтому создание `DelayQueue<Delayed>` вполне корректно.

Итак, обобщенные типы безопаснее и проще в использовании, чем типы, которые требуют приведений в клиентском коде. При разработке новых типов убедитесь, что они могут быть использованы без таких приведений. Часто это будет означать, что типы следует делать обобщенными. Если у вас есть некоторые существующие типы, которые должны быть обобщенными, но не являются таковыми, обобщите их. Это упростит жизнь новым пользователям этих типов и не нарушит работоспособность существующих клиентов (раздел 5.1).

5.5. Предпочитайте обобщенные методы

Так же, как и классы, методы могут быть обобщенными. Статические вспомогательные методы, которые работают с параметризованными типами, обычно являются обобщенными. Все методы “алгоритмов” в `Collection` (такие, как `binarySearch` и `sort`) являются обобщенными.

Написание обобщенных методов подобно написанию обобщенных типов. Рассмотрим следующий метод, который возвращает объединение двух множеств:

```
// Использование несформированных типов неприемлемо (раздел 5.1)
public static Set union(Set s1, Set s2)
{
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

Этот метод компилируется с двумя предупреждениями:

```
Union.java:5: warning: [unchecked] unchecked call to
HashSet(Collection<? extends E>) as a member of raw type HashSet
    Set result = new HashSet(s1);
                  ^
```

```
Union.java:6: warning: [unchecked] unchecked call to
addAll(Collection<? extends E>) as a member of raw type Set
    result.addAll(s2);
                ^
```

Чтобы устранить эти предупреждения и сделать метод безопасным с точки зрения типов, изменим объявление так, чтобы в нем был *параметр типа*, представляющий тип элемента всех трех множеств (двух аргументов и возвращаемого значения), и используем этот параметр типа во всем методе. **Список параметров типа, который объявляет параметры типа, находится между модификаторами метода и его возвращаемым типом.** В данном примере список параметров типа — `<E>`, а возвращаемый тип — `Set<E>`. Соглашение об именовании параметров типа одинаково для обобщенных методов и обобщенных типов (разделы 5.4, 9.12).

```
// Обобщенный метод
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
{
    Set<E> result = new HashSet<>(s1);
    result.addAll(s2);
    return result;
}
```

По крайней мере для простых обобщенных методов это все, что нужно. Этот метод компилируется без каких-либо предупреждений и обеспечивает безопасность типов, а также простоту использования. Вот простая программа, демонстрирующая применение этого метода. Она не содержит приведенных и компилируется без ошибок или предупреждений:

```
// Простая программа, использующая обобщенный метод
public static void main(String[] args)
{
    Set<String> guys = Set.of("Tom", "Dick", "Harry");
    Set<String> stooges = Set.of("Larry", "Moe", "Curly");
    Set<String> aflCio = union(guys, stooges);
    System.out.println(aflCio);
}
```

При запуске программа выводит

```
[Moe, Tom, Harry, Larry, Curly, Dick]
```

(Порядок этих элементов в выводе зависит от реализации.)

Ограничение метода `union` заключается в том, что типы всех трех множеств (входные параметры и возвращаемое значение) должны быть в точности одинаковы. Метод можно сделать более гибким с помощью *ограниченных типов с символами подстановки* (раздел 5.6).

В некоторых случаях нужно будет создать объект, являющийся неизменяемым, но применимый ко многим различным типам. Поскольку обобщенные типы реализуются с использованием затирания (раздел 5.3), можно использовать один объект для всех требуемых параметризаций типов, но нужно написать статический фабричный метод для многократной раздачи объектов для каждой запрошенной параметризации типа. Такой проектный шаблон, именуемый *обобщенной фабрикой синглтонов* (*generic singleton factory*), используется для функциональных объектов (раздел 7.1), таких как `Collections.reverseOrder`, а иногда и для коллекций, таких как `Collections.emptySet`.

Предположим, что вы хотите написать раздатчик для функций тождественности (идентичности). Библиотеки предоставляют `Function.identity`, поэтому писать собственный метод нет никаких оснований (раздел 9.3), но это весьма поучительно. Было бы расточительно создавать новый объект функции тождественности при каждом запросе, потому что это объект без сохранения состояния. Если бы обобщенные типы Java были типами, доступными при выполнении, вам потребовалась бы единственная функция тождественности для каждого типа, но из-за затирания будет достаточно обобщенного синглтона. Вот как он выглядит:

```
// Проектный шаблон обобщенной фабрики синглтонов
private static UnaryOperator<Object> IDENTITY_FN = (t) -> t;

@SuppressWarnings("unchecked")
public static <T> UnaryOperator<T> identityFunction()
{
    return (UnaryOperator<T>) IDENTITY_FN;
}
```

Приведение `IDENTITY_FN` к `(UnaryFunction<T>)` генерирует предупреждение о непроверяемом приведении, так как `UnaryOperator<Object>` не является `UnaryOperator<T>` для каждого `T`. Но функция тождества особая: она возвращает неизменный аргумент, поэтому мы знаем, что использовать его как `UnaryFunction<T>`, независимо от значения `T`, безопасно с точки зрения типов. Таким образом, мы можем уверенно подавить вывод предупреждений, генерируемых этим приведением. После того как мы это сделаем, код будет компилироваться без ошибок или предупреждений.

Вот пример программы, которая использует наш обобщенный синглтон и как `UnaryOperator<String>`, и как `UnaryOperator<Number>`. Как

обычно, в этом коде нет приведений, и он компилируется без ошибок или предупреждений:

```
// Пример программы, использующей обобщенный синглтон
public static void main(String[] args)
{
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryOperator<String> sameString = identityFunction();

    for (String s : strings)
        System.out.println(sameString.apply(s));

    Number[] numbers = { 1, 2.0, 3L };
    UnaryOperator<Number> sameNumber = identityFunction();

    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

Допустимо, хотя и сравнительно редко используется, чтобы параметр типа был ограничен некоторым выражением с участием самого этого параметра типа (известно как *рекурсивное ограничение типа* (recursive type bound)). Рекурсивное ограничение типа обычно используется в связи с интерфейсом `Comparable`, который определяет естественное упорядочение типа (раздел 3.5). Этот интерфейс показан далее:

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

Параметр типа `T` определяет тип, с которым можно сравнивать элементы типа, реализующего `Comparable<T>`. На практике почти все типы можно сравнить только с элементами их собственного типа. Так, например, `String` реализует `Comparable<String>`, `Integer` реализует `Comparable<Integer>` и т.д.

Многие методы принимают коллекцию элементов, реализующих `Comparable`, для ее сортировки, поиска внутри нее, вычисления минимума или максимума и т.п. Чтобы все это делать, необходимо, чтобы каждый элемент коллекции был сопоставим с любым другим ее элементом, иными словами, чтобы элементы списка были *взаимно сравниваемыми*. Вот как выразить это ограничение:

```
// Применение рекурсивного ограничения типа
// для выражения взаимной сравниваемости
public static <E extends Comparable<E>> E max(Collection<E> c);
```

Ограничение типа `<E extends Comparable<E>>` можно прочесть как “любой тип E, который можно сравнивать с самим собой”, что более или менее точно соответствует понятию взаимной сравниваемости.

Вот метод, который можно использовать с предыдущим объявлением. Он вычисляет максимальное значение в коллекции согласно естественному порядку ее элементов, и компилируется без ошибок или предупреждений:

```
// Возвращает максимальное значение коллекции -
// использует рекурсивное ограничение типа
public static <E extends Comparable<E>> E max(Collection<E> c)
{
    if (c.isEmpty())
        throw new IllegalArgumentException("Пустая коллекция");

    E result = null;

    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return result;
}
```

Обратите внимание, что этот метод генерирует исключение `IllegalArgumentException`, если список пуст. Лучшей альтернативой был бы возврат `Optional<E>` (раздел 8.7).

Рекурсивные ограничения типов могут быть гораздо более сложными, но, к счастью, с такими приходится встречаться очень редко. Если вы понимаете эту идиому, ее вариант с подстановочными символами (раздел 5.6) и идиому *имитации собственного типа* (раздел 2.2), вы будете способны справиться с большинством рекурсивных ограничений типов, с которыми столкнетесь на практике.

Резюмируя, можно сказать, что обобщенные методы, подобно обобщенным типам, безопаснее и проще в использовании, чем методы, требующие от своих клиентов использовать явные приведения входных параметров и возвращаемых значений. Как и при работе с типами, вам следует убедиться, что ваши методы могут использоваться без приведения, а это часто означает, что они должны быть обобщенными. И, как и в случае типов, следует обобщать существующие методы, которые требуют приведений при использовании. Это упрощает жизнь новым пользователям и не нарушает работоспособность существующих клиентов (раздел 5.1).

5.6. Используйте ограниченные символы подстановки для повышения гибкости API

Как отмечалось в разделе 5.3, параметризованные типы являются *инвариантными*. Другими словами, для любых двух различных типов `Type1` и `Type2`, тип `List<Type1>` не является ни подтипом, ни супертипом для `List<Type2>`. Хотя то, что `List<String>` не является подтипом типа `List<Object>`, кажется противоречащим интуиции, это действительно имеет смысл. Вы можете поместить любой объект в `List<Object>`, но в `List<String>` вы можете поместить только строки. Поскольку `List<String>` не может делать все, что делает `List<Object>`, он не является его подтипом (согласно принципу подстановки Лисков, раздел 3.1).

Иногда требуется большая гибкость, чем та, которую могут предоставить инвариантные типы. Рассмотрим класс `Stack` из раздела 5.4. Чтобы освежить свою память, взгляните на его открытый API:

```
public class Stack<E>
{
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Предположим, что мы хотим добавить метод, который принимает последовательность элементов и вносит их в стек. Вот первая попытка:

```
// Метод pushAll без типа с символами
// подстановки - этого недостаточно!
public void pushAll(Iterable<E> src)
{
    for (E e : src)
        push(e);
}
```

Этот метод компилируется без ошибок, но не является полностью удовлетворительным. Если тип элемента `Iterable src` в точности соответствует типу элементов стека, метод прекрасно работает. Но предположим, что у вас `Stack<Number>`, а вызывается `push(intVal)`, где `intVal` имеет тип `Integer`. Это сработает, потому что `Integer` является подтипом `Number`. Казалось бы логичным, что должен работать и этот код:

```
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Однако если вы попытаете это сделать, то получите сообщение об ошибке, потому что параметризованные типы являются инвариантными:

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
    numberStack.pushAll(integers);
                        ^
```

К счастью, выход есть. Для выхода из подобных ситуаций язык предоставляет особый вид параметризованного типа, именуемый *ограниченным типом с символами подстановки* (bounded wildcard type). Тип входного параметра `pushAll` должен быть не “`Iterable от E`”, а “`Iterable некоторого подтипа E`”, и есть такой тип с символом подстановки, который означает именно это: `Iterable<? extends E>`. (Ключевое слово `extends` немного вводит в заблуждение: вспомните из раздела 5.4, что *подтип* определяется так, что каждый тип является подтипом самого себя, несмотря на то, что он сам себя не расширяет.) Изменим `pushAll` для использования этого типа:

```
// Тип с символом подстановки в параметре,
// служащем в качестве производителя E
public void pushAll(Iterable <? extends E > src)
{
    for (E e : src)
        push(e);
}
```

При этом изменении без замечаний происходит компиляция не только `Stack`, но и кода клиента, который не компилировался с исходным объявлением `pushAll`. Поскольку компиляция `Stack` и его клиента не приводит ни к каким предупреждениям, вы знаете, что получившийся код безопасен с точки зрения типов.

Теперь предположим, что вы хотите написать метод `popAll`, работающий в паре с `pushAll`. Метод `popAll` извлекает каждый элемент из стека и добавляет его в заданную коллекцию. Вот как могут выглядеть первые попытки написания метода `popAll`:

```
// Метод popAll без применения типов с символами
// подстановки – этого недостаточно!
public void popAll(Collection<E> dst)
{
    while (!isEmpty())
        dst.add(pop());
}
```

Вновь метод компилируется без ошибок и отлично работает, если тип элементов целевой коллекции в точности соответствует типу элементов стека. Но

вновь это не является полностью удовлетворительным решением. Предположим, что имеются `Stack<Number>` и переменная типа `Object`. Если снимать элементы со стека и сохранять их в переменной, код компилируется и выполняется без ошибок. Так почему бы все так и не оставить?

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

Если попытаться скомпилировать этот клиентский код с показанной выше версией `popAll`, то будет получена ошибка, очень похожая на ту, которую мы получили в первой версии `pushAll: Collection<Object>` не является подтипом типа `Collection<Number>`. И вновь нас спасают символы подстановки. Тип входного параметра `popAll` должен быть не “коллекция E”, а “коллекция некоторого супертиска E” (где супертиск определен так, что E является супертиском для самого себя [25, 4.10]). И вновь имеется тип с символом подстановки, который означает в точности то, что нам надо: `Collection<? super E>`. Давайте изменим `popAll` соответствующим образом:

```
// Тип с символом подстановки для параметра,
// который служит потребителем E
public void popAll(Collection <? super E > dst)
{
    while (!isEmpty())
        dst.add(pop());
}
```

При таком изменении и код класса `Stack`, и клиентский код компилируются без каких-либо замечаний.

Урок очевиден. Для максимальной гибкости для входных параметров, представляющих производителей или потребителей, следует использовать типы с символами подстановки. Если входной параметр является и производителем, и потребителем, то типы с символами подстановки не принесут никакой пользы: здесь требуется точное совпадение типов, а это то, что получается без применения символов подстановки.

Мнемоническое правило на английском языке, помогающее запомнить, какой символ подстановки использовать, звучит так:

PECS — producer-extends, consumer-super
(производитель — extends, потребитель — super)

Другими словами, если параметризованный тип представляет производителя T, используйте `<? extends T>`; если он представляет потребителя T, используйте `<? super T>`. В нашем примере `Stack` параметр `src` метода `pushAll` производит экземпляры E для использования стеком, поэтому соответствующий

тип для `src` имеет вид `Iterable<? extends E>`; параметр `dst` метода `addAll` потребляет экземпляры `E` из стека, поэтому для `dst` соответствующий тип имеет вид `Collection<? super E>`. Правило PECS мнемонически отражает основополагающий принцип, определяющий использование типов с символами подстановки. Нафталин (Naftalin) и Уадлер (Wadler) называют его принципом “*взять и отдать*” (Get and Put Principle) [34, 2.4].

С учетом этого мнемонического правила давайте посмотрим на некоторые объявления методов и конструкторов из предыдущих разделов этой главы. Конструктор `Chooser` в разделе 5.3 объявлен следующим образом:

```
public Chooser(Collection<T> choices)
```

Этот конструктор использует коллекцию `choices` только для **производства** значений типа `T` (и их сохранения для дальнейшего использования), так что его объявление должно использовать тип с символами подстановки, который **расширяет T**. Ниже приведено объявление получающегося в результате конструктора:

```
// Тип с символом подстановки для параметра,  
// который служит производителем T  
public Chooser(Collection<? extends T> choices)
```

Будет ли это изменение значить что-либо на практике? Да, будет. Предположим, у вас есть `List<Integer>` и вы хотите передать его в конструктор `Chooser<Number>`. Такой код не скомпилируется при исходном объявлении, но при добавлении ограниченного типа с символами подстановки все будет иначе.

Теперь давайте взглянем на метод `union` из раздела 5.5. Вот его объявление:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Оба параметра, и `s1`, и `s2`, представляют собой производителей `E`, так что мнемоника PECS говорит нам о том, что объявление должно принять следующий вид:

```
public static <E> Set<E> union(Set<? extends E> s1,  
                             Set<? extends E> s2)
```

Обратите внимание, что возвращаемый тип — по-прежнему `Set<E>`. **Не используйте ограниченные типы с символами подстановки в качестве возвращаемых типов.** Вместо дополнительной гибкости для ваших пользователей это заставит их использовать типы с символами подстановки в клиентском коде. При использовании нового объявления следующий код будет компилироваться без ошибок:

```
Set<Integer> integers = Set.of(1, 3, 5);
Set<Double> doubles = Set.of(2.0, 4.0, 6.0);
Set<Number> numbers = union(integers, doubles);
```

При надлежащем использовании типы с символами подстановки почти невидимы для пользователей класса. Они заставляют методы принимать параметры, которые они должны принимать, и отвергать те, которые они должны отвергать. **Если пользователь класса вынужден думать о типах с символами подстановки, вероятно, что-то неправильно с API этого класса.**

До Java 8 правила вывода типов были недостаточно умны, чтобы обработать предыдущий фрагмент кода, который требует от компилятора использовать контекстуально определенный тип возвращаемого значения (или *целевой тип*) для вывода типа E. Целевой тип вызова `union`, показанный ранее, представляет собой `Set<Number>`. При попытке компиляции фрагмента в более ранней версии Java (с соответствующей заменой фабрики `Set.of`) вы получите длинное, запутанное сообщение об ошибке наподобие следующего:

```
Union.java:14: error: incompatible types
    Set<Number> numbers = union(integers, doubles);
                                ^
required: Set<Number>
found:     Set<INT#1>
where INT#1,INT#2 are intersection types:
    INT#1 extends Number,Comparable<? extends INT#2>
    INT#2 extends Number,Comparable<?>
```

К счастью, есть способ справиться с такого рода ошибками. Если компилятор не выводит правильный тип, вы всегда можете сказать ему, какой тип использовать, с помощью *явного аргумента типа* (*explicit type argument*) [25, 15.12]. Даже до введения ожидаемых типов (*target type*) в Java 8 это не приходилось делать часто (и это очень хорошо, потому что явные аргументы типа выглядят не очень красиво). С добавлением явного аргумента типа, как показано далее, фрагмент кода будет компилироваться без замечаний в версиях, предшествующих Java 8:

```
// Явный параметр типа – требовался до Java 8
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Теперь обратим внимание на метод `max` из раздела 5.5. Вот исходное объявление:

```
public static <T extends Comparable<T>> T max(List<T> list)
```

А вот — переделанное объявление с использованием типов с символами подстановки:

```
public static <T extends Comparable<? super T>> T max(
    List<? extends T> list)
```

Чтобы получить переделанное объявление из оригинала, мы дважды прибегаем к эвристике PECS. Прямое применение — к параметру `list`. Он производит экземпляр `T`, поэтому мы изменяем тип `List<T>` на `List<? extends T>`. Применение к параметру типа `T` сложнее. Это первый случай, когда мы встречаемся с применением символов подстановки к параметру типа. Первоначально `T` был определен как расширяющий `Comparable<T>`, но сравнение `T` потребляет экземпляры `T` (и производит целые числа, указывающие отношение порядка). Таким образом, параметризованный тип `Comparable<T>` заменяется ограниченным типом с символами подстановки `Comparable<? super T>`. Сравнения всегда являются потребителями, поэтому в общем случае следует использовать **`Comparable<? super T>`**, а не **`Comparable<T>`**. То же самое верно для компараторов — в общем случае следует использовать **`Comparator<? super T>`**, а не **`Comparator<T>`**.

Пересмотренное объявление `max`, вероятно, — наиболее сложное объявление метода в этой книге. Дает ли эта сложность что-то в действительности? Да, дает. Вот простой пример списка, с которым не будет работать исходное объявление, но который разрешен пересмотренным объявлением:

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

Причина того, что исходное объявление метода не работает с таким списком, — `ScheduledFuture` не реализует `Comparable<ScheduledFuture>`. Вместо этого он является подынтерфейсом `Delayed`, который расширяет `Comparable<Delayed>`. Другими словами, экземпляр `ScheduledFuture` не просто сравним с другими экземплярами `ScheduledFuture`; он сравним с любым экземпляром `Delayed`, и этого достаточно, чтобы заставить исходное объявление его отклонить. В общем случае символы подстановки требуются для поддержки типов, которые не реализуют `Comparable` (или `Comparator`) непосредственно, но расширяют тип, который это делает.

Есть еще одна тема, связанная с символами подстановки, которую стоит обсудить. Имеется дуальность между параметрами типа и символами подстановки, и многие методы могут быть объявлены с использованием тех или других. Например, вот два возможных объявления статического метода для замены двух индексированных элементов списка. Первый использует неограниченный параметр типа (раздел 5.5), а второй — неограниченный символ подстановки:

```
// Два возможных объявления метода swap
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Какое из этих двух объявлений предпочтительнее и почему? В открытом API второе объявление лучше, потому что оно проще. Вы передаете список — любой список — и метод меняет местами индексированные элементы. Нет параметра типа, о котором нужно беспокоиться. Как правило, **если параметр типа появляется в объявлении метода только один раз, его следует заменить символом подстановки**. Если это параметр неограниченного типа, замените его неограниченным символом подстановки; если это параметр ограниченного типа, замените его ограниченным символом подстановки.

Во втором объявлении метода `swap` имеется одна проблема. Простая реализация не скомпилируется:

```
public static void swap(List<?> list, int i, int j)
{
    list.set(i, list.set(j, list.get(i)));
}
```

Попытка компиляции приводит к следующему малополезному сообщению об ошибке:

```
Swap.java:5: error: incompatible types: Object cannot be
converted to CAP#1
    list.set(i, list.set(j, list.get(i)));
                        ^
    where CAP#1 is a fresh type-variable:
        CAP#1 extends Object from capture of ?
```

Кажется неправильным, что мы не можем поместить элемент обратно в список, из которого мы только что его взяли. Проблема заключается в том, что типом `list` является `List<?>`, а в `List<?>` нельзя поместить никакое значение, кроме `null`. К счастью, есть способ реализовать этот метод, не прибегая к небезопасным приведениям или несформированным типам. Идея заключается в том, чтобы написать вспомогательный закрытый метод для *захвата* типа с символом подстановки. Чтобы захватить тип, вспомогательный метод должен быть обобщенным. Вот как он выглядит:

```
public static void swap(List<?> list, int i, int j)
{
    swapHelper(list, i, j);
}
// Закрытый вспомогательный метод для захвата символа подстановки
private static <E> void swapHelper(List<E> list, int i, int j)
{
    list.set(i, list.set(j, list.get(i)));
}
```

Метод `swapHelper` знает, что `list` представляет собой `List<E>`. Таким образом, он знает, что любое значение, которое он получает из этого списка, имеет тип `E` и что безопасно внести любое значение типа `E` в список. Эта немного запутанная реализация `swap` компилируется без замечаний. Это позволяет нам экспортировать объявление на основе символа подстановки, в то же время получая выгоду от более сложного обобщенного метода внутри. Клиенты метода `swap` не должны сталкиваться с объявлением более сложного метода `swapHelper`, но они от этого выигрывают. Стоит отметить, что вспомогательный метод имеет точно ту же сигнатуру, которую мы отвергли как слишком сложную для открытого метода.

Подытожим сказанное. Использование типов с символами подстановки в вашем API, будучи более сложным, делает API гораздо более гибким. Если вы пишете библиотеку, которая будет широко использоваться, надлежащее применение типов с символами подстановки следует рассматривать как обязательное. Помните основное правило: производитель — `extends`, потребитель — `super` (PECS). Помните также, что все сравнения и компараторы² являются потребителями.

5.7. Аккуратно сочетайте обобщенные типы и переменное количество аргументов

И методы с переменным количеством аргументов (раздел 8.5), и обобщенные типы были добавлены в платформе Java 5, так что следовало бы ожидать их корректного взаимодействия; к сожалению, это не так. Цель переменного количества аргументов — позволить клиентам передавать методу переменное количество аргументов, но эта технология представляет собой *утечку абстракции*: при вызове такого метода создается массив для хранения параметров; этот массив, который должен бы быть деталью реализации, оказывается видимым. Как результат вы получаете запутанные предупреждения компилятора, когда параметры имеют обобщенные или параметризованные типы.

Вспомните из раздела 5.3, что недоступный во время выполнения тип — это тип, представление времени выполнения которого содержит меньше информации, чем представление времени компиляции, и что почти все обобщенные и параметризованные типы являются недоступными во время выполнения. Если метод объявляет параметр переменной длины недоступного во время выполнения типа, компилятор выдает для такого объявления

² Здесь автор имеет в виду интерфейсы `Comparable` и `Comparator` для реализации сравнений. — *Примеч. ред.*

предупреждение. Если метод вызывается с переменным числом параметров, выведенные типы которых недоступны во время выполнения, компилятор выдает предупреждение для такого вызова. Эти предупреждения выглядят примерно следующим образом:

```
warning: [unchecked] Possible heap pollution from
parameterized vararg type List<String>
```

Когда переменная параметризованного типа ссылается на объект, не принадлежащий этому типу, происходит *загрязнение кучи* (heap pollution) [25, 4.12.2]. Это может привести к сбою автоматически сгенерированных компилятором приведений, что нарушает основные гарантии системы обобщенных типов.

Например, рассмотрим следующий метод, который представляет собой немного замаскированный вариант фрагмента кода из раздела 5.3:

```
// Смесь обобщенных типов и переменного количества
// аргументов может нарушить безопасность типов!
static void dangerous(List<String>... stringLists)
{
    List<Integer> intList = List.of(42);
    Object[] objects = stringLists;
    objects[0] = intList;           // Загрязнение кучи
    String s = stringLists[0].get(0); // ClassCastException
}
```

Этот метод, не имеющий видимых приведений, генерирует исключение `ClassCastException` при вызове с одним или несколькими аргументами. Его последняя строка имеет невидимое приведение, которое генерируется компилятором. Это приведение завершается неудачей, демонстрируя нарушение безопасности типов, так что **опасно хранить значение в обобщенном массиве-параметре переменной длины**.

Этот пример приводит к интересному вопросу: почему корректно объявление метода с обобщенным параметром переменной длины, если явное создание обобщенного массива не разрешено? Другими словами, почему метод, показанный выше, генерирует только предупреждение, в то время как фрагмент кода в разделе 5.3 выдает ошибку? Ответ заключается в том, что методы с переменным количеством параметров обобщенных или параметризованных типов могут быть весьма полезны на практике, поэтому разработчики языка решили оставить эту несогласованность. Фактически библиотеки Java экспортируют несколько таких методов, включая `Arrays.asList(T... a)`, `Collections.addAll(Collection<? super T> c, T... elements)` и `EnumSet.of(E first, E... rest)`. В отличие от метода `dangerous`, показанного выше, эти методы библиотеки являются безопасными с точки зрения типов.

До Java 7 не было ничего, что автор метода с обобщенным параметром переменной длины мог бы сделать с предупреждениями в месте вызова. Это сделало такие API неудобными в использовании. Пользователи должны были либо смириться с предупреждениями, либо, что предпочтительнее, подавлять их с помощью аннотации `@SuppressWarnings("unchecked")` в каждой точке вызова (раздел 5.2). Это утомительно, ухудшает удобочитаемость и скрывает предупреждения, которые связаны с реальными проблемами.

В Java 7 были добавлены аннотации `SafeVarargs`, которые позволяли автору метода с обобщенным параметром с переменной длиной подавлять клиентские предупреждения автоматически. По сути, аннотация **`SafeVarargs`** представляет собой обещание безопасности с точки зрения типов от автора метода. В обмен на это обещание компилятор обязуется не предупреждать пользователей метода, что вызовы могут оказаться небезопасными.

Очень важно не аннотировать метод с помощью `@SafeVarargs`, если только он на самом деле не является безопасным. Что требуется для того, чтобы обеспечить эту безопасность? Вспомним, что при вызове метода для хранения произвольного количества параметров создается обобщенный массив. Если метод ничего не хранит в этом массиве (что привело бы к перезаписи параметров) и не позволяет ссылке на этот массив выбраться “наружу” (что привело бы к разрешению использования непроверенного кода для обращения к этому массиву), то метод безопасен. Другими словами, если массив параметра с переменной длиной используется только для того, чтобы передавать переменное количество аргументов от вызывающего метода (в чем в конечном итоге и состоит его цель), то такой метод является безопасным.

Стоит отметить, что можно нарушать безопасность типов без сохранения чего-либо в массиве параметра с переменной длиной. Рассмотрим следующий обобщенный метод с переменным количеством аргументов, который возвращает массив, содержащий его параметры. На первый взгляд, это может выглядеть маленькой удобной утилитой:

```
// ОПАСНО: открывает ссылку на массив
// параметра с переменной длиной!
static <T> T[] toArray(T... args)
{
    return args;
}
```

Этот метод просто возвращает массив параметров. Метод может выглядеть безопасным, но это не так! Тип данного массива определяется типами времени компиляции аргументов, передаваемых в метод, и компилятор может не иметь достаточно информации для точного определения. Поскольку метод возвращает свой массив параметров, он может распространить загрязнение кучи по стеку вызовов.

Для конкретности рассмотрим следующий обобщенный метод, который принимает три аргумента типа `T` и возвращает массив, содержащий два выбранных случайным образом аргумента:

```
static <T> T[] pickTwo(T a, T b, T c)
{
    switch (ThreadLocalRandom.current().nextInt(3))
    {
        case 0:
            return toArray(a, b);
        case 1:
            return toArray(a, c);
        case 2:
            return toArray(b, c);
    }
    throw new AssertionError(); // Попасть сюда нельзя
}
```

Этот метод сам по себе не опасен и не будет генерировать предупреждение, за исключением того, что он вызывает метод `toArray`, который имеет обобщенный параметр переменной длины.

При компиляции этого метода компилятор генерирует код для создания массива `toArray` параметра с переменной длиной, в который передаются два экземпляра `T`. Этот код выделяет массив типа `Object []`, который представляет собой наиболее конкретный тип, гарантированно способный хранить эти экземпляры независимо от того, какие типы объектов передаются в `pickTwo` в точке вызова. Метод `toArray` просто возвращает этот массив в метод `pickTwo`, который, в свою очередь, возвращает его коду, вызвавшему метод `pickTwo`. Поэтому `pickTwo` всегда будет возвращать массив типа `Object []`.

Рассмотрим теперь метод `main`, который вызывает `pickTwo`:

```
public static void main(String[] args)
{
    String[] attributes = pickTwo("Good", "Fast", "Cheap");
}
```

В этом методе нет ничего неправильного, так что он будет компилироваться без каких-либо предупреждений. Но когда вы его запустите, он сгенерирует исключение `ClassCastException`, хотя и не содержит никаких видимых приведений. Невидимым для вас является сгенерированное компилятором скрытое приведение к `String []` значения, возвращаемого `pickTwo`, чтобы оно могло быть сохранено в `attributes`. Приведение завершается неудачей, потому что `Object []` не является подтипом типа `String []`. Это довольно обескураживающая ошибка, потому что из метода удалены два уровня, которые фактически приводят к загрязнению кучи (`toArray`), а массив параметра переменной длины не изменяется после того, как в нем сохраняются фактические параметры.

Этот пример предназначен для демонстрации того, как **опасно давать другому методу доступ к массиву обобщенного параметра переменной длины**, с двумя исключениями: безопасна передача массива в другой метод с переменным количеством аргументов, который правильно помечен как `@SafeVarargs`, и безопасна передача массива методу с фиксированным количеством аргументов, который просто вычисляет некоторую функцию от содержимого массива.

Вот типичный пример безопасного обобщенного параметра переменной длины. Данный метод в качестве аргументов принимает произвольное количество списков и возвращает единый список, содержащий элементы всех входных списков. Поскольку метод помечен как `@SafeVarargs`, он не генерирует никакие предупреждения, ни при объявлении, ни в точке вызова:

```
// Безопасный метод с обобщенным параметром переменной длины
@SafeVarargs
static <T> List<T> flatten(List <? extends T > ... lists)
{
    List<T> result = new ArrayList<>();

    for (List <? extends T > list : lists)
        result.addAll(list);

    return result;
}
```

Правило определения, когда использовать аннотацию `SafeVarargs`, простое: **используйте `@SafeVarargs` с каждым методом с параметром переменной длины обобщенного или параметризованного типа**, чтобы его пользователи не были перегружены ненужными и запутанными предупреждениями компилятора. Это означает, что *никогда* не следует писать небезопасные методы с переменным количеством аргументов, такие как `dangerous` или `toArray`. Каждый раз, когда компилятор предупреждает вас о возможном загрязнении кучи из-за обобщенного параметра переменной длины в методе, который вы контролируете, проверяйте безопасность этого метода. Напомним, что обобщенный метод с переменным количеством аргументов безопасен, если:

1. он ничего не сохраняет в массиве параметра переменной длины;
2. он не делает массив (или его клон) видимым ненадежному коду.

Если любой из этих запретов нарушается, эту ситуацию следует исправить.

Обратите внимание, что аннотация `SafeVarargs` является законной только для методов, которые не могут быть перекрыты, потому что невозможно гарантировать, что каждое возможное перекрытие метода будет безопасным.

В Java 8 эта аннотация разрешается только для статических методов и окончательных (`final`) методов экземпляров; в Java 9 она разрешена и для закрытых методов экземпляров.

Альтернативой использованию аннотации `SafeVarargs` является применение советов из раздела 5.3 по замене параметра переменной длины (который представляет собой замаскированный массив) параметром `List`. Вот как выглядит этот подход при применении к нашему методу `flatten`. Обратите внимание, что изменилось только объявление параметра:

```
// List представляет собой безопасную с точки зрения типов
// альтернативу обобщенному параметру переменной длины
static <T> List<T> flatten(List<List<? extends T>> lists)
{
    List<T> result = new ArrayList<>();
    for (List<? extends T> list : lists)
        result.addAll(list);
    return result;
}
```

Далее этот метод может использоваться в сочетании со статическим фабричным методом `List.of`, чтобы разрешить переменное количество аргументов. Обратите внимание, что этот подход опирается на тот факт, что объявление `List.of` аннотировано с помощью `@SafeVarargs`:

```
audience = flatten(List.of(friends, romans, countrymen));
```

Преимуществом этого подхода является то, что компилятор может *доказать*, что метод является безопасным с точки зрения типов. Вам не придется ручаться за его безопасность с помощью аннотации `SafeVarargs`, как и беспокоиться о возможной ошибке, допущенной при определении безопасности метода. Основным недостатком данного подхода является то, что клиентский код получается многословнее и может оказаться немного медленнее.

Этот трюк может также использоваться в ситуациях, в которых невозможно написать безопасный метод с переменным количеством аргументов, как в случае с методом `toArray` выше. Его аналогом с использованием `List` является метод `List.of`, так что мы даже не должны его писать; авторы библиотек Java сделали эту работу вместо нас. Метод `pickTwo` превращается в следующий:

```
static <T> List<T> pickTwo(T a, T b, T c)
{
    switch (rnd.nextInt(3))
    {
        case 0:
            return List.of(a, b);
        case 1:
            return List.of(a, c);
    }
}
```

```

        case 2:
            return List.of(b, c);
    }
    throw new AssertionError();
}

```

А метод `main` принимает вид

```

public static void main(String[] args)
{
    List<String> attributes = pickTwo("Good", "Fast", "Cheap");
}

```

Получающийся в результате код безопасен с точки зрения типов, поскольку использует только обобщенные типы, но не массивы.

Итак, следует помнить, что методы с переменным количеством аргументов и обобщенные типы взаимодействуют не очень хорошо, потому что параметр переменной длины представляет собой утечку абстракции, построенную поверх массива, а правила типов для массивов отличаются от правил для обобщенных типов. Хотя обобщенные параметры переменной длины не являются безопасными с точки зрения типов, они являются вполне законными. Если вы решили написать метод с обобщенным (или параметризованным) параметром переменной длины, сначала убедитесь, что метод является безопасным с точки зрения типов, а затем аннотируйте его с использованием `@SafeVarargs`, чтобы его не было противно использовать из-за предупреждений компилятора.

5.8. Применяйте безопасные с точки зрения типов гетерогенные контейнеры

Распространенные примеры использования обобщенных типов включают коллекции, такие как `Set<E>` и `Map<K, V>`, и одноэлементные контейнеры, такие как `ThreadLocal<T>` и `AtomicReference<T>`. Во всех этих примерах имеется параметризованный контейнер. Это ограничивает вас фиксированным количеством параметров типа в контейнере. Обычно это именно то, что и требуется. `Set` имеет единственный параметр типа, представляющий тип его элементов; `Map` имеет два параметра типа, представляющие типы ключа и значения, и т.д.

Тем не менее иногда требуется большая гибкость. Например, строка базы данных может содержать произвольное количество столбцов, и было бы неплохо получать ко всем ним доступ, безопасный с точки зрения типов. К счастью, есть очень легкий способ добиться этого. Идея заключается в том, чтобы

параметризовать *ключ*, а не *контейнер*. Затем предоставить параметризованный ключ контейнеру для вставки или извлечения значений. Система обобщенных типов используется для гарантии того, что тип значения согласуется с его ключом.

В качестве простого примера такого подхода рассмотрим класс `Favorites`, который позволяет своим клиентам хранить и извлекать предпочтительные экземпляры произвольно большого количества типов. Объект `Class` для типа будет играть роль части параметризованного ключа. Причина, по которой этот подход работает, заключается в том, что класс `Class` является обобщенным. Тип литерала класса есть не просто `Class`, а `Class<T>`. Например, `String.class` имеет тип `Class<String>`, а `Integer.class` — тип `Class<Integer>`. Когда литерал класса передается между методами для передачи информации о типе как во время компиляции, так и во время выполнения, он называется токеном типа (*type token*) [6].

API класса `Favorites` прост. Он выглядит подобно простому отображению с тем отличием, что параметризованным является ключ, а не отображение. Клиент предоставляет объект типа `Class` при установке и получении избранных объектов. Вот как выглядит API:

```
// Схема безопасного с точки зрения
// типов гетерогенного контейнера - API
public class Favorites
{
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Вот пример программы, которая использует класс `Favorites`, сохраняя, извлекая и выводя избранные экземпляры `String`, `Integer` и `Class`:

```
// Схема безопасного с точки зрения
// типов гетерогенного контейнера - клиент
public static void main(String[] args)
{
    Favorites f = new Favorites();
    f.putFavorite(String.class, "Java");
    f.putFavorite(Integer.class, 0xcafebabe);
    f.putFavorite(Class.class, Favorites.class);
    String favoriteString = f.getFavorite(String.class);
    int favoriteInteger = f.getFavorite(Integer.class);
    Class<?> favoriteClass = f.getFavorite(Class.class);
    System.out.printf("%s %x %s\n", favoriteString,
                      favoriteInteger, favoriteClass.getName());
}
```

Как и следовало ожидать, программа выводит

```
Java cafebabe Favorites
```

Кстати, обратите внимание, что метод `printf` в Java отличается от одноименной функции C тем, что вы должны использовать `%n` там, где в C используется `\n`. Спецификатор `%n` генерирует платформу-зависимые разделители строк, которыми на многих (но не на всех) платформах являются `\n`.

Экземпляр `Favorites` *безопасен с точки зрения типов*: он никогда не будет возвращать `Integer`, если запросить у него `String`. Он также *гетерогенен*: в отличие от обычного отображения, все ключи имеют различные типы. Поэтому мы называем `Favorites` *безопасным с точки зрения типов гетерогенным контейнером*.

Реализация `Favorites` на удивление невелика. Вот она во всей своей полноте:

```
// Схема безопасного с точки зрения
// типов гетерогенного контейнера - реализация
public class Favorites
{
    private Map<Class<?>, Object> favorites = new HashMap<>();
    public <T> void putFavorite(Class<T> type, T instance)
    {
        favorites.put(Objects.requireNonNull(type), instance);
    }
    public <T> T getFavorite(Class<T> type)
    {
        return type.cast(favorites.get(type));
    }
}
```

Здесь есть несколько тонкостей. Каждый экземпляр `Favorites` опирается на закрытый член `favorites` типа `Map<Class<?>, Object>`. Вы можете решить, что невозможно добавить что-либо в это отображение из-за неограниченного типа с символом подстановки, но на самом деле все совсем не так. Следует отметить, что этот тип с символом подстановки является вложенным: это не тип отображения, являющийся типом с символом подстановки, а тип его ключа. Это означает, что все ключи могут иметь *различные* параметризованные типы: один из них может быть `Class<String>`, другой — `Class<Integer>`, и т.д. Это и есть источник гетерогенности.

Следующее, что нужно отметить: тип значения отображения `favorites` — просто `Object`. Другими словами, отображение не гарантирует взаимоотношения типов между ключами и значениями, т.е. что каждое значение имеет тип, представленный его ключом. Фактически система типов в Java не является достаточно мощной для того, чтобы выразить такое отношение. Но мы

знаем, что это так, и воспользуемся этим преимуществом, когда придет время извлечения избранных объектов.

Реализация `putFavorite` тривиальна: метод просто помещает в `favorites` отображение данного объекта `Class` на данный избранный экземпляр. Как уже отмечалось, такая реализация устраняет “связь типов” между ключом и значением; теряется знание, что значение является экземпляром ключа. Но это нормально, потому что метод `getFavorites` может восстановить эту связь (и так и делает).

Реализация метода `getFavorite` сложнее, чем метода `putFavorite`. Он получает из отображения `favorites` значение, соответствующее данному объекту `Class`. Это верная ссылка на объект для возврата, но она имеет неправильный тип времени компиляции: это `Object` (тип значения отображения `favorites`), а мы должны вернуть `T`. Таким образом, реализация `getFavorite` динамически приводит ссылку на объект к типу, представленному объектом `Class`, с использованием метода `cast` класса `Class`.

Метод `cast` является динамическим аналогом оператора приведения в `Java`. Он просто проверяет, что его аргумент является экземпляром типа, представленного объектом `Class`. Если это так, он возвращает аргумент; в противном случае он генерирует исключение `ClassCastException`. Мы знаем, что вызов приведения в `getFavorite` не генерирует исключения `ClassCastException`, в предположении, что клиентский код компилируется без ошибок. То есть мы знаем, что значения в отображении `favorites` всегда соответствуют типам их ключей.

Так что же делает метод `cast` с учетом того, что он просто возвращает свой аргумент? Сигнатура метода `cast` в полной мере использует преимущества того факта, что класс `Class` является обобщенным. Тип его возвращаемого значения представляет собой параметр объекта `Class`:

```
public class Class<T>
{
    T cast(Object obj);
}
```

Это именно то, что нужно методу `getFavorite`. Это позволит нам сделать `Favorites` безопасным с точки зрения типов, не прибегая к непроверяемым приведениям к типу `T`.

Стоит отметить два ограничения класса `Favorites`. Вредоносный клиент может легко нарушить безопасность типов экземпляра `Favorites`, используя объект `Class` в несформированном виде. Но получающийся в результате код клиента будет во время компиляции генерировать предупреждение о непроверяемом коде. Это ничем не отличается от реализаций обычных коллекций `HashSet` и `HashMap`. Вы можете легко поместить `String`

в `HashSet<Integer>`, используя несформированный тип `HashSet` (раздел 5.1). Тем не менее можно обеспечить безопасность с точки зрения типов времени выполнения, если вы готовы за нее платить. Способ гарантировать, что инвариант типа `Favorites` никогда не будет нарушен, состоит в том, что метод `putFavorite` должен проверять, что `instance` действительно является экземпляром типа, представленного `type`, и мы уже знаем, как это делается. Просто используйте динамическое приведение:

```
// Достижение безопасности типов времени
// выполнения с помощью динамического приведения
public <T> void putFavorite(Class<T> type, T instance)
{
    favorites.put(type, type.cast(instance));
}
```

В `java.util.Collections` имеются классы-оболочки, исполняющие тот же трюк. Они называются `checkedSet`, `checkedList`, `checkedMap` и т.д. Их статические фабрики могут принимать один (или два) объекта `Class` в дополнение к коллекции (или отображению). Статические фабрики представляют собой обобщенные методы, гарантирующие, что типы времени компиляции объектов `Class` и коллекции совпадают. Оболочки добавляют доступность при выполнении в коллекции, которые они обертывают. Например, оболочка генерирует исключение `ClassCastException` во время выполнения, если кто-то пытается поместить `Coin` в коллекцию `Collection<Stamp>`. Эти оболочки полезны для отслеживания клиентского кода, который добавляет элемент неверного типа в коллекцию, в приложении, в котором смешаны обобщенные и несформированные типы.

Второе ограничение класса `Favorites` заключается в том, что он не может быть использован с типом, недоступным во время выполнения (раздел 5.3). Другими словами, вы можете хранить свои избранные `String` или `String[]`, но не избранный `List<String>`. Если вы попытаетесь сохранить избранный `List<String>`, ваша программа не будет компилироваться. Причина заключается в том, что вы не можете получить объект `Class` для `List<String>`. Литерал класса `List<String>.class` представляет собой синтаксическую ошибку, но это хорошо. `List<String>` и `List<Integer>` совместно используют один объект `Class`, которым является `List.class`. Если бы “литералы типа” `List<String>` и `List<Integer>` были допустимыми и возвращали одну и ту же ссылку на объект, это привело бы к хаосу внутри объекта `Favorites`. Существует не совсем удовлетворительный способ обойти это ограничение.

Токены типа, используемые `Favorites`, являются неограниченными: `getFavorite` и `putFavorite` принимают любые объекты `Class`. Иногда может

потребуется ограничить типы, которые могут быть переданы в метод. Это может быть достигнуто с помощью *ограниченного токена типа* (bounded type token), который представляет собой просто токен типа, который ограничивает представимые типы с помощью ограниченного параметра типа (раздел 5.5) или ограниченного символа подстановки (раздел 5.6).

API аннотаций (раздел 6.6) широко использует ограниченные токены типа. Например, далее показан метод чтения аннотации во время выполнения. Этот метод поступает из интерфейса `AnnotatedElement`, который реализуется рефлексивными типами, представляющими классы, методы, поля и другие элементы программы:

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

Аргумент `annotationType` является ограниченным токеном типа, представляющим тип аннотации. Метод возвращает аннотацию элемента данного типа, если таковая имеется, или `null`, если ее нет. По сути, аннотированный элемент является безопасным с точки зрения типов гетерогенным контейнером, ключи которого являются типами аннотаций.

Предположим, что есть объект типа `Class<?>`, и его нужно передать в метод, который требует ограниченный токен типа, такой как `getAnnotation`. Можно привести этот объект к типу `Class<? extends Annotation>`, но это приведение непроверяемое, так что будет сгенерировано предупреждение времени компиляции (раздел 5.2). К счастью, `Class` предоставляет метод экземпляра, который выполняет приведение такого рода безопасно (и динамически). Этот метод называется `asSubclass`, и он приводит объект класса `Class`, для которого он был вызван, к представлению подкласса класса, представленного его аргументом. Если приведение выполнено успешно, метод возвращает свой аргумент; в противном случае генерируется исключение `ClassCastException`.

Вот как использовать метод `asSubclass` для чтения аннотации, тип которой неизвестен во время компиляции. Этот метод будет компилироваться без ошибок и предупреждений:

```
// Применение asSubclass для безопасного приведения
// к ограниченному токену типа
static Annotation getAnnotation(AnnotatedElement element,
                               String annotationTypeName)
{
    Class<?> annotationType = null; // Неограниченный токен типа

    try
    {
        annotationType = Class.forName(annotationTypeName);
```

```
    }  
    catch (Exception ex)  
    {  
        throw new IllegalArgumentException(ex);  
    }  
    return element.getAnnotation(  
        annotationType.asSubclass(Annotation.class));  
}
```

Итак, нормальное использование обобщенных типов, демонстрируемое API коллекций, ограничивает вас фиксированным количеством параметров типа в контейнере. Вы можете обходить это ограничение, помещая параметр типа в ключ, а не в контейнер. Можно использовать объекты `Class` в качестве ключей таких гетерогенных контейнеров, безопасных с точки зрения типов. Объект `Class`, используемый таким образом, называется токеном типа. Можно также использовать ключи пользовательских типов. Например, можно было бы иметь тип `DatabaseRow`, представляющий строку базы данных (контейнер) и обобщенный тип `Column<T>` в качестве его ключа.

Перечисления и аннотации

Java поддерживает два специальных семейства ссылочных типов: разновидность классов, именуемых *типами перечислений* (enum type), и разновидность интерфейсов, именуемых *типами аннотаций* (annotation type). В этой главе рассматриваются наилучшие практики для использования этих семейств типов.

6.1. Используйте перечисления вместо констант `int`

Тип перечисления — это тип, разрешенные значения которого состоят из фиксированного набора констант, таких как времена года, планеты Солнечной системы или масти в колоде карт. До того как типы перечислений были добавлены в язык программирования, распространенной схемой представления типов перечислений было объявление группы именных констант типа `int`, по одной для каждого члена типа:

```
// Перечисления с помощью констант int - крайне неудовлетворительно!  
public static final int APPLE_FUJI           = 0;  
public static final int APPLE_PIPPIN        = 1;  
public static final int APPLE_GRANNY_SMITH  = 2;  
public static final int ORANGE_NAVEL        = 0;  
public static final int ORANGE_TEMPLE       = 1;  
public static final int ORANGE_BLOOD        = 2;
```

Эта технология, известная как *шаблон перечисления int* (int enum pattern), обладает множеством недостатков. Она не обеспечивает ни безопасность с точки зрения типов, ни выразительность. Компилятор не будет жаловаться при передаче яблока методу, который ожидает апельсин, сравнивает яблоки с апельсинами с помощью оператора `==` или поступает еще хуже:

```
// Цитрусово-яблочное пюре :(  
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Обратите внимание, что имя каждой константы для яблока начинается с `APPLE_`, а имя каждой константы для апельсина начинается с `ORANGE_`. Дело в том, что Java не предоставляет пространств имен групп перечислений `int`. Префиксы предотвращают конфликты имен, когда две группы перечислений `int` содержат идентично именованные константы, например между `ELEMENT_MERCURY` и `PLANET_MERCURY`¹.

Программы, использующие перечисления `int`, являются хрупкими. Поскольку перечисления `int` представляют собой *константные переменные* [25, 4.12.4], их целочисленные значения компилируются в клиентский код, использующий эти значения [25, 13.1]. Если значение, связанное с целочисленной константой перечисления, изменяется, его клиенты должны быть перекомпилированы. Если этого не сделать, клиентские программы будут по-прежнему работать, но их поведение будет неверным.

Нет легкого способа транслировать константы перечислений `int` в выводимые строки. Если вы выведете такую константу или отобразите ее в отладчике, все, что вы увидите, — это число, что не слишком полезно. Нет никакого надежного способа обойти все целочисленные константы перечислений в группе или хотя бы получить размер этой группы констант.

Вы можете столкнуться с вариантом этой схемы, в котором вместо констант `int` используются константы `String`. Этот вариант, известный как *шаблон перечисления String* (`String enum pattern`), еще менее желательно использовать. Предоставление выводимых строк для констант происходит ценой проблем с производительностью, потому что теперь надо сравнивать не целочисленные значения, а строки. Кроме того, эта схема может привести неопытных пользователей к жестко “прошитым” в клиентском коде строковым константам, а не к использованию имен полей. Если такая жестко прошитая строковая константа содержит опечатку, то она будет пропущена компилятором и проблема проявится только при выполнении.

К счастью, Java предоставляет альтернативу, которая помогает избежать недостатков шаблонов перечислений и предлагает ряд дополнительных преимуществ. Это *типы перечислений* (`enum type`) [25, 8.9]. Вот как они выглядят в их простейшей форме:

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

При поверхностном взгляде может показаться, что типы перечислений в Java похожи на аналоги в других языках, таких как C, C++ и C#, но это сходство обманчиво. Типы перечислений Java — это полноценные классы, обладающие

¹ Имеются в виду одинаково звучащие названия химического элемента ртути и планеты Меркурий. — *Примеч. пер.*

большими возможностями, чем их аналоги в других языках, в которых типы перечислений являются, по сути, значениями `int`.

Основная идея, лежащая в основе типов перечислений Java, проста: они являются классами, которые экспортируют по одному экземпляру для каждой константы перечисления, используя открытое статическое окончательное поле. Типы перечислений являются окончательными, потому что у них нет конструкторов. Клиенты не могут ни создавать экземпляры типов перечислений, ни расширять их, и не может быть никаких иных экземпляров, кроме объявленных констант перечисления. Другими словами, создание экземпляров типов перечислений находится под строгим контролем. Они являются обобщением сингلتонов (раздел 2.3), которые, по сути, являются одноэлементными перечислениями.

Перечисления обеспечивают безопасность с точки зрения типов времени компиляции. Если вы объявили параметр типа `Apple`, то гарантируется, что любая ненулевая ссылка на объект, переданная в качестве аргумента, является одним из трех разрешенных значений `Apple`. Попытка передать значение неверного типа приведет к ошибке времени компиляции, так же как и попытки присвоить выражение одного перечисления переменной другого или использовать оператор `==` для сравнения значений различных типов перечислений.

Типы перечислений с одинаковыми именами констант мирно сосуществуют, поскольку у каждого типа имеется собственное пространство имен. Вы можете добавлять или изменять порядок констант в типе перечисления без необходимости повторно компилировать клиентов, потому что поля, экспортирующие константы, предоставляют изолирующий слой между типом перечисления и его клиентами: значения констант не компилируются в клиентский код, как в случае с шаблоном перечислений `int`. Наконец, можно перевести типы перечислений в выводимые строки с помощью метода `toString`.

В дополнение к тому, что типы перечислений позволяют избавиться от недостатков перечислений `int`, они позволяют добавлять произвольные методы и поля и реализовывать произвольные интерфейсы. Они предоставляют высококачественные реализации всех методов `Object` (глава 3, “Методы, общие для всех объектов”), реализуют интерфейсы `Comparable` (раздел 3.5) и `Serializable` (глава 12, “Сериализация”), а их сериализованный вид разработан таким образом, чтобы выдерживать большинство изменений в типе перечисления.

Зачем нужно добавлять методы или поля к типам перечислений? Например, вы можете захотеть связать данные с их константами. Наши типы `Apple` или `Orange`, например, могут извлечь выгоду из метода, который вернет цвет фрукта или его изображение. Вы можете добавить к типу перечисления любой метод, кажущийся подходящим. Тип перечисления может начать существование как простая коллекция констант перечисления и со временем развиться в полноценную абстракцию.

В качестве примера типа перечисления с широкими возможностями рассмотрим восемь планет Солнечной системы. У каждой планеты есть масса и радиус, и эти два атрибута позволяют рассчитать гравитационное поле (ускорение свободного падения) на поверхности планеты. Это, в свою очередь, позволяет рассчитать вес любого объекта на поверхности планеты, зная его массу. Далее показано, как выглядит этот тип перечисления. Числа в скобках после каждой константы перечисления — это параметры, которые передаются конструктору. В данном случае это масса и радиус планеты.

// Тип перечисления с данными и поведением

```
public enum Planet
{
    MERCURY(3.302e+23, 2.439e6),
    VENUS (4.869e+24, 6.052e6),
    EARTH (5.975e+24, 6.378e6),
    MARS (6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN (5.685e+26, 6.027e7),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass;           // кг
    private final double radius;        // м
    private final double surfaceGravity; // м/с^2

    // Гравитационная постоянная, м^3 / кг·с^2
    private static final double G = 6.67300E-11;

    // Конструктор
    Planet(double mass, double radius)
    {
        this.mass = mass;
        this.radius = radius;
        surfaceGravity = G * mass / (radius * radius);
    }
    public double mass()
    {
        return mass;
    }
    public double radius()
    {
        return radius;
    }
    public double surfaceGravity()
    {
        return surfaceGravity;
    }
}
```

```

public double surfaceWeight(double mass)
{
    return mass * surfaceGravity; // F = ma
}
}

```

Довольно легко написать тип перечисления с большими возможностями, такой как Planet. **Чтобы связать данные с константами перечисления, следует объявить поля экземпляров и написать конструктор, который получает данные и сохраняет их в полях.** Типы перечислений по своей природе неизменяемы, так что все поля должны быть окончательными, объявленными как final (раздел 4.3). Поля могут быть открытыми, но лучше сделать их закрытыми и снабдить открытыми методами доступа (раздел 4.2). В случае Planet конструктор также рассчитывает и сохраняет значение гравитационного поля на поверхности, но это просто оптимизация. Гравитационное поле может быть рассчитано заново с использованием массы и радиуса при каждом вызове метода surfaceWeight, который получает массу объекта и возвращает его вес на представленной константой планете.

Перечисление Planet удивительно мощное при его простоте. Вот короткая программка, которая получает земной вес объекта (в любых единицах) и выводит таблицу весов этого объекта на всех восьми планетах (в тех же единицах):

```

public class WeightTable
{
    public static void main(String[] args)
    {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for (Planet p : Planet.values())
            System.out.printf("Вес на планете %s равен %f\n",
                               p, p.surfaceWeight(mass));
    }
}

```

Заметим, что у Planet, как и у всех типов перечислений, есть статический метод values, который возвращает массив его значений в той последовательности, в которой они были объявлены. Также обратите внимание, что метод toString возвращает объявленные названия каждого значения перечисления, позволяя легко вывести их с помощью методов println и printf. Если вас не устраивает данное строковое представление, измените его, перекрывая метод toString. Вот результат выполнения нашей небольшой программы WeightTable (не перекрывающей метод toString) с аргументом командной строки 185:

```

Вес на планете MERCURY равен 69.912739
Вес на планете VENUS равен 167.434436

```

Вес на планете EARTH равен 185.000000
Вес на планете MARS равен 70.226739
Вес на планете JUPITER равен 467.990696
Вес на планете SATURN равен 197.120111
Вес на планете URANUS равен 167.398264
Вес на планете NEPTUNE равен 210.208751

С 2006 года (через два года после того, как перечисления были добавлены в Java) Плутон перестал быть планетой. Это вызывает вопрос “Что происходит при удалении элемента из типа перечисления?” Ответ заключается в том, что любая клиентская программа, которая не обращается к удаленным элементам, будет по-прежнему нормально работать. Так, например, наша программа `WeightTable` будет просто распечатывать таблицу, меньшую на одну строку. Но что если клиентская программа обращается к удаленному элементу (в данном случае — к `Planet.PLUTO`)? Если вы перекомпилируете клиентскую программу, компиляция завершится неудачно с выводом сообщения об ошибке в строке, в которой выполняется обращение к бывшей планете. Если клиентская программа не перекомпилирована, то данная строка сгенерирует исключение во время выполнения. Это наилучшее поведение, на которое можно надеяться, куда лучше, чем то, которое достигается при использовании перечислений `int`.

Некоторое поведение, связанное с константами перечисления, может потребоваться использовать только в пределах класса или пакета, в котором определено перечисление. Такое поведение лучше всего реализовать как закрытый метод, или метод, доступный на уровне пакета. С каждой константой будет связана скрытая коллекция поведений, которая позволяет классу или пакету, содержащему перечисление, реагировать на константы перечисления соответствующим образом. Как и в прочих классах, если только у вас нет веских причин предоставлять метод перечисления для своих клиентов, его нужно объявить закрытым или, при необходимости, доступным в пределах пакета (раздел 4.1).

Чтобы перечисление в общем случае было полезным, оно должно быть классом верхнего уровня; если его применение связано с определенным классом верхнего уровня, оно должно быть классом-членом класса верхнего уровня (раздел 4.10). Например, перечисление `java.math.RoundingMode` представляет режимы округления десятичных дробей. Эти режимы округления используются классом `BigDecimal`, но они представляют полезную абстракцию, которая не является фундаментально связанной с `BigDecimal`. Делая `RoundingMode` перечислением верхнего уровня, проектировщики библиотеки поощряют всех программистов, нуждающихся в режимах округления, к повторному использованию этого перечисления, что приводит к увеличению согласованности API.

Методики, продемонстрированной в примере с Planet, достаточно для применения в большинстве типов перечислений, но иногда требуется большее. С каждой константой Planet связаны различные данные, но иногда требуется связать с каждой константой фундаментально различное *поведение*. Например, предположим, что вы пишете тип перечисления, представляющий операции на базе арифметического калькулятора с четырьмя функциями и хотите предоставить метод для выполнения арифметических операций, представленных каждой константой. Одним из способов достижения этого является конструкция switch со значениями перечисления:

```
// Тип перечисления с переключением между своими значениями - спорно
public enum Operation
{
    PLUS, MINUS, TIMES, DIVIDE;
    // Выполнение арифметической операции, представленной константой
    public double apply(double x, double y)
    {
        switch (this)
        {
            case PLUS:
                return x + y;
            case MINUS:
                return x - y;
            case TIMES:
                return x * y;
            case DIVIDE:
                return x / y;
        }
        throw new AssertionError("Неизвестная операция: " + this);
    }
}
```

Этот код работает, но он не слишком хорош. Он не будет компилироваться без конструкции throw, потому что технически конец метода может быть достигнут (хотя на самом деле он не будет достигнут никогда [25, 14.21]). Что еще хуже, это довольно хрупкий код. Если вы добавите новую константу перечисления, но забудете добавить соответствующий case в switch, то тип перечисления все равно скомпилируется, но будет приводить к ошибке при выполнении, когда вы попытаетесь выполнить новую операцию.

К счастью, есть более удачный способ связывания различного поведения с каждой константой перечисления: объявить абстрактный метод apply в типе перечисления и перекрыть его конкретным методом для каждой константы в теле класса, зависящего от константы (constant-specific class body). Такие

методы известны как *реализации методов, зависящих от констант* (constant-specific method implementations):

```
// Тип перечисления с реализациями методов, зависящих от констант
public enum Operation
{
    PLUS {public double apply(double x, double y){return x + y;}},
    MINUS {public double apply(double x, double y){return x - y;}},
    TIMES {public double apply(double x, double y){return x * y;}},
    DIVIDE{public double apply(double x, double y){return x / y;}};

    public abstract double apply(double x, double y);
}
```

Если вы добавите новую константу во вторую версию Operation, маловероятно, что вы забудете предоставить соответствующий метод apply, потому что он следует сразу же за объявлением константы. В том маловероятном случае, если вы все же забудете это сделать, компилятор напомнит вам об этом, потому что абстрактные методы в типе перечисления должны быть перекрыты конкретными методами у всех констант.

Реализации методов, зависящих от констант, могут быть объединены с зависящими от констант данными. Например, вот версия Operation, которая перекрывает метод toString для возвращения символа, обычно связанного с операцией:

```
// Тип перечисления с телами классов и данными,
// зависящими от констант
public enum Operation
{
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;

    Operation(String symbol) { this.symbol = symbol; }

    @Override public String toString() { return symbol; }
}
```

```

    public abstract double apply(double x, double y);
}

```

Приведенная реализация `toString` упрощает вывод арифметических выражений, как показано в приведенной ниже короткой программе:

```

public static void main(String[] args)
{
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);

    for (Operation op : Operation.values())
        System.out.printf("%f %s %f = %f%n",
                           x, op, y, op.apply(x, y));
}

```

Выполнение этой программы для 2 и 4 в качестве аргументов командной строки приводит к следующему выводу:

```

2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000

```

Типы перечислений имеют автоматически генерируемый метод `valueOf(String)`, который транслирует имя константы в саму константу. Если вы перекрываете метод `toString` в типе перечисления, подумайте о перекрытии метода `fromString` для трансляции пользовательского строкового представления в соответствующую константу. Приведенный далее код (с соответствующим образом измененным именем типа) выполняет эту операцию для любого перечисления, если каждая константа имеет уникальное строковое представление:

```

// Реализация метода fromString для типа перечисления
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));

// Возвращает Operation для строки, если такое значение имеется
public static Optional<Operation> fromString(String symbol)
{
    return Optional.ofNullable(stringToEnum.get(symbol));
}

```

Обратите внимание, что константы `Operation` помещаются в отображение `stringToEnum` из инициализатора статического поля, который выполняется после создания константы перечисления. В приведенном выше коде поток

(глава 7, “Лямбда-выражения и потоки”) применяется к массиву, возвращаемому методом `values()`; до Java 8 мы бы создали пустое хеш-отображение и выполнили обход значений массива значений, вставляя отображения строк на константы перечисления в хеш-отображение (и при желании вы можете поступить именно таким образом). Но обратите внимание, что попытки каждой константы поместить себя в отображение из собственного конструктора не работают. Это вызовет ошибку компиляции, и это очень хорошо, потому что если бы это было разрешено, то привело бы к генерации исключения `NullPointerException` во время выполнения. Конструкторам перечислений не разрешен доступ к статическим полям перечислений, за исключением константных переменных (раздел 6.1). Это ограничение необходимо, поскольку статические поля еще не инициализированы при выполнении конструкторов перечисления. Частным случаем этого ограничения является то, что константы перечислений не могут обращаться одна к другой из своих конструкторов.

Также обратите внимание, что метод `fromString` возвращает `Optional<String>`. Это позволяет методу указать, что переданная ему строка не представляет допустимую операцию, и заставляет клиента учитывать эту возможность (раздел 8.7).

Недостатком реализаций методов, зависящих от констант, является то, что они затрудняют совместное использование кода константами перечислений. Например, рассмотрим перечисление, представляющее дни недели в пакете заработной платы. Это перечисление содержит метод, вычисляющий оплату труда работника за конкретный день, с учетом базового оклада работника (в час) и количества рабочих минут в этот день. В пять будних дней рабочее время, превышающее обычный рабочий день, приводит к оплате сверхурочных; для двух выходных дней любое рабочее время оплачивается как сверхурочное. С помощью конструкции `switch` этот расчет легко реализуется, если использовать несколько меток `case` в каждом из двух фрагментов кода:

```
// Перечисление с использованием switch
// для совместного использования кода - спорно
enum PayrollDay
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int MINS_PER_SHIFT = 8 * 60;
    int pay(int minutesWorked, int payRate)
    {
        int basePay = minutesWorked * payRate;
        int overtimePay;
```

```
switch (this)
{
    case SATURDAY:
    case SUNDAY: // Выходной день
        overtimePay = basePay / 2;
        break;

    default: // Будний день
        overtimePay =
            minutesWorked <= MINS_PER_SHIFT ?
                0 :
                (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
}
return basePay + overtimePay;
}
```

Этот код, несомненно, краток, но опасен с точки зрения поддержки. Предположим, вы добавите к перечислению элемент, возможно — особое значение для представления отпуска, но забудете добавить соответствующий `case` в конструкцию `switch`. Программа все равно будет компилироваться, но метод `pay` будет начислять оплату за выход на работу во время отпуска так же, как если бы это был обычный рабочий день.

Чтобы безопасно выполнять расчеты оплаты с помощью реализации метода, зависящего от константы, нужно дублировать расчет оплаты сверхурочных для каждой константы или переместить расчет в два вспомогательных метода (один — для рабочих дней, другой — для выходных) и вызывать подходящий вспомогательный метод для каждой константы. Оба варианта ведут к увеличению объема кода, существенно уменьшая его удобочитаемость и увеличивая вероятность ошибки.

Раздувание кода можно уменьшить, заменив абстрактный метод `overtimePay` в `PayrollDay` конкретным методом, который выполняет расчет сверхурочных в выходные дни. Но это ведет к тому же недостатку, что и конструкция `switch`: добавив еще один день без перекрытия метода `overtimePay`, вы просто молча унаследуете вычисления для обычного рабочего дня.

В действительности нам нужно нечто, что *заставит* выбирать стратегию оплаты сверхурочных каждый раз при добавлении константы перечисления. К счастью, есть красивый способ этого достичь. Суть в том, чтобы перенести расчет оплаты сверхурочных в закрытое вложенное перечисление и передать экземпляр этого *перечисления-стратегии* конструктору перечисления `PayrollDay`. Затем перечисление `PayrollDay` делегирует расчет оплаты сверхурочных перечислению-стратегии, избегая необходимости в конструкции `switch` или реализации метода, зависящего от констант, в `PayrollDay`. Хотя

Этот подход менее краток, чем применение конструкции `switch`, он безопаснее и гибче:

```
// Схема применения стратегии-перечисления
enum PayrollDay
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;
    PayrollDay(PayType payType)
    {
        this.payType = payType;
    }
    PayrollDay()
    {
        this(PayType.WEEKDAY);    // По умолчанию
    }
    int pay(int minutesWorked, int payRate)
    {
        return payType.pay(minutesWorked, payRate);
    }
}

// Тип стратегии-перечисления
private enum PayType {
    WEEKDAY {
        int overtimePay(int minsWorked, int payRate)
        {
            return minsWorked <= MINS_PER_SHIFT ? 0 :
                (minsWorked - MINS_PER_SHIFT) * payRate / 2;
        }
    },
    WEEKEND {
        int overtimePay(int minsWorked, int payRate)
        {
            return minsWorked * payRate / 2;
        }
    };
    abstract int overtimePay(int mins, int payRate);
    private static final int MINS_PER_SHIFT = 8 * 60;
    int pay(int minsWorked, int payRate)
    {
        int basePay = minsWorked * payRate;
        return basePay + overtimePay(minsWorked, payRate);
    }
}
```

Если конструкция `switch` для перечислений не является хорошим выбором для реализации поведения, зависящего от констант, для чего же она хороша?

Конструкции `switch` с перечислениями хорошо подходит для дополнения типов перечислений поведением, зависимым от констант. Например, предположим, что перечисление `Operation` находится не под вашим контролем и вы хотели бы, чтобы у него был метод экземпляра для возврата операции, обратной данной. Вы можете имитировать данный эффект с помощью следующего статического метода:

```
// Применение switch с перечислением
// для имитации отсутствующего метода
public static Operation inverse(Operation op)
{
    switch (op)
    {
        case PLUS:
            return Operation.MINUS;
        case MINUS:
            return Operation.PLUS;
        case TIMES:
            return Operation.DIVIDE;
        case DIVIDE:
            return Operation.TIMES;
        default:
            throw new AssertionError("Неизвестная операция: "+op);
    }
}
```

Вы должны использовать эту методику и для типов перечислений, которые находятся под вашим контролем, если метод просто не принадлежит типу перечисления. Такой метод может потребоваться для некоторого конкретного применения, но при этом должен быть недостаточно полезным, чтобы заслуживать включения в тип перечисления.

В общем случае перечисления сопоставимы по производительности с константами `int`. Незначительным недостатком перечислений с точки зрения производительности перечисления является то, что загрузка и инициализация типов перечислений требуют времени и памяти, но маловероятно, что на практике это будет заметным.

Так когда же следует использовать типы перечислений? **Используйте перечисления всегда, когда требуется набор констант, члены которых известны во время компиляции.** Конечно, сюда входят “естественно перечислимые типы”, такие как планеты, дни недели или шахматные фигуры. Но сюда входят и другие множества, для которых во время компиляции известны все возможные значения, такие как пункты меню, коды операций и параметры командной строки. **Нет необходимости, чтобы множество констант перечисления оставалось неизменным все время.** Возможность перечислений была

разработана специально для того, чтобы обеспечить бинарно-совместимое развитие типов перечислений.

Вывод очевиден: преимущества типов перечислений по сравнению с константами типа `int` сложно переоценить. Перечисления более удобочитаемы, безопасны и имеют больше возможностей. Многие перечисления требуют явных конструкторов и членов, но многие другие извлекают пользу из связанных с каждой константой данных и методов, поведение которых определяется этими данными. Намного меньше пользы типы перечислений получают от связывания нескольких различных поведений с одним методом. В этом относительно редком случае предпочитайте методы, зависящие от констант, перечислениям с использованием конструкции `switch`. Подумайте о применении стратегий-перечислений, если некоторые (но не все) перечислимые константы обладают общим поведением.

6.2. Используйте поля экземпляров вместо порядковых значений

Многие перечисления естественным образом ассоциируются с единственным значением `int`. Все перечисления имеют метод `ordinal`, который возвращает числовую позицию каждой константы перечисления в своем типе. Вы можете соблазниться выводить связанное значение типа `int` из порядкового значения:

```
// Неверное применение ordinal() для
// получения связанного значения — НЕ ДЕЛАЙТЕ ТАК
public enum Ensemble
{
    SOLO,    DUET,    TRIO,    QUARTET,  QUINTET,
    SEXTET,  SEPTET,  OCTET,  NONET,    DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

Хотя это перечисление и работает, его поддержка — сущий кошмар. Если константы переупорядочиваются, метод `numberOfMusicians` перестает работать. Если вы захотите добавить вторую константу перечисления, связанную со значением типа `int`, которое уже было использовано, вам ничего не поможет. Например, было бы неплохо добавить константу для *двойного квартета*, который, как и октет, состоит из восьми музыкантов, но в данном случае нет никакой возможности это сделать.

Кроме того, нельзя добавить константу для значения `int`, не добавив константу для всех промежуточных значений `int`. Например, предположим, что вы хотите добавить константу, представляющую *тройной квартет*, который состоит из двенадцати музыкантов. Но не существует стандартного термина для ансамбля, состоящего из одиннадцати музыкантов, так что вам придется добавить фиктивную константу для неиспользуемого значения `int` (а именно, 11). Это как минимум уродливо. Если не используются многие значения `int`, это еще и непрактично.

К счастью, имеется простое решение этих проблем. **Никогда не выводите значение, связанное с перечислением, из его порядкового номера; вместо этого храните его в поле экземпляра:**

```
public enum Ensemble
{
    SOLO(1),    DUET(2),    TRIO(3),    QUARTET(4),    QUINTET(5),
    SEXTET(6), SEPTET(7),  OCTET(8),   DOUBLE_QUARTET(8),
    NONET(9),  DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

Спецификация Enum говорит о методе `ordinal`: “большинство программистов не должны использовать этот метод. Он разработан для использования в структурах данных общего назначения на основе перечислений, таких как `EnumSet` и `EnumMap`”. Если только вы не пишете код с такой структурой данных, лучше полностью избегать метода `ordinal`.

6.3. Используйте `EnumSet` вместо битовых полей

Если элементы перечислимого типа используются главным образом в множествах, то традиционно используется схема с применением перечисления `int` (раздел 6.1), назначающая различные степени 2 каждой константе:

```
// Константы перечисления в виде битовых полей - УСТАРЕЛО!
public class Text
{
    public static final int STYLE_BOLD          = 1 << 0; // 1
    public static final int STYLE_ITALIC       = 1 << 1; // 2
    public static final int STYLE_UNDERLINE    = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8
}
```

```
// Параметр представляет собой побитовое ИЛИ из нуля
// или большего количества констант STYLE_
public void applyStyles(int styles) { ... }
}
```

Это представление позволяет использовать побитовую операцию ИЛИ для объединения нескольких констант в множество, известное как *битовое поле* (bit field):

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

Представления в виде битовых полей позволяют эффективно выполнять операции над множествами, такие как объединение и пересечение, используя побитовую арифметику. Но битовые поля обладают всеми недостатками констант перечислений `int` и даже большими. Битовое поле труднее интерпретировать, чем простую перечислимую константу `int`, если она распечатана в виде числа. Нет простого способа обхода для всех элементов, представленных битовым полем. Наконец, во время написания API вы должны предсказать максимальное количество битов, которое когда-либо понадобится, и соответственно выбрать тип для битового поля (обычно `int` или `long`). Как только вы выбрали тип, вы не можете превысить его ширину (32 или 64 бита) без изменений в API.

Некоторые программисты, предпочитающие использовать типы перечислений вместо констант `int`, все еще используют битовые поля, если им необходимо передать наборы констант. Делать так нет никакой необходимости, поскольку имеется лучшая альтернатива. Пакет `java.util` предоставляет класс `EnumSet` для эффективного представления множеств значений, извлеченных из единственного перечислимого типа. Этот класс реализует интерфейс `Set`, предоставляя все богатство, безопасность типов и возможность взаимодействия, которые можно получить с любой другой реализацией `Set`. Но внутренне каждый `EnumSet` представляет собой битовый вектор. Если у базового типа перечисления 64 или менее элементов (для большинства это так), то весь `EnumSet` представлен одним значением типа `long`, так что его производительность сравнима с производительностью битового поля. Групповые операции, такие как `removeAll` и `retainAll`, реализованы с помощью побитовой арифметики, так как если бы вы работали вручную с битовыми полями. Но теперь вы изолированы от сложности и ошибок ручной работы с битами: `EnumSet` выполняет всю эту сложную работу для вас.

Вот как выглядит предыдущий пример, если изменить его так, чтобы он использовал перечисления и их множества вместо битовых полей. Он короче, яснее и безопаснее:

```
// EnumSet - современная замена битовых полей
public class Text
{
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Можно передать любой Set, но EnumSet наиболее понятен
    public void applyStyles(Set<Style> styles) { ... }
}
```

Вот клиентский код, который передает экземпляр EnumSet методу applyStyles. Класс EnumSet предоставляет богатый набор статических фабрик для легкого создания множества, одна из которых и показана в этом коде:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Обратите внимание, что метод applyStyles принимает Set<Style>, а не EnumSet<Style>. Хотя представляется вероятным, что все клиенты будут передавать методу именно EnumSet, в общем случае хорошая практика заключается в том, чтобы принимать тип интерфейса, а не тип реализации (раздел 9.8). Это обеспечивает возможность клиенту передать некую другую реализацию Set.

Итак, поскольку перечислимые типы будут использоваться в наборах, нет причин для их представления в виде битовых полей. Класс EnumSet объединяет в себе краткость и производительность битовых полей со всеми преимуществами типов перечислений, описанных в разделе 6.1. Единственный реальный недостаток EnumSet — невозможность создать неизменяемый EnumSet (по крайней мере, в версии Java 9), но он может быть исправлен в будущих версиях. А пока что вы можете обернуть EnumSet в оболочку Collections.unmodifiableSet, но при этом пострадают и краткость, и производительность.

6.4. Используйте EnumMap вместо индексирования порядковыми номерами

Иногда можно встретить код, который использует метод ordinal (раздел 6.2) для индексирования массива или списка. Рассмотрим, например, упрощенный класс, представляющий растение:

```
class Plant
{
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }
```

```

final String name;
final Lifecycle lifeCycle;

Plant(String name, Lifecycle lifeCycle) {
    this.name = name;
    this.lifeCycle = lifeCycle;
}

@Override public String toString() {
    return name;
}
}

```

Теперь предположим, что у вас есть массив растений, представляющий сад, и вы хотите перечислить эти растения, организовав их по типу жизненного цикла (однолетние, многолетние или двухлетние). Для этого вы создаете три множества, по одному для каждого типа, и выполняете обход сада, помещая каждое растение в соответствующее множество. Некоторые программисты могли бы сделать это, помещая наборы в массив, проиндексированный порядковым номером типа:

```

// Использование ordinal() для индексации в массиве - НЕ ДЕЛАЙТЕ ТАК!
Set<Plant>[] plantsByLifeCycle =
    (Set<Plant>[]) new Set[Plant.Lifecycle.values().length];

for (int i = 0; i < plantsByLifeCycle.length; i++)
    plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden)
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);

// Вывод результатов
for (int i = 0; i < plantsByLifeCycle.length; i++)
{
    System.out.printf("%s: %s%n",
        Plant.Lifecycle.values()[i],
        plantsByLifeCycle[i]);
}

```

Этот способ работает, но в нем много проблем. Так как массив несовместим с обобщенными типами (раздел 5.3), программе потребуется непроверяемое приведение, так что она будет компилироваться с предупреждениями. Поскольку массиву неизвестно, что собой представляет его индекс, вывод необходимо помечать вручную. Но самая серьезная проблема заключается в том, что когда вы получите доступ к массиву, индексированному порядковыми номерами перечисления, то на вас ляжет ответственность за использование корректных значений `int`; тип `int` не обеспечивает безопасность перечислений с точки

зрения типов. Если вы используете неверное значение, программа будет молча делать не то, что нужно, или — если вам повезет — сгенерирует исключение `ArrayIndexOutOfBoundsException`.

Имеется намного более удачный способ добиться того же результата. Массив эффективно выполняет роль отображения перечисления на значения, так что можно с тем же успехом использовать `Map`. Заметим, что есть очень быстрая реализация `Map`, спроектированная для использования с ключами-перечислениями, известная как `java.util.EnumMap`. Вот как выглядит программа, переписанная с использованием `EnumMap`:

```
// Использование EnumMap для связи данных с перечислениями
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class);
for (Plant.LifeCycle lc : Plant.LifeCycle.values())
    plantsByLifeCycle.put(lc, new HashSet<>());
for (Plant p : garden)
    plantsByLifeCycle.get(p.lifeCycle).add(p);
System.out.println(plantsByLifeCycle);
```

Эта программа короче, яснее, безопаснее и по скорости сравнима с исходной версией. Здесь нет небезопасных приведений, нет необходимости вручную помечать вывод, поскольку ключи отображения — перечисления, которые знают, как транслировать самих себя в выводимые строки, и нет никакой возможности для ошибок при вычислении индексов массивов. Причина, по которой `EnumMap` сравним по скорости с массивом, индексированным порядковыми значениями, заключается в том, что `EnumMap` использует такой массив внутренне. Но он скрывает детали этой реализации от программиста, объединяя богатство и безопасность типов, которые свойственны `Map`, со скоростью массивов. Обратите внимание, что конструктор `EnumMap` получает объект `Class` типа ключа: это *ограниченный токен типа*, который предоставляет информацию времени выполнения об обобщенном типе (раздел 5.8).

Предыдущая программа может быть еще более сокращена с помощью потока (раздел 7.4) для управления отображением. Вот простейший код на основе потока, который во многом дублирует поведение предыдущего примера:

```
// Простой подход на основе потока – вряд ли создаст EnumMap!
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.lifeCycle)));
```

Проблема этого кода заключается в том, что он выбирает собственную реализацию отображения, и на практике это будет не `EnumMap`, так что с точки зрения потребления памяти и времени работы этот код не будет соответствовать версии с явно указанным `EnumMap`. Чтобы исправить эту проблему, используйте `Collectors.groupingBy` с тремя параметрами, что позволяет

вызывающему объекту указать используемую реализацию карты с помощью параметра `mapFactory`:

```
// Использование потока и EnumMap для связи данных с перечислениями
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.lifeCycle,
        () -> new EnumMap<>(LifeCycle.class), toSet())));
```

К такой оптимизации не стоит прибегать в “игрушечных” программах наподобие показанной, но она может иметь решающее значение в программе, которая интенсивно использует отображение.

Поведение версий на основе потока несколько отличается от поведения версии с использованием `EnumMap`. Версия с использованием `EnumMap` всегда создает вложенные отображения для каждого жизненного цикла растений, в то время как версии на основе потока создают вложенное отображение, только если сад содержит одно или несколько растений с данным жизненным циклом. Так, например, если сад содержит только однолетние и многолетние растения, но не двухлетние, то размер `plantsByLifeCycle` будет равен трем в версии с использованием `EnumMap` и двум — в обеих версиях на основе потока.

Вы можете встретить и массив массивов, индексированный (дважды!) по порядковым значениям, используемым для представления отображения от двух значений перечислений. Например, показанная далее программа использует такой массив для связывания двух агрегатных состояний с фазовым переходом (переход жидкого состояния в твердое называется замерзанием, жидкого в газообразное — кипением и т.д.):

```
// Использование ordinal() для индексирования
// массива массивов - НЕ ДЕЛАЙТЕ ТАК!
public enum Phase
{
    SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

        // Строки индексированы порядковыми значениями from,
        // столбцы - порядковыми значениями to
        private static final Transition[][] TRANSITIONS = {
            { null, MELT, SUBLIME },
            { FREEZE, null, BOIL },
            { DEPOSIT, CONDENSE, null }
        };

        // Возвращает фазовый переход из одного
        // агрегатного состояния в другое
        public static Transition from(Phase from, Phase to)
        {
```

```

        return TRANSITIONS[from.ordinal()][to.ordinal()];
    }
}

```

Эта программа работает и может даже элегантно выглядеть, но это впечатление обманчиво. Как и в более простом примере с растениями в саду, у компилятора нет возможности знать отношения между порядковыми номерами и индексами массива. Если вы сделаете ошибку в таблице перехода или забудете обновить ее при изменении типов перечислений `Phase` или `Phase.Transition`, то вы получите ошибку при выполнении программы. Ошибка может принять вид исключения `ArrayIndexOutOfBoundsException`, `NullPointerException` или, что еще хуже, просто некорректных результатов работы. А размер таблицы будет квадратично зависеть от количества агрегатных состояний, даже если количество ненулевых записей будет значительно меньшим.

И вновь, все можно сделать гораздо лучше, если воспользоваться `EnumMap`. Поскольку каждый фазовый переход из одного агрегатного состояния в другое индексируется *парой* перечислений, лучше всего представить отношения в виде отображения, в котором одно перечисление (исходное агрегатное состояние) отображается на другое отображение, в котором целевое агрегатное состояние отображается на результат (фазовый переход). Две фазы, связанные с фазовым переходом, лучше всего записывать, связывая с перечислением для фазовых переходов, которое затем используется для инициализации вложенного `EnumMap`:

```

// Использование вложенного EnumMap
// для связи данных с парами перечислений
public enum Phase
{
    SOLID, LIQUID, GAS;
    public enum Transition
    {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);
        private final Phase from;
        private final Phase to;
        Transition(Phase from, Phase to) {
            this.from = from;
            this.to = to;
        }
    }

    // Инициализация отображения фазовых переходов
    private static final Map<Phase, Map<Phase, Transition>>
        m = Stream.of(values()).collect(groupingBy(t -> t.from,

```

```

        () -> new EnumMap<>(Phase.class),
        toMap(t -> t.to, t -> t,
            (x,y)->y, ()->new EnumMap<>(Phase.class))););

    public static Transition from(Phase from, Phase to)
    {
        return m.get(from).get(to);
    }
}
}

```

Код инициализации фазового перехода, может, немного сложен. Тип отображения имеет вид `Map<Phase, Map<Phase, Transition>>`, что означает “отображение исходного агрегатного состояния на отображение целевого агрегатного состояния на фазовый переход”. Отображение отображений инициализируется с использованием каскадной последовательности двух коллекторов. Первый коллектор группирует переходы по исходному агрегатному состоянию, а второй создает `EnumMap` с отображением целевого агрегатного состояния на фазовый переход. Функция слияния во втором коллекторе $(x, y) \rightarrow y$ не используется; она требуется только потому, что для получения `EnumMap` мы должны указать фабрику отображений, а `Collectors` предоставляет телескопические фабрики. В предыдущем издании этой книги использовалось явное итерирование для инициализации отображения фазового перехода. Код был более подробным, но, вероятно, более понятным.

Теперь предположим, что вы хотите добавить в систему новое агрегатное состояние: *плазму*, или ионизированный газ. Есть только два фазовых перехода, связанных с этой фазой: *ионизация*, которая преобразует газ в плазму, и *деионизация*, которая возвращает плазму в газ. Для обновления программы на основе массива вам пришлось бы добавить одну новую константу в `Phase` и две в `Phase.Transition`, а также заменить исходный девятиэлементный массив новой версией массива с шестнадцатью элементами. Если вы добавите в массив слишком много или слишком мало элементов или поместите элементы не в том порядке, то вам не повезет: программа скомпилируется, но не будет работать. Для обновления версии на основе `EnumMap` все, что вам надо сделать, — это добавить `PLASMA` в список агрегатных состояний, а также `IONIZE(GAS, PLASMA)` и `DEIONIZE(PLASMA, GAS)` в список фазовых переходов:

```

// Добавление нового агрегатного состояния
// с использованием реализации вложенного EnumMap
public enum Phase
{
    SOLID, LIQUID, GAS, PLASMA;

```

```
public enum Transition {
    MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
    BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
    SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
    IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);
    ... // Остальная часть кода - без изменений
}
```

Обо всем остальном программа позаботится сама, и у вас не будет возможности сделать ошибку. Внутренне отображение отображений реализовано как массив массивов, так что вы практически ничего не потеряете ни в смысле памяти, ни в смысле времени работы, получая взамен большую ясность, безопасность и простоту поддержки.

Для краткости в приведенных выше примерах используется `null` для указания отсутствия изменения агрегатного состояния (когда `to` и `from` идентичны). Это не является хорошей практикой и может привести к исключению `NullPointerException` во время выполнения. Проектирование ясного, элегантного решения этой проблемы оказывается удивительно сложным, а получающаяся в результате программа — достаточно длинной, чтобы отвлечь вас от основного материала этого раздела.

Итак, редко имеет смысл использовать порядковые номера для индексирования массивов: вместо этого лучше использовать `EnumMap`. Если представляемое вами отношение является многомерным, используйте `EnumMap<... , EnumMap<...>>`. Это частный случай общего принципа, гласящего о том, что прикладным программистам крайне редко требуется (если требуется вообще) использовать `Enum.ordinal` (раздел 6.2).

6.5. Имитируйте расширяемые перечисления с помощью интерфейсов

Почти во всех отношениях типы перечислений превосходят схему безопасных с точки зрения типов перечислений, описанную в первом издании этой книги [3]. Однако при этом имеется одно исключение, которое касается расширяемости, возможной в исходной схеме, но не поддерживаемой конструкцией языка. Другими словами, используя упомянутую схему, можно было сделать так, чтобы один тип перечисления расширял другой; используя же только средства языка, это сделать невозможно. И это не случайно. По большей части расширение типов перечислений оказывается плохой идеей. Запутывает то, что элементы расширенного типа являются экземплярами базового типа, но не наоборот. Не существует простого способа перечислить все элементы базового

типа и его расширений. Наконец, расширяемость усложняет многие аспекты дизайна и реализации.

С учетом сказанного существует по крайней мере один убедительный случай, оправдывающий использование расширяемого типа перечисления; это *коды операций*. Код операции относится к перечислимому типу, элементы которого представляют операции на некоторой машине, такому как тип `Operation` в разделе 6.1, представляющий функции простого калькулятора. Иногда желательно позволить пользователям API предоставлять собственные операции, эффективно расширяя набор операций, предоставляемых API.

К счастью, есть прекрасный способ достичь такого эффекта с использованием типов перечислений. Основная идея состоит в получении преимуществ из того факта, что типы перечислений могут реализовывать произвольные интерфейсы, определяя интерфейс для типа кодов операций и перечисление, являющееся стандартной реализацией интерфейса. Например, вот как выглядит расширяемая версия типа `Operation` из раздела 6.1:

// Имитация расширяемого перечисления с использованием интерфейса

```
public interface Operation
{
    double apply(double x, double y);
}
public enum BasicOperation implements Operation
{
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;

    BasicOperation(String symbol)
    {
        this.symbol = symbol;
    }

    @Override public String toString()
    {
        return symbol;
    }
}
```

Хотя тип перечисления (`BasicOperation`) не является расширяемым, тип интерфейса (`Operation`) таковым является, и это тот тип интерфейса, который используется для представления операций в API. Можно определить другой тип перечисления, который реализует этот интерфейс, и использовать экземпляры этого нового типа вместо экземпляров базового. Например, предположим, что вы хотите определить расширение показанного ранее типа операции, добавив операции возведения в степень и получения остатка от деления. Все, что вам нужно сделать, — написать тип перечисления, который реализует интерфейс `Operation`:

```
// Имитированное расширенное перечисление
public enum ExtendedOperation implements Operation
{
    EXP("^")
    {
        public double apply(double x, double y)
        {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%")
    {
        public double apply(double x, double y)
        {
            return x % y;
        }
    };

    private final String symbol;

    ExtendedOperation(String symbol)
    {
        this.symbol = symbol;
    }

    @Override public String toString()
    {
        return symbol;
    }
}
```

Теперь можно использовать ваши новые операции везде, где можно использовать базовые операции, при условии, что API написаны так, что принимают тип интерфейса (`Operation`), а не реализации (`BasicOperation`). Обратите внимание, что вам не нужно объявлять абстрактный метод `apply` в перечислении, как в нерасширяемом перечислении с реализацией метода, зависящего от конкретного экземпляра. Дело в том, что абстрактный метод (`apply`) является членом интерфейса (`Operation`).

Результирующий код немного менее сложен, а метод `test` немного более гибок: он позволяет вызывающему объекту объединить операции от нескольких типов реализации. С другой стороны, вы отказываетесь от возможности использования `EnumSet` (раздел 6.3) и `EnumMap` (раздел 6.4) для указанных операций.

Обе показанные ранее программы будут давать следующий вывод при запуске с аргументами командной строки 4 и 2:

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

Небольшим недостатком использования интерфейсов для эмуляции расширяемых перечислений является то, что реализации не могут быть наследованы от одного типа перечисления в другой. Если код реализации не основан ни на каком состоянии, он может быть помещен в интерфейс с помощью реализации по умолчанию (раздел 4.6). В нашем примере `Operation` необходимо продублировать логику хранения и извлечения символа, связанного с операцией, в `BasicOperation` и `ExtendedOperation`. В данном случае это не имеет значения, потому что дублируется очень небольшой код. Если количество совместно используемой функциональности велико, можно инкапсулировать ее во вспомогательный класс или во вспомогательный статический метод для устранения дублирования кода.

Описанная в этом разделе схема используется в библиотеках Java. Например, тип перечисления `java.nio.file.LinkOption` реализует интерфейсы `CopyOption` и `OpenOption`.

Таким образом, **хотя написать расширяемые типы перечислений нельзя, их можно эмулировать путем написания интерфейса с базовым типом перечисления, который реализует этот интерфейс.** Это позволяет клиентам писать собственные перечисления (или другие типы), которые реализуют этот интерфейс. Экземпляры этих типов могут затем использоваться везде, где могут использоваться экземпляры базового типа перечисления, в предположении, что API записываются с использованием интерфейса.

6.6. Предпочитайте аннотации схемам именования

Исторически было общепринятым использование *схем именования* (naming patterns) для указания, что некоторые элементы программы требуют специальной обработки инструментарием или каркасом. Например, до версии 4 каркас тестирования JUnit требовал от своих пользователей, чтобы разрабатываемые методы тестирования имели имена, начинающиеся с символов `test` [2]. Этот

способ работает, но обладает рядом недостатков. Во-первых, опечатки приводят к “тихим” сбоям. Например, предположим, что вы случайно назвали тестовый метод `tsetSafetyOverride`, а не `testSafetyOverride`. JUnit 3 не будет жаловаться, но и тест не выполнит, что приведет к ложному чувству безопасности.

Во-вторых, нет никакого способа гарантировать, что схемы именования используются только в соответствующих элементах программы. Например, предположим, что вы назвали класс `TestSafetyMechanisms` в надежде, что JUnit 3 автоматически проверит все его методы независимо от их названий. И вновь JUnit 3 не будет жаловаться, но и не будет выполнять какие-либо тесты.

В-третьих, схема именования не предоставляет хорошего способа связать значения параметров с элементами программы. Например, предположим, что вы хотите поддержать категорию тестов, которые успешны только в случае генерации конкретного исключения. Тип исключения, по существу, является параметром теста. Имя типа исключения можно закодировать в имени метода теста, используя некоторую схему именования, но это будет уродливо и ненадежно (раздел 9.6). Компилятор не может знать, что должен проверить соответствие строки имени генерируемому исключению. Если именованный класс не существует или не является исключением, вы не узнаете об этом до тех пор, пока не попытаетесь выполнить тест.

Аннотации [25, 9.7] красиво решают все эти проблемы, и JUnit работает с ними, начиная с версии 4. В этом разделе мы напишем собственный игрушечный тест, чтобы показать, как работают аннотации. Предположим, что вы хотите определить тип аннотации для обозначения простых тестов, которые запускаются автоматически и не проходят, если генерируется исключение. Вот как может выглядеть такая аннотации типа с именем `Test`:

```
// Объявление типа аннотации-маркера
import java.lang.annotation.*;

/**
 * Указывает, что аннотированный метод является тестовым.
 * Используется только для статических методов без параметров.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Объявление для типа аннотации `Test` само является аннотированным с помощью аннотаций `Retention` и `Target`. Такие аннотации объявлений типов аннотаций известны как *мета-аннотации* (meta-annotations). Метааннотация `@Retention(RetentionPolicy.RUNTIME)` указывает, что аннотации `Test`

должны быть доступны во время выполнения. Без нее аннотации `Test` будут невидимы для инструментария тестирования. Метааннотация `@Target(ElementType.METHOD)` указывает, что аннотация `Test` является допустимой только в объявлениях методов: она не может применяться в объявлениях классов, полей или других элементов программы.

Комментарий перед объявлением аннотации `Test` гласит: “Используется только для статических методов без параметров”. Было бы хорошо, если бы компилятор мог обеспечить выполнение этого условия, но это невозможно, если только вы не напишете для этого *процессор аннотаций*. Дополнительную информацию на эту тему ищите в документации по `javax.annotation.processing`. В отсутствие такого процессора при размещении аннотации `Test` в объявлении метода экземпляра или метода с одним или несколькими параметрами тестовая программа будет по-прежнему компилироваться, предоставляя тестирующему инструментарию разбираться с проблемой во время выполнения.

Вот как аннотация `Test` выглядит на практике. Она называется *аннотацией-маркером* (marker annotation), потому что не имеет параметров, а просто “маркирует” аннотируемый элемент. Если программист ошибется в написании `Test` или применит аннотацию `Test` к элементу программы, отличному от объявления метода, программа не скомпилируется:

```
// Программа, содержащая аннотации-маркеры
public class Sample
{
    @Test public static void m1() {} // Тест должен быть пройден
    public static void m2() { }
    @Test public static void m3() // Тест не должен быть пройден
    {
        throw new RuntimeException("Boom");
    }
    public static void m4() { }
    @Test public void m5() { } // НЕВЕРНОЕ ИСПОЛЬЗОВАНИЕ :
    // нестатический метод
    public static void m6() { }
    @Test public static void m7() // Тест не должен быть пройден
    {
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

Класс `Sample` имеет семь статических методов, четыре из которых аннотированы как тесты. Два из них, `m3` и `m7`, генерируют исключения, а два, `m1` и `m5`, этого не делают. Но один из аннотированных методов, не генерирующих

исключение (m5), является методом экземпляра и поэтому представляет собой неправильное использование аннотации. В целом, `Sample` содержит четыре теста: один должен быть успешно пройден, два — нет, и еще один тест является недопустимым. Четыре метода, не аннотированные как `Test`, будут игнорироваться тестирующим инструментарием.

Аннотации `Test` не влияют непосредственно на семантику класса `Sample`. Они служат только для предоставления информации для использования заинтересованными в ней программами. В целом, аннотации не изменяют семантику аннотированного кода, но обеспечивают его особую обработку таким инструментарием, как этот простой тестировщик:

```
// Программа для обработки аннотаций-маркеров
import java.lang.reflect.*;
public class RunTests
{
    public static void main(String[] args) throws Exception
    {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods())
        {
            if (m.isAnnotationPresent(Test.class))
            {
                tests++;
                try
                {
                    m.invoke(null);
                    passed++;
                }
                catch (InvocationTargetException wrappedExc)
                {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                }
                catch (Exception exc)
                {
                    System.out.println("Invalid @Test: " + m);
                }
            }
        }
        System.out.printf("Пройдено: %d, Сбоев: %d\n",
            passed, tests - passed);
    }
}
```

Этот инструментарий тестирования принимает в командной строке полное имя класса и рефлексивно выполняет все методы класса, аннотированные с помощью аннотации `Test`, путем вызова `Method.invoke`. Метод `isAnnotationPresent` сообщает инструменту, какие методы следует запустить. Если тестовый метод генерирует исключение, рефлексивная функциональность оборачивает его в `InvocationTargetException`. Тестировщик перехватывает это исключение и выводит сообщение о сбое, содержащее исходное исключение, сгенерированное тестируемым методом, которое извлекается из `InvocationTargetException` с помощью метода `getCause`.

Если попытка рефлексивного вызова тестируемого метода генерирует любое исключение, отличное от `InvocationTargetException`, это указывает на недопустимое использование аннотации `Test`, которое не было обнаружено во время компиляции. Такое некорректное использование включает аннотацию метода экземпляра, метода с одним или несколькими параметрами или недоступного метода. Второй блок `catch` тестировщика перехватывает эти ошибки использования аннотации `Test` и выводит соответствующее сообщение об ошибке. Вот вывод `RunTests` при тестировании `Sample`:

```
public static void Sample.m3() failed: RuntimeException: Boom
Invalid @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Пройдено: 1, Сбоев: 3
```

Теперь добавим поддержку тестов, которые завершаются успешно, только если они генерируют определенное исключение. Для этого нам нужен новый тип аннотации:

```
// Тип аннотации с параметром
import java.lang.annotation.*;
/**
 * Указывает, что аннотированный метод является
 * тестирующим методом, который для успешного завершения
 * должен генерировать указанное исключение
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> value();
}
```

Тип параметра этой аннотации — `Class<? extends Throwable>`. Этот тип с символом подстановки достаточно громоздкий. На простом человеческом языке это означает “объект `Class` для некоторого класса, который расширяет класс `Throwable`” и позволяет пользователю аннотации указать любой тип исключения (или ошибки). Это использование является примером

ограниченного токена типа (bounded type token) (раздел 5.8). Ниже показано, как эта аннотация выглядит на практике. Обратите внимание, что литералы классов используются в качестве значения для параметра аннотации:

```
// Программа, содержащая аннотацию с параметром
public class Sample2
{
    @ExceptionTest(ArithmeticException.class)
    public static void m1() // Тест должен быть пройден
    {
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() // Тест не должен быть пройден
    { // (неверное исключение)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { } // Тест не должен быть пройден
    // (нет исключения)
}
```

Теперь давайте изменим инструментарий тестирования для обработки новой аннотации. Для этого добавим следующий код в метод main:

```
if (m.isAnnotationPresent(ExceptionTest.class))
{
    tests++;

    try
    {
        m.invoke(null);
        System.out.printf("Сбой теста %s: нет исключения%n",m);
    }
    catch (InvocationTargetException wrappedEx)
    {
        Throwable exc = wrappedEx.getCause();
        Class <? extends Throwable > excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc))
        {
            passed++;
        }
        else
        {
            System.out.printf(
                "Сбой теста %s: ожидался %s, получен %s%n",
                m, excType.getName(), exc);
        }
    }
}
```

```

    }
    catch (Exception exc)
    {
        System.out.println("Invalid @Test: " + m);
    }
}

```

Этот код похож на тот, который мы использовали для обработки аннотации `Test`, с одним отличием: он извлекает значение параметра аннотации и использует его для проверки, правильного ли типа исключение генерируется проверяемым методом. Явного приведения здесь нет, а следовательно, нет никакой опасности генерации исключения `ClassCastException`. Тот факт, что тестовая программа скомпилирована, гарантирует, что параметры аннотаций представляют допустимые типы исключений, с одной оговоркой: если параметры аннотаций были действительны во время компиляции, но файл класса, представляющий тип указанного исключения, отсутствует во время выполнения, то программа тестирования будет генерировать исключение `TypeNotPresentException`.

Сделаем очередной шаг с нашим примером исключения. Легко представить себе тест, который считается успешно пройденным, если он генерирует любое из нескольких указанных исключений. Механизм аннотаций имеет функциональную возможность, которая позволяет легко поддерживать такое использование. Предположим, что мы изменяем тип параметра аннотации `ExceptionTest` массивом объектов `Class`:

```

// Тип аннотации с параметром массива
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest
{
    Class<? extends Exception>[] value();
}

```

Синтаксис для массива параметров в аннотации достаточно гибкий. Он оптимизирован для одноэлементных массивов. Все предыдущие аннотации `ExceptionTest` по-прежнему действительны и с новой версией `ExceptionTest` с параметром-массивом и дают в результате одноэлементный массив. Чтобы указать массив из нескольких элементов, окружите эти элементы фигурными скобками и разделите их запятыми:

```

// Код, содержащий аннотацию с параметром-массивом
@ExceptionTest({IndexOutOfBoundsException.class,
                NullPointerException.class })
public static void doublyBad()
{
    List<String> list = new ArrayList<>();
}

```

```

// Спецификация разрешает этому методу генерировать
// IndexOutOfBoundsException или NullPointerException
list.addAll(5, null);
}

```

Достаточно просто изменить программу тестирования для обработки новой версии `ExceptionTest`. Вот код, который заменяет исходную версию:

```

if (m.isAnnotationPresent(ExceptionTest.class))
{
    tests++;
    try
    {
        m.invoke(null);
        System.out.printf("Сбой теста %s: нет исключения%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        int oldPassed = passed;
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Сбой теста %s: %s %n", m, exc);
    }
}

```

Начиная с Java 8 имеется еще один способ сделать многозначные аннотации. Вместо объявления типа аннотации с параметром-массивом можно аннотировать объявление аннотации с помощью метааннотации `@Repeatable`, чтобы указать, что аннотация может применяться к одному элементу несколько раз. Эта метааннотация принимает единственный параметр, который является объектом класса *типа, содержащего аннотацию* (containing annotation type), единственным параметром которого является массив типа аннотации [25, 9.6.3]. Вот как выглядит объявление аннотации, если мы применим этот подход к нашей аннотации `ExceptionTest`. Обратите внимание, что тип, содержащий аннотацию, должен быть аннотирован с соблюдением соответствующих стратегии и цели, иначе объявление не скомпилируется:

```

// Повторяемый тип аннотации
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)

```

```

public @interface ExceptionTest
{
    Class <? extends Exception > value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer
{
    ExceptionTest[] value();
}

```

Вот как выглядит наш тест `doublyBad` с использованием повторяющихся аннотаций вместо аннотации с массивом:

```

// Код, содержащий повторяющуюся аннотацию
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }

```

Обработка повторяющихся аннотаций требует аккуратности. Повторяющиеся аннотации генерируют синтетическую аннотацию типа, содержащего аннотацию. Метод `getAnnotationsByType` скрывает этот факт и может использоваться для доступа к повторяющимся и не повторяющимся аннотациям с типом повторяемых аннотаций. Но `isAnnotationPresent` делает явным то, что повторяющаяся аннотация не имеет типа аннотации, но является типом, содержащим аннотацию. Если элемент имеет повторяющиеся аннотации некоторого типа и вы используете метод `isAnnotationPresent` для проверки, что элемент имеет аннотацию этого типа, то вы обнаружите, что это не так. Следовательно, использование этого метода для проверки наличия типа аннотации приведет к тому, что программа будет молча игнорировать повторяющиеся аннотации. Аналогично использование этого метода для проверки типа, содержащего аннотацию, приведет к тому, что программа будет молча игнорировать не повторяющиеся аннотации. Чтобы обнаружить повторяющиеся и не повторяющиеся аннотации с помощью `isAnnotationPresent`, вы должны проверить как тип аннотации, так и тип, содержащий эту аннотацию. Вот как выглядит соответствующая часть нашей программы `RunTests` после внесения изменений для использования повторяемой версии аннотации `ExceptionTest`:

```

// Обработка повторяемых аннотаций
if (m.isAnnotationPresent(ExceptionTest.class)
    || m.isAnnotationPresent(ExceptionTestContainer.class))
{
    tests++;
}

```

```

try
{
    m.invoke(null);
    System.out.printf("Сбой теста %s: нет исключения%n", m);
}
catch (Throwable wrappedExc)
{
    Throwable exc = wrappedExc.getCause();
    int oldPassed = passed;
    ExceptionTest[] excTests =
        m.getAnnotationsByType(ExceptionTest.class);
    for (ExceptionTest excTest : excTests)
    {
        if (excTest.value().isInstance(exc))
        {
            passed++;
            break;
        }
    }

    if (passed == oldPassed)
        System.out.printf("Сбой теста %s: %s %n", m, exc);
}
}

```

Повторяемые аннотации были добавлены для улучшения удобочитаемости исходного кода, который логически применяет несколько экземпляров одного и того же типа аннотации к данному элементу программы. Если вы считаете, что они улучшают удобочитаемость вашего исходного кода, используйте их, но помните, что есть и другие шаблоны объявления и обработки повторяющихся аннотаций и что обработка повторяющихся аннотаций подвержена ошибкам.

Каркас тестирования в этом разделе — просто игрушка, но он ясно демонстрирует превосходство аннотаций над схемами именования, и это только поверхностное знакомство с предоставляемыми ими возможностями. Если вы пишете инструментарий, который требует от программистов добавления информации в исходный код, определите соответствующие типы аннотаций. **Просто не существует причин для использования схем именования, когда вместо этого можно использовать аннотации.**

Впрочем, большинству программистов не требуется самостоятельно определять типы аннотаций. Но **все программисты должны использовать предопределенные типы аннотаций, предоставляемые Java** (разделы 6.7, 5.2). Кроме того, рассмотрите вопрос об использовании аннотаций, предоставляемых интегрированной средой разработки или инструментами статического анализа. Такие аннотации могут улучшить качество диагностической

информации, представляемой этим инструментарием. Обратите, однако, внимание, что эти аннотации пока еще не стандартизированы, поэтому вам может потребоваться выполнить некоторую работу при смене инструментария или изменении стандарта.

6.7. Последовательно используйте аннотацию `Override`

Java-библиотеки содержат несколько типов аннотаций. Для типичного программиста наиболее важным из них является `@Override`. Эта аннотация может использоваться только в объявлении метода и указывает, что объявление аннотированного метода перекрывает объявление метода в супертипе. Если вы последовательно используете эту аннотацию, она будет защищать вас от большого класса отвратительных ошибок. Рассмотрим программу, в которой класс `Bigram` представляет *биграмму* — упорядоченную пару букв:

```
// Можете ли вы обнаружить ошибку?
public class Bigram
{
    private final char first;
    private final char second;
    public Bigram(char first, char second)
    {
        this.first = first;
        this.second = second;
    }
    public boolean equals(Bigram b)
    {
        return b.first == first && b.second == second;
    }
    public int hashCode()
    {
        return 31 * first + second;
    }
    public static void main(String[] args)
    {
        Set<Bigram> s = new HashSet<>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

Основная программа добавляет в множество двадцать шесть биграмм, каждая из которых состоит из двух идентичных строчных букв. Затем она выводит размер множества. Вы могли бы ожидать, что программа выведет число 26, так как множество не может содержать дубликаты. Но если вы попытаетесь запустить программу, то обнаружите, что она выводит не 26, а 260. Что же пошло не так?

Очевидно, что автор класса `Bigram` намеревался перекрыть метод `equals` (раздел 3.1) и даже не забыл перекрыть `hashCode` (раздел 3.2). К сожалению, наш незадачливый программист не перекрыл `equals`, а перегрузил его (раздел 8.4). Для перекрытия `Object.equals` необходимо определить метод `equals`, параметр которого имеет тип `Object`, но параметр метода `equals` класса `Bigram` — не `Object`, поэтому `Bigram` наследует метод `equals` класса `Object`. Этот метод проверяет *тождественность* объектов, так же как и оператор `==`. Каждая из десяти копий каждого биграмма отличается от остальных девяти, и потому они рассматриваются `Object.equals` как неравные, что и объясняет, почему программа выводит 260.

К счастью, компилятор может помочь вам найти эту ошибку, но только если вы поможете ему, указав, что вы намерены перекрыть `Object.equals`. Для этого следует аннотировать `Bigram.equals` с помощью аннотации `@Override`, как показано далее:

```
@Override public boolean equals(Bigram b)
{
    return b.first == first && b.second == second;
}
```

Если вы добавите эту аннотацию и попытаете перекомпилировать программу, компилятор выдаст сообщение об ошибке наподобие показанного:

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

Вы сразу же поймете, что поступили неправильно, хлопнете себя по лбу и замените неверную реализацию `equals` верной (раздел 3.1):

```
@Override public boolean equals(Object o)
{
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

Таким образом, следует использовать аннотацию `Override` для каждого объявления метода, которое, как вы полагаете, перекрывает объявление суперкласса. Имеется одно небольшое исключение из этого правила. Если вы пишете класс, который не помечен как абстрактный, и вы уверены, что он перекрывает абстрактный метод в его суперклассе, то не беспокойтесь об аннотации `Override` для этого метода. В классе, который не объявлен как абстрактный, компилятор будет выдавать сообщение об ошибке, если вы не перекроете метод абстрактного суперкласса. Однако вы, возможно, пожелаете обратить внимание на все методы класса, которые перекрывают методы суперкласса; в этом случае вы можете свободно аннотировать и эти методы. Большинство сред разработки можно настроить таким образом, чтобы они добавляли аннотацию `Override` автоматически, когда вы решаете перекрыть метод.

Большинство сред разработки обеспечивают еще одну причину для последовательного использования аннотации `Override`. Если вы включите соответствующую проверку, интегрированная среда разработки будет генерировать предупреждение при наличии метода, у которого нет аннотации `Override`, но который перекрывает метод суперкласса. При последовательном использовании аннотации `Override` интегрированная среда разработки будет предупреждать вас о непреднамеренном перекрытии. Они дополняют сообщения компилятора об ошибках и предупреждают о непреднамеренных сбоях перекрытия. При использовании и интегрированной среды разработки, и компилятора вы можете быть уверены, что перекрываете методы везде, где хотите, и нигде больше.

Аннотация `Override` может использоваться с объявлениями методов, которые перекрывают объявления как в интерфейсах, так и в классах. С появлением методов по умолчанию хорошей практикой является использование `Override` в конкретных реализациях методов интерфейса для гарантии корректности сигнатуры. Если вы знаете, что интерфейс не имеет методов по умолчанию, можете опустить аннотацию `Override` для конкретных реализаций методов интерфейсов во избежание лишнего многословия.

Однако в абстрактном классе или интерфейсе *стоит* аннотировать *все* методы, которые перекрывают методы суперкласса или суперинтерфейса, не важно — конкретного или абстрактного. Например, интерфейс `Set` не добавляет новые методы к интерфейсу `Collection`, поэтому он должен включать аннотацию `Override` для всех своих объявлений методов, чтобы гарантировать, что он непреднамеренно не добавит никаких новых методов в интерфейс `Collection`.

Итак, компилятор может защитить вас от большого количества ошибок при использовании аннотации `Override` с каждым объявлением метода, который перекрывает объявление супертипа, с одним исключением. В конкретных

классах не нужно комментировать методы, которые, как вы считаете, перекрывают объявления абстрактных методов (хотя их аннотирование не приносит никакого вреда).

6.8. Используйте интерфейсы-маркеры для определения типов

Интерфейс-маркер (или маркерный интерфейс; `marker interface`) представляет собой интерфейс, не содержащий объявлений методов, а лишь указывающий (“маркирующий”) класс, реализующий интерфейс, как имеющий определенные свойства. Например, рассмотрим интерфейс `Serializable` (глава 12, “Сериализация”). Путем реализации этого интерфейса класс указывает, что его экземпляры могут записываться в `ObjectOutputStream` (быть “сериализованными”).

Возможно, вы слышали, что аннотации-маркеры (раздел 6.6) делают интерфейсы-маркеры устаревшими. Это неверное утверждение. Интерфейсы-маркеры обладают двумя преимуществами по сравнению с аннотациями-маркерами. Первое, и основное, заключается в том, что **интерфейсы-маркеры определяют тип, который реализуется экземплярами маркированного класса; аннотации-маркеры этим свойством не обладают.** Существование типа интерфейсов-маркеров позволяет вам отлавливать ошибки во время компиляции, которые при использовании аннотаций-маркеров вы можете не заметить до выполнения программы.

Возможности сериализации в Java (глава 6, “Перечисления и аннотации”) использует интерфейс-маркер `Serializable` для указания, что тип является сериализуемым. Метод `ObjectOutputStream.writeObject`, который сериализует переданный ему объект, требует, чтобы его аргумент был сериализуемым. Таким образом, неуместные попытки сериализовать объект могут быть выявлены во время компиляции (путем проверки типов). Именно в обнаружении ошибок во время компиляции состоит предназначение интерфейсов-маркеров, но, к сожалению, API `ObjectOutputStream.write` не использует преимущества интерфейса `Serializable`: его аргумент объявлен как имеющий тип `Object`, поэтому попытки сериализовать несериализуемый объект не приведут к проблемам до времени выполнения.

Еще одно преимущество интерфейсов-маркеров перед аннотациями-маркерами заключается в том, что они могут быть более точно нацелены. Если тип аннотации `@Target` объявлен как `ElementType.TYPE`, она может применяться к *любому* классу или интерфейсу. Предположим, что у вас есть маркер, который применим только к реализации определенного интерфейса.

Определив его как интерфейс-маркер, вы можете расширить единственный интерфейс, к которому он применим, гарантируя, что все маркированные типы также являются подтипами единственного интерфейса, к которому он применим.

Возможно, интерфейс `Set` является именно таким *ограниченным интерфейсом-маркером*. Он применим только к подтипам `Collection`, но не добавляет новые методы, помимо тех, которые определены в `Collection`. Вообще говоря, его нельзя рассматривать как интерфейс-маркер, потому что он уточняет контракты нескольких методов `Collection`, включая `add`, `equals` и `hashCode`. Но легко представить себе интерфейс-маркер, который применяется только к подтипам некоторого конкретного интерфейса и *не* уточняет контракты ни одного из методов интерфейса. Такой интерфейс-маркер может описать некоторый инвариант всего объекта или указать, что экземпляры подходят для обработки методом некоторого иного класса (как интерфейс `Serializable` указывает, что экземпляры могут быть обработаны `ObjectOutputStream`).

Главным преимуществом аннотаций-маркеров перед интерфейсами-маркерами является то, что они являются частью более мощной системы аннотаций. Таким образом, аннотации-маркеры обеспечивают согласованность в каркасах на основе аннотаций.

Так когда же следует использовать аннотацию-маркер, а когда — интерфейс-маркер? Ясно, что необходимо использовать аннотации, если маркер применяется к любому элементу программы, кроме класса или интерфейса, поскольку только классы и интерфейсы могут быть сделаны реализующими или расширяющими интерфейсы. Если маркер применяется только к классам и интерфейсам, задайте себе вопрос “Может быть, я хочу ограничиться одним или несколькими методами, которые принимают только объекты, имеющие данную маркировку?” Если это так, следует предпочесть интерфейс-маркер. Это позволит вам использовать интерфейс в качестве типа параметра для рассматриваемых методов, что в результате обеспечит проверку типов во время компиляции. Если вы можете убедить себя, что вы никогда не захотите написать метод, который принимает только объекты с данной маркировкой, то, вероятно, лучше использовать аннотации-маркеры. Если, кроме того, маркировка является частью каркаса, который интенсивно использует аннотации, то очевидно, что правильным выбором будет аннотация-маркер.

Итак, свое применение находят и интерфейсы-маркеры, и аннотации-маркеры. Чтобы определить тип, который не имеет каких-либо новых методов, связанных с ним, лучше использовать интерфейс-маркер. Если же вы хотите пометить элементы программы, отличные от классов и интерфейсов, или вписать маркеры в каркасы с интенсивным использованием аннотаций, то правильным

выбором является аннотация-маркер. Если вы пишете тип аннотации-маркера, целью которого является `ElementType.TYPED`, то потратьте время на выяснение, действительно ли это должен быть тип аннотации или же более целесообразным будет применение интерфейса-маркера.

В некотором смысле этот раздел является противоположностью раздела 4.8, который гласит “Если вы не хотите определять тип, не используйте интерфейс”. Этот раздел в первом приближении гласит иное: “Если вы хотите определить тип, используйте интерфейс”.

Лямбда-выражения и потоки

В Java 8 были добавлены функциональные интерфейсы, лямбда-выражения и ссылки на методы, упрощающие создание функциональных объектов. В тандеме с этими изменениями был добавлен API потоков для предоставления библиотечной поддержки обработки последовательностей элементов данных. В этой главе мы рассмотрим, как наилучшим образом использовать эти возможности.

7.1. Предпочитайте лямбда-выражения анонимным классам

Исторически интерфейсы (или, реже, абстрактные классы) с одним абстрактным методом использовались как *функциональные типы* (function types). Их экземпляры, известные как *функциональные объекты* (function objects), представляют функции или действия. Со времени выпуска JDK 1.1 в 1997 году основным средством для создания функциональных объектов были *анонимные классы* (anonymous class) (раздел 4.10). Вот фрагмент кода для сортировки списка строк по длине, использующий анонимный класс для создания функции сравнения для сортировки (которая определяет порядок сортировки):

```
// Экземпляр анонимного класса в качестве
// функционального объекта - устарело!
Collections.sort(words,
    new Comparator<String>()
    {
        public int compare(String s1, String s2)
        {
            return Integer.compare(s1.length(), s2.length());
        }
    });
```

Анонимные классы были адекватными для классических проектных шаблонов объектно-ориентированного проектирования, требующих наличия

функциональных объектов, в особенности для такого проектного шаблона, как *Стратегия* (Strategy) [12]. Интерфейс Comparator представляет *абстрактную стратегию* (abstract strategy) для сортировки; показанный выше анонимный класс является *конкретной стратегией* (concrete strategy) для сортировки строк. Многословность реализации анонимных классов, однако, сделала функциональное программирование в Java не слишком привлекательной перспективой.

В Java 8 язык формализовал концепцию интерфейсов с единственным абстрактным методом как отдельную, заслуживающую особой обработки. Эти интерфейсы сейчас известны как *функциональные интерфейсы* (functional interfaces), а язык позволяет создавать их экземпляры с помощью *лямбда-выражений* (lambda expressions, иногда для краткости именуемых просто *лямбдами*). Лямбда-выражения функционально аналогичны анонимным классам, но гораздо более краткие. Вот как выглядит показанный ранее фрагмент кода с анонимным классом при замене последнего лямбда-выражением. Все лишнее убрано, и поведение становится совершенно очевидным:

```
// Лямбда-выражение в качестве функционального объекта
// (замена анонимного класса)
Collections.sort(words,
    (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Обратите внимание, что типы лямбда-выражения (Comparator<String>), его параметров (s1 и s2, оба — String) и его возвращаемого значения (int) в коде не представлены. Компилятор выводит эти типы из контекста с помощью процесса, известного как *вывод типа* (type inference). В некоторых случаях компилятор не может вывести типы, и их следует указывать явно. Правила вывода типа сложны: они занимают целую главу книги [25, глава 18]. Немногие программисты понимают эти правила в деталях, но это нормально. **Опустите типы всех параметров лямбда-выражения, если только они не делают вашу программу яснее.** Если компилятор выдает ошибку, говорящую, что он не может вывести тип параметра лямбда-выражения, *тогда* укажите его. Иногда может потребоваться приведение типа возвращаемого значения или всего лямбда-выражения, но это бывает редко.

Следует добавить одно предостережение относительно вывода типов. В разделе 5.1 требуется не использовать несформированные типы, в разделе 5.4 говорится о пользе обобщенных типов, а в разделе 5.5 рекомендуется предпочитать обобщенные методы. Этот совет вдвойне важен при использовании лямбда-выражений, потому что компилятор получает большую часть информации о типе, которая позволяет выполнять вывод, от обобщенных типов. Если вы не предоставляете эту информацию, компилятор будет не в состоянии

выполнить вывод типа, и вам придется вручную указывать типы в ваших лямбда-выражениях, увеличивая многословность последних. В качестве примера показанный выше фрагмент кода не будет компилироваться, если переменная `words` будет объявлена как имеющая несформированный тип `List` вместо параметризованного типа `List<String>`.

Кстати, компаратор в фрагменте кода может быть сделан даже более кратким, если вместо лямбда-выражения использовать *метод построения компаратора* (разделы 3.5 и 7.2):

```
Collections.sort(words, comparingInt(String::length));
```

На самом деле фрагмент можно сделать еще короче, воспользовавшись методом `sort`, который был добавлен в интерфейс `List` в Java 8:

```
words.sort(comparingInt(String::length));
```

Добавление лямбда-выражений в язык делает практичным использование функциональных объектов, в которых ранее это не имело бы смысла. Например, рассмотрим тип перечисления `Operation` из раздела 6.1. Поскольку каждому значению перечисления требуется иное поведение его метода `apply`, мы использовали тела классов, зависящие от констант, и перекрыли метод `apply` в каждой константе перечисления. Чтобы освежить вашу память, приведем соответствующий код:

```
// Тип перечисления с телами и данными классов,
// зависящими от констант (раздел 6.1)
public enum Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    Operation(String symbol) { this.symbol = symbol; }
    @Override public String toString() { return symbol; }

    public abstract double apply(double x, double y);
}
```

Раздел 6.1 гласит, что поля экземпляра перечисления предпочтительнее тел классов, зависящих от констант. Лямбда-выражения облегчают реализацию поведения, зависящего от констант, с использованием первых вместо последних. Просто передайте реализацию поведения каждой константы в виде лямбда-выражения ее конструктору. Конструктор сохраняет лямбда-выражение в поле экземпляра, а метод `apply` перенаправляет вызовы лямбда-выражению. Результирующий код получается проще и понятнее, чем исходная версия:

```
// Перечисление с полями — функциональными объектами
// и поведением, зависящим от констант
public enum Operation
{
    PLUS("+", (x, y) -> x + y),
    MINUS("-", (x, y) -> x - y),
    TIMES("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);
    private final String symbol;
    private final DoubleBinaryOperator op;
    Operation(String symbol, DoubleBinaryOperator op)
    {
        this.symbol = symbol;
        this.op = op;
    }
    @Override public String toString()
    {
        return symbol;
    }
    public double apply(double x, double y)
    {
        return op.applyAsDouble(x, y);
    }
}
```

Обратите внимание, что мы используем интерфейс `DoubleBinaryOperator` для лямбда-выражений, которые представляют собой поведение констант перечислений. Это один из многих предопределенных функциональных интерфейсов в `java.util.function` (раздел 7.3). Он представляет функцию, которая получает два аргумента типа `double` и возвращает результат типа `double`.

Рассматривая перечисление `Operation` на основе лямбда-выражений, вы можете решить, что методы, зависящие от констант, отжили свое, но это не так. В отличие от методов и классов, у лямбда-выражений отсутствуют имена и документация; если вычисление не является не требующим пояснений или превышает несколько строк, не используйте для него лямбда-выражение.

Для лямбда-выражения идеальна одна строка кода, а три строки представляют собой разумный максимум. Если вы нарушаете это правило, это может нанести серьезный ущерб удобочитаемости ваших программ. Если лямбда-выражение длинное или сложное для чтения, либо найдите способ его упростить, либо выполните рефакторинг программы для его устранения. Кроме того, аргументы, передаваемые в конструкторы перечислений, вычисляются в статическом контексте. Таким образом, лямбда-выражения в конструкторах перечислений не могут обращаться к членам экземпляров перечислений. Тела классов, зависящие от констант, по-прежнему актуальны, если тип перечисления имеет сложное поведение, зависящее от констант, которое не удастся реализовать парой строк, или требующее доступ к полям или методам экземпляра.

Точно так же вы можете подумать, что в эпоху лямбда-выражений устарели и анонимные классы. Это ближе к истине, но есть несколько вещей, которые можно сделать с анонимными классами, и нельзя — с лямбда-выражениями. Лямбда-выражения ограничены функциональными интерфейсами. Если нужно создать экземпляр абстрактного класса, то это можно сделать с помощью анонимного класса, но не лямбда-выражения. Аналогично анонимные классы можно использовать для создания экземпляров интерфейсов с несколькими абстрактными методами. Наконец, лямбда-выражение не может получить ссылку на себя. В лямбда-выражении ключевое слово `this` относится к включающему экземпляру, который обычно является тем, что вам нужно. В анонимном классе ключевое слово `this` ссылается на экземпляр анонимного класса. Если вам необходим доступ к функциональному объекту из его тела, то вы должны использовать анонимный класс.

Лямбда-выражения используют общее с анонимными классами свойство, что их нельзя надежно сериализовать и десериализовать между реализациями. Таким образом, **вы должны редко (если вообще придется это делать) сериализовать лямбда-выражение (или экземпляр анонимного класса)**. Если у вас есть функциональный объект, который вы хотите сделать сериализуемым, такой как `Comparator`, используйте экземпляр частного закрытого статического вложенного класса (раздел 4.10).

Резюме: по состоянию на Java 8 лямбда-выражения на сегодняшний день являются наилучшим средством представления малых функциональных объектов. **Не используйте анонимные классы для функциональных объектов, если только вы не должны создавать экземпляры типов, не являющиеся функциональными интерфейсами.** Кроме того, помните, что лямбда-выражения позволяют легко представлять малые функциональные объекты, что открывает двери для методов функционального программирования, которых ранее практически не было в Java.

7.2. Предпочитайте ссылки на методы лямбда-выражениям

Основным преимуществом лямбда-выражений перед анонимными классами является то, что они более кратки. Но Java предоставляет средство создания функциональных объектов, еще более простое, чем лямбда-выражения: *ссылки на методы* (method references). Вот фрагмент кода из программы, которая поддерживает отображения произвольных ключей на значения `Integer`. Если значение интерпретируется как число экземпляров ключа, то программа представляет собой реализацию мультимножества. Функция в фрагменте кода связывает число 1 с ключом, если он отсутствует в отображении, и увеличивает связанное с ним значение, если ключ уже есть в отображении:

```
map.merge(key, 1, (count, incr) -> count + incr);
```

Обратите внимание, что этот код использует метод `merge`, который был добавлен в интерфейс `Map` в Java 8. Если данный ключ не существует в отображении, метод просто вставляет в него заданное значение. Если соответствующий элемент уже присутствует в отображении, то `merge` применяет заданную функцию к текущему значению и заменяет текущее значение результатом. Этот код представляет собой типичный случай использования метода `merge`.

Код легко читается, но все же остается несколько шаблонным. Параметры `count` и `incr` занимают достаточно много места, не выполняя реального суммирования. В действительности все, что говорит нам это лямбда-выражение, — это то, что функция возвращает сумму двух аргументов. Начиная с Java 8 `Integer` (и все другие упакованные численные примитивные типы) предоставляют статический метод `sum`, который делает в точности то же самое. Мы можем просто передать ссылку на этот метод и получить тот же результат с меньшими визуальными помехами:

```
map.merge(key, 1, Integer::sum);
```

Чем больше параметров у метода, тем больше исходного текста можно сэкономить с помощью ссылки на метод. Однако в некоторых лямбда-выражениях выбираемые вами имена параметров представляют собой полезную документацию, которая делает лямбда-выражения более удобочитаемыми и удобными, чем ссылка на метод, даже если лямбда-выражение оказывается более длинным.

Нет ничего такого, что вы могли бы сделать с помощью ссылки на метод, но не могли бы сделать с помощью лямбда-выражения (с одним малозначимым исключением — см. [25, 9.9-2], если вас это интересует). Как уже говорилось, ссылки на метод обычно дают более короткий и понятный код. Они также предоставляют путь решения вопроса со слишком длинным или слишком сложным лямбда-выражением: можно просто извлечь код лямбда-выражения в

новый метод и заменить лямбда-выражение ссылкой на этот метод. Вы можете дать методу хорошее имя и подробно его документировать.

Если вы программируете с использованием интегрированной среды разработки, она будет предлагать вам заменить лямбда-выражение ссылкой на метод везде, где это возможно. Обычно (но не всегда) следует принимать это предложение интегрированной среды разработки. Иногда лямбда-выражение будет более кратким, чем ссылка на метод. Чаще всего это происходит, когда метод находится в том же классе, что и лямбда-выражение. Например, рассмотрим фрагмент, который, допустим, имеется в классе с именем `GoshThisClassNameIsHumongous`:

```
service.execute(GoshThisClassNameIsHumongous::action);
```

Эквивалент с лямбда-выражением имеет следующий вид:

```
service.execute(() -> action());
```

Фрагмент со ссылкой на метод ни короче, ни яснее фрагмента с лямбда-выражением, поэтому в данном случае оно предпочтительнее. Аналогично интерфейс `Function` предоставляет обобщенный статический фабричный метод, возвращающий тождественную функцию, `Function.identity()`. Но, как правило, короче и яснее *не* использовать этот метод, прибегая к встроенному лямбда-эквиваленту `x->x`.

Многие ссылки на методы ссылаются на статические методы, но есть четыре разновидности, которые этого не делают. Два из них представляют собой *ограниченные* (`bound`) и *неограниченные* (`unbound`) ссылки на методы экземпляра. В ограниченных ссылках получающий объект указан в ссылке на метод. Ограниченные ссылки по своей природе аналогичны статическим ссылкам: функциональный объект принимает те же аргументы, что и указываемый метод. В неограниченных ссылках получающий объект указывается при применении функционального объекта через дополнительный параметр перед объявленными параметрами метода. Неограниченные ссылки часто используются как отображения и функции-фильтры в конвейерах потоков (раздел 7.4). Наконец, существуют две разновидности ссылок на *конструкторы*, для классов и массивы. Ссылки на конструкторы служат в качестве объектов-фабрик. Все пять разновидностей ссылок на методы приведены в следующей таблице.

Тип ссылки на метод	Пример	Эквивалентное лямбда-выражение
Статическая	<code>Integer::parseInt</code>	<code>str -> Integer.parseInt(str)</code>
Ограниченная	<code>Instant.now()::isAfter</code>	<code>Instant then = Instant.now(); t -> then.isAfter(t)</code>

Тип ссылки на метод	Пример	Эквивалентное лямбда-выражение
Неограниченная	<code>String::toLowerCase</code>	<code>str -> str.toLowerCase()</code>
Конструктор класса	<code>TreeMap<K,V>::new</code>	<code>() -> new TreeMap<K,V></code>
Конструктор массива	<code>int[]::new</code>	<code>len -> new int[len]</code>

Итак, ссылка на метод часто представляет собой более краткую альтернативу лямбда-выражению. Там, где ссылки на методы короче и понятнее, используйте их; там, где это не так, используйте лямбда-выражения.

7.3. Предпочитайте использовать стандартные функциональные интерфейсы

Сейчас, когда Java имеет лямбда-выражения, значительно изменились лучшие практики написания API. Например, проектный шаблон *Шаблонный метод* (Template Method) [12], в котором подкласс перекрывает *примитивный метод* для специализации поведения суперкласса, становится гораздо менее привлекательным. Современная альтернатива состоит в том, чтобы предоставить статическую фабрику или конструктор, который принимает функциональный объект, для достижения того же эффекта. В общем случае вы будете писать больше конструкторов и методов, которые принимают функциональные объекты в качестве параметров. Выбор правильного типа функционального параметра требует внимания.

Рассмотрим класс `LinkedHashMap`. Его можно использовать как кеш путем перекрытия его защищенного метода `removeEldestEntry`, который вызывается методом `put` всякий раз, когда в отображение добавляется новый ключ. Когда этот метод возвращает значение `true`, отображение удаляет свою старейшую запись, которая передается методу. Показанное далее перекрытие позволяет отображению вырасти до 100 записей, а затем удаляет старейшую запись каждый раз, когда добавляется новый ключ, поддерживая хранение сотни последних записей:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
{
    return size() > 100;
}
```

Эта методика отлично работает, но вы можете сделать все гораздо лучше с помощью лямбда-выражений. Если бы `LinkedHashMap` был написан сегодня, он бы имел статическую фабрику или конструктор, который принимал бы функциональный объект. Глядя на объявление `removeEldestEntry`, вы можете решить, что функциональный объект должен получать `Map.Entry<K, V>` и возвращать значение `boolean`, но это совсем не так: метод `removeEldestEntry` вызывает `size()`, чтобы получить количество элементов в отображении, и это работает, потому что метод `removeEldestEntry` является методом экземпляра отображения. Функциональный объект, который передается конструктору, не является методом экземпляра отображения и не может его захватить, поскольку, когда вызывается фабрика или конструктор, объект еще не существует. Таким образом, отображение должно передать функциональному объекту само себя, а также старейшую запись. Если объявить такой функциональный интерфейс, он будет выглядеть следующим образом:

```
// функциональный интерфейс, не являющийся необходимым.
// Используйте вместо него стандартный интерфейс.
@FunctionalInterface interface EldestEntryRemovalFunction<K,V>
{
    boolean remove(Map<K,V> map, Map.Entry<K,V> eldest);
}
```

Этот интерфейс будет работать нормально, но вы не должны его использовать, потому что вам не нужно объявлять новый интерфейс для этой цели. Пакет `java.util.function` предоставляет вам большую коллекцию стандартных функциональных интерфейсов. Если один из стандартных функциональных интерфейсов выполняет нужную вам работу, в общем случае следует использовать именно его, а не специализированный функциональный интерфейс. Это сделает ваш API более легким для изучения путем уменьшения его концептуального охвата и предоставит значительные выгоды для взаимодействия, поскольку многие стандартные функциональные интерфейсы предоставляют полезные методы по умолчанию. Например, интерфейс `Predicate` предоставляет методы для комбинации предикатов. В нашем примере `LinkedHashMap` следует предпочесть стандартный интерфейс `BiPredicate<Map<K, V>, Map.Entry<K, V>>` пользовательскому интерфейсу `EldestEntryRemovalFunction`.

В `java.util.Function` имеется сорок три интерфейса. Нельзя ожидать, что вы запомните их все, но если вы запомните шесть базовых интерфейсов, то остальные можно будет вывести, когда в них возникнет нужда. Основные интерфейсы работают с объектами ссылочных типов. Интерфейс `Operator` представляет функции, типы результата и аргумента которых совпадают. Интерфейс `Predicate` представляет функцию, которая принимает аргумент и

возвращает значение типа `boolean`. Интерфейс `Function` представляет функцию, типы аргумента и возвращаемого значения которой различаются. Интерфейс `Supplier` представляет функцию, которая не принимает аргументы, но возвращает (или “поставляет”) значение. Наконец, интерфейс `Consumer` представляет функцию, которая принимает аргумент, но не возвращает ничего, по сути, “потребляя свой” аргумент. Шесть основных функциональных интерфейсов приведены в следующей таблице.

Интерфейс	Сигнатура функции	Пример
<code>UnaryOperator<T></code>	<code>T apply(T t)</code>	<code>String::toLowerCase</code>
<code>BinaryOperator<T></code>	<code>T apply(T t1, T t2)</code>	<code>BigInteger::add</code>
<code>Predicate<T></code>	<code>boolean test(T t)</code>	<code>Collection::isEmpty</code>
<code>Function<T,R></code>	<code>R apply(T t)</code>	<code>Arrays::asList</code>
<code>Supplier<T></code>	<code>T get()</code>	<code>Instant::now</code>
<code>Consumer<T></code>	<code>void accept(T t)</code>	<code>System.out::println</code>

Есть также три варианта каждого из шести базовых интерфейсов для работы с примитивными типами `int`, `long` и `double`. Их имена являются производными от имен базовых интерфейсов, предваряемыми типом-примитивом. Так, например, предикат, который принимает `int`, — `IntPredicate`, а бинарный оператор, который принимает два значения типа `long` и возвращает значение `long`, имеет имя `LongBinaryOperator`. Ни один из этих вариантов типов не является параметризованным, за исключением вариантов `Function`, которые параметризуются типом возвращаемого значения. Например, `LongFunction<int[]>` получает значение типа `long` и возвращает `int[]`.

Есть девять дополнительных вариантов интерфейса `Function` для использования при примитивных типах результата. Источник и результат всегда различаются, потому что функция, возвращаемый тип которой соответствует типу аргумента, — `UnaryOperator`. Если и исходный тип, и тип результата являются примитивными, то имя `Function` предваряется `SrcToResult`, например `LongToIntFunction` (шесть вариантов). Если исходный тип является примитивным, а результат — ссылкой на объект, то `Function` предваряется `<Src>ToObj`, например `DoubleToObjFunction` (три варианта).

Есть двухаргументные версии трех базовых функциональных интерфейсов, для которых они имеют смысл: `BiPredicate<T, U>`, `BiFunction<T, U, R>` и `BiConsumer<T, U>`. Имеются также варианты `BiFunction`, возвращающие три соответствующих примитивных типа: `ToIntBiFunction<T, U>`, `ToLongBiFunction<T, U>` и `ToDoubleBiFunction<T, U>`. Есть двухаргументные варианты `Consumer`, которые принимают одну ссылку на объект и один примитивный тип: `ObjDoubleConsumer<T>`, `ObjIntConsumer<T>` и

`ObjLongConsumer<T>`. Всего имеется девять двухаргументных версий базовых интерфейсов.

Наконец, имеется интерфейс `BooleanSupplier`, вариант `Supplier`, который возвращает значение типа `boolean`. Это единственное явное упоминание типа `boolean` в любом из имен стандартных функциональных интерфейсов, но возвращаемые значения типа `boolean` поддерживаются посредством `Predicate` и его четырех разновидностей. Интерфейс `BooleanSupplier` и сорок два интерфейса, описанные в предыдущих абзацах, в сумме дают сорок три стандартных функциональных интерфейса. Для запоминания многовато, но, с другой стороны, с основной частью функциональных интерфейсов, которые вы будете использовать, вы будете встречаться регулярно, так что у вас не должно быть слишком много проблем при поиске нужного интерфейса.

Большинство стандартных функциональных интерфейсов существуют только для обеспечения поддержки примитивных типов. **Не поддавайтесь искушению использовать базовые функциональные интерфейсы с упакованными примитивами вместо примитивных функциональных интерфейсов.** Несмотря на работоспособность, этот вариант нарушает рекомендации раздела 9.5, в котором требуется предпочитать примитивные типы упакованным примитивным типам. Влияние использования упакованных примитивов для массовых операций на производительность может оказаться смертельно опасным.

Теперь вы знаете, что обычно следует использовать стандартные функциональные интерфейсы вместо собственных. Но когда же *следует* писать собственные интерфейсы? Конечно же, нужно писать свой интерфейс, если ни один из стандартных интерфейсов не делает то, что вам нужно, например если вам требуется предикат, который принимает три параметра, или предикат, генерирующий проверяемое исключение. Но иногда приходится писать собственный функциональный интерфейс даже при наличии стандартного структурно идентичного интерфейса.

Рассмотрим нашего старого друга — `Comparator<T>`, который структурно идентичен интерфейсу `ToIntBiFunction<T, T>`. Даже если этот последний интерфейс существовал, когда первый был добавлен в библиотеку, использовать его было бы неправильным. Существует несколько причин, по которым `Comparator` заслуживает собственного интерфейса. Во-первых, его имя обеспечивает документирование происходящего всякий раз, когда он используется в API, а используется он часто. Во-вторых, интерфейс `Comparator` имеет строгие требования к допустимым экземплярам, которые соответствуют его *общему контракту*. Путем реализации интерфейса вы обеспечиваете соответствие этому контракту. В-третьих, интерфейс оснащен полезными методами по умолчанию для преобразования и объединения компараторов.

Вы должны серьезно подумать о написании специального функционального интерфейса вместо использования стандартного, если вам нужен функциональный интерфейс, который имеет одну или несколько общих характеристик с `Comparator`.

- Он будет часто использоваться, и его описательное имя может принести пользу.
- С ним связан строгий контракт.
- Он выигрывает от пользовательских методов по умолчанию.

Если вы решили написать собственный функциональный интерфейс, помните, что это интерфейс, а значит, к его разработке следует относиться с большой осторожностью (раздел 4.7).

Обратите внимание, что интерфейс `EldestEntryRemovalFunction` помечен с помощью аннотации `@FunctionalInterface`. Этот тип аннотации сходен по духу с `@Override`. Это заявление о намерениях программиста, которое служит трем целям: говорит читателям класса и его документации, что этот интерфейс разработан для включения лямбда-выражений; заставляет вас быть честным, поскольку интерфейс не будет компилироваться, если только он не содержит в точности один абстрактный метод; не позволяет сопровождающим код программистам случайно добавить абстрактные методы в интерфейс по мере его развития. **Всегда аннотируйте ваши функциональные интерфейсы с помощью аннотации `@FunctionalInterface`.**

Осталось поставить последнюю точку в рассказе об использовании функциональных интерфейсов в API. Не предоставляйте метод с несколькими перегрузками, которые принимают различные функциональные интерфейсы в одной и той же позиции аргумента, если это может привести к возможной неоднозначности в команде клиента. Это не просто теоретическая проблема. Метод `submit` в `ExecutorService` может принимать `Callable<T>` или `Runnable`, и можно написать клиентскую программу, которая требует наличия приведения для указания правильной перегрузки (раздел 8.4). Самый простой способ избежать этой проблемы — не писать перегрузки, которые принимают различные функциональные интерфейсы в одной и той же позиции аргумента. Это частный случай совета из раздела 8.4 — разумно использовать перегрузки.

Помните: теперь, когда в языке Java имеются лямбда-выражения, важно, чтобы вы проектировали свои API с учетом наличия лямбда-выражений. Вы можете принимать типы функциональных интерфейсов в качестве входных данных и возвращать их в качестве выходных. Обычно лучше использовать стандартные интерфейсы, предоставляемые `java.util.function.Function`, но не забывайте об относительно редких случаях, когда лучше писать собственные функциональные интерфейсы.

7.4. Разумно используйте потоки

API потоков был добавлен в Java 8, чтобы упростить выполнение объемных операций — последовательно или параллельно. Этот API предоставляет две ключевые абстракции: *поток* (stream), который представляет собой конечную или бесконечную последовательность элементов данных, и *конвейер потока* (stream pipeline), который представляет собой многоэтапные вычисления над этими элементами. Элементы потока могут поступать из разных источников. Распространенные источники включают в себя коллекции, массивы, файлы, программное обеспечение обработки регулярных выражений, генераторы псевдослучайных чисел и другие потоки. Элементы данных в потоке могут быть ссылками на объекты или примитивными значениями (поддерживаются три примитивных типа: int, long и double).

Конвейер потока состоит из исходного потока, за которым следуют нуль или более *промежуточных операций* (intermediate operations) и одна *завершающая операция* (terminal operation). Каждая промежуточная операция некоторым образом преобразует поток, как, например, отображая каждый элемент на функцию от этого элемента или отфильтровывая элементы, не удовлетворяющие некоторым условиям. Все промежуточные операции преобразовывают один поток в другой, тип элементов которого может быть таким же, как и у входного потока, или отличаться от него. Завершающая операция выполняет окончательные вычисления в потоке, полученном в результате последней промежуточной операции, например сохраняет его элементы в коллекцию, возвращает определенный элемент или выводит все его элементы на экран.

Конвейеры потоков вычисляются *отложено* (lazily): вычисления не начинаются до тех пор, пока не будет вызвана завершающая операция, и никогда не вычисляются элементы данных, которые не нужны для выполнения завершающей операции. Такие отложенные вычисления делают возможной работу с бесконечными потоками. Обратите внимание, что конвейер потока без завершающей операции, по сути, представляет собой отсутствие вычислений, так что не забудьте ее добавить.

API потоков *свободен* (fluent): он предназначен для того, чтобы позволить всем вызовам, которые составляют конвейер, быть собранными в одно выражение. Фактически несколько конвейеров можно связывать в единственное выражение.

По умолчанию конвейеры потоков выполняются последовательно. Заставить конвейер выполняться параллельно так же просто, как вызвать метод parallel для любого потока в конвейере, но такое действие оказывается уместным довольно редко (раздел 7.7).

API потоков достаточно универсален, так что практически любое вычисление может быть выполнено с использованием потоков, но то, что вы можете это сделать, не означает, что вы должны так делать. При правильном использовании потоки могут сделать программы короче и понятнее; но при использовании их ненадлежащим образом они могут сделать программы трудно читаемыми и поддерживаемыми. Нет точных и быстрых правил для выяснения, когда следует использовать потоки, но имеются подходящие эвристики.

Рассмотрим следующую программу, которая считывает слова из файла словаря и выводит все группы анаграмм, размер которых соответствует определенному пользователем минимуму. Напомним, что два слова являются анаграммами, если они состоят из одних и тех же букв, расположенных в ином порядке. Программа считывает каждое слово из словаря, находящегося в определенном пользователем файле, и помещает слова в отображение. Ключом является слово с его буквами в алфавитном порядке, так что ключ для "staple" — "aelpst", как и для "petals", ключом является "aelpst", так что эти два слова являются анаграммами, и все анаграммы имеют один и тот же вид при размещении букв в алфавитном порядке. Значением отображения является список, содержащий все слова с общим представлением букв в алфавитном порядке. После обработки словаря каждый список содержит полную группу анаграмм. Затем программа перебирает представление values() отображения и выводит все списки, размер которых соответствует пороговому:

```
// Итеративный вывод всех больших групп анаграмм в словаре
public class Anagrams
{
    public static void main(String[] args) throws IOException
    {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);
        Map<String, Set<String>> groups = new HashMap<>();

        try (Scanner s = new Scanner(dictionary))
        {
            while (s.hasNext())
            {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                    (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
    }
}
```

```
private static String alphabetize(String s)
{
    char[] a = s.toCharArray();
    Arrays.sort(a);
    return new String(a);
}
}
```

Один этап в этой программе необходимо особо отметить. Включение каждого слова в отображение, выделенное полужирным шрифтом, использует метод `computeIfAbsent`, который был добавлен в Java 8. Этот метод ищет ключ в отображении: если ключ имеется в наличии, метод просто возвращает связанное с ним значение. Если ключа нет, то метод вычисляет значение путем применения данного функционального объекта к ключу, связывает это значение с ключом и возвращает вычисленное значение. Метод `computeIfAbsent` упрощает реализацию отображений, которые связывают с каждым ключом несколько значений.

Теперь рассмотрим следующую программу, которая решает ту же задачу, но с интенсивным использованием потоков. Обратите внимание, что вся программа, за исключением кода, который открывает файл словаря, содержится в одном выражении. Единственная причина, по которой словарь открыт в отдельном выражении, — возможность использования инструкции `try-c-ресурсами`, которая гарантирует, что файл словаря будет корректно закрыт:

```
// Злоупотребление потоками — не делайте так!
public class Anagrams
{
    public static void main(String[] args) throws IOException
    {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary))
        {
            words.collect(
                groupingBy(word -> word.chars().sorted()
                    .collect(StringBuilder::new,
                        (sb, c) -> sb.append((char) c),
                        StringBuilder::append).toString())
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .map(group -> group.size() + ": " + group)
                .forEach(System.out::println);
        }
    }
}
```

Если вы сочтете этот код трудным для чтения, не волнуйтесь: вы не одиноки. Этот код короче, но куда менее удобочитаем, в особенности для программистов, не являющихся экспертами в использовании потоков. **Злоупотребление потоками затрудняет чтение и поддержку программ.**

К счастью, есть золотая середина. Приведенная далее программа решает ту же задачу, потребляя потоки, но не злоупотребляя ими. Результатом является программа, которая короче и понятнее, чем оригинал:

```
// Корректное применение потоков повышает ясность и краткость
public class Anagrams
{
    public static void main(String[] args) throws IOException
    {
        Path dictionary = Paths.get(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        try (Stream<String> words = Files.lines(dictionary))
        {
            words.collect(groupingBy(word -> alphabetize(word)))
                .values().stream()
                .filter(group -> group.size() >= minGroupSize)
                .forEach(g -> System.out.println(g.size() + ": " + g));
        }
    }
    // Метод alphabetize тот же, что и в исходной версии
}
```

Даже если вы мало знакомы с потоками, эту программу нетрудно понять. Она открывает файл словаря в блоке `try`-с-ресурсами, получая поток, состоящий из всех строк файла. Переменная `stream` называется `words`, чтобы показать, что каждый элемент в потоке представляет собой слово. Конвейер этого потока не имеет никаких промежуточных операций; его завершающая операция собирает все слова в отображение, группирующее слова по представлению их букв в алфавитном порядке (раздел 7.5). Это именно то отображение, которое было построено в обеих предыдущих версиях программы. Затем `Stream<List<String>>` открывается для представления `values()` отображения. Элементы этого потока, конечно же, представляют собой группы анаграмм. Поток фильтруется таким образом, чтобы все группы, размер которых меньше `minGroupSize`, игнорировались; наконец, остальные группы выводятся завершающей операцией `forEach`.

Обратите внимание на тщательный выбор имен параметров лямбда-выражений. Параметр `g` в действительности должен был бы быть назван `group`, но получаемая в результате строка кода могла бы оказаться слишком широкой для книги. **В отсутствие явного указания типов тщательное именование**

параметров лямбда-выражения имеет важное значение для удобочитаемости конвейеров потоков.

Обратите также внимание, что “алфавитизация” слова делается отдельным методом `alphabetize`. Это повышает удобочитаемость, предоставляя имя для операции и убирая детали реализации из основной программы. **Применение вспомогательных методов для удобочитаемости в конвейерах потоков даже более важно, чем в итеративном коде**, потому что конвейеры не содержат явной информации о типах именованных временных переменных.

Метод `alphabetize` также можно перепроектировать с использованием потоков, но метод на основе потока был бы менее ясным, его было бы более трудно правильно написать, и, вероятно, он был бы медленнее. Эти недостатки являются результатом отсутствия поддержки потоков для примитивного типа `char` в Java (отсюда не следует, что Java должен поддерживать потоки `char`; это было бы невозможно сделать). Для демонстрации опасности обработки значений `char` с помощью потоков рассмотрим следующий код:

```
"Hello world!".chars().forEach(System.out::print);
```

Вы могли бы ожидать вывода `Hello world!`, но, если вы выполните его, то обнаружите, что он выводит `721011081081113211911111410810033`. Это происходит потому, что элементы потока, возвращенного кодом `"Hello world!".chars()`, являются не значениями типа `char`, а значениями типа `int`, поэтому вызывается перегрузка `print` для `int`. Это и в самом деле выглядит странно — что метод с именем `chars` возвращает поток значений `int`. Программу *можно* исправить с помощью приведения для принудительного вызова правильной перегрузки:

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

Но в идеале вы должны **воздерживаться от использования потоков для обработки значений типа `char`**.

Начав использовать потоки, вы можете почувствовать желание преобразовать все свои циклы в потоки, но не поддавайтесь этому искушению. Хотя это может быть возможным, но, скорее всего, это только принесет вред удобочитаемости и сопровождаемости кода. Как правило, даже умеренно сложные задачи лучше всего выполняются с помощью некоторой комбинации потоков и итераций, как проиллюстрировано выше программой `Anagrams`. Так что **выполняйте рефакторинг существующего кода для использования потоков и используйте их в новом коде только там, где это имеет смысл**.

Как показано в программах в этом разделе, конвейеры потоков выражают повторные вычисления с помощью функциональных объектов (обычно — лямбда-выражений или ссылок на методы), в то время как итеративный код выражает повторные вычисления с использованием блоков кода. Есть некоторые

вещи, которые можно сделать с помощью блоков кода, но которые не получится сделать из функциональных объектов.

- Из блока кода можно прочесть или изменить любую локальную переменную в области видимости; из лямбда-выражения можно прочесть только финальные или фактически финальные переменные [25, 4.12.4], и нельзя изменять никакие локальные переменные.
- В блоке кода можно выполнить оператор `return` для выхода из охватывающего метода, прервать или продолжить выполнение охватывающего цикла с помощью операторов `break` и `continue`, или сгенерировать любое проверяемое исключение, которое этот метод объявил как могущее быть сгенерированным; из лямбда-выражения нельзя сделать ни одно из перечисленных действий.

Если вычисление лучше всего выражается с использованием этих методов, то, вероятно, это не лучший код для применения потоков. И наоборот, потоки позволяют очень легко сделать некоторые другие вещи.

- Единообразное преобразование последовательностей элементов.
- Фильтрация последовательностей элементов.
- Объединение последовательностей элементов с помощью единственной операции (например, сложение, конкатенация или вычисление минимума).
- Накопление последовательности элементов в коллекции, возможно, с группировкой по некоторым общим атрибутам.
- Поиск в последовательности элементов некоторого элемента, удовлетворяющего некоторым критериям поиска.

Если вычисление лучше всего выражается с использованием этих методов, то, вероятно, это хороший кандидат для применения потоков.

Есть одна вещь, которую трудно сделать с использованием потоков, — одновременное обращение к соответствующим элементам на разных этапах конвейера: после отображения значения на некоторое другое значение исходное значение теряется. Одним из способов обхода этого ограничения является отображение каждого значения на *объект пары*, содержащий исходное и новое значения, но это решение не является удовлетворительным, в особенности если такие объекты пар требуются для нескольких этапов конвейера. Получающийся в результате код оказывается беспорядочным и многословным, что сводит на нет основное предназначение потоков. В ситуациях, когда это применимо, в качестве обходного пути, когда нужен доступ к значению на более ранней стадии вычислений, лучше инвертировать использованное отображение.

Например, напишем программу для вывода первых двадцати *простых чисел Мерсенна*. Напомним, что простые числа Мерсенна — числа вида $2^p - 1$. Если p — простое, то соответствующее число Мерсенна *может* быть простым; если это так, это — простое число Мерсенна. В качестве первоначального потока в нашем конвейере мы хотим видеть простые числа. Вот метод, возвращающий (бесконечный) поток. Мы предполагаем, что для легкого доступа к статическим членам `BigInteger` был использован статический импорт:

```
static Stream<BigInteger> primes()
{
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

Имя метода (`primes`, простые числа) представляет собой множественное число имени существительного, описывающего элементы потока. Такое именование настоятельно рекомендуется для всех методов, которые возвращают потоки, потому что оно повышает удобочитаемость конвейеров потоков. Этот метод использует статическую фабрику `Stream.iterate`, которая принимает два параметра: первый элемент в потоке и функцию для получения следующего элемента в потоке из предыдущего. Вот программа для вывода первых 20 простых чисел Мерсенна:

```
public static void main(String[] args)
{
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

Эта программа является простым кодированием текстового описания, приведенного выше: она начинается с простых чисел, вычисляет соответствующие числа Мерсенна, отфильтровывает все числа, кроме простых (магическое число 50 управляет вероятностным тестом простоты), ограничивает результирующий поток двадцатью элементами и выводит их.

Предположим теперь, что мы хотим, чтобы каждому простому числу Мерсенна при выводе предшествовала его экспонента (p). Это значение присутствует только в начальном потоке, а потому оно недоступно при работе завершающей операции, выводящей результаты. К счастью, легко вычислить экспоненту числа Мерсенна путем инверсии операции, примененной в первой промежуточной операции. Экспонента представляет собой просто число битов в бинарном представлении числа, поэтому следующая завершающая операция генерирует требуемый результат:

```
.forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

Есть много задач, для которых не очевидно, что следует использовать: потоки или итерации. Например, рассмотрим задачу инициализации новой колоды карт. Предположим, что `Card` представляет собой неизменяемый класс значения, который инкапсулирует достоинство карты `Rank` и ее масть `Suit` (оба они являются типами перечислений). Эта задача является представителем всех задач, в которых требуется вычисление всех пар элементов, которые могут быть выбраны из двух множеств. Математики называют их *декартовым произведением* двух множеств. Вот как выглядит итеративная реализация с помощью вложенных циклов `for`, которая должна выглядеть для вас очень знакомой:

```
// Итеративное вычисление декартова произведения
private static List<Card> newDeck()
{
    List<Card> result = new ArrayList<>();
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            result.add(new Card(suit, rank));
    return result;
}
```

А вот реализация на основе потока, которая использует промежуточную операцию `flatMap`. Эта операция отображает каждый элемент в потоке на поток, а затем объединяет все эти новые потоки в единственный поток (сливая их). Обратите внимание, что эта реализация содержит вложенное лямбда-выражение, выделенное полужирным шрифтом:

```
// Вычисление декартова произведения на основе потока
private static List<Card> newDeck()
{
    return Stream.of(Suit.values())
        .flatMap(suit ->
            Stream.of(Rank.values())
                .map(rank -> new Card(suit, rank)))
        .collect(toList());
}
```

Какая из двух версий `newDeck` лучше? Все сводится к личным предпочтениям и среде, в которой вы программируете. Первая версия более простая и, пожалуй, выглядит более естественной. Большая часть программистов Java в состоянии понимать и поддерживать этот код, но все же некоторые программисты будут чувствовать себя более комфортно со второй (поточковой) версией. Она немного короче и не слишком сложная для понимания для достаточно хорошо разбирающегося в потоках и функциональном программировании программиста. Если вы не знаете, какую версию предпочесть, то, пожалуй, более безопасным выбором будет итеративная версия. Если же вы предпочитаете

потоковую версию и считаете, что и другие программисты, которые будут работать с вашим кодом, разделяют ваши предпочтения, используйте ее.

Итак, одни задачи лучше выполняются с помощью потоков, а другие — с помощью итераций. Многие задачи лучше всего выполняются с помощью сочетания этих двух подходов. Нет никаких жестких правил выбора используемого для решения задачи подхода, но есть некоторые полезные эвристики. Во многих случаях будет очевидно, какой подход следует использовать; в ряде случаев это не так. **Если вы не знаете, что лучше применять — потоки или итерации, попробуйте оба варианта и посмотрите, какой из них работает лучше.**

7.5. Предпочитайте в потоках функции без побочных эффектов

Если вы новичок в потоках, вам может показаться трудной работа с ними. Даже простое выражение вычислений в виде конвейера потока может быть трудным делом. Даже при успехе, если ваша программа будет работать, может оказаться так, что вы с трудом понимаете, как это происходит. Потоки — это не просто API, это парадигма, основанная на функциональном программировании. Для того чтобы получить выразительность, скорость и в некоторых случаях — возможности параллельных вычислений, которые могут предложить потоки, необходимо принять, помимо API, саму парадигму.

Наиболее важной частью парадигмы потоков является структурирование вычислений как последовательности преобразований, в которой результат каждого этапа должен быть как можно ближе к *чистой функции* (pure function) от результата предыдущего этапа. Чистая функция — это функция, результат которой зависит только от входных данных, и при этом как не зависит от какого-либо изменяемого состояния, так и не обновляет никакое состояние. Для того чтобы этого добиться, любые функциональные объекты, которые передаются в потоковые операции (как промежуточные, так и завершающую), должны быть свободны от побочных эффектов.

Иногда можно увидеть потоковый код, который выглядит подобно приведенному далее фрагменту (строющему таблицу частоты слов в текстовом файле):

```
// Использование потокового API, но не потоковой
// парадигмы - не делайте так!
Map<String, Long> freq = new HashMap<>();

try (Stream<String> words = new Scanner(file).tokens())
{
```

```
words.forEach(word -> {
    freq.merge(word.toLowerCase(), 1L, Long::sum); });
}
```

Что не так с этим кодом? В конце концов, он использует потоки, лямбда-выражения и ссылки на методы и возвращает правильный ответ. Но, попросту говоря, это вообще не потоковый код. Это итеративный код, маскирующийся под потоковый. Он не получает никаких выгод от применения потокового API и (немного) длиннее, труднее для чтения и сопровождения, чем соответствующий итеративный код. Проблема вытекает из того факта, что этот код делает свою работу в завершающей операции `forEach`, используя лямбда-выражение, которое изменяет внешнее состояние (таблицу частот). Операция `forEach`, которая делает нечто большее, чем представляет результат вычисления, выполняемого потоком, является “душком в коде”, как и лямбда-выражение, изменяющее состояние. Так как же должен выглядеть этот код?

// Корректное использование потоков для инициализации таблицы частот

```
Map<String, Long> freq;

try (Stream<String> words = new Scanner(file).tokens())
{
    freq = words
        .collect(groupingBy(String::toLowerCase, counting()));
}
```

Этот фрагмент кода делает то же самое, что и предыдущий, но с правильным использованием потокового API. Он короче и понятнее. Так почему же он сразу не был написан таким способом? Потому что предыдущий код использует хорошо знакомые инструменты. Java-программисты хорошо знают, как использовать циклы, и завершающая операция `forEach` аналогична им. Однако операция `forEach` находится среди наименее мощных завершающих операций и наименее дружелюбна к потокам. Она носит явно итеративный характер, а следовательно, не поддается распараллеливанию. **Операция `forEach` должна использоваться только для того, чтобы вывести результат потоковых вычислений, но не для выполнения вычислений.** Иногда имеет смысл использовать `forEach` для некоторых других целей, таких как добавление результатов потоковых вычислений к уже имеющейся коллекции.

Усовершенствованный код использует *коллектор* (`collector`), который представляет собой новую концепцию, которую следует изучить, чтобы использовать потоки. API `Collectors` подавляет: он содержит 39 методов, некоторые из которых имеют целых пять параметров типа. Хорошая новость заключается в том, что получить большую пользу от этого API можно и не вникая во все

его сложности. Для начала можно игнорировать интерфейс `Collectors` и рассматривать коллектор как черный ящик, инкапсулирующий стратегию *приведения* (*reduction*). В данном контексте приведение означает, что элементы потока объединяются в один объект. Объект, генерируемый коллектором, обычно является коллекцией.

Коллекторы для сбора элементов потока в истинную коллекцию `Collection` просты. Существует три таких коллектора: `toList()`, `toSet()` и `toCollection(collectionFactory)`. Они возвращают список, множество и коллекцию указанного программистом типа соответственно. Вооружившись этим знанием, мы можем написать конвейер потока для извлечения первой десятки из нашей таблицы частот:

```
// Конвейер для получения первой десятки слов из таблицы частот
List<String> topTen = freq.keySet().stream()
    .sorted(comparing(freq::get).reversed())
    .limit(10)
    .collect(toList());
```

Обратите внимание, что мы не должны квалифицировать метод `toList` его классом `Collectors`. Это обычное и мудрое решение — статически импортировать все члены `Collectors`, потому что это делает конвейеры потоков более удобочитаемыми.

Единственная сложная часть этого кода — компаратор, который мы передаем в метод `sorted`, а именно — `comparing(freq::get).reversed()`. Метод `comparing` представляет собой метод создания компаратора (раздел 3.5), который принимает функцию извлечения ключа. Эта функция принимает слово, а “извлечение” на самом деле является поиском в таблице: ограниченная ссылка на ссылку `freq::get` ищет слово в таблице частот и возвращает количество появлений слова в файле. Наконец, мы вызываем `reversed` для компаратора, так что сортировка слов выполняется от наиболее частых к наименее частым. После этого оказывается очень просто ограничить поток десятью словами и собрать их в список.

В предыдущих фрагментах кода используется метод `stream` класса `Scanner` для получения потока через сканер. Этот метод был добавлен в Java 9. Если вы используете более ранние версии, можете транслировать сканер, который реализует `Iterator`, в поток с помощью адаптера, подобного показанному в разделе 7.6 (`streamOf(Iterable<E>)`).

Что можно сказать о других тридцати шести методах `Collectors`? Большинство из них существуют для того, чтобы позволять собирать потоки в отображения, что гораздо сложнее, чем просто собирать их в истинные коллекции. Каждый элемент потока связан с ключом *и значением*, и несколько элементов потока могут быть связаны с одним и тем же ключом.

Простейшим коллектором отображения является `toMap(keyMapper, valueMapper)`, который принимает две функции, одна из которых отображает элемент потока на ключ, а вторая — на значение. Мы использовали этот коллектор в нашей реализации `fromString` в разделе 6.1 для того, чтобы получить отображение из строковой формы перечисления в само перечисление:

```
// Использование коллектора toMap для создания
// отображения строки на перечисление
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(
        toMap(Object::toString, e -> e));
```

Эта простая форма `toMap` идеальна, если каждый элемент в потоке отображается на уникальный ключ. Если несколько элементов потока отображаются на один и тот же ключ, конвейер завершит работу с исключением `IllegalStateException`.

Более сложные формы `toMap`, так же как и метод `groupingBy`, дают различные способы предоставления стратегий для обработки таких коллизий. Одним из этих способов является предоставление метода `toMap` с *функцией слияния*. Функция слияния представляет собой `BinaryOperator<V>`, где `V` — тип значения отображения. Любые дополнительные значения, связанные с ключом, объединяются с существующим значением с помощью функции слияния; так, например, если функция слияния представляет собой умножение, то в конечном итоге вы получите значение, которое является произведением всех значений, связанных с ключом.

Трехаргументная разновидность `toMap` позволяет также сделать отображение ключа на выбранный элемент, связанный с этим ключом. Например, предположим, что у нас есть поток альбомов с записями различных исполнителей, и мы хотим выполнить отображение исполнителя на альбом-бестселлер. Эту работу можно выполнить с помощью следующего коллектора.

```
// Коллектор для генерации отображения ключа на выбранный элемент
Map<Artist, Album> topHits = albums.collect(
    toMap(Album::artist, a->a, maxBy(comparing(Album::sales))));
```

Обратите внимание, что компаратор использует статический фабричный метод `maxBy`, который статически импортируется из `BinaryOperator`. Этот метод преобразует `Comparator<T>` в `BinaryOperator<T>`, который вычисляет максимальное значение с помощью указанного компаратора. В этом случае компаратор возвращается методом создания компаратора `comparing`, который принимает функцию извлечения ключа `Album::sales`. Это может показаться несколько запутанным, но код выглядит красиво. Грубо говоря, он гласит: “преобразовать поток альбомов в отображение, отображая каждого

исполнителя на альбом, который является лучшим по продажам”. Это удивительно близко к постановке задачи.

Другое использование трехаргументной разновидности `toMap` — создание коллектора, который при наличии коллизий обеспечивает стратегию “последний записанный побеждает”. Для многих потоков результаты будут недетерминированными, но если все значения, которые могут быть связаны с ключом функцией отображения, идентичны или являются приемлемыми, то поведение коллектора может быть именно тем, что вам требуется:

```
// Коллектор для обеспечения стратегии
// "последний записанный побеждает"
toMap(keyMapper, valueMapper, (v1, v2) -> v2)
```

Третья, и последняя, версия `toMap` принимает четвертый аргумент, который является фабрикой отображений, для использования в случае, когда требуется указать реализацию конкретного отображения, такого как `EnumMap` или `TreeMap`.

Имеются также варианты первых трех разновидностей `toMap` с именем `toConcurrentMap`, которые эффективно работают параллельно и производят экземпляры `ConcurrentHashMap`.

В дополнение к методу `toMap` API `Collectors` предоставляет метод `groupingBy`, который возвращает коллекторы для создания отображений, которые группируют элементы в категории, основанные на *функции-классификаторе* (*classifier function*). Функция-классификатор принимает элемент и возвращает категорию, к которой он относится. Эта категория служит в качестве ключа элемента отображения. Простейший вариант метода `groupingBy` принимает только классификатор и возвращает отображение, значениями которого являются списки всех элементов в каждой категории. Это коллектор, который мы использовали в программе `Anagram` в разделе 7.4 для создания отображения “алфавитизированных” на списки слов:

```
words.collect(groupingBy(word -> alphabetize(word)))
```

Если вы хотите, чтобы `groupingBy` возвращал коллектор, который создает отображение со значениями, отличными от списков, то в дополнение к классификатору можете указать *нисходящий коллектор* (*downstream collector*). Такой коллектор производит значение из потока, содержащего все элементы в категории. Простейшее использование этого параметра заключается в передаче методу `toSet()`, что приводит к отображению, значения которого представляют собой множества, а не списки элементов.

В качестве альтернативы можно передать `toCollection(collectionFactory)`, что позволит вам создавать коллекции, в которые помещается каждая

категория элементов. Это дает вам возможность выбрать любой нужный тип коллекции. Еще одно простое применение двухаргументной разновидности `groupingBy` состоит в передаче `counting()` в качестве нисходящего коллектора. Это приводит к отображению, которое связывает каждую категорию с количеством элементов в категории, а не к коллекции, содержащей элементы. Это то, что вы видели в примере с таблицей частот в начале этого раздела:

```
Map<String, Long> freq = words
    .collect(groupingBy(String::toLowerCase, counting()));
```

Третья версия `groupingBy` позволяет указать фабрику отображений в дополнение к нисходящему коллектору. Обратите внимание, что этот метод нарушает стандартную телескопическую схему списка аргументов: параметр `mapFactory` предшествует (а не следует за) параметру `downStream`. Эта версия `groupingBy` дает вам контроль над содержащим отображением, а также над содержащимися коллекциями; так, например, можно указать коллектор, который возвращает `TreeMap`, значениями которого являются `TreeSets`.

Метод `groupingByConcurrent` предоставляет варианты всех трех перегрузок `groupingBy`. Эти варианты эффективно работают параллельно и производят экземпляры `ConcurrentHashMap`. Существует также редко используемый напарник `groupingBy`, именуемый `partitioningBy`. Вместо метода классификатора он принимает предикат и возвращает отображение, ключ которого имеет тип `Boolean`. Существуют две перегрузки этого метода, одна из которых принимает, помимо предиката, нисходящий коллектор.

Коллекторы, возвращаемые методом `counting`, предназначены *только* для использования в качестве нисходящих коллекторов. Такая же функциональность доступна непосредственно в `Stream` посредством метода `count`, так что **не существует причин для написания `collect(counting())`**. Есть еще пятнадцать методов `Collectors` с этим свойством. Они включают девять методов, имена которых начинаются с `summing`, `averaging` и `summarizing` (функциональность которых доступна для соответствующих примитивных типов потоков). Они также включают все перегрузки метода `reducing`, а также методы `filtering`, `mapping`, `flatMaping` и `collectingAndThen`. Большинство программистов могут спокойно игнорировать большинство этих методов. С точки зрения дизайнера эти коллекторы представляют собой попытку частично дублировать функциональность потоков в коллекторах, так что нисходящие коллекторы могут выступать в качестве “мини-потоков”.

Существует три метода `Collectors`, о которых мы еще не упоминали. Хотя они находятся в `Collectors`, они не связаны с коллекциями. Первые два из них — `minBy` и `maxBy`, которые принимают компаратор и возвращают минимальный или максимальный элемент потока, определенный с помощью компаратора. Они представляют собой незначительные обобщения методов `min` и

max в интерфейсе Stream и являются коллекторами, аналогичными бинарным операторам, возвращаемым соответствующими методами в BinaryOperator. Напомним, что мы использовали BinaryOperator.maxBy в нашем примере с альбомами-бестселлерами.

Последний метод Collectors — joining, который действует только на потоки экземпляров CharSequence, такие как строки. В разновидности без параметров он возвращает коллектор, который просто соединяет элементы путем конкатенации. Его одноаргументная разновидность принимает единственный параметр CharSequence с именем delimiter и возвращает коллектор, который соединяет элементы потока, вставляя между соседними элементами разделитель. Если вы передадите как разделитель запятую, коллектор возвратит строку значений, разделенных запятыми (но будьте осторожны: строка может быть неоднозначной, если любой из элементов в потоке содержит запяты). Трехаргументная разновидность принимает в дополнение к разделителю префикс и суффикс. Получаемый коллектор генерирует строки наподобие тех, которые получаются, когда вы выводите коллекции, например [came, saw, conquered].

Суть программирования конвейеров потоков — функциональные объекты без побочных действий. Это относится ко всем из множества функциональных объектов, передаваемых потокам и связанным объектам. Завершающая операция forEach должна использоваться только для вывода результата вычислений, выполняемых в потоке, но не выполнять сами вычисления. Для того чтобы правильно использовать потоки, вы должны изучить коллекторы. Наиболее важными фабриками коллекторов являются toList, toSet, toMap, groupingBy и joining.

7.6. Предпочитайте коллекции потокам в качестве возвращаемых типов

Многие методы возвращают последовательности элементов. До Java 8 очевидными возвращаемыми типами таких методов были интерфейсы Collection, Set и List; Iterable; а также типы массивов. Обычно было достаточно легко решить, какой из этих типов следует вернуть. Нормой являлся интерфейс коллекции. Если метод, существующий исключительно для того, чтобы обеспечить возможность работы цикла for для каждого элемента коллекции, или возвращаемая последовательность не могли быть сделана реализующей некоторый метод Collection (как правило, contains(Object)), то использовался интерфейс Iterable. Если возвращаемые элементы представляли собой примитивные значения или имелись строгие требования к

производительности, использовались массивы. В Java 8 в платформу были добавлены потоки, что существенно осложнило задачу выбора соответствующего возвращаемого типа для метода, возвращающего последовательность.

Вы можете услышать мнение, что теперь очевидным выбором для возврата последовательности элементов являются потоки, но, как говорится в разделе 7.4, потоки не делают итерации устаревшими: написание хорошего кода требует разумного объединения потоков и итераций. Если API возвращает только поток, а некоторые пользователи хотят итерировать возвращенную последовательность с помощью цикла, эти пользователи будут не без оснований расстроены. Это будет особенно разочаровывающим, потому что интерфейс `Stream` содержит единственный абстрактный метод в интерфейсе `Iterable`, а спецификация `Stream` для данного метода совместима с `Iterable`. Единственное, что предотвращает использование цикла для итерирования потока, — это то, что `Stream` не расширяет `Iterable`.

К сожалению, хорошего решения этой проблемы нет. На первый взгляд может показаться, что будет работать передача ссылки на метод `iterator` из `Stream`. Результирующий код, возможно, немного запутанный и непрозрачный, но не безрассудный:

```
// Не компилируется из-за ограничений вывода типов Java
for (ProcessHandle ph : ProcessHandle.allProcesses():iterator)
{
    // Работа
}
```

К сожалению, при попытке скомпилировать этот код вы получите сообщение об ошибке:

```
Test.java: 6: error: method reference not expected here
for (ProcessHandle ph : ProcessHandle.allProcesses():iterator) {
    ^
```

Чтобы скомпилировать код, следует привести ссылку на метод к соответствующим образом параметризованному `Iterable`:

```
// Обратительный обходной путь для итерирования потока
for (ProcessHandle ph : (Iterable<ProcessHandle>)
    ProcessHandle.allProcesses():iterator)
```

Этот клиентский код работает, но он слишком зашумленный и непрозрачный для использования на практике. Лучшее решение заключается в использовании метода адаптера. JDK не предоставляет такого метода, но его легко написать, используя тот же прием, что и в фрагментах выше. Обратите внимание, что приведение в методе адаптера не требуется, поскольку вывод типа Java в этом контексте работает должным образом:

```
// Адаптер Stream<E> к Iterable<E>
public static <E> Iterable<E> iterableOf(Stream<E> stream)
{
    return stream::iterator;
}
```

С помощью этого адаптера можно выполнить итерирование любого потока с помощью цикла `for`:

```
for (ProcessHandle p : iterableOf(ProcessHandle.allProcesses()))
{
    // Работа
}
```

Обратите внимание, что потоковая версия программы `Anagrams` в разделе 6.1 использует метод `Files.lines` для чтения словаря, в то время как итеративная версия использует сканер. Метод `Files.lines` превосходит сканер, который молча проглатывает любые исключения, которые могут возникнуть во время чтения файла. В идеале было бы хорошо использовать `Files.lines` и в итерационной версии. Это своего рода компромисс, на который идут программисты, если API обеспечивает только потоковый доступ к последовательности, а они хотят итерировать последовательность с помощью цикла `for`.

И наоборот, программист, который хочет обрабатывать последовательность с помощью конвейера потока, будет обоснованно огорчен API, который предоставляет только `Iterable`. И вновь JDK не предоставляет подходящего адаптера, но его достаточно легко написать:

```
// Адаптер Iterable<E> к Stream<E>
public static <E> Stream<E> streamOf(Iterable<E> iterable)
{
    return StreamSupport.stream(iterable.spliterator(), false);
}
```

Если вы создаете метод, который возвращает последовательность объектов, и знаете, что он будет использоваться только в конвейере потока, тогда, конечно, можете беспрепятственно вернуть поток. Аналогично, методу, возвращающему последовательность, которая будет использоваться только для итерирования, следует возвращать `Iterable`. Но если вы пишете открытый API, который возвращает последовательность, то должны предоставить возможности как для пользователей, которые хотят работать с конвейерами потоков, так и для тех, кто хочет итерировать (если только у вас нет веских оснований полагать, что большинство ваших пользователей захотят использовать один и тот же механизм).

Интерфейс `Collection` является подтипом `Iterable` и имеет метод `stream`, поэтому он обеспечивает и итеративный, и потоковый доступ. Таким

образом, `Collection` или подходящий подтип в общем случае является наилучшим типом возвращаемого значения для открытого метода, возвращающего последовательность. Массивы также легко обеспечивают итеративный, и потоковый доступ с помощью методов `Arrays.asList` и `Stream.of`. Если последовательность, которую вы возвращаете, достаточно небольшая для беспрепятственного размещения в памяти, вероятно, лучше возвращать одну из стандартных реализаций коллекций, например `ArrayList` или `HashSet`. Но не храните в памяти большую последовательность только для того, чтобы вернуть ее как коллекцию.

Если возвращаемая последовательность велика, но может быть представлена в сжатом виде, рассмотрите возможность реализации коллекции специального назначения. Например, предположим, что вы хотите вернуть *показательное множество* данного множества, которое состоит из всех его подмножеств. Показательное множество для множества $\{a, b, c\}$ имеет вид $\{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Если множество состоит из n элементов, его показательное множество содержит 2^n элементов. Таким образом, вы не должны даже думать о хранении показательного множества в стандартной реализации коллекции. Однако легко реализовать необходимую пользовательскую коллекцию с помощью `AbstractList`.

Хитрость заключается в том, чтобы использовать индекс каждого элемента в показательном множестве как битовый вектор, где n -й бит индекса указывает на наличие или отсутствие n -го элемента исходного множества. По сути, есть естественное отображение двоичных чисел от 0 до $2^n - 1$ на элементы показательного множества для n -элементного множества. Вот код:

```
// Возвращает показательное множество входного
// множества как пользовательскую коллекцию
public class PowerSet
{
    public static final <E> Collection<Set<E>> of(Set<E> s)
    {
        List<E> src = new ArrayList<>(s);
        if (src.size() > 30)
            throw new IllegalArgumentException(
                "Множество слишком велико " + s);
        return new AbstractList<Set<E>>()
        {
            @Override public int size()
            {
                return 1 << src.size(); // 2 в степени srcSize
            }
            @Override public boolean contains(Object o)
            {
                return o instanceof Set && src.containsAll((Set)o);
            }
        }
    }
}
```


(*a*, *b*, *c*), (*b*, *c*) и (*c*). Понимание, что подсписками списка являются просто суффиксы его префиксов (или, что то же самое, префиксы его суффиксов) и пустой список, ведет к ясной, достаточно краткой реализации:

```
// Возвращает поток всех подсписков входного списка
public class SubLists
{
    public static <E> Stream<List<E>> of(List<E> list)
    {
        return Stream.concat(
            Stream.of(Collections.emptyList()),
            prefixes(list).flatMap(SubLists::suffixes));
    }
    private static <E> Stream<List<E>> prefixes(List<E> list)
    {
        return IntStream.rangeClosed(1, list.size())
            .mapToObj(end -> list.subList(0, end));
    }
    private static <E> Stream<List<E>> suffixes(List<E> list)
    {
        return IntStream.range(0, list.size())
            .mapToObj(start -> list.subList(start, list.size()));
    }
}
```

Обратите внимание, что метод `Stream.concat` использован для добавления пустого списка в возвращаемый поток. Также обратите внимание, что метод `flatMap` (раздел 7.4) использован для создания единого потока, состоящего из всех суффиксов всех префиксов. Наконец, заметьте, что мы генерируем префиксы и суффиксы путем отображения потока последовательных значений `int`, возвращаемых `IntStream.range` и `IntStream.rangeClosed`. Эта идиома, грубо говоря, представляет собой поток, эквивалентный стандартному циклу по целочисленным индексам. Таким образом, наша реализация подсписка по духу сходна с очевидным вложенным циклом `for`:

```
for (int start = 0; start < src.size(); start++)
    for (int end = start + 1; end <= src.size(); end++)
        System.out.println(src.subList(start, end));
```

Этот цикл можно транслировать непосредственно в поток. Результат оказывается более кратким, чем наша предыдущая реализация, но, возможно, немного менее удобочитаемым. По духу он аналогичен потоковому коду для декартова произведения в разделе 7.4:

```
// Возвращает поток всех подписков входного списка
public static <E> Stream<List<E>> of(List<E> list)
{
    return IntStream.range(0, list.size())
        .mapToObj(start ->
            IntStream.rangeClosed(start + 1, list.size())
                .mapToObj(end -> list.subList(start, end)))
        .flatMap(x -> x);
}
```

Подобно предшествующему ему циклу `for`, этот код *не* создает пустой список. Для того чтобы исправить этот недостаток, можно либо использовать `concat`, как мы делали в предыдущей версии, либо заменить `1` на `(int) Math.signum(start)` в вызове `rangeClosed`.

Любая из этих реализаций потока подписков работоспособна, но обе требуют от некоторых пользователей использовать адаптер `Stream-в-Iterable` или использовать поток в местах, где итерации будут более естественными. Но адаптер `Stream-в-Iterable` не только вносит беспорядок в код клиента, но и замедляет работу цикла (в 2,3 раза на моей машине). Целевая реализация `Collection` (не показанная здесь) оказывается существенно более многословной, но работает на моей машине примерно в 1,4 раза быстрее, чем наша реализация на основе потока.

Резюме: при написании метода, который возвращает последовательность элементов, нужно помнить, что некоторым из ваших пользователей может потребоваться обработать их как поток, в то время как другие могут захотеть выполнить над ними итерирование. Постарайтесь удовлетворить и тех, и других. Если возможно возвращать коллекцию, сделайте это. Если у вас уже есть элементы в коллекции или количество элементов в последовательности достаточно небольшое, чтобы оправдать создание новой коллекции, верните стандартную коллекцию, такую как `ArrayList`. В противном случае рассмотрите вопрос о реализации пользовательской коллекции, как мы делали это для показательного множества. Если возврат коллекции невозможен, возвращайте `Stream` или `Iterable` — то, что вам кажется более естественным. Если в будущей версии `Java` объявление интерфейса `Stream` изменится и он будет расширять `Iterable`, то вы сможете беспрепятственно возвращать потоки, поскольку они будут позволять как потоковую, так и итерационную обработку.

7.7. Будьте внимательны при параллелизации потоков

Среди основных языков программирования `Java` всегда был на переднем крае обеспечения простоты решения задач параллельного программирования.

Когда язык программирования Java был выпущен в 1996 году, он имел встроенную поддержку потоков выполнения, с синхронизацией и возможностями `wait/notify`. В Java 5 была введена библиотека `java.util.concurrent`, с параллельными коллекциями и каркасом исполнителя. В Java 7 представлен пакет ветвления, высокопроизводительный каркас для параллельных вычислений. В Java 8 введены потоки, которые могут быть распараллелены с помощью единственного вызова метода `parallel`. Написание параллельных программ на языке Java становится все проще, но написание правильных и быстрых параллельных программ остается таким же сложным, как и ранее. Нарушения безопасности и надежности остаются доминирующим фактором в параллельном программировании, и параллельные конвейеры потоков не являются исключением.

Рассмотрим программу из раздела 7.4:

```
// Поточковая программа для генерации первых 20 простых чисел Мерсенна
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
static Stream<BigInteger> primes()
{
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

На моей машине эта программа сразу же начинает выводить числа и завершается за 12,5 с. Предположим, что я наивно попытаюсь ускорить программу путем добавления вызова `parallel()` в конвейере потока. Как вы думаете, что произойдет с производительностью программы? Станет ли она выполняться на несколько процентов быстрее? На несколько процентов медленнее? К сожалению, в результате она не выводит вообще ничего, но использование процессора поднимается до 90% и остается таковым на неопределенный срок, демонстрируя ошибку живучести (*liveness failure*). Программа, возможно, в конечном итоге и завершается, но через полчаса ожидания я просто прекратил ее мучения.

Что же происходит? Попросту говоря, библиотека потоков не имеет представления, как распараллелить этот конвейер, а эвристика не работает. Даже при самых благоприятных обстоятельствах **параллелизация конвейера вряд ли увеличит его производительность, если источник получен от `Stream.iterate` или используется промежуточная операция `limit`**. Наш конвейер сталкивается с *обеими* из указанных проблем. Что еще хуже, стратегия параллелизации по умолчанию ведет себя непредсказуемо при использовании

limit, полагая, что нет ничего страшного в вычислении дополнительных элементов и отбрасывании любых ненужных результатов. В нашем случае требуется примерно в два раза больше времени для поиска каждого очередного простого числа Мерсенна по сравнению с поиском предыдущего. Таким образом, стоимость вычислений одного дополнительного элемента примерно равна суммарной стоимости вычислений всех предыдущих элементов, и этот выглядящий безобидным конвейер оказывается поставленным алгоритмом автоматического распараллеливания на колени. Мораль этой истории проста: **не распараллеливайте огульно конвейеры потоков**. Последствия для производительности программы могут оказаться катастрофическими.

Как правило, **выигрыш в производительности от применения параллелизма оказывается наибольшим в случае потоков над экземплярами `ArrayList`, `HashMap`, `HashSet` и `ConcurrentHashMap`; массивами; диапазонами `int` и `long`**. Общим у всех этих структур данных является то, что все они могут точно и дешево быть разделены на поддиапазоны любых желаемых размеров, что позволяет легко разделить работу между параллельными потоками. Абстракцией, используемой библиотекой потоков для выполнения этой задачи, является *spliterator* (итератор-разделитель), который возвращается методом `spliterator` у `Stream` и `Iterable`.

Еще одним важным фактором, которым обладают все эти структуры данных, является то, что они обеспечивают очень хорошую *локальность ссылок* при последовательной обработке: ссылки последовательных элементов хранятся в памяти рядом одна с другой. Объекты, на которые ссылаются данные ссылки, могут не быть расположены в памяти по соседству, что уменьшает локальность ссылок. Локальность ссылки имеет критически важное значение для распараллеливания массовых операций: без нее потоки тратят большую часть своего времени на простой в ожидании данных, передаваемых из памяти в кеш. Структурами данных с наилучшей локальностью ссылок являются примитивные массивы, потому в этом случае по соседству в памяти хранятся сами данные.

Природа завершающей операции конвейера потока также влияет на эффективность параллельного выполнения. Если значительное (по сравнению с общей работой конвейера) количество работы выполняется в завершающей операции, и это по своей природе последовательные операции, то распараллеливание конвейера будет иметь ограниченную эффективность. Наилучшими завершающими операциями с точки зрения параллелизма являются *приведения*, когда все элементы, подающиеся из конвейера, объединяются с помощью одного из методов `reduce` класса `Stream` или таких предопределенных приведений, как `min`, `max`, `count` и `sum`. Такие сокращенные операции¹

¹ Которые не вычисляют все операнды, если это не требуется. — *Примеч. ред.*

(shortcircuiting), как `anyMatch`, `allMatch` и `noneMatch`, также поддаются распараллеливанию. Операции, выполняемые в методе `collect` класса `Stream`, которые известны как *изменяющие приведения* (mutable reductions), не являются хорошими кандидатами для распараллеливания, потому что накладные расходы объединения коллекции оказываются высокими.

Если вы пишете собственную реализацию `Stream`, `Iterable` или `Collection` и хотите добиться высокой производительности параллелизма, перекройте метод `splitterator` и интенсивно протестируйте параллельную производительность получаемых потоков. Написание такого высококачественного метода является сложным делом и выходит за рамки этой книги.

Распараллеливание потоков может привести не только к низкой производительности, включая ошибки живучести, но и к неверным результатам и непредсказуемому поведению (ошибкам безопасности). Ошибки безопасности могут быть результатом распараллеливания конвейера, который использует функции отображения, фильтры и другие предоставляемые программистом функциональные объекты, которые могут не соответствовать спецификации. Спецификация `Stream` накладывает жесткие требования на эти функциональные объекты. Например, функциональные объекты накопления и объединения, передаваемые операции `reduce` потока `Stream`, должны быть ассоциативными, без состояний и взаимодействий. Если вы нарушите эти требования (некоторые из них обсуждаются в разделе 7.5), то при последовательной работе конвейера, скорее всего, получите правильные результаты. Но при параллельной работе, скорее всего, правильные результаты получить не удастся, более того, возможны катастрофические последствия для работы программы.

Помимо этого, стоит отметить, что даже если распараллеленная программа вывода простых чисел Мерсенна и завершится, то она не выведет простые числа в правильном (по возрастанию) порядке. Чтобы сохранить порядок вывода из последовательной версии, вам придется заменить завершающую операцию `forEach` операцией `forEachOrdered`, которая гарантированно обходит параллельные потоки в том *порядке, в котором они встречаются*.

Даже если предположить, что вы используете эффективно разделяемый исходный поток, распараллеливаемую или дешевую завершающую операцию и не взаимодействующие функциональные объекты, вы не получите существенного ускорения от распараллеливания, если только конвейер не выполняет достаточное количество реальной работы для компенсации накладных расходов, связанных с параллельной работой. В качестве *очень* грубой оценки, количество элементов в потоке, умноженное на количество строк кода, выполняемых для каждого элемента, должно составлять по крайней мере сто тысяч [30].

Важно помнить, что распараллеливание потока представляет собой не более чем оптимизацию производительности. Как и в случае любой оптимизации, необходимо протестировать производительность до и после внесения изменений для уверенности в том, что эту оптимизацию вообще стоит применять (раздел 9.11). В идеале, вы должны выполнить тест в условиях реальной работы. Обычно все параллельные конвейеры потока в программе работают в едином пуле ветвления. Один неправильно ведущий себя конвейер может отрицательно повлиять на производительность других, несвязанных с ним частей системы.

Если это звучит, как то, что шансы при распараллеливании конвейеров против вас, то это потому, что все так и есть. Мой знакомый, который поддерживает код из многих миллионов строк с интенсивным использованием потоков, нашел лишь несколько мест, где параллельные потоки оказались эффективными. Это *не* означает, что вы должны воздерживаться от распараллеливания потоков. **При соответствующих обстоятельствах можно достичь ускорения, почти линейно зависящего от количества ядер процессора, просто добавив вызов `parallel` в конвейер потока.** Особенно такое ускорение оказывается к месту в некоторых областях, таких как машинное обучение и обработка данных.

В качестве простого примера конвейера потока, в котором параллелизм оказывается эффективным, рассмотрим функцию вычисления $\pi(n)$ — количества простых чисел, не превышающих n :

```
// Конвейер потока для вычисления количества простых
// чисел — эффективное использование распараллеливания
static long pi(long n)
{
    return LongStream.rangeClosed(2, n)
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

На моей машине вычисление $\pi(10^8)$ с помощью этой функции заняло 31 с. Простое добавление вызова `parallel()` снизило это время до 9,2 с:

```
// Конвейер потока для вычисления количества простых
// чисел — параллельная версия
static long pi(long n)
{
    return LongStream.rangeClosed(2, n)
        .parallel()
        .mapToObj(BigInteger::valueOf)
        .filter(i -> i.isProbablePrime(50))
        .count();
}
```

Другими словами, параллелизация вычислений ускоряет их в 3,7 раза на моей машине с четырехъядерным процессором. Следует заметить, что на практике для больших значений n вычисление $\pi(n)$ выполняется иными методами. Есть гораздо более эффективные алгоритмы, в частности знаменитая формула Лемера.

Если вы собираетесь распараллелить поток случайных чисел, начните с экземпляра `SplittableRandom`, а не `ThreadLocalRandom` (или, по сути, устаревшего `Random`). `SplittableRandom` предназначен именно для такого использования и обладает потенциалом для линейного ускорения. `ThreadLocalRandom` предназначен для использования единственным потоком и будет адаптирован для функционирования в качестве источника параллельного потока, но не такого быстрого, как `SplittableRandom`. `Random` синхронизируется при каждой операции, что ведет к чрезмерным затратам, лишаящим распараллеливание смысла.

Итак, даже не пытайтесь распараллеливать конвейер потока, если только у вас нет веских оснований полагать, что это сохранит правильность вычислений и увеличит их скорость. Результатом ненадлежащим образом выполненного распараллеливания потока может быть отказ программы или катастрофическое падение ее производительности. Если вы считаете, что параллелизм может быть оправдан, убедитесь, что ваш код остается правильным при параллельном выполнении, и выполните тщательные измерения производительности в реальных условиях. Если ваш код остается правильным, а эксперименты подтверждают повышение производительности, тогда и только тогда распараллеливайте поток в рабочем коде.

В этой главе рассматриваются некоторые аспекты проектирования методов: как работать с параметрами и возвращаемыми значениями, как разрабатывать сигнатуры и документировать методы. Значительная часть материала в этой главе относится как к конструкторам, так и к методам. Подобно главе 4, “Классы и интерфейсы”, эта глава фокусируется на удобстве использования, надежности и гибкости.

8.1. Проверяйте корректность параметров

Большинство методов и конструкторов имеют определенные ограничения на значения, которые могут быть переданы в качестве параметров. Например, нередко указывается, что значения индекса должны быть неотрицательными, а ссылки на объекты — отличными от `null`. Следует точно и ясно документировать все эти ограничения и начинать работу метода с проверки их выполнения. Это частный случай более общего принципа, согласно которому следует выявлять ошибки как можно ранее после того, как они произошли. В противном случае обнаружение ошибки станет менее вероятным, а определение ее источника — более трудоемким.

Если методу передано неверное значение параметра, но в начале работы он проверяет свои параметры, то метод быстро и аккуратно завершится генерацией соответствующего исключения. Если же метод не проверяет свои параметры, могут произойти разные вещи. Метод может завершиться, сгенерировав непонятное исключение посреди работы. Еще хуже, если метод завершится нормально, но вычислит при этом неверный результат. Но самое худшее — когда метод завершается нормально, но оставляет некоторый объект в испорченном состоянии, что впоследствии в непредсказуемый момент времени вызовет ошибку в совершенно иной части программы, никак не связанной с этим

методом. Другими словами, отказ от проверки параметров может привести к нарушению принципа *атомарности сбоев* (failure atomicity) (раздел 10.8).

Для документирования в открытых и защищенных методах исключений, которые могут быть сгенерированы при нарушении ограничений на значения параметров, используйте дескриптор `@throws Javadoc` (раздел 10.6). Как правило, генерируемое в такой ситуации исключение — `IllegalArgumentException`, `IndexOutOfBoundsException` или `NullPointerException` (раздел 10.4). После того как вы документировали ограничения на параметры метода и исключения, которые будут генерироваться в случае нарушения этих ограничений, обеспечение соблюдения этих ограничений — очень простой вопрос. Вот типичный пример:

```
/**
 * Возвращает объект BigInteger, значение которого — остаток от
 * деления данного числа на m. Этот метод отличается от метода
 * remainder тем, что всегда возвращает неотрицательное значение
 * типа BigInteger.
 *
 * @param m — модуль, должен быть положительным
 * @return переданное значение mod m
 * @throws ArithmeticException если m <= 0
 */
public BigInteger mod(BigInteger m)
{
    if (m.signum() <= 0)
        throw new ArithmeticException("Модуль <= 0: " + m);

    ... // Выполнение вычислений
}
```

Обратите внимание, что документирующий комментарий *не* говорит, что “mod генерирует `NullPointerException`, если `m` представляет собой `null`”, хотя метод поступает именно так в результате побочного действия вызова `m.signum()`. Это исключение *документировано* в комментарии уровня охватывающего класса `BigInteger`. Комментарии уровня класса относятся ко всем параметрам всех открытых методов класса. Это хороший способ избежать беспорядка при документировании `NullPointerException` в каждом методе по отдельности. Он может сочетаться с использованием `@Nullable` или подобных аннотаций для указания, что определенный параметр может иметь значение `null`, но такая практика не является стандартной, и для этой цели используется несколько аннотаций.

Метод `Objects.requireNonNull`, добавленный в Java 7, гибкий и удобный, так что больше нет никакой причины для проверки значения `null` вручную. При желании можно указать собственное исключение,

подробно сообщаемое о происшедшем. Этот метод возвращает входное значение, так что вы можете выполнить проверку на `null` в момент использования значения:

```
// Использование проверки на null при использовании
this.strategy = Objects.requireNonNull(strategy, "strategy");
```

Можно также игнорировать возвращаемое значение и использовать `Objects.requireNonNull` в качестве автономной проверки там, где это отвечает вашим потребностям.

В Java 9 к `java.util.Objects` добавлена проверка диапазона. Эта возможность включает три метода: `checkFromIndexSize`, `checkFromToIndex` и `checkIndex`. Эта возможность не столь гибкая, как метод проверки на `null`. Она не позволяет задать собственное сообщение исключения и предназначена исключительно для использования с индексами списков и массивов. Она также не обрабатывает закрытые диапазоны (которые содержат обе конечные точки). Но если это то, что вам нужно, — это очень полезные вспомогательные методы.

Для неэкспортируемого метода вы как автор пакета контролируете обстоятельства, при которых вызывается метод, так что вы можете и должны гарантировать, что методу будут передаваться только допустимые значения параметров. Таким образом, закрытые методы могут проверять свои параметры с помощью *утверждений* (assertions), как показано далее:

```
// Закрытая вспомогательная функция для рекурсивной сортировки
private static void sort(long a[], int offset, int length)
{
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Выполнение вычислений
}
```

По сути, эти утверждения являются заявлениями, что указанное условие *будет* верно независимо от того, как пакет будет использоваться клиентами. В отличие от обычных проверок корректности, утверждения генерируют исключение `AssertionError`, если условия не выполняются. И в отличие от обычных проверок корректности, они не выполняют никаких действий и не имеют никакой стоимости при их отключении с помощью флага командной строки `java -ea` (или `-enableassertions`). Дополнительную информацию об утверждениях можно найти в учебнике [1].

Особенно важно проверять правильность параметров, которые методом не используются, а откладываются для последующей обработки. Например, рассмотрим статический фабричный метод из раздела 4.6, который получает

массив чисел типа `int` и возвращает представление этого массива в виде экземпляра `List`. Если клиент передаст методу значение `null`, будет сгенерировано исключение `NullPointerException`, поскольку метод содержит явную проверку (вызов `Objects.requireNonNull`). Если бы проверка отсутствовала, то метод возвращал бы ссылку на вновь созданный экземпляр `List`, которая приводила бы к генерации исключения `NullPointerException`, когда клиент попытается ею воспользоваться. К сожалению, к этому моменту определить происхождение проблемного экземпляра `List` будет уже трудно, что может значительно усложнить задачу отладки.

Конструкторы представляют собой частный случай принципа, согласно которому вы должны проверить правильность параметров, сохраненных для последующего использования. Правильность параметров конструктора необходимо проверить для того, чтобы избежать построения объекта, который нарушает инварианты класса.

Из правила, обязывающего перед выполнением вычислений проверять параметры метода, имеются исключения. Важным исключением является ситуация, когда явная проверка правильности оказывается дорогостоящей или непрактичной операцией, и при этом параметры неявно проверяются непосредственно в процессе выполнения вычислений. Например, рассмотрим метод, сортирующий список объектов, такой как `Collections.sort(List)`. Все объекты в представленном списке должны быть взаимно сравнимыми. В процессе сортировки списка каждый объект в нем будет сравниваться с каким-либо другим объектом из того же списка. Если объекты не будут взаимно сравнимыми, в результате одного из таких сравнений будет сгенерировано исключение `ClassCastException`, а это именно то, что должен делать в такой ситуации метод `sort`. Таким образом, нет смысла выполнять упреждающую проверку взаимной сравнимости элементов списка. Заметим, однако, что неразборчивое использование такого подхода может привести к потере *атомарности сбоев* (раздел 10.2).

Иногда в ходе вычислений неявно выполняется проверка корректности, но если параметры не проходят проверку, генерируется неверное исключение. Иными словами, исключение, которое генерируется вычислениями, может не соответствовать исключению, описанному в документации как генерируемое методом. В этих обстоятельствах следует использовать идиому *трансляции исключения* (*exception translation*), описанную в разделе 10.5, для преобразования естественного исключения в корректное.

Из данного раздела не следует делать вывод, что произвольные ограничения параметров являются хорошим решением. Наоборот, вы должны создавать методы как можно более общими, насколько это возможно и практично. Чем меньше ограничений вы накладываете на параметры, тем лучше, при условии,

что метод в состоянии выполнить осмысленные действия для каждого принимаемого значения параметра. Однако зачастую некоторые ограничения обусловлены реализуемой абстракцией.

Таким образом, каждый раз, когда вы пишете метод или конструктор, вы должны подумать над тем, какие ограничения имеются для его параметров. Необходимо отразить эти ограничения в документации, а также реализовать в начале работы метода явную проверку их выполнения. Важно, чтобы это стало вашей привычкой. Это те небольшие затраты труда, которые сторицей окупятся при первом же обнаружении неправильного параметра.

8.2. При необходимости создавайте защитные копии

Одна из особенностей языка программирования Java, которая делает его столь приятным в использовании, — он является *безопасным языком*. Это означает, что в отсутствие машинных методов (native method) программа защищена от переполнения буфера, выхода за границы массива, неконтролируемых указателей и других ошибок повреждения памяти, которые следует учитывать при работе с такими небезопасными языками, как C и C++. В безопасном языке можно писать классы и быть уверенным, что их инварианты будут сохранены независимо от того, что происходит в любой другой части системы. Это невозможно в языках, которые рассматривают всю память как один гигантский массив.

Но даже в безопасном языке, если не приложить со своей стороны некоторые усилия, вы не изолированы от других классов. **Вы должны писать программы оборонительно — исходя из предположения, что клиенты вашего класса будут предпринимать все возможное для того, чтобы разрушить его инварианты.** Это совершенно верно в ситуации, когда кто-то пытается взломать вашу систему безопасности, но и без того вашему классу, скорее всего, придется иметь дело с непредвиденным поведением других классов, обусловленным случайными ошибками программиста, пользующегося вашим API. В любом случае имеет смысл потратить время и писать классы, которые будут надежно работать даже при неправильном поведении клиентов.

Хотя один класс не может изменить внутреннее состояние объекта другого класса без определенной помощи с его стороны, оказать такую помощь, не желая того, на удивление просто. Например, рассмотрим следующий класс, задачей которого является представление неизменного периода времени:

```
// Некорректный "неизменяемый" класс для промежутка времени
public final class Period
{
```

```

private final Date start;
private final Date end;
/**
 * @param start — начало промежутка времени
 * @param end — конец промежутка времени;
 *             не должен предшествовать start
 * @throws IllegalArgumentException если start после end
 * @throws NullPointerException если start или end равен null
 */
public Period(Date start, Date end)
{
    if (start.compareTo(end) > 0)
        throw new IllegalArgumentException(
            start + " после " + end);

    this.start = start;
    this.end = end;
}
public Date start()
{
    return start;
}
public Date end()
{
    return end;
}
... // Остальная часть кода опущена
}

```

На первый взгляд может показаться, что этот класс неизменяемый и обеспечивающий инвариант, гласящий, что начало промежутка времени не может быть позже его конца. Однако нарушить этот инвариант очень легко — используя тот факт, что класс `Date` является изменяемым:

```

// Изменение внутренних данных экземпляра Period
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78); // Изменение внутреннего состояния p!

```

По состоянию на Java 8 очевидный способ решения этой проблемы — использование `Instant` (или `LocalDateTime` или `ZonedDateTime`) вместо `Date`, потому что `Instant` (и другие классы `java.time`) являются неизменяемыми (раздел 4.3). **`Date` является устаревшим классом и не должен использоваться в новом коде.** Но проблема имеется по-прежнему: есть моменты, когда вам придется использовать изменяемые типы значений в API и внутренних представлениях, и методы, обсуждаемые в этом разделе, подходят и для них.

Для защиты внутреннего содержимого экземпляра `Period` от такого рода атак важно сделать *защитную копию (defensive copy)* каждого изменяемого параметра конструктора и использовать копии в качестве компонентов экземпляра `Period` вместо оригиналов:

```
// Исправленный конструктор - с защитным копированием параметров
public Period(Date start, Date end)
{
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(
            this.start + " после " + this.end);
}
```

С новым конструктором описанная выше атака не даст никакого результата с новым экземпляром `Period`. Обратите внимание, что копии делаются до проверки параметров (раздел 8.1), так что проверка корректности выполняется над копией, а не над оригиналом. Хотя это может показаться неестественным, это необходимо, так как защищает класс от изменения параметров другим потоком во время *окна уязвимости (window of vulnerability)* между моментами проверки параметров и их копирования. В сообществах компьютерной безопасности эта атака известна как *атака между проверкой и использованием (time-of-check/time-of-use, или TOCTOU)* [51].

Заметим также, что для создания копий мы не пользовались методом `clone` класса `Date`. Поскольку `Date` не является окончательным классом, нет гарантии, что метод `clone` возвратит объект, класс которого именно `java.util.Date`: он может вернуть экземпляр ненадежного подкласса, созданного специально для нанесения ущерба. Такой подкласс может, например, записывать ссылку на каждый экземпляр в момент создания последнего в закрытый статический список, а затем предоставить злоумышленнику доступ к этому списку. В результате злоумышленник получит полный контроль над всеми экземплярами копий. Чтобы предотвратить атаки такого рода, не используйте метод `clone` для создания копии параметра, тип которого позволяет ненадежным сторонам создавать подклассы.

Хотя замененный конструктор успешно защищает от описанной ранее атаки, все еще возможно изменить экземпляр `Period`, потому что его методы доступа обеспечивают доступ к его изменяемому внутреннему содержимому:

```
// Вторая атака внутреннего содержимого экземпляра Period
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Изменение внутреннего состояния p!
```

Для защиты от второй атаки просто измените методы доступа так, чтобы они **возвращали копии изменяемых внутренних полей**:

```
// Исправленные методы доступа – делайте копии внутренних полей
public Date start()
{
    return new Date(start.getTime());
}
public Date end()
{
    return new Date(end.getTime());
}
```

С новым конструктором и новыми методами доступа класс `Period` стал действительно неизменяемым. Теперь некомпетентному или злоумышленному программисту не удастся нарушить инвариант, гласящий, что начало промежутка времени предшествует его концу. Это так, поскольку, за исключением самого класса `Period`, никакой другой класс не имеет возможности получить доступ ни к какому изменяемому полю экземпляра `Period`. Указанные поля действительно инкапсулированы в этом объекте.

В методах доступа, в отличие от конструктора, для создания копий используется метод `clone`. Такое решение приемлемо, поскольку мы точно знаем, что внутренние объекты `Date` в классе `Period` действительно имеют тип `java.util.Date`, а не какой-то потенциально ненадежный его подкласс. Несмотря на это в общем случае для копирования экземпляра лучше использовать конструктор или статическую фабрику по причинам, изложенным в разделе 3.4.

Копирование параметров выполняется не только для неизменяемых классов. Всякий раз, когда вы пишете метод или конструктор, который сохраняет во внутренней структуре ссылку на объект, созданный клиентом, задумайтесь, не является ли этот объект потенциально изменяемым. Если это так, проанализируйте, будет ли ваш класс устойчив к изменениям в объекте после того, как последний будет введен в структуру данных. Если ответ отрицательный, то следует создать копию этого объекта и поместить ее в структуру данных вместо оригинала. Например, если ссылку на объект, предоставленный клиентом, предполагается использовать как элемент во внутреннем экземпляре `Set` или как ключ во внутреннем экземпляре `Map`, следует учитывать, что инварианты этого множества или отображения могут быть нарушены, если после добавления в них объект вдруг будет изменен.

То же самое верно и для копирования внутренних компонентов до их возвращения клиентам. Независимо от того, является ли ваш класс неизменяемым — вы должны дважды подумать, прежде чем вернуть ссылку на внутренний изменяемый компонент. Вероятно, все же следует вернуть его копию. Помните, что массивы ненулевой длины всегда являются изменяемыми. Таким

образом, вы всегда должны делать копию внутреннего массива перед его возвратом клиенту. Кроме того, можно вернуть неизменяемое представление массива. Оба эти метода рассматриваются в разделе 4.1.

Главным выводом, вытекающим из сказанного, является то, что вы должны везде, где это возможно, использовать в качестве компонентов ваших объектов неизменяемые объекты, чтобы вам не нужно было беспокоиться об их копировании (раздел 4.3). В нашем примере `Period` при использовании версии Java 8 и более поздних следует применять `Instant` (или `LocalDateTime`, или `ZonedDateTime`). Если вы используете более ранние версии, то одним из вариантов является хранение примитива `long`, возвращаемого `Date.getTime()`, вместо ссылки на `Date`.

Возможно определенное снижение производительности, связанное с копированием, которое не всегда оправдано. Если класс доверяет своему вызывающему объекту и знает, что тот не изменит никакие его внутренние компоненты (например, потому, что класс и его клиент являются частью единого пакета), то может оказаться целесообразным отказаться от защитного копирования. В такой ситуации документация класса должна явно указать, что вызывающий объект не должен изменять затрагиваемые параметры или возвращаемые значения.

Даже за рамками одного пакета не всегда целесообразно делать копию изменяемого параметра до его интеграции в объект. Есть некоторые методы и конструкторы, вызов которых означает явное *перемещение* объекта, на который указывает параметр. При вызове такого метода клиент обещает, что он больше не будет изменять объект непосредственно. Метод или конструктор, ожидающий получение владения предоставляемым клиентом изменяемым объектом, должен ясно указывать это в своей документации.

Классы, содержащие методы или конструкторы, вызов которых означает передачу управления объектом, не способны защититься от злоумышленников. Такие классы можно использовать только тогда, когда есть взаимное доверие между классом и его клиентами или когда нарушение инвариантов класса не способно нанести ущерба никому, кроме самого клиента. Примером последней ситуации является схема класса-оболочки (раздел 4.4). В зависимости от природы класса-оболочки клиент может нарушить инварианты этого класса с помощью непосредственного обращения к объекту уже после того, как тот оказался в оболочке, однако обычно это не наносит вреда никому, кроме самого клиента.

Итак, если у класса есть изменяемые компоненты, которые он получает от клиента или возвращает ему, то необходимо защитное копирование таких компонентов класса. Если затраты на копирование слишком высоки и класс доверяет своим клиентам, зная, что они не внесут неприемлемые изменения

в компоненты, то защитное копирование можно заменить документированием, отражающим ответственность клиентов за неизменность задействованных компонентов.

8.3. Тщательно проектируйте сигнатуры методов

Этот раздел представляет собой набор советов по проектированию API, которые не заслуживают собственного раздела. Собранные вместе, они помогут сделать ваш API проще в освоении и использовании и менее подверженным ошибкам.

Тщательно выбирайте имена методов. Имена всегда должны подчиняться выбранному соглашению об именовании (раздел 9.12). Основная цель должна заключаться в выборе имен, которые понятны и согласуются с другими именами в том же пакете. Вторичная цель состоит в выборе имен, согласующихся с более широким контекстом их применения. Избегайте длинных имен методов. Если у вас возникли сомнения, обратитесь к API библиотеки Java в качестве руководства. Хотя в них имеются и несогласованности (они неизбежны, учитывая размер и масштабы этих библиотек), здесь по большей части есть консенсус.

Не заходите слишком далеко в погоне за удобством методов. Каждый метод должен заниматься только собственной частью работы. Избыток методов делает класс слишком сложным для изучения, использования, документирования, тестирования и сопровождения. В отношении интерфейсов это верно вдвойне: большое количество методов усложняет жизнь и разработчикам, и пользователям. Для каждого действия, поддерживаемого вашим классом или интерфейсом, создавайте полнофункциональный метод. Сокращенный вариант рассматривайте лишь в том случае, если он будет часто использоваться. **Если есть сомнения, забудьте об этом.**

Избегайте длинных списков параметров. В общем случае на практике четыре параметра нужно рассматривать как абсолютный максимум, и чем меньше параметров, тем лучше. Большинство программистов не способны запоминать более длинные списки параметров. Если метод превышает указанный предел, вашим API будет невозможно пользоваться, не обращаясь постоянно к его документации. **Особенно вредны длинные последовательности параметров одного и того же типа** — и не только потому, что пользователь не сможет запомнить порядок их следования. Беда в том, что если он по ошибке поменяет их местами, программа все равно будет компилироваться и работать — только делать она будет совсем не то, что хотел ее автор.

Есть три способа сокращения слишком длинных списков параметров. Первый заключается в разбиении метода на несколько методов меньшего размера, каждому из которых нужно лишь какое-то подмножество его параметров. Если выполнить разбиение неаккуратно, может получиться слишком много методов, однако этот же прием помогает *сократить* количество методов путем увеличения их ортогональности. Например, рассмотрим интерфейс `java.util.List`. Он не предоставляет методы для поиска индекса первого или последнего элемента в подсписке, каждому из которых потребовались бы три параметра. Вместо этого он предоставляет метод `subList`, который принимает два параметра и возвращает *представление* подсписка. Для получения желаемого результата метод `subList` можно скомбинировать с методами `indexOf` или `lastIndexOf`, каждый из которых принимает по одному параметру. Более того, метод `subList` можно сочетать с *любым* другим методом, работающим с экземпляром `List`, для выполнения самых разных операций над подсписками. Полученный API имеет очень высокое отношение мощности к размеру.

Второй прием сокращения слишком длинных списков параметров заключается в создании *вспомогательных классов*, хранящих группы параметров. Обычно эти вспомогательные классы являются статическими классами-членами (раздел 4.10). Этот прием рекомендуется использовать, когда становится понятно, что часто возникающая последовательность параметров на самом деле представляет некую отдельную сущность. Предположим, например, что вы пишете класс, реализующий карточную игру, и выясняется, что постоянно приходится передавать последовательность из двух параметров: достоинства карты и ее масти. И ваш API, и содержимое вашего класса, вероятно, только выиграют, если для представления карты вы создадите вспомогательный класс и каждую такую последовательность параметров замените одним параметром этого вспомогательного класса.

Третий прием, сочетающий в себе аспекты первых двух, использует проектный шаблон *Строитель* (Builder) (раздел 2.2) от построения объекта до вызова метода. Если у вас есть метод с многими параметрами, в особенности если некоторые из них необязательны, имеет смысл определить объект, который будет представлять все параметры и позволять клиенту выполнять многократные вызовы методов установки значений для данного объекта. Каждый такой метод будет устанавливать один параметр или маленькую, связанную группу параметров. Как только нужный параметр будет установлен, клиент вызывает “исполняющий” метод, который выполняет окончательные проверки параметров и фактическое вычисление.

Предпочитайте в качестве типов параметров интерфейсы, а не классы (раздел 9.8). Если имеется соответствующий интерфейс для определения параметра, используйте его, а не класс, реализующий этот интерфейс. Например,

нет никаких причин для написания метода, который принимает в качестве входного параметра `HashMap`, — используйте вместо него `Map`. Это позволяет передавать в метод `HashMap`, `TreeMap`, `ConcurrentHashMap`, подотображение `TreeMap` или любую реализацию `Map`, которая еще только будет написана. Используя класс вместо интерфейса, вы ограничиваете клиента конкретной реализацией и заставляете выполнять ненужное и потенциально затратное копирование, если входные данные находятся в некоторой иной форме.

Предпочитайте двухэлементные типы перечислений для параметров `boolean`, если значение `boolean` не очевидно из названия метода. Перечисления делают код более легким для чтения и написания. Кроме того, они позволяют легко добавить дополнительные параметры позже. Например, может иметься тип `Thermometer` со статической фабрикой, которая принимает следующее перечисление:

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Запись `Thermometer.newInstance(TemperatureScale.CELSIUS)` не только более понятна, чем `Thermometer.newInstance(true)`, но впоследствии вы можете добавить в `TemperatureScale`, например, `KELVIN`, и при этом вам не придется добавлять в `Thermometer` новую статическую фабрику. Можно также выполнить рефакторинг и вынести зависимости температурных шкал в методы констант перечислений (раздел 6.1). Например, каждая константа температурной шкалы может иметь метод, который принимает значение `double` и превращает его в градусы Цельсия.

8.4. Перегружайте методы разумно

Приведенная далее программа представляет собой благонамеренную попытку классифицировать коллекции в соответствии с тем, чем они являются: множествами, списками или некоторыми иными видами коллекций:

// Неверно! – Что выведет данная программа?

```
public class CollectionClassifier
{
    public static String classify(Set<?> s)
    {
        return "Set";
    }
    public static String classify(List<?> lst)
    {
        return "List";
    }
}
```

```

public static String classify(Collection<?> c)
{
    return "Unknown Collection";
}
public static void main(String[] args)
{
    Collection<?>[] collections =
    {
        new HashSet<String>(),
        new ArrayList<BigInteger>(),
        new HashMap<String, String>().values()
    };

    for (Collection<?> c : collections)
        System.out.println(classify(c));
}
}

```

Вы можете ожидать, что программа выведет Set, затем List и Unknown Collection, но это не так. Она три раза выводит Unknown Collection. Почему так получается? Потому что метод `classify` *перегружен* (overloaded), а выбор того, какая из перегрузок будет вызвана, выполняется во время компиляции. На всех трех итерациях цикла тип времени компиляции параметра один и тот же: `Collection<?>`. Тип времени выполнения отличается на каждой итерации, но это не влияет на выбор перегрузки. Поскольку тип времени компиляции параметра представляет собой `Collection<?>`, единственной применимой перегрузкой является третья, `classify(Collection<?>)`, и именно эта перегрузка выполняется на каждой итерации цикла.

Поведение этой программы такое контринтуитивное потому, что **выбор среди перегруженных методов является статическим, в то время как выбор перекрытого метода — динамический**. Корректная версия *перекрытого* метода выбирается во время выполнения на основе типа времени выполнения объекта, метод которого вызывается. Напомним, что метод является *перекрытым* (overridden), когда подкласс содержит объявление метода с той же самой сигнатурой, что и объявление метода у предка. Если метод экземпляра перекрыт в подклассе и этот метод вызывается для экземпляра подкласса, то выполняется *перекрытый метод* подкласса, независимо от типа времени компиляции экземпляра подкласса. Рассмотрим следующую программу:

```

class Wine
{
    String name()
    {
        return "wine";
    }
}

```

```

class SparklingWine extends Wine
{
    @Override String name()
    {
        return "sparkling wine";
    }
}
class Champagne extends SparklingWine
{
    @Override String name()
    {
        return "champagne";
    }
}
public class Overriding
{
    public static void main(String[] args)
    {
        List<Wine> wineList = List.of(
            new Wine(), new SparklingWine(), new Champagne());

        for (Wine wine : wineList)
            System.out.println(wine.name());
    }
}

```

Метод `name` объявлен в классе `Wine` и перекрыт в подклассах `SparklingWine` и `Champagne`. Как и ожидается, эта программа выводит `wine`, `sparkling wine` и `champagne`, несмотря на то, что на каждой итерации цикла тип времени компиляции этого экземпляра — `Wine`. Тип времени компиляции объекта не влияет на то, какой из перекрытых методов будет выполняться; всегда выполняется “наиболее конкретный” перекрытый метод. Сравните это с перегрузкой, при которой тип времени выполнения объекта не влияет на то, какая перегрузка будет выполнена; выбор производится во время компиляции и основывается исключительно на типах параметров времени компиляции.

В примере с `CollectionClassifier` программа должна была распознавать тип параметра, автоматически переключаясь на соответствующий перегруженный метод на основании типа параметра времени выполнения так же, как метод `name` в примере с `Wine`. Перегрузка метода просто не обеспечивает соответствующую функциональность. В предположении, что требуется статический метод, можно исправить программу `CollectionClassifier`, заменив все три варианта перегрузки метода `classify` единственным методом, который выполняет явную проверку `instanceof`:

```
public static String classify(Collection<?> c) {
    return c instanceof Set ? "Set" :
           c instanceof List ? "List" : "Unknown Collection";
}
```

Поскольку перекрытие является нормой, а перегрузка — исключением, именно перекрытие определяет ожидания программистов при вызове метода. Как показал пример `CollectionClassifier`, перегрузка может не оправдать эти ожидания. Не следует писать код, поведение которого может быть неочевидным для программиста среднего уровня. Особенно это касается API. Если рядовой пользователь API не знает, какой из перегруженных методов будет вызван для данного набора параметров, то работа с таким API, вероятно, будет сопровождаться ошибками. Причем ошибки эти проявятся, скорее всего, только в поведении программы во время выполнения, и для многих программистов их диагностика будет сложной задачей. Поэтому **необходимо избегать запутывающих применений перегрузки.**

Вопрос о том, что же именно является запутывающим при использовании перегрузки, открыт для обсуждения. **Безопасная, консервативная стратегия состоит в том, чтобы никогда не экспортировать две перегрузки с одинаковым числом параметров.** Если метод использует переменное количество параметров, консервативная стратегия заключается в том, чтобы не перегружать его вовсе, за исключением, описанным в разделе 8.5. Если вы будете придерживаться этих ограничений, у других программистов никогда не будет сомнений в том, какая перегрузка применяется при том или ином наборе фактических параметров. Эти ограничения не являются ужасно обременительными, потому что **вы всегда можете дать методам разные имена вместо того, чтобы их перегружать.**

Рассмотрим, например, класс `ObjectOutputStream`. У него есть варианты метода `write` для каждого примитивного типа и для ряда ссылочных типов. Вместо перегрузки метода `write` все эти варианты имеют различные имена, такие как `writeBoolean(boolean)`, `writeInt(int)` и `writeLong(long)`. Дополнительным преимуществом этой схемы именования по сравнению с перегрузкой является то, что можно обеспечить методы чтения с соответствующими именами, например `readBoolean()`, `readInt()` и `readLong()`. Класс `ObjectInputStream` действительно предоставляет такие методы чтения.

В случае конструкторов у вас нет возможности использовать различные имена: при наличии нескольких конструкторов в классе они *всегда* перегружаются. Правда, в отдельных ситуациях вместо конструктора можно предоставить экспортируемую статическую фабрику (раздел 2.1), но это не всегда возможно. При применении конструкторов вам также не нужно беспокоиться о взаимодействии перегрузки и перекрытия, так как конструкторы не могут быть

перекрыты. Поскольку вам, вероятно, придется экспортировать несколько конструкторов с одним и тем же количеством параметров, полезно узнать, в каких случаях это безопасно.

Экспорт нескольких перегрузок с одним и тем же количеством параметров вряд ли запутает программистов, *если* всегда очевидно, какой вариант перегрузки будет применяться к данному набору фактических параметров. Это случай, когда у каждой пары вариантов перегрузки есть хотя бы один формальный параметр с “совершенно иным” типом. Два типа являются совершенно разными, если очевидна невозможность приведения ненулевого выражения к обоим типам. В этих условиях выбор варианта перегрузки для данного набора фактических параметров полностью определен типами параметров времени выполнения, и на него не могут повлиять их типы времени компиляции, так что исчезает главный источник путаницы. Например, класс `ArrayList` имеет конструктор, принимающий параметр типа `int`, и конструктор, принимающий параметр типа `Collection`. Трудно представить себе ситуацию, когда могла бы возникнуть путаница с выбором одного из этих двух конструкторов.

До Java 5 все примитивные типы являлись “совершенно иными” по отношению ко всем ссылочным типам, но при наличии автоматической упаковки это становится неверным, что приводит к реальным проблемам. Рассмотрим следующую программу:

```
public class SetList
{
    public static void main(String[] args)
    {
        Set<Integer> set = new TreeSet<>();
        List<Integer> list = new ArrayList<>();

        for (int i = -3; i < 3; i++)
        {
            set.add(i);
            list.add(i);
        }

        for (int i = 0; i < 3; i++)
        {
            set.remove(i);
            list.remove(i);
        }

        System.out.println(set + " " + list);
    }
}
```

Сначала эта программа добавляет целые числа в диапазоне от -3 до 2 включительно в отсортированное множество и в список. Затем она делает три одинаковых вызова `remove` для удаления из множества и списка. Если вы принадлежите к большинству, то, вероятно, ожидаете, что программа удалит неотрицательные значения (0 , 1 и 2) из списка и множества, и выведет

```
[-3, -2, -1] [-3, -2, -1]
```

На самом деле программа удалит неотрицательные значения из множества и нечетные значения из списка и выведет

```
[-3, -2, -1] [-2, 0, 2]
```

Это поясняет, почему такое поведение называют запутывающим.

Вот что здесь происходит. Вызов `set.remove(i)` выбирает перегрузку `remove(E)`, где `E` представляет собой тип элемента множества (`Integer`), и выполняет автоматическую упаковку `i` из `int` в `Integer`. Это ожидаемое поведение, так что программа завершается удалением положительных значений из множества. С другой стороны, вызов `list.remove(i)` выбирает перегрузку `remove(int i)`, которая выполняет удаление элемента в указанной *позиции* списка. Если начать со списка `[-3, -2, -1, 0, 1, 2]` и удалить нулевой элемент, затем первый, а потом второй, то мы останемся со списком `[-2, 0, 2]`. Вот тайна и раскрыта. Для исправления проблемы нужно привести аргумент метода `list.remove` к `Integer`, заставляя выбрать правильную перегрузку. В качестве альтернативы можно вызвать `Integer.valueOf` для `i` и передать результат в `list.remove`. В любом варианте программа выведет

```
[-3, -2, -1] [-3, -2, -1]
```

как и ожидалось:

```
for (int i = 0; i < 3; i++)
{
    set.remove(i);
    list.remove((Integer) i); // или remove(Integer.valueOf(i))
}
```

Запутывающее поведение, продемонстрированное в предыдущем примере, связано с тем, что у интерфейса `List<E>` есть две перегрузки метода `remove`: `remove(E)` и `remove(int)`. До Java 5, когда интерфейс `List` был “обобщен”, у него был метод `remove(Object)`, а не `remove(E)`, а соответствующие типы параметров — `Object` и `int` — кардинально различались. Но с появлением обобщенных типов и автоматической упаковки эти два типа параметров больше не столь различны. Другими словами, появление обобщенных типов и преобразований в языке нанесло ущерб интерфейсу `List`. К счастью,

пострадало лишь незначительное количество API в библиотеках Java, но эта история ясно показывает, что обобщенные типы и автоматическая упаковка повышают важность внимательности и осторожности при применении перегрузки.

Добавление лямбда-выражений и ссылок на методы в Java 8 увеличивает возможности путаницы при перегрузке. Например, рассмотрим следующие два фрагмента:

```
new Thread(System.out::println).start();
```

```
ExecutorService exec = Executors.newCachedThreadPool();
exec.submit(System.out::println);
```

В то время как вызов конструктора `Thread` и вызов метода `submit` выглядят похожими, первый компилируется, а последний — нет. Аргументы в обоих случаях идентичны (`System.out::println`), а конструктор и метод имеют перегрузку, которая принимает `Runnable`. Что же здесь происходит? Удивительный ответ заключается в том, что метод `submit` имеет перегрузку, которая принимает `Callable<T>`, а конструктор `Thread` — нет. Вы можете подумать, что это не имеет никакой разницы, потому что все перегрузки `println` возвращают `void`, поэтому ссылка на метод не может быть `Callable`. Это имеет смысл, но алгоритм разрешения перегрузки работает иначе. Возможно, не менее удивительно то, что вызов метода `submit` был бы корректным, если бы метод `println` не был перегружен. Это сочетание перегрузки ссылки на метод (`println`) и вызываемого метода (`submit`) не позволяет алгоритму разрешения перегрузки вести себя так, как вы ожидаете.

Технически говоря, проблема заключается в том, что `System.out::println` является *неточной ссылкой на метод* (inexact method reference) [25, 15.13.1] и что “определенные выражения аргументов, которые содержат неявно типизированные лямбда-выражения или неточные ссылки на метод, игнорируются тестами на применимость, поскольку их значение не может быть определено до тех пор, пока не будет выбран целевой тип” [25, 15.12.2]. Не волнуйтесь, если вы не понимаете этот пассаж; он предназначен для разработчиков компиляторов. Ключевым моментом является то, что перегрузка методов или конструкторов с различными функциональными интерфейсами в одной и той же позиции аргумента вызывает путаницу. Таким образом, **не перегружайте методы, принимающие различные функциональные интерфейсы в одной и той же позиции аргумента**. В терминах данного раздела различные функциональные интерфейсы не являются совершенно разными. Java-компилятор предупредит вас о такого рода проблеме перегрузки, если вы передадите в командной строке параметр `-Xlint:overloads`.

Типы массивов и типы классов, отличные от `Object`, совершенно различны. Кроме того, совершенно различаются типы массивов и типы интерфейсов, отличные от `Serializable` и `Cloneable`. Считается, что два различных класса *не связаны* (*unrelated*), если ни один класс не является потомком другого [25, 5.5]. Например `String` и `Throwable` не связаны. Никакой объект не может быть экземпляром двух несвязанных классов, поэтому несвязанные классы также являются совершенно различными.

Есть и другие пары типов, которые не могут быть преобразованы ни в одном направлении [25, 5.1.12], но если вы выходите за рамки простых описанных выше случаев, то многим программистам будет трудно понять, какая перегрузка относится к тому или иному множеству фактических параметров. Правила, определяющие, какая перегрузка будет выбрана, крайне сложны и с каждой новой версией языка становятся все сложнее и объемнее. Лишь немногие программисты понимают все их тонкости.

Иногда обстоятельства могут сложиться так, что вы захотите нарушить рекомендации из данного раздела, особенно при эволюции существующих классов. Например, рассмотрим класс `String`, у которого был метод `contentEquals(StringBuffer)`, начиная с версии Java 4. В версии Java 5 в класс был добавлен новый интерфейс `CharSequence` для обеспечения общего интерфейса для `StringBuffer`, `StringBuilder`, `String`, `CharBuffer` и других подобных типов. В то же время, когда был добавлен `CharSequence`, `String` получил перегрузку метода `contentEquals`, который получает `CharSequence`.

Получившаяся перегрузка очевидным образом нарушает рекомендации данного раздела, но не причиняет вреда, поскольку оба перегруженных метода делают одно и то же при запуске с одной и той же ссылкой на объект. Программисты могут не знать, какая именно перегрузка выполняется, но никаких последствий это не вызывает, пока они ведут себя одинаково. Стандартный способ гарантировать это поведение — вызвать более общую перегрузку из более конкретной:

```
// Гарантирует, что 2 метода имеют идентичное поведение
public boolean contentEquals(StringBuffer sb)
{
    return contentEquals((CharSequence) sb);
}
```

В то время как библиотеки для платформы Java в основном следуют приведенным здесь советам, все же можно найти ряд классов, в которых они нарушаются. Например, класс `String` экспортирует два перегруженных статических фабричных метода `valueOf(char[])` и `valueOf(Object)`, которые при получении ссылки на один и тот же объект выполняют совершенно разную

работу. Этому нет внятного объяснения, и следует рассматривать такие методы как аномалию, способную вызвать настоящую неразбериху.

В итоге прибегать к перегрузке методов только потому, что это можно сделать, не следует. В общем случае лучше воздерживаться от перегрузки методов с сигнатурами с одинаковым количеством параметров. Но иногда, особенно при работе с конструкторами, следовать этому совету невозможно. В таком случае постарайтесь избежать ситуации, при которой с использованием приведения типов один и тот же набор параметров может использоваться разными вариантами перегрузки. Если же такой ситуации избежать нельзя, например, из-за того, что вы переделываете существующий класс для реализации нового интерфейса, обеспечьте одинаковое поведение всех вариантов перегрузки, получающих одни и те же параметры. Если этого не сделать, программистам будет тяжело эффективно использовать перегруженный метод или конструктор и понять, почему он не работает.

8.5. Используйте методы с переменным количеством аргументов с осторожностью

Методы с переменным количеством аргументов, официально известные как методы *переменной арности* (variable arity) [25, 8.4.1], принимают нуль или более аргументов указанного типа. Эта функциональная возможность работает следующим образом: сначала создается массив, размер которого определяется количеством аргументов, передаваемых в месте вызова, а затем значения аргументов записываются в массив и, наконец, массив передается методу.

Например, вот метод с переменным количеством аргументов, который принимает последовательность аргументов типа `int` и возвращает их сумму. Как и следовало ожидать, значение `sum(1, 2, 3)` равно 6, а `sum()` — нулю:

```
// Простое использование метода
// с переменным количеством аргументов
static int sum(int... args)
{
    int sum = 0;

    for (int arg : args)
        sum += arg;

    return sum;
}
```

Иногда нужно написать метод, который требует *одного* или нескольких аргументов некоторого типа, а не *нуля* или более. Например, предположим, что вы хотите написать функцию, которая вычисляет минимальный из аргументов. Такая функция не будет точно определенной, если клиент не передаст в нее аргументы. Длину массива аргументов можно проверить во время выполнения:

```
// НЕВЕРНЫЙ способ использования метода с переменным
// количеством аргументов для передачи одного
// или большего количества аргументов
static int min(int... args)
{
    if (args.length == 0)
        throw new IllegalArgumentException("Слишком мало аргументов");

    int min = args[0];

    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];

    return min;
}
```

Это решение имеет ряд проблем. Наиболее серьезной является то, что если клиент вызывает этот метод без аргументов, то сбой произойдет во время выполнения, а не во время компиляции. Еще одной проблемой является то, что это некрасиво. Вы вынуждены включить явную проверку корректности `args` и не можете использовать цикл `for`, если только не инициализируете `min` значением `Integer.MAX_VALUE`, что также достаточно уродливо.

К счастью, есть гораздо лучший способ достижения желаемого эффекта. Объявите метод как принимающий два параметра, один — обычный параметр указанного типа, а второй — параметр данного типа переменной длины. Это решение устраняет все недостатки предыдущего:

```
// Правильный способ использования метода с переменным
// количеством аргументов для передачи одного
// или большего количества аргументов
static int min(int firstArg, int... remainingArgs)
{
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Использование переменного количества аргументов было разработано для метода `printf`, который был добавлен к платформе в то же время, что и методы с переменным количеством элементов, и базовые средства рефлексии (раздел 9.9). Метод `printf` и рефлексия активно используют параметры переменной длины.

Соблюдайте осторожность при использовании параметров переменной длины в критических в отношении производительности ситуациях. Каждый вызов метода с переменным количеством аргументов требует размещения массива и его инициализации. Если вы определили эмпирически, что не можете позволить себе такую высокую цену, но вам нужна гибкость параметров переменной длины, то можете прибегнуть к описанной далее схеме. Предположим, вы определили, что 95% вызовов метода имеют три или менее аргументов. Объявите пять перегрузок метода: по одному для количества аргументов от нуля до трех и один метод с переменным количеством аргументов, который будет использоваться, когда количество аргументов превышает три:

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

Теперь вы знаете, что платить за создание массивов придется лишь в 5% от всех вызовов — там, где количество параметров будет больше трех. Как и в большинстве случаев оптимизации производительности, этот прием обычно неприемлем, но там, где он годится, он становится спасательным кругом.

Статическая фабрика для `EnumSet` использует этот метод для уменьшения стоимости создания множеств перечислений до минимума. Этот прием в данной ситуации пригоден, поскольку множества перечислений предоставляют конкурентоспособную с точки зрения производительности замену битовым полям (раздел 6.3).

Резюмируя, можно отметить, что методы с переменным количеством аргументов зачастую очень удобны, но не стоит забывать об их возможном отрицательном влиянии на производительность. При необходимости предваряйте параметр переменной длины обязательными параметрами.

8.6. Возвращайте пустые массивы и коллекции, а не `null`

Нередко можно увидеть методы, которые выглядят примерно следующим образом:

```
// Возвращает null для указания пустой коллекции. Не делайте так!  
private final List<Cheese> cheesesInStock = ...;  
/**  
 * @return Список, содержащий все сыры в магазине,  
 * или null, если сыров в продаже нет.  
 */  
public List<Cheese> getCheeses()  
{  
    return cheesesInStock.isEmpty() ? null  
        : new ArrayList<>(cheesesInStock);  
}
```

Нет никакой причины для выделения как особого случая ситуации, когда в продаже нет сыра. Такой подход приведет лишь к дополнительному коду клиента для обработки возможного возвращаемого значения `null`, например:

```
List<Cheese> cheeses = shop.getCheeses();  
if (cheeses != null && cheeses.contains(Cheese.STILTON))  
    System.out.println("Jolly good, just the thing.");
```

Такого рода многословность необходима почти при каждом вызове метода, который вместо массива или коллекции нулевой длины возвращает `null`. Это чревато ошибками, так как разработчик клиента мог забыть написать специальный код для обработки результата `null`. Такая ошибка может годами оставаться незамеченной, поскольку подобные методы, как правило, возвращают один или несколько объектов. Следует также упомянуть о том, что возврат `null` вместо пустого контейнера приводит к усложнению самого метода, возвращающего контейнер.

Иногда можно услышать возражения, что возврат значения `null` предпочтительнее возврата массива или коллекции нулевой длины потому, что позволяет избежать расходов на размещение в памяти пустого контейнера. Этот аргумент несостоятелен по двум причинам. Во-первых, на этом уровне нет смысла беспокоиться о производительности, если только профилирование программы не покажет, что именно этот метод является основной причиной падения производительности (раздел 9.11). Во-вторых, *можно* возвращать пустой массив или коллекцию и без выделения памяти. Вот типичный код, возвращающий, возможно, пустую коллекцию. Обычно это все, что вам нужно:

```
// Корректный способ возврата, возможно, пустой коллекции  
public List<Cheese> getCheeses()  
{  
    return new ArrayList<>(cheesesInStock);  
}
```

В том маловероятном случае, когда у вас есть свидетельства о том, что выделение пустой коллекции вредит производительности, вы можете избежать выделения, возвращая одну и ту же *неизменяемую* пустую коллекцию многократно, так как неизменяемые объекты могут свободно использоваться совместно (раздел 4.3). Вот код, который делает это, используя метод `Collections.emptyList`. Если вы возвращали множество, вы должны использовать `Collections.emptySet`; если отображение — `Collections.emptyMap`. Но помните, что это оптимизация, и она требуется редко. Если вы полагаете, что нуждаетесь в ней, выполните измерения производительности до и после, чтобы гарантировать, что этот подход действительно вам помогает:

```
// Оптимизация — избежать выделения пустой коллекции
public List<Cheese> getCheeses()
{
    return cheesesInStock.isEmpty() ? Collections.emptyList()
        : new ArrayList<>(cheesesInStock);
}
```

Ситуация для массивов идентична ситуации для коллекций. Никогда не возвращайте `null` вместо массива нулевой длины. Обычно вы должны просто возвращать массив корректной длины, которая может быть равна нулю. Обратите внимание, что мы передаем массив нулевой длины в метод `toArray`, чтобы указать желаемый возвращаемый тип, который представляет собой `Cheese[]`:

```
// Корректный способ возврата, возможно, пустого массива
public Cheese[] getCheeses()
{
    return cheesesInStock.toArray(new Cheese[0]);
}
```

Если вы считаете, что выделение массива нулевой длины вредит производительности, то можете возвращать один и тот же массив нулевой длины многократно, потому что все массивы нулевой длины являются неизменяемыми:

```
// Оптимизация — избежать выделения пустого массива
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
public Cheese[] getCheeses()
{
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

В оптимизированной версии мы передаем в каждый вызов `toArray` *один и тот же* пустой массив, и этот массив будет возвращаться из `getCheeses`, когда `cheesesInStock` пуст. *Не* выделяйте массив для передачи в `toArray` в

надежде на улучшение производительности заранее. Исследования показали, что это контрпродуктивно [42]:

```
// Не делайте так - выделение массива
// заранее вредит производительности!
return cheesesInStock.toArray(new Cheese[cheesesInStock.size()]);
```

Итак, **никогда не возвращайте null вместо пустого массива или коллекции**. Это делает ваш API более трудным в использовании и более склонным к ошибкам, при этом не давая никакого преимущества в производительности.

8.7. Возвращайте Optional с осторожностью

До Java 8 существовали два подхода, к которым можно было прибегнуть при написании метода, который при определенных обстоятельствах не мог вернуть значение. Можно сгенерировать исключение или вернуть null (при условии, что возвращается объект ссылочного типа). Ни один из этих подходов не является идеальным. Исключения должны быть зарезервированы для исключительных ситуаций (раздел 10.1); кроме того, генерация исключения — операция дорогостоящая, поскольку требует захвата всего стека вызовов при генерации исключения. Возврат значения null не имеет этих недостатков, но имеет собственные. Если метод возвращает значение null, клиенты должны поддерживать специальный код для обработки возможного возврата null, если только программист не сможет *доказать*, что возврат null в его вызове невозможен. Если клиент пренебрегает проверкой возвращаемого значения на null и сохраняет возвращаемое в некоторой структуре данных, то в некоторый момент в будущем возможна генерация исключения NullPointerException в месте кода, в котором уже будет поздно что-либо делать с этой проблемой.

В Java 8 появился третий подход к написанию методов, которые могут в некоторых ситуациях не возвращать значение. Класс `Optional<T>` представляет собой неизменяемый контейнер, который может хранить единственную не-null-ссылку на `T` или не хранить вообще ничего. Объект, не хранящий никакого значения, называется *пустым*. Находящееся в непустом объекте `Optional` значение называется *присутствующим*. `Optional`, по сути, представляет собой неизменяемую коллекцию, которая может содержать не более одного элемента. Класс `Optional<T>` не реализует `Collection<T>`, но он мог бы, в принципе, это делать.

Метод, который концептуально возвращает `T`, но может быть не в состоянии сделать это при определенных обстоятельствах, можно объявить как

возвращающий `Optional<T>`. Это позволяет методу вернуть пустой `Optional<T>` как указание, что он не может вернуть допустимый результат. Такой метод более гибкий и проще в использовании, чем генерирующий исключение, и менее склонен к ошибкам, чем возвращающий значение `null`.

В разделе 5.5 мы приводили следующий метод для вычисления максимального значения коллекции в соответствии с естественным порядком ее элементов:

```
// Возврат максимального значения коллекции – генерируя
// исключение в случае пустой коллекции
public static <E extends Comparable<E>> E max(Collection<E> c)
{
    if (c.isEmpty())
        throw new IllegalArgumentException("Пустая коллекция");

    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);
    return result;
}
```

Этот метод генерирует исключение `IllegalArgumentException`, если данная коллекция является пустой. Мы упоминали в разделе 5.5, что лучшей альтернативой был бы возврат `Optional<E>`. Вот как выглядит метод при внесении соответствующих изменений:

```
// Возврат максимального значения коллекции как Optional<E>
public static <E extends Comparable<E>>
Optional<E> max(Collection<E> c)
{
    if (c.isEmpty())
        return Optional.empty();

    E result = null;

    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return Optional.of(result);
}
```

Как видите, вернуть `Optional` — это просто. Все, что вам нужно сделать, — создать объект с помощью подходящей статической фабрики. В данной программе мы используем их две: `Optional.empty()` возвращает пустой объект, а `Optional.of(value)` возвращает объект, содержащий

указанное значение, не являющееся null. Передача значения null в `Optional.of(value)` является ошибкой. Если вы сделаете это, метод сгенерирует исключение `NullPointerException`. Метод `Optional.ofNullable(value)` принимает значение, которое может быть null, и возвращает пустой объект в случае передачи значения null. **Никогда не возвращайте значение null из метода, возвращающего Optional:** тем самым теряется сам смысл использования `Optional`.

Многие завершающие операции потоков возвращают `Optional`. Если мы перепишем метод `max` для использования потока, то операция `max` потока `Stream` будет выполнять работу по генерации `Optional` вместо нас (хотя мы должны передать ему явный компаратор):

```
// Возврат максимального значения коллекции
// как Optional<E> - использует поток
public static <E extends Comparable<E>>
Optional<E> max(Collection<E> c)
{
    return c.stream().max(Comparator.naturalOrder());
}
```

Так как же решить — вернуть `Optional` вместо возврата null или генерации исключения или нет? **Объекты Optional по духу аналогичны проверяемым исключениям** (раздел 10.3) в том, что они *заставляют* пользователя API признать тот факт, что возвращаемого значения может не быть. Генерация непроверяемого исключения или возврат null позволяет пользователю игнорировать этот случай, что потенциально чревато катастрофическими последствиями. Однако генерация проверяемых исключений требует дополнительного стереотипного кода клиента.

Если метод возвращает `Optional`, клиент может выбрать, какие действия предпринять, если метод не может вернуть значение. Можно указать значение по умолчанию:

```
// Использование Optional для предоставления значения по умолчанию
String lastWordInLexicon = max(words).orElse("Нет слов...");
```

Или можно сгенерировать любое подходящее исключение. Обратите внимание, что мы передаем в исключение фабрику, а не фактическое исключение. Это позволяет избежать создания исключения, если оно на самом деле не генерируется:

```
// Использование Optional для генерации выбранного исключения
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

Если можно *доказать*, что объект `Optional` непустой, то можно получить значение и без указания действия для случая пустого `Optional`. Но если вы

ошибетесь, то ваш код будет генерировать исключение `NoSuchElementException`:

```
// Использование Optional, когда известно,
// что возвращаемое значение точно есть
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

Иногда вы можете столкнуться с ситуацией, когда получение значения по умолчанию оказывается дорогостоящим, и хотите этой стоимости, если только она не является необходимой. Для таких ситуаций `Optional` предоставляет метод, который принимает `Supplier<T>` и вызывает его только при необходимости. Этот метод называется `orElseGet`, но, возможно, он должен был бы быть назван `orElseCompute` потому, что он тесно связан с тремя методами `Map`, имена которых начинаются с `compute`. Есть несколько методов `Optional` для более специализированных случаев: `filter`, `map`, `flatMap` и `ifPresent`. В Java 9 к ним были добавлены еще два метода: `or` и `ifPresentOrElse`. Если описанные базовые методы не подходят для вашей ситуации, просмотрите документацию этих более сложных методов — возможно, они делают именно то, что вам нужно.

В случае, если ни один из этих методов не отвечает вашим потребностям, `Optional` предоставляет метод `isPresent()`, который может рассматриваться как предохранительный клапан. Этот метод возвращает `true`, если `Optional` содержит значение, и `false`, если объект пуст. Этот метод можно использовать, чтобы выполнять любую обработку, которая вам нужна, но убедитесь в том, что вы используете его с умом. Многие применения `isPresent` можно с выгодой заменить одним из указанных выше методов. Как правило, получающийся в результате код короче, более четкий и идиоматичный.

Например, рассмотрим следующий фрагмент кода, который выводит идентификатор родительского процесса, или N/A, если процесс не имеет родителя. Показанный фрагмент кода использует класс `ProcessHandle`, введенный в Java 9:

```
Optional<ProcessHandle> parentProcess = ph.parent();
System.out.println("Parent PID: " + (parentProcess.isPresent() ?
    String.valueOf(parentProcess.get().pid()) : "N/A"));
```

Показанный выше фрагмент кода можно заменить следующим, использующим функцию `map` класса `Optional`:

```
System.out.println("Parent PID: " +
    ph.parent().map(h -> String.valueOf(h.pid())).orElse("N/A"));
```

При программировании с потоками не редкость обнаружить `Stream<Optional<T>>` и потребовать, чтобы для дальнейшей работы `Stream<T>`

содержал все элементы в непустом `Optional`. Если вы используете Java 8, можете поступить следующим образом:

```
streamOfOptionals
    .filter(Optional::isPresent)
    .map(Optional::get)
```

В Java 9 класс `Optional` оснащен методом `stream()`. Этот метод представляет собой адаптер, который превращает `Optional` в `Stream`, содержащий элемент, если таковой присутствует в `Optional`, или не содержащий элементы, если объект пуст. В сочетании с методом `flatMap` в `Stream` (раздел 7.4) этот метод обеспечивает краткую замену показанного ранее фрагмента:

```
streamOfOptionals
    .flatMap(Optional::stream)
```

Не все возвращаемые типы получают выгоду от применения `Optional`. **Типы контейнеров, включая коллекции, отображения, потоки, массивы и `Optional`, не должны оборачиваться в `Optional`.** Вместо возврата пустого `Optional<List<T>>` вы должны возвращать просто пустой `List<T>` (раздел 8.6). Возврат пустого контейнера устраняет необходимость в клиентском коде для обработки `Optional`. Класс `ProcessHandle` имеет метод `arguments`, который возвращает `Optional<String[]>`, но этот метод следует рассматривать как аномалию, которую не стоит рассматривать как пример.

Так когда же объявлять метод как возвращающий `Optional<T>` вместо `T`? Как правило, **следует объявлять метод как возвращающий `Optional<T>`, если он не в состоянии вернуть результат, а клиенты должны выполнять специальную обработку, когда результат не возвращается.** Возврат `Optional<T>` имеет свою стоимость. `Optional` представляет собой объект, который должен быть выделен и инициализирован, а для чтения значения из `Optional` требуется дополнительное косвенное обращение. Это делает `Optional` непригодным для использования в некоторых ситуациях, когда производительность оказывается критической. Попадает ли конкретный метод в эту категорию — можно определить только путем тщательных измерений (раздел 9.11).

Возврат `Optional`, который содержит упакованный примитивный тип, оказывается слишком дорогим по сравнению с возвратом примитивного типа, потому что `Optional` имеет два уровня упаковки вместо нуля. Поэтому разработчики библиотек сочли нужным предоставить аналоги `Optional<T>` для примитивных типов `int`, `long` и `double`. Этими дополнительными типами являются `OptionalInt`, `OptionalLong` и `OptionalDouble`. Они содержат большинство (но не все) методов `Optional<T>`. Поэтому **вы никогда не должны возвращать `Optional` для упакованных примитивных типов,**

с возможным исключением для “младших примитивных типов” Boolean, Byte, Character, Short и Float.

До сих пор мы рассматривали варианты возврата объектов Optional и их обработки после возврата. Мы не обсуждали другие возможные применения, потому что большинство других вариантов использования Optional подозрительны. Например, никогда не следует использовать Optional в качестве значений отображений. Если вы это сделаете, у вас будет два способа выражения логического отсутствия ключа в карте: ключ может либо отсутствовать в отображении, либо присутствовать, но отображаться на пустой Optional. Это оказывается ненужной сложностью с большим потенциалом для путаницы и ошибок. В общем случае почти всегда нецелесообразно использовать Optional как ключ, значение или элемент коллекции или массива.

Это оставляет без ответа большой вопрос. Имеет ли вообще в каких-то ситуациях смысл хранить Optional в поле экземпляра? Часто это “душок”, свидетельствующий о том, что, возможно, вам необходим подкласс, содержащий поля с Optional. Но иногда это может оказаться оправданным. Рассмотрим случай нашего класса NutritionFacts из раздела 2.2. Экземпляр NutritionFacts содержит много ненужных полей. Вы не можете иметь подкласс для каждой возможной комбинации этих полей. Кроме того, поля имеют примитивные типы, что делает неудобным непосредственное выражение отсутствия значения. Лучший API для NutritionFacts должен возвращать Optional из каждого метода доступа для необязательного поля, поэтому имеет смысл просто хранить эти Optional как поля объекта.

Итак, если вы пишете метод, который может не всегда возвращать значение, и считаете важным, чтобы пользователи метода учитывали эту возможность при каждом вызове метода, то вы, вероятно, должны возвращать объект Optional. Однако следует помнить, что возврат Optional оказывает влияние на производительность. Если для некоторого метода производительность является критичной, возможно, лучше возвращать значение null или генерировать исключение. Наконец, Optional редко используется в любом качестве, отличным от возвращаемого значения.

8.8. Пишите документирующие комментарии для всех открытых элементов API

Если API предназначен для использования, он должен быть документирован. Традиционно документация API генерировалась вручную и поддержка синхронизации документации с кодом была неприятной рутинной работой. Среда программирования Java упрощает эту задачу с помощью утилиты

Javadoc. Javadoc генерирует документацию API автоматически из исходного кода со специальным образом отформатированными *документирующими комментариями*, более широко известными как *doc-комментарии*.

Хотя соглашения о doc-комментариях официально не являются частью языка, де-факто они представляют собой API, который должен знать каждый программист Java. Эти соглашения описаны в разделе *How to Write Doc Comments* (Как писать документирующие комментарии) веб-страницы [23]. Хотя эта страница не обновлялась еще с выпуска Java 4, она по-прежнему представляет собой бесценный ресурс. В Java 9 был добавлен один важный дескриптор, `{@index}`; в Java 8 был добавлен дескриптор `{@implSpec}`; и еще два дескриптора были добавлены в Java 5: `{@literal}` и `{@code}`. Эти дескрипторы отсутствуют на вышеупомянутой веб-странице, но обсуждаются в этом разделе.

Чтобы должным образом документировать свой API, следует предварительно каждый экспортируемый класс, интерфейс, конструктор, метод и объявление поля документирующим комментарием. Если класс является сериализуемым, следует также документировать его сериализованную форму (раздел 12.3). В отсутствие документирующих комментариев лучше, что может сделать Javadoc, — это воспроизвести объявление как единственную документацию рассматриваемого элемента API. Работа с API с отсутствующими документирующими комментариями затруднена и ведет к ошибкам. Открытые классы не должны использовать конструкторы по умолчанию, потому что нет никакого способа предоставить для них документирующие комментарии. Чтобы написать простой в сопровождении код, следует также написать документирующие комментарии для большинства неэкспортируемых классов, интерфейсов, конструкторов, методов и полей, хотя эти комментарии не должны быть написаны так тщательно, как для экспортируемых элементов API.

Документирующий комментарий метода должен лаконично описывать контракт между этим методом и его клиентом. Контракт должен точно оговаривать, *что* делает данный метод, а не *как* он это делает. Исключение составляют методы в классах, предназначенных для наследования (раздел 4.5). В документирующем комментарии необходимо перечислить все *предусловия* (precondition), т.е. условия, которые должны выполняться для того, чтобы клиент мог вызвать этот метод, и *постусловия* (postcondition), т.е. утверждения, которые будут истинными после успешного завершения вызова. Обычно предусловия неявно описываются дескрипторами `@throws` для непроверяемых исключений. Каждое непроверяемое исключение соответствует нарушению некоего предусловия. Предусловия могут также быть указаны вместе с параметрами, которых они касаются, в соответствующих дескрипторах `@param`.

Помимо предусловий и постусловий, в методах должны быть документированы любые *побочные эффекты*. Побочный эффект (side effect) — это наблюдаемое изменение состояния системы, которое является неочевидным условием для достижения постусловия. Например, если метод запускает фоновый поток, то это должно быть отражено в документации.

Для полного описания контракта метода документирующий комментарий метода должен включать дескрипторы `@param` для каждого параметра, дескриптор `@return` (если только метод не возвращает `void`), и дескриптор `@throws` для каждого исключения, которое может быть сгенерировано этим методом, — как проверяемого, так и непроверяемого (раздел 10.6). Если текст в дескрипторе `@return` идентичен описанию метода, его можно опустить (если это разрешает используемый вами стандарт кодирования).

По соглашению текст, следующий за дескрипторами `@param` и `@return`, должен представлять собой именованную конструкцию (noun phrase), описывающую значение параметра или возвращаемого значения. Текст, следующий за тегом `@throws`, должен состоять из слова *если* и именной конструкции, описывающей условия, при которых генерируется данное исключение. По соглашению после текстов, следующих за дескрипторами `@param`, `@return` и `@throws`, точка не ставится. Все эти соглашения иллюстрирует следующий документирующий комментарий:

```
/**
 * Возвращает элемент в указанной позиции этого списка.
 *
 * <p>Этот метод <i>не</i> гарантирует работу за константное время.
 * В некоторых реализациях он может выполняться за время,
 * пропорциональное позиции элемента.
 *
 * @param index Индекс возвращаемого элемента; должен быть
 *             неотрицательным и меньше размера списка
 * @return Элемент в указанной позиции списка
 * @throws IndexOutOfBoundsException если индекс
 *             выходит за пределы диапазона
 *             ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

Обратите внимание на использование HTML-дескрипторов в этом комментарии (`<p>` и `<i>`). Утилита Javadoc преобразует документирующие комментарии в HTML, так что произвольные элементы HTML в документирующем комментарии в конечном итоге оказываются в генерируемом HTML-документе. Иногда программисты заходят столь далеко, что даже вставляют HTML-таблицы в документирующие комментарии, хотя это бывает редко.

Обратите также внимание на использование дескриптора Javadoc {@code} вокруг фрагмента кода в описании @throws. Он необходим для решения двух задач: во-первых, он приводит к тому, что фрагмент кода отображается соответствующим шрифтом, а во-вторых, он запрещает обработку HTML-разметки во вложенных дескрипторах Javadoc в фрагменте кода. Это означает возможность, например, использования знака “меньше” (<) в фрагменте кода несмотря на то, что это метасимвол HTML. Для добавления многострочного примера кода в документирующий комментарий используйте дескриптор Javadoc {@code} внутри дескриптора HTML <pre>. Другими словами, пример многострочного кода должен начинаться с <pre>{@code и заканчиваться символами }</pre>. Это сохраняет разрывы строк в коде и устраняет необходимость использовать управляющие последовательности для метасимволов HTML, но не для знака @, для которого необходимо использовать управляющую последовательность (например, если фрагмент кода использует аннотации).

Наконец, отметим появление в doc-комментарии слова *этом* (this). По соглашению это слово всегда указывает объект, которому принадлежит вызываемый метод, соответствующий данному комментарию.

Как упоминалось в разделе 4.1, при создании класса для наследования необходимо документировать его *схемы использования собственных методов*, чтобы программисты знали семантику перекрываемых методов. Такое использование собственных методов должно документироваться с помощью дескриптора @implSpec, добавленного в Java 8. Напомним, что обычные документирующие комментарии описывают контракт между методом и его клиентом; комментарии @implSpec, напротив, описывают контракт между методом и его подклассом, позволяя подклассам полагаться на реализацию поведения при наследовании метода или вызове его через super. Вот как это выглядит на практике:

```
/**
 * Возвращает true, если эта коллекция пуста.
 *
 * @implSpec
 * Эта реализация возвращает {@code this.size() == 0}.
 *
 * @return true если коллекция пуста
 */
public boolean isEmpty()
{
    ...
}
```

По состоянию на Java 9 утилита Javadoc по-прежнему игнорирует дескриптор @implSpec, если только вы не передаете в командной строке переключатель

-tag "implSpec:a:Implementation Requirements:". Надеемся, что это будет исправлено в следующих выпусках.

Не забудьте, что необходимо предпринять специальные действия для генерации документации, содержащей метасимволы HTML, такие как знаки `<`, `>` и `&`. Наилучший способ вставить эти символы в документацию — заключить их в дескриптор `{@literal}`, который подавляет обработку HTML-разметки и вложенные дескрипторы Javadoc. Это напоминает применение дескриптора `{@code}` с тем отличием, что он не выводит текст особым шрифтом для кода. Например, фрагмент Javadoc

* Геометрический ряд сходится, если `{@literal |r| < 1}`.

генерирует документацию “Геометрический ряд сходится, если $|r| < 1$ ”. Дескриптор `{@literal}` мог быть размещен только вокруг знака “меньше”, а не вокруг всего неравенства, причем документация получилась бы такой же, но тогда документирующий комментарий было бы сложнее читать в исходном коде. Этот пример иллюстрирует общий принцип: **документирующие комментарии должны быть читаемыми как в исходном тексте, так и в сгенерированной документации**. Если и того, и другого достичь не удастся, то предпочтительнее, конечно, удобочитаемость сгенерированной документации.

Первым предложением любого документирующего комментария является *краткое описание* (summary description) того элемента, к которому этот комментарий относится. Например, краткое описание в приведенном ранее примере комментария имеет вид “Возвращает элемент в указанной позиции этого списка”. Краткое описание должно описывать функциональность соответствующей сущности. Во избежание путаницы **никакие два члена или конструктора в одном классе или интерфейсе не должны иметь одинаковое краткое описание**. Обращайте особое внимание на перегруженные методы, описание которых часто хочется начать с одного и того же предложения (что неприемлемо в документации).

Внимательно следите за тем, чтобы в первом предложении документирующего комментария не было лишней точки, так как точка является признаком завершения краткого описания. Например, документирующий комментарий, начинающийся с фразы “Ученая степень, как, например, к.т.н. или д.ф.-м.н.” приведет к появлению в кратком описании фразы “Ученая степень, как, например, к.т.н.” Краткое описание заканчивается первой точкой, за которой идут пробел, символ табуляции или конца строки (или первый дескриптор) [24]. Здесь, как видите, после последней точки в *к.т.н.* идет пробел. Лучшим решением будет не использовать в кратком описании точек или окружать их и связанный с ними текст дескриптором `{@literal}`:

```
/**
 * Ученая степень, как, например, {@literal к.т.н.} или д.ф.-м.н.
 */
public class Degree
{
    ...
}
```

Утверждение о том, что краткое описание является первым *предложением* документирующего комментария, немного вводит в заблуждение. Соглашение гласит, что оно редко бывает полным предложением. Для методов и конструкторов краткое описание должно быть глагольной фразой (включая любой объект), описывающей действия, выполняемые методом, например, такой.

- `ArrayList(int initialCapacity)` — конструирует пустой список с указанной начальной емкостью.
- `Collection.size()` — возвращает количество элементов в этой коллекции.

Как показано в этих примерах, используется глагол в изъявительном наклонении третьего лица (“возвращает число”) вместо глагола в повелительном наклонении второго лица (“возврат числа”).

Для классов, интерфейсов и полей краткое описание должно быть именной конструкцией, описывающей сущность, представляемую экземпляром класса или интерфейса, или самим полем, например так.

- `Instant` — мгновенная точка на временной линии.
- `Math.PI` — значение типа `double`, находящееся ближе других к значению π (отношению длины окружности к ее диаметру).

В Java 9 в HTML, создаваемый Javadoc, добавляется предметный указатель. Этот указатель, облегчающий навигацию по большому количеству документации API, принимает форму поля поиска в правом верхнем углу страницы. Когда вы вводите в поле текст, то получаете раскрывающийся список соответствующих страниц. Элементы API, таких как классы, методы и поля, индексируются автоматически. Но иногда вы можете добавить в предметный указатель дополнительные пункты, важные для вашего API. Для этой цели был добавлен дескриптор `{@index}`. Индексирование термина в документирующем комментарии очень простое — достаточно обернуть его в указанный дескриптор, как показано в следующем фрагменте:

* Этот метод скомпилирован согласно стандарту `{@index IEEE 754}`.

Обобщенные типы, перечисления и аннотации требуют особого внимания в документирующих комментариях. При документировании обобщенного типа или метода убедитесь, что вы документируете все параметры:

```
/**
 * Объект, отображающий ключи на значения. Отображение
 * не может содержать дублирующиеся ключи; каждый ключ
 * отображается не более чем на одно значение.
 *
 * (Остальное опущено)
 *
 * @param <K> тип ключей этого отображения
 * @param <V> тип значений этого отображения
 */
public interface Map<K, V>
{
    ...
}
```

При документировании типа перечисления убедитесь, что вы документируете константы, так же как типы и любые открытые методы. Обратите внимание, что если комментарий краткий, его можно полностью поместить в одну строку:

```
/**
 * Категории инструментов симфонического оркестра.
 */
public enum OrchestraSection
{
    /** Деревянные духовые инструменты – флейта, кларнет, гобой. */
    WOODWIND,
    /** Медные духовые инструменты – валторна, труба. */
    BRASS,
    /** Ударные инструменты – литавры, тарелки. */
    PERCUSSION,
    /** Струнные инструменты – скрипка, виолончель. */
    STRING;
}
```

При документировании типа аннотации убедитесь, что вы документируете все члены, так же как и сам тип. Документируйте члены с помощью именных групп, как если бы это были поля. Для кратких описаний типа используйте глагольные группы, которые говорят, что означает, что элемент программы имеет аннотацию этого типа:

```
/**
 * Указывает, что аннотированный метод является
 * тестовым методом, который должен генерировать
```

```

* требуемое исключение для успешного завершения.
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest
{
    /**
     * Исключение, которое аннотированный тестовый метод
     * должен генерировать для успешного завершения. (Тест
     * может генерировать любой подтип типа, описываемого
     * этим объектом класса.)
     */
    Class <? extends Throwable > value();
}

```

Документирующие комментарии уровня пакета следует помещать в файл с именем `package-info.java`. В дополнение к документирующему комментарию уровня пакета `package-info.java` должен содержать объявление пакета и может содержать аннотации данного объявления. Аналогично, если вы решите использовать систему модулей (раздел 4.1), то документирующие комментарии уровня модуля следует поместить в файл `module-info.java`.

Двумя аспектами API, которые зачастую игнорируются, являются потоковая безопасность и возможность сериализации. **Является ли класс безопасным с точки зрения потоков или нет — в любом случае уровень его безопасности с точки зрения потоков должен быть документирован**, как описано в разделе 11.5. Если класс сериализуем, нужно документировать его сериализованный вид, как описано в разделе 12.3.

Утилита Javadoc имеет возможность “наследовать” комментарии к методам. Если элемент API не имеет документирующего комментария, Javadoc находит среди приемлемых наиболее близкий комментарий, отдавая при этом предпочтение интерфейсам, а не суперклассам. Подробности алгоритма поиска комментариев можно найти в *The Javadoc Reference Guide* [24]. Вы можете также наследовать *части* документирующих комментариев из супертипов, используя дескриптор `{@inheritDoc}`. Среди прочего это означает, что классы могут повторно использовать документирующие комментарии от интерфейсов, которые они реализуют, а не копировать эти комментарии. Эта возможность позволяет снизить бремя поддержки нескольких наборов почти идентичных документирующих комментариев, но она сложна в использовании и имеет некоторые ограничения. Детальное обсуждение этого вопроса выходит за рамки данной книги.

Следует добавить еще одно предостережение, связанное с документирующими комментариями. В то время как комментарии должны сопровождать все экспортируемые элементы API, этого не всегда достаточно. Для сложных

API, состоящих из множества взаимосвязанных классов, зачастую требуется дополнять документирующие комментарии внешним документом, описывающим общую архитектуру данного API. Если такой документ существует, то соответствующие документирующие комментарии к классу или пакету должны включать ссылку на него.

Javadoc автоматически проверяет соблюдение многих рекомендаций из данного раздела. В Java 7 для такого поведения следовало использовать параметр командной строки `-Xdoclint`; в Java 8 и 9 проверка включена по умолчанию. Подключаемые модули интегрированной среды, такие как `checkstyle`, идут еще дальше в проверке соблюдения этих рекомендаций [7]. Вы также можете уменьшить вероятность появления ошибок в документирующем комментарии, пропустив HTML-файлы, созданные Javadoc, через *программу проверки корректности HTML* (HTML validity checker). Таким образом можно обнаружить многие некорректные применения дескрипторов HTML. Несколько таких программ проверки доступны для скачивания; можно также проверить корректность HTML в Интернете с помощью службы проверки разметки W3C [52]. При проверке HTML имейте в виду, что по состоянию на Java 9 Javadoc способен генерировать как HTML5, так и HTML 4.01, хотя по умолчанию он по-прежнему генерирует HTML 4.01. Используйте параметр командной строки `-html5`, если хотите заставить Javadoc генерировать HTML5.

Соглашения, описанные в данном разделе, представляют собой основы. Более детальная информация о том, как писать документирующие комментарии, содержится в руководстве *How To Write Doc Comments* [23].

Если вы придерживаетесь изложенных в данном разделе основных принципов, то сгенерированная документация должна обеспечить четкое описание вашего API. Единственный способ убедиться в этом — **чтение веб-страниц, созданных утилитой Javadoc**. Это стоит делать для каждого API, который будет использоваться другими пользователями. Так же, как тестирование программы почти неизбежно приводит к внесению некоторых изменений в код, чтение документации, как правило, приводит к по крайней мере некоторым незначительным изменениям в документирующих комментариях.

Резюмируя, можно утверждать, что документирующие комментарии — самое лучшее, самое эффективное средство документирования API. Написание комментариев должно рассматриваться как обязательное для всех экспортируемых элементов API. Выберите стиль, который не противоречит стандартным соглашениям. Помните, что в комментариях к документации можно использовать любой код HTML, причем метасимволы HTML необходимо маскировать с помощью управляющих последовательностей.

Общие вопросы программирования

Данная глава посвящена основным элементам языка программирования Java. В ней рассматриваются локальные переменные, управляющие структуры, библиотеки, типы данных и две возможности, выходящие за рамки собственно языка программирования: *рефлексия* (reflection) и *машинно-зависимые методы* (native methods). Наконец, здесь обсуждаются оптимизация и соглашения по именованию.

9.1. Минимизируйте область видимости локальных переменных

Этот раздел по своей сути схож с разделом 4.1, “Минимизируйте доступность классов и членов”. Сужая область видимости локальных переменных, вы повышаете удобочитаемость и сопровождаемость своего кода и снижаете вероятность возникновения ошибок.

Старые языки программирования, такие как C, требуют, чтобы локальные переменные были объявлены в начале блока. У программистов зачастую вырабатывается эта привычка, но от нее, право, стоит отказаться. Напомним, что язык программирования Java позволяет объявлять переменную в любом месте, где можно использовать инструкцию (как и язык C, начиная со стандарта C99).

Наиболее мощный способ минимизации области видимости локальной переменной заключается в ее объявлении там, где она впервые используется. Объявление переменной до ее использования только засоряет программу: появляется еще одна строка, отвлекающая читателя, который пытается разобраться в том, что делает программа. К моменту фактического использования переменной программист может успеть забыть ее тип и/или ее начальное значение.

Слишком раннее объявление локальной переменной может привести не только к тому, что ее область видимости начинается слишком рано, но и к

тому, что она заканчивается слишком поздно. Область видимости локальной переменной начинается в том месте, где она объявлена, и заканчивается с завершением блока, содержавшего это объявление. Если переменная объявлена за пределами блока, в котором она используется, то она остается видимой и после того, как программа выйдет из этого блока. Если переменная случайно использована до или после области предполагаемого применения, последствия могут быть катастрофическими.

Почти каждое объявление локальной переменной должно содержать инициализатор. Если у вас недостаточно информации для правильной инициализации переменной, следует отложить объявление до тех пор, когда этой информации будет достаточно. Исключение из этого правила связано с использованием инструкций `try/catch`. Если переменная инициализируется с использованием выражения, вычисление которого может привести к генерации исключения, то такая инициализация переменной должна осуществляться внутри блока `try`. Если переменная должна использоваться за пределами блока `try`, объявлять ее следует перед блоком `try`, там, где она еще не может быть “правильно инициализирована” (пример можно найти в разделе 9.9).

Циклы предоставляют отличную возможность сужения области видимости переменных. Цикл `for` позволяет объявлять *переменные цикла* (*loop variables*), ограничивая их видимость только той областью, где они действительно нужны. (Эта область состоит из тела цикла и кода в скобках между ключевым словом `for` и телом цикла.) Поэтому **следует предпочитать цикл `for` циклу `while` при условии, что после завершения цикла содержимое переменной цикла не требуется.**

Например, вот как выглядит предпочтительная идиома обхода элементов некоторой коллекции (раздел 9.2):

```
// Предпочтительная идиома обхода элементов коллекции или массива
for (Element e : c)
{
    ... // Выполнение действий с e
}
```

Если вам нужен доступ к итератору, например, для вызова его метода `remove`, предпочтительная идиома использует традиционный цикл `for` вместо цикла `for` для коллекции:

```
// Идиома обхода в случае необходимости доступа к итератору
for (Iterator<Element> i = c.iterator(); i.hasNext(); )
{
    Element e = i.next();
    ... // Выполнение действий с e и i
}
```

Чтобы понять, почему эти циклы `for` предпочтительнее цикла `while`, рассмотрим следующий фрагмент кода, который содержит два цикла `while` и одну ошибку:

```
Iterator<Element> i = c.iterator();
while (i.hasNext())
{
    doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while (i.hasNext())
{ // Ошибка!
    doSomethingElse(i2.next());
}
```

Второй цикл содержит типичную ошибку, связанную с копированием и вставкой (для последующего исправления) фрагмента программы: инициализируется новая переменная цикла `i2`, но используется старая переменная `i`, которая, к сожалению, остается в области видимости. Полученный код компилируется без замечаний и выполняется без генерации исключений, но работает не так, как ожидал программист. Вместо того чтобы выполнить итерирование коллекции `c2`, второй цикл завершается немедленно, создавая ложное впечатление, что коллекция `c2` пуста. Поскольку программа ничего об этой ошибке не сообщает, последняя может оставаться незамеченной долгое время.

Если подобная ошибка копирования и вставки будет сделана для цикла `for`, то получающийся в результате код даже не будет компилироваться. Переменная элемента (или итератора) из первого цикла не будет находиться в области видимости второго цикла. Вот как выглядит ситуация с традиционным циклом `for`:

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); )
{
    Element e = i.next();
    ... // Do something with e and i
}
...
// Ошибка времени компиляции - имя i не найдено
for (Iterator<Element> i2 = c2.iterator(); i2.hasNext(); )
{
    Element e2 = i2.next();
    ... // Выполнение действий с e2 и i2
}
```

Более того, если вы пользуетесь идиомой цикла `for`, существенно уменьшается вероятность того, что вы допустите ошибку при копировании и вставке,

поскольку нет никаких причин использовать в двух циклах различные названия переменных. Эти циклы абсолютно независимы, а потому нет никакого вреда от повторного применения одного и того же имени переменной элемента (или итератора). На самом деле это даже стильно.

Цикл `for` имеет еще одно преимущество перед циклом `while`: он короче, что повышает его удобочитаемость.

Вот еще одна идиома, минимизирующая область видимости локальной переменной:

```
for (int i = 0, n = expensiveComputation(); i < n; i++)
{
    ... // Выполнение некоторых действий с i;
}
```

Здесь важно заметить наличие *двух* переменных цикла, `i` и `n`; они обе имеют верную область видимости. Вторая переменная, `n`, используется для хранения граничного значения первой переменной, тем самым позволяя избежать (возможно, дорогостоящих) излишних вычислений на каждой итерации. Как правило, эта идиома должна использоваться, когда проверка условия цикла включает вызов метода, который гарантированно возвращает один и тот же результат на каждой итерации цикла.

Последний прием, позволяющий уменьшить область видимости локальных переменных, заключается в **сохранении малого размера и точной направленности методов**. Если в пределах одного и того же метода вы объединяете две операции, то локальные переменные, относящиеся к одной из них, могут попасть в область видимости другой. Во избежание этого просто разделите метод на два, по одному методу для каждой операции.

9.2. Предпочитайте циклы `for` для коллекции традиционным циклам `for`

Как уже говорилось в разделе 7.4, одни задания лучше выполнять с использованием потоков, в то время как для выполнения других лучше подходят итерации. Вот применение традиционного цикла `for` для обхода коллекции:

```
// Не лучший способ обхода коллекции!
for (Iterator<Element> i = c.iterator(); i.hasNext(); )
{
    Element e = i.next();
    ... // Выполнение некоторых действий с e
}
```

А вот как выглядит применение традиционного цикла `for` для обхода массива:

```
// Не лучший способ обхода массива!
for (int i = 0; i < a.length; i++)
{
    ... // Выполнение некоторых действий с a[i]
}
```

Эти идиомы лучше, чем циклы `while` (раздел 9.1), но они не идеальны. Итераторы и индексные переменные создают лишние помехи: все, что вам нужно для работы, — это сами элементы. Кроме того, они создают возможности для внесения ошибок. Итератор встречается в каждом цикле трижды, а индексная переменная — четырежды, что дает широкие возможности использовать неверную переменную. Если это произойдет, нет никакой гарантии, что компилятор заметит проблему. Наконец, эти два цикла достаточно различны, излишне обращают внимание программиста на тип контейнера и создают (не очень серьезные) сложности при изменении этого типа.

Цикл по коллекции (`for-each`, официально известный как “расширенный цикл `for`”) решает все указанные проблемы. Он избавляет программистов от путаницы и возможностей для внесения ошибок, полностью скрывая итератор или индексную переменную. Получающаяся в результате идиома равно применима ко всем коллекциям и массивам и упрощает процесс переключения реализации от одного типа контейнера к другому:

```
// Предпочтительная идиома для обхода коллекций и массивов
for (Element e : elements)
{
    ... // Выполнение некоторых действий с e
}
```

Когда вы видите двоеточие (:), читайте его как “из” или “в”. Таким образом, показанный выше цикл можно прочесть как “для каждого элемента *e* из *elements*”. Никаких накладных расходов на применение такого цикла нет, даже для массивов: генерируемый им код, по сути, идентичен коду, который вы бы написали вручную.

Преимущество цикла по коллекции над традиционным циклом оказывается еще большим в случае вложенных итераций. Вот распространенная ошибка, которую допускают программисты при выполнении вложенных итераций:

```
// Сможете ли вы найти ошибку?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
static Collection<Suit> suits = Arrays.asList(Suit.values());
```

```
static Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

Не огорчайтесь, если вы не смогли найти ошибку. Время от времени ее допускают многие эксперты в программировании. Проблема в том, что метод `next` вызывается слишком много раз с итератором внешней коллекции (`suits`). Он должен вызываться во внешнем цикле, так, чтобы на каждую масть приходился один вызов, но вместо этого он вызывается во внутреннем цикле, так что он вызывается по одному разу для каждой карты. После того как вы пройдете по всем мастям, цикл сгенерирует исключение `NoSuchElementException`.

Если вам реально не повезет и размер внешней коллекции окажется кратным размеру внутренней коллекции — возможно, потому что это одна и та же коллекция, — то цикл завершится нормально, но он будет делать совсем не то, что вы от него хотите. Например, рассмотрим неудачную попытку напечатать все возможные комбинации пары при бросании игровых костей:

```
// Та же ошибка, но с другим симптомом!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = EnumSet.allOf(Face.class);

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

Программа не генерирует исключения, но вместо ожидаемых 36 комбинаций выводит шесть “дублей” (от “ONE ONE” до “SIX SIX”).

Для исправления ошибки можно добавить переменную в области видимости внешнего цикла для хранения внешнего элемента:

```
// Исправлено, но исправлено уродливо — можно сделать лучше!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
{
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

Если вместо этого использовать цикл по коллекции, проблема просто исчезает. Получающийся в результате код красив и лаконичен:

// Предпочтительная идиома для вложенных обходов коллекций и массивов

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

Увы, имеются три ситуации, в которых *нельзя* применять цикл по коллекции.

- **Деструктивная фильтрация** — если нужно обойти коллекцию с удалением выбранных элементов, то нужно использовать явный итератор, чтобы иметь возможность вызова его метода `remove`. Часто можно избежать явного обхода коллекции, используя ее метод `removeIf`, добавленный в Java 8.
- **Преобразование** — если требуется обойти список или массив и заменить некоторые или все значения его элементов, то для этого нужен итератор списка или индекс массива.
- **Параллельное итерирование** — если нужен параллельный обход нескольких коллекций, требуется явное управление итератором или индексной переменной, чтобы все итераторы или индексные переменные могли изменяться одновременно (как было непреднамеренно продемонстрировано выше в ошибочном коде с картами и игральными костями).

Если вы окажетесь в одной из этих ситуаций, используйте обычный цикл `for` и не забывайте о ловушках, описанных в этом разделе.

Цикл `for` для коллекций позволяет обходить не только коллекции и массивы; он позволяет выполнять обход любого объекта, который реализует интерфейс `Iterable`, который состоит из единственного метода. Вот как выглядит этот интерфейс:

```
public interface Iterable<E>
{
    // Возвращает итератор для элементов данного объекта
    Iterator<E> iterator();
}
```

Реализовать интерфейс `Iterable` достаточно сложно, если вы пишете собственную реализацию `Iterator` “с нуля”, но если вы пишете тип, который представляет группу элементов, вы обязательно должны подумать над тем, чтобы он реализовывал интерфейс `Iterable`, даже если вы не намерены реализовывать `Collection`. Это позволит вашим пользователям выполнять обход вашего типа с помощью цикла `for` для коллекций, за что они, несомненно, будут вам благодарны.

Итак, цикл `for` для коллекций предоставляет неоспоримые преимущества по сравнению с традиционным циклом `for` в ясности, гибкости кода и предотвращении ошибок, без дополнительных накладных расходов. Используйте циклы `for` для коллекций вместо традиционных циклов `for` везде, где только можно.

9.3. Изучите и используйте возможности библиотек

Предположим, вы хотите генерировать случайные целые числа от нуля до некоторой верхней границы. Столкнувшись с такой распространенной задачей, многие программисты пишут небольшой метод, который имеет приблизительно следующий вид:

```
// Распространено, но глубоко неверно!
static Random rnd = new Random();
static int random(int n)
{
    return Math.abs(rnd.nextInt()) % n;
}
```

Возможно, этот метод хорошо выглядит, но он имеет три недостатка. Первый заключается в том, что если `n` — небольшая степень двойки, то последовательность случайных чисел будет повторяться после достаточно короткого периода. Второй недостаток заключается в том, что если `n` не является степенью двойки, то одни числа в среднем будут возвращаться чаще других. Если `n` велико, этот эффект может проявляться достаточно отчетливо. Это явление демонстрируется следующей программой, которая генерирует миллион случайных чисел в тщательно выбранном диапазоне, а затем выводит количество чисел, попавших в нижнюю половину этого диапазона:

```
public static void main(String[] args)
{
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;

    for (int i = 0; i < 1000000; i++)
        if (random(n) < n / 2)
            low++;

    System.out.println(low);
}
```

Если бы метод `random` работал правильно, программа выводила бы число, близкое к полумиллиону, однако, запустив эту программу, вы обнаружите, что

она выводит число, близкое к 666666. Две трети чисел, сгенерированных методом `random`, попадают в нижнюю половину диапазона!

Третий недостаток представленного метода `random` заключается в том, что он может, пусть и изредка, сбойть катастрофически — выдавать результат, выходящий за пределы указанного диапазона. Это происходит потому, что метод пытается преобразовать значение, возвращенное методом `rnd.nextInt()`, в неотрицательное целое число, используя для этого метод `Math.abs`. Если `nextInt()` возвращает `Integer.MIN_VALUE`, то `Math.abs` также возвращает `Integer.MIN_VALUE`, и оператор получения остатка от деления (`%`) в предположении, что `n` не является степенью двойки, возвращает отрицательное число. Это почти наверняка вызовет сбой в вашей программе, но воспроизвести его будет трудно.

Чтобы написать такую версию метода `random`, в которой были бы исправлены все три недостатка, необходимо неплохо разбираться в генераторах псевдослучайных чисел, теории чисел и арифметике в дополнительных кодах. К счастью, все это вам не нужно — все уже сделано до вас. Необходимый метод называется `Random.nextInt(int)`. Нет нужды разбираться в деталях, как работает этот метод (хотя, если вам это интересно, вы можете изучить документацию или исходный текст метода). Программист с подготовкой в области теории алгоритмов затратил немало времени на разработку, реализацию и тестирование этого метода, а затем показал его экспертам в данной области знаний, которые убедились в его правильности. После этого библиотека прошла стадию предварительного тестирования и была опубликована, и миллионы программистов активно использовали ее в течение пары десятилетий. Пока что ошибок в указанном методе найдено не было. Но если какой-либо дефект и обнаружится, он будет немедленно исправлен — в следующей же версии. **Вместе со стандартной библиотекой вы используете знания написавших ее экспертов, а также опыт тех, кто работал с ней до вас.**

Начиная с Java 7 вы больше не должны использовать `Random`. Для большинства применений следует выбирать генератор случайных чисел `ThreadLocalRandom`. Он генерирует высококачественные случайные числа и он очень быстр. На моей машине он быстрее `Random` в 3,6 раза. Для пулов и параллельных потоков используйте `SplittableRandom`.

Второе преимущество применения библиотек заключается в том, что не нужно терять время на написание решений для разовых задач, имеющих лишь косвенное отношение к вашей работе. Если вы принадлежите к большинству программистов, то вы должны тратить время на разработку своего приложения, а не на подготовку его фундамента.

Третье преимущество использования стандартных библиотек заключается в том, что их производительность имеет тенденцию со временем повышаться,

причем без каких-либо усилий с вашей стороны. Множество людей пользуется библиотеками; они применяются в стандартных промышленных тестах, поэтому организации, которые осуществляют поддержку библиотек, заинтересованы в том, чтобы заставить их работать быстрее. Многие библиотеки платформы Java переписывались годами, зачастую многократно, что привело к существенному увеличению их производительности.

Четвертое преимущество использования библиотек заключается в том, что со временем библиотеки приобретают новые функциональные возможности. Если в библиотеке чего-то не хватает, сообщество разработчиков даст знать об этом недостатке, и отсутствующая, но необходимая функциональность может быть добавлена в следующей же версии.

Последнее преимущество применения стандартных библиотек состоит в том, что ваш код соответствует господствующим в данный момент тенденциям. Такой код проще для чтения, поддержки и повторного использования множеством разработчиков.

С учетом всех упомянутых преимуществ казалось бы логичным применение программистом библиотек, а не собственных разработок, однако многие программисты этого не делают. Почему? Видимо, потому, что не знают обо всех возможностях имеющихся библиотек. **С каждой новой версией в библиотеки включается множество новых функций, и стоит быть в курсе всех новшеств.** При выпуске каждой новой версии Java в вебе публикуются страницы с описанием всех новых возможностей. Внимательно изучайте эти веб-страницы [20, 21]. Предположим, что вы хотите написать программу для вывода указанного в командной строке URL (грубо — то, что делает команда Linux `curl`). До Java 9 этот код был утомительно многословным, но в Java 9 к классу `InputStream` был добавлен метод `transferTo`. Вот полная программа для выполнения упомянутых действий с использованием этого нового метода:

```
// Вывод содержимого URL с использованием transferTo из Java 9
public static void main(String[] args) throws IOException
{
    try (InputStream in = new URL(args[0]).openStream())
    {
        in.transferTo(System.out);
    }
}
```

Библиотеки слишком велики для полного изучения документации [22], но **каждый программист должен знать основные возможности пакетов `java.lang`, `java.util` и `java.io`, а также их подпакетов.** Знания прочих библиотек могут добываться по мере необходимости. Даже краткое описание

функциональных возможностей библиотек (развивавшихся не одно десятилетие) выходит за рамки данного раздела.

Некоторые библиотеки требуют отдельного упоминания. Каркас коллекций и библиотека потоков (разделы 7.4–7.7) должны быть частью базового набора инструментов каждого программиста, как и возможности параллелизма, предоставляемые библиотекой `java.util.concurrent`. Этот пакет содержит как высокоуровневые утилиты для упрощения многопоточного программирования, так и низкоуровневые примитивы, позволяющие экспертам писать собственные высокоуровневые параллельные абстракции. Высокоуровневая часть `java.util.concurrent` рассматривается в разделах 11.3 и 11.4.

Иногда возможности библиотеки не отвечают вашим потребностям. И чем специфичнее ваши запросы, тем это вероятнее. При том что вашим первым побуждением должно быть применение библиотек, если вы изучили возможности, предлагаемые библиотеками в вашей области, и обнаружили, что они не соответствуют вашим потребностям, используйте альтернативную реализацию. В функциональности, предоставляемой любым конечным множеством библиотек, всегда найдутся пробелы. Если вы не можете найти то, что вам нужно, в библиотеках платформы Java, обратитесь к высококачественным библиотекам сторонних разработчиков, таким как, например, отличная библиотека с открытым исходным кодом Guava от Google [15]. И только если вы не в состоянии найти необходимую вам функциональность ни в одной библиотеке, у вас нет иного выбора, как реализовать ее самостоятельно.

Не изобретайте велосипед! Если вам нужно сделать нечто, что представляется вполне обычным, учтите, что в библиотеках может иметься класс, который это делает. Если он есть, используйте его; если вы не знаете, есть ли он, проверьте. В общем случае программный код в библиотеке наверняка окажется лучше кода, который вы напишете сами, а со временем он может стать еще лучше. Мы не ставим под сомнение ваши способности как программиста, однако библиотечному коду уделяется гораздо больше внимания, чем может позволить себе средний разработчик при реализации той же функциональности.

9.4. Если вам нужны точные ответы, избегайте `float` и `double`

Типы `float` и `double` предназначены, в первую очередь, для научных и инженерных расчетов. Они реализуют *бинарную арифметику с плавающей точкой* (binary floating-point arithmetic), тщательно спроектированную для быстрого получения правильного приближения для широкого диапазона значений.

Однако эти типы не дают точных результатов и не должны использоваться там, где требуются точные результаты. В частности, типы **float** и **double** не годятся для денежных расчетов, поскольку с их помощью невозможно точно представить число 0,1 (или любую иную отрицательную степень числа 10).

Предположим, например, что у вас в кармане лежат доллар и три цента и вы платите за что-то 42 цента. Сколько денег у вас остается? Вот фрагмент программы, написанной наивным программистом для ответа на данный вопрос:

```
System.out.println(1.03 - 0.42);
```

Увы, программа выводит 0.6100000000000001. И это не единственный случай. Предположим, что в вашем кармане доллар и вы покупаете 9 прокладок для крана по 10 центов. Какую сдачу вы получите?

```
System.out.println(1.00 - 9 * 0.10);
```

В соответствии с выводом данного фрагмента вы получите сдачу, равную 0.09999999999999998 доллара.

Вы можете подумать, что эту проблему можно решить, округляя результаты вычислений перед выводом. К сожалению, это не всегда срабатывает. Например, пусть у вас в кармане есть доллар и вы видите полку, на которой выстроены в ряд конфеты за 10, 20, 30 центов и далее — вплоть до доллара. Вы покупаете по одной конфете каждого вида, начиная с той, которая стоит 10 центов, затем — 20 центов и так далее, пока у вас есть возможность купить очередную конфету. Сколько конфет вы купите и какую получите сдачу? Решим эту задачу следующим образом:

```
// Неверно — применение типа с плавающей
// точкой для денежных вычислений!
public static void main(String[] args)
{
    double funds = 1.00;
    int itemsBought = 0;

    for (double price = 0.10; funds >= price; price += 0.10)
    {
        funds -= price;
        itemsBought++;
    }

    System.out.println("Куплено " + itemsBought + " штук.");
    System.out.println("Сдача: $" + funds);
}
```

Если вы выполните эту программу, то обнаружите, что можете купить только три конфеты и у вас останется 0.3999999999999999 доллара. Но это

неверный ответ! Правильный способ решения этой задачи — применение **BigDecimal**, **int** или **long** для финансовых вычислений.

Вот простое преобразование предыдущей программы для использования типа **BigDecimal** вместо **double**. Обратите внимание, что использован класс **BigDecimal**, принимающий параметр **String**, а не **double**. Этот требуется для предотвращения внесения неточных значений в вычисления [4, Puzzle 2]:

```
public static void main(String[] args)
{
    final BigDecimal TEN_CENTS = new BigDecimal(".10");
    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");

    for (BigDecimal price = TEN_CENTS;
         funds.compareTo(price) >= 0;
         price = price.add(TEN_CENTS))
    {
        funds = funds.subtract(price);
        itemsBought++;
    }

    System.out.println("Куплено " + itemsBought + " штук.");
    System.out.println("Сдача: $" + funds);
}
```

При выполнении преобразованной программы вы выясните, что можете купить четыре конфеты, истратив все имеющиеся у вас деньги. Это правильный ответ на поставленную задачу.

У использования **BigDecimal**, однако, имеется два недостатка: его менее удобно использовать, чем примитивный арифметический тип, и он существенно медленнее. Последнее не так важно при решении отдельной небольшой задачи, но неудобство применения может серьезно вас раздражать.

Альтернативой применения **BigDecimal** является использование **int** или **long** (в зависимости от величин, с которыми требуется работать) с самостоятельным отслеживанием десятичной точки. В данном примере очевидный подход состоит в выполнении вычислений в центах, а не в долларах. Вот как выглядит соответствующее изменение программы:

```
public static void main(String[] args)
{
    int itemsBought = 0;
    int funds = 100;

    for (int price = 10; funds >= price; price += 10)
    {
        funds -= price;
    }
}
```

```

        itemsBought++;
    }

    System.out.println("Куплено " + itemsBought + " штук.");
    System.out.println("Сдача: " + funds + " центов.");
}

```

Итак, для вычислений, требующих точного результата, не используйте типы с плавающей точкой (`float` и `double`). Если вы хотите, чтобы система сама отслеживала положение десятичной точки, и вас не пугают неудобства, связанные с отказом от примитивного типа, используйте `BigDecimal`. Применение этого класса имеет еще и то преимущество, что он дает вам полный контроль над округлением: для любой операции, завершающейся округлением, вы можете выбрать один из восьми режимов округления. Это может пригодиться при выполнении финансовых расчетов с жестко заданным алгоритмом округления. Если же для вас важна производительность и вас не пугает необходимость самостоятельно отслеживать положение десятичной точки, а обрабатываемые значения не слишком велики, используйте тип `int` или `long`. Если значения содержат не более девяти десятичных цифр, можно применять тип `int`; если же значения не превышают восемнадцати десятичных цифр, используйте тип `long`. Если же в значениях более восемнадцати цифр, вам придется работать с `BigDecimal`.

9.5. Предпочитайте примитивные типы упакованным примитивным типам

Система типов Java состоит из двух частей — *примитивов*, таких как `int`, `double` и `boolean`, и *ссылочных типов*, таких как `String` и `List`. Каждый примитивный тип имеет соответствующий ссылочный тип, именуемый *упакованным примитивным типом* (`boxed primitive`). Упакованными примитивными типами, соответствующими типам `int`, `double` и `boolean`, являются `Integer`, `Double` и `Boolean`.

Как упоминалось в разделе 2.6, автоматическая упаковка и распаковка размывают, но не стирают различия между примитивными и упакованными примитивными типами. Между ними имеются реальные различия, так что важно всегда помнить, с каким типом мы имеем дело, и делать осознанный выбор между ними.

Между примитивными и упакованными примитивными типами есть три основных различия. Во-первых, у примитивных типов есть только значения, в то время как у упакованных примитивных типов имеется еще и идентичность

(identities), отличная от их значений. Иными словами, два экземпляра упакованных примитивных типов могут иметь одинаковые значения, но разные идентичности. Во-вторых, примитивные типы имеют только полнофункциональные значения, в то время как каждый упакованный примитивный тип имеет одно нефункциональное значение — `null` — в дополнение ко всем функциональным значениям соответствующего примитивного типа. И наконец, примитивные типы более эффективны с точки зрения потребления памяти и времени работы, чем упакованные примитивные типы. Все эти различия могут привести к реальным проблемам при неосторожном использовании упакованных примитивных типов.

Рассмотрим следующий компаратор, созданный для представления в возрастающем порядке значений `Integer`. (Напомним, что метод `compare` компаратора возвращает число — отрицательное, положительное или ноль — в зависимости от того, является ли первый аргумент метода меньше, равным или больше второго аргумента.) На практике обычно писать компаратор не требуется, так как он реализует естественное упорядочение значений `Integer`, которое можно получить и без компаратора, но зато с компаратором мы получим интересный и поучительный пример:

```
// Плохой компаратор — можете ли вы указать его слабые места?
Comparator<Integer> naturalOrder =
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

Этот компаратор выглядит работоспособным и пройдет множество тестов. Например, его можно использовать с `Collections.sort` для корректной сортировки списка из миллиона элементов, независимо от наличия в нем одинаковых элементов. Но на самом деле этот компаратор глубоко ошибочен. Чтобы в этом убедиться, просто выведите значение `naturalOrder.compare(new Integer(42), new Integer(42))`. Оба экземпляра `Integer` представляют одно и то же значение (42), так что выводиться должно значение 0, но на самом деле выводится 1, что указывает, что первое значение `Integer` больше второго!

В чем же дело? Первая проверка в `naturalOrder` работает правильно. Вычисление выражения `i < j` заставляет экземпляры `Integer`, на которые ссылаются `i` и `j`, быть *автоматически распакованными*, т.е. из них извлекаются их примитивные значения. Вычисление продолжается и проверяет, является ли первое примитивное значение типа `int` меньше второго. Предположим, что это не так. Тогда следующая проверка вычисляет выражение `i == j`, которое выполняет *сравнение идентичностей* (identity comparison) двух ссылок на объекты. Если `i` и `j` ссылаются на различные экземпляры `Integer`, которые представляют одно и то же значение типа `int`, это сравнение вернет `false`, и

компаратор ошибочно вернет значение 1, указывающее, что первое значение `Integer` больше второго. **Применение оператора `==` к упакованным примитивным типам почти всегда ошибочно.**

На практике при необходимости компаратора для описания естественного упорядочения для типа следует просто вызвать `Comparator.naturalOrder()`, а если вы пишете компаратор самостоятельно, то должны использовать методы построения компараторов или статические методы сравнения примитивных типов (раздел 3.5). Нашу проблему можно исправить, добавляя две локальные переменные для хранения значений примитивного типа `int`, соответствующих упакованным типам `Integer`, и выполнять все сравнения над этими переменными. Это позволяет избежать ошибочного сравнения идентичностей:

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed; // Автораспаковка
    return i < j ? -1 : (i == j ? 0 : 1);
};
```

Теперь рассмотрим следующую небольшую программу:

```
public class Unbelievable
{
    static Integer i;
    public static void main(String[] args)
    {
        if (i == 42)
            System.out.println("Невероятно");
    }
}
```

Нет, она не выводит слово “Невероятно”, но делает почти столь же странные вещи. Она генерирует исключение `NullPointerException` при вычислении выражения `i==42`. Проблема в том, что `i` имеет тип `Integer`, а не `int`, и, подобно всем неконстантным полям-ссылкам, имеет начальное значение `null`. Когда программа вычисляет выражение `i==42`, она сравнивает `Integer` и `int`. Почти всегда при смешивании обычных и упакованных примитивных типов в одной операции упакованный примитивный тип автоматически распаковывается. При автоматической распаковке нулевого объекта генерируется исключение `NullPointerException`. Как показывает данная программа, такое может случиться где угодно. Исправить программу просто — как объявить, что `i` имеет тип `int`, а не `Integer`.

Наконец, рассмотрим программу из раздела 2.6:

```
// Очень медленная программа!  
// Можете ли вы найти, где создается объект?  
public static void main(String[] args)  
{  
    Long sum = 0L;  
    for (long i = 0; i < Integer.MAX_VALUE; i++)  
    {  
        sum += i;  
    }  
    System.out.println(sum);  
}
```

Эта программа намного медленнее, чем должна бы быть, потому что в ней локальная переменная `sum` случайно объявлена как имеющая упакованный примитивный тип `Long`, а не обычный примитивный тип `long`. Программа компилируется без ошибок или предупреждений, но переменная постоянно упаковывается и распаковывается, снижая производительность.

Во всех трех рассмотренных программах имеется одна и та же проблема: программист проигнорировал различия между обычными примитивными типами и упакованными примитивными типами, что и привело к неприятным последствиям. В первых двух случаях это привело к ошибкам времени выполнения, а в третьем — к серьезному снижению производительности.

Так когда же следует использовать упакованные примитивные типы? У них есть несколько корректных вариантов применения. Во-первых, в качестве элементов, ключей и значений коллекций. Вы не можете поместить обычные примитивные типы в коллекцию, поэтому вам придется использовать упакованные примитивные типы. Это частный случай более общего. Вы должны использовать упакованные примитивные типы в качестве параметров типа в параметризованных типах и методах (глава 5, “Обобщенное программирование”), потому что язык не позволяет вам использовать обычные примитивные типы. Например, вы не можете объявить переменную типа `ThreadLocal<int>`, а должны использовать `ThreadLocal<Integer>`. Наконец, вы должны использовать упакованные примитивные типы при вызове рефлексивных методов (раздел 9.9).

Резюме: используйте обычные примитивные типы вместо упакованных примитивных типов везде, где это возможно. Обычные примитивные типы проще и быстрее. Если вам приходится использовать упакованные примитивные типы, будьте осторожны! **Автоматическая упаковка уменьшает длину исходного текста, но не опасность использования упакованных примитивных типов.** Когда ваша программа сравнивает два упакованных примитивных типа с помощью оператора `==`, то происходит сравнение идентичностей, что, скорее всего, совсем не то, что вам нужно. Когда программа выполняет

вычисления, в которых одновременно задействованы и обычные, и упакованные примитивные типы, она выполняет распаковку, а когда ваша программа выполняет распаковку, она может сгенерировать исключение `NullPointerException`. Наконец, когда программа упаковывает примитивные значения, это может привести к дорогостоящему и ненужному созданию объектов.

9.6. Избегайте применения строк там, где уместнее другой тип

Тип `String` создавался для представления текста, и он прекрасно справляется с этой работой. Поскольку строки широко распространены и хорошо поддерживаются языком Java, возникает естественное желание использовать их для решения других задач, для которых они не предназначались. В этой статье обсуждается несколько операций, которые не следует проделывать со строками.

Строки — плохая замена другим типам значений. Когда данные попадают в программу из файла, сети или с клавиатуры, они зачастую имеют вид строки. Естественным является стремление оставить их в том же виде, однако это оправданно лишь тогда, когда данные по своей природе являются текстом. Если это числовые данные, они должны быть преобразованы в соответствующий числовой тип, такой как `int`, `float` или `BigInteger`. Ответ на вопрос “да/нет” следует преобразовать в значение типа `boolean`. В общем случае, если есть подходящий тип значения, примитивный или ссылочный, вы обязаны им воспользоваться. Если такового нет, вы должны его написать. Этот совет кажется очевидным, но его часто игнорируют.

Строки — плохая замена для перечислений. Как говорилось в разделе 6.1, перечисления гораздо лучше подходят в качестве констант перечислимых типов, чем строки.

Строки — плохая замена для агрегатных типов. Если некая сущность состоит из нескольких компонентов, то попытка представить ее одной строкой обычно представляет собой плохое решение. В качестве примера приведем строку кода из реальной системы (имена идентификаторов изменены, чтобы не выдавать виновника):

```
// Неподходящее применение строки в качестве агрегатного типа
String compoundKey = className + "#" + i.next();
```

Этот подход имеет множество недостатков. Если в одном из полей встретится символ, используемый для разделения полей, это может привести к беспорядку. Для получения доступа к отдельным полям следует выполнить анализ строки, а это медленная, трудоемкая и чреватая ошибками операция. Вы не

можете предоставить методы `equals`, `toString` и `compareTo`, а вынуждены принять поведение, предлагаемое классом `String`. Лучшим подходом является написание класса для представления агрегата; зачастую такой класс является закрытым статическим классом-членом (раздел 4.10).

Строки — плохая замена надежным (устойчивым к подделке) ключам. Иногда строки используются для обеспечения доступа к некоторой функциональности. Например, рассмотрим переменные, локальные по отношению к потоку. Этот механизм позволяет создавать переменные, которые в каждом потоке имеют собственное значение. Библиотеки Java поддерживают эту возможность начиная с версии 1.2, но до этого программистам приходилось выкручиваться самим. Столкнувшись с необходимостью реализации такого функционала, много лет назад несколько программистов независимо друг от друга пришли к одному и тому же решению, в котором идентификация каждой локальной по отношению к потоку переменной осуществляется через строку-ключ, предоставляемую клиентом:

```
// Неверное применение строки в качестве ключа!
public class ThreadLocal
{
    private ThreadLocal() { } // Неинстанцируемый

    // Устанавливает для именованной
    // переменной значение текущего потока.
    public static void set(String key, Object value);

    // Возвращает из именованной
    // переменной значение текущего потока.
    public static Object get(String key);
}
```

Проблема при таком подходе заключается в том, что эти строковые ключи совместно используют общее пространство имен локальных по отношению к потоку переменных. Чтобы такой подход работал, клиентские строки должны быть уникальными: если два независимых клиента, работающих с этим пакетом, решат использовать для своих переменных одно и то же имя, то, сами того не подозревая, они будут совместно использовать одну и ту же переменную, что в общем случае приведет к сбою работы обоих клиентов. Кроме того, нарушается безопасность: чтобы получить незаконный доступ к данным другого клиента, злоумышленник может намеренно воспользоваться тем же ключом, что и клиент.

API можно исправить, если эту строку заменить неподделяемым ключом¹:

¹ В английском языке используется термин *capability*. — *Примеч. ред.*

```
public class ThreadLocal
{
    private ThreadLocal() { } // Неинстанцируемый
    public static class Key // (Мандат)
    {
        Key() { }
    }
    // Генерирует уникальный, неподделываемый ключ
    public static Key getKey()
    {
        return new Key();
    }
    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

Хотя это решает обе проблемы с API на основе строк, можно сделать еще лучше. В действительности статические методы здесь не нужны. Наоборот, это могут быть методы самого ключа. При этом ключ больше не является ключом для локальной по отношению к потоку переменной: он сам *является* переменной, локальной по отношению к потоку. Теперь класс верхнего уровня больше ничего для вас не делает, так что от него можно избавиться и переименовать вложенный класс в ThreadLocal:

```
public final class ThreadLocal
{
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

Этот API не является безопасным с точки зрения типов, потому что вы должны выполнять приведение значения из Object к фактическому типу при его извлечении из локальной по отношению к потоку переменной. Невозможно сделать безопасным с точки зрения типов исходный API, так же как сложно сделать безопасным API на основе ключа, но очень просто сделать этот API безопасным, сделав класс ThreadLocal параметризованным (раздел 5.4):

```
public final class ThreadLocal<T>
{
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

Это, грубо говоря, и есть API, который предоставляет `java.lang.ThreadLocal`. Помимо того, что этот интерфейс разрешает проблемы API на основе строк, он быстрее и элегантнее любого API на основе ключей.

Таким образом, не поддавайтесь естественному желанию представить объект в виде строк, если для него имеется или может быть написан более подходящий тип данных. Если строки используются неправильно, они оказываются более громоздкими, менее гибкими, более медленными и подверженными ошибкам, чем другие типы. Как правило, строки ошибочно применяются вместо примитивных типов, перечислений и составных типов.

9.7. Помните о проблемах производительности при конкатенации строк

Оператор конкатенации строк (+) — удобное средство объединения нескольких строк в одну. Он превосходно справляется с генерацией отдельной строки для вывода или с созданием строкового представления для небольшого объекта фиксированного размера, но плохо масштабируется. **Время, которое необходимо оператору конкатенации для последовательного объединения n строк, пропорционально квадрату n .** Это печальное следствие того факта, что строки являются *неизменяемыми* (раздел 4.3). При конкатенации двух строк выполняется копирование содержимого обеих строк.

Например, рассмотрим метод, который создает строковое представление выписываемого счета, последовательно объединяя строки каждого пункта в счете:

```
// Некорректное применение конкатенации
// строка — низкая производительность!
public String statement()
{
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i); // Конкатенация строк
    return result;
}
```

При большом количестве пунктов метод работает очень медленно. **Чтобы добиться приемлемой производительности, используйте `StringBuilder` вместо `String` для хранения создаваемой строки:**

```
public String statement()
{
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
```

```

for (int i = 0; i < numItems(); i++)
    b.append(lineForItem(i));

return b.toString();
}

```

Со времени Java 6 была выполнена большая работа по ускорению конкатенации; тем не менее разница в производительности впечатляет. Если число `numItems` возвращает 100, а `lineForItem` возвращает 80-символьную строку, то на моей машине второй метод работает в 6,5 раз быстрее первого. Поскольку первый метод демонстрирует квадратичную зависимость от количества пунктов, а второй — линейную, разница в производительности при большем количестве пунктов будет расти. Заметим, что второй метод начинается с предварительного размещения в памяти объекта `StringBuilder`, достаточно большого, чтобы в нем полностью поместился результат вычислений, устраняя необходимость в автоматическом росте буфера. Даже если отказаться от этого и использовать `StringBuilder` с размером по умолчанию, он все равно будет работать в 5,5 раз быстрее, чем первый метод.

Мораль проста: **не пользуйтесь оператором конкатенации для объединения большого числа строк**, разве что только когда производительность не имеет для вас никакого значения. Лучше применять метод `append` из класса `StringBuilder`. В качестве альтернативы можно использовать символьный массив или обрабатывать строки по одной, не объединяя их.

9.8. Для ссылки на объекты используйте их интерфейсы

Раздел 8.3 гласит: в качестве типов параметров следует указывать интерфейсы, а не классы. В более общем случае, ссылаясь на объекты, вы должны отдавать предпочтение не классам, а интерфейсам. Если имеются подходящие типы интерфейсов, то параметры, возвращаемые значения, переменные и поля следует объявлять, используя типы интерфейсов. Единственный случай, когда вам нужно ссылаться на класс объекта, — при его создании с использованием конструктора. В качестве конкретного примера рассмотрим случай класса `LinkedHashSet`, который является реализацией интерфейса `Set`. Приобретите привычку писать так:

```
// Верно — использует в качестве типа интерфейс
Set<Son> sonSet = new LinkedHashSet<>();
```

а не так:

```
// Плохо — использует в качестве типа класс!
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

Если вы выработаете привычку использовать в качестве типов интерфейсы, ваша программа будет гораздо более гибкой. Если вы решите изменить реализацию, то все, что вы должны для этого сделать, — изменить имя класса в конструкторе (или использовать другую статическую фабрику). Например, первое объявление можно изменить следующим образом:

```
Set<Son> sonSet = new HashSet<>();
```

Весь окружающий код при этом продолжит работать. Окружающий код не был осведомлен о старом типе реализации, так что он не должен заметить внесенного изменения.

Однако нельзя упускать из виду следующее: если исходная реализация интерфейса предлагала некоторую особенную функциональность, не предусмотренную общим контрактом этого интерфейса, а код зависел от этой функциональности, крайне важно, чтобы новая реализация интерфейса обеспечивала ту же функциональность. Например, если окружающий первое объявление код зависел от стратегии упорядочения `LinkedHashSet`, то подставить в объявлении `HashSet` вместо `LinkedHashSet` будет некорректно, поскольку `HashSet` не дает гарантий относительно порядка итераций.

Тогда зачем менять тип реализации? А затем, что вторая реализация предлагает более высокую производительность либо обеспечивает необходимую дополнительную функциональность, которую не обеспечивает исходная реализация. В качестве примера предположим, что поле содержит экземпляр `HashMap`. Замена его `EnumMap` обеспечит лучшую производительность и порядок итераций, согласующийся с естественным порядком ключей, но вы можете использовать `EnumMap`, только если тип ключа является типом перечисления. Замена `HashMap` на `LinkedHashMap` обеспечит предсказуемый порядок итераций при производительности, сравнимой с производительностью `HashMap`, без каких-либо особых требований к типу ключа.

Вы можете решить, что ничего страшного в объявлении переменной с использованием ее типа реализации нет, поскольку вы можете изменить тип объявления и тип реализации одновременно, однако нет никакой гарантии, что такое изменение приведет к программе, которая будет компилироваться. Если код клиента использовал метод исходного типа реализации, который отсутствует в заменяемом его типе, или если код клиента передает экземпляр методу, который требует исходный тип реализации, то после внесения изменений код не будет компилироваться. Объявление переменной с типом интерфейса заставит вас быть честным.

Вполне допустимо ссылаться на объект с использованием класса, а не интерфейса, если подходящий интерфейс отсутствует. Рассмотрим, например, *классы значений*, такие как `String` и `BigInteger`. Классы значений

редко пишутся с расчетом на несколько реализаций. Часто они являются окончательными и редко имеют соответствующие интерфейсы. Совершенно корректно использовать такой класс значения в качестве параметра, переменной, поля или возвращаемого типа.

Второй случай, когда нет подходящего интерфейсного типа, связан с объектами, принадлежащими каркасу, фундаментальными типами которого являются классы, а не интерфейсы. Если объект принадлежит такому *каркасу, основанному на классах*, то для ссылки на него предпочтительнее применять *базовый класс*, зачастую являющийся абстрактным, а не класс реализации. В эту категорию попадают многие классы `java.io`, такие как `OutputStream`.

Последний случай отсутствия подходящего типа интерфейса — когда классы, реализующие интерфейс, предоставляют также дополнительные методы, которых нет в интерфейсе; например, `PriorityQueue` имеет метод `comparator`, который отсутствует в интерфейсе `Queue`. Такой класс должен использоваться для ссылки на свои экземпляры, *только* если программа использует эти дополнительные методы (и эти случаи должны быть очень редкими).

Три приведенных случая не являются исчерпывающим списком, а лишь дают представление о ситуациях, когда для ссылки на объект следует использовать его класс. На практике должно быть вполне очевидно, есть ли у данного объекта соответствующий интерфейс. Если есть, ваша программа будет более гибкой, если вы будете использовать его для ссылки на этот объект. **Если подходящего интерфейса нет, используйте наименее конкретный класс иерархии, обеспечивающий необходимую функциональность.**

9.9. Предпочитайте интерфейсы рефлексии

Ядро средств рефлексии, `java.lang.reflect`, предоставляет программный доступ к произвольным классам. Для данного объекта типа `Class` вы можете получить экземпляры `Constructor`, `Method` и `Field`, представляющие конструкторы, методы и поля класса, представленного экземпляром `Class`. Эти объекты предоставляют программный доступ к именам членов класса, типам полей, сигнатурам методов и т.д.

Кроме того, экземпляры `Constructor`, `Method` и `Field` позволяют *рефлексивно* манипулировать аналогами, лежащими в их основе: вы можете создавать экземпляры, вызывать методы и получать доступ к полям соответствующего класса, вызывая методы экземпляров `Constructor`, `Field` и `Method`. Например, `Method.invoke` позволяет вызвать любой метод любого объекта любого класса (при соблюдении обычных ограничений, связанных с безопасностью). Рефлексия позволяет одному классу использовать другой класс, даже

если последний во время компиляции первого еще не существовал. Однако такая мощь имеет свою цену.

- **Вы теряете все преимущества проверки типов времени компиляции, включая проверку исключений.** Если программа рефлексивно пытается вызвать несуществующий или недоступный метод, это приведет к сбою времени выполнения, если только не предпринять специальные меры предосторожности.
- **Код, необходимый для рефлексивного доступа к классам, неуклюжий и многословный.** Его утомительно писать и трудно читать.
- **Проблемы с производительностью.** Рефлексивный вызов метода гораздо медленнее обычного вызова метода. Насколько именно медленнее — трудно сказать, так как здесь в игру вступает множество факторов. На моей машине вызов метода без входных параметров, возвращающего значение типа `int`, оказывается в одиннадцать раз медленнее при рефлексивном вызове.

Имеется очень немного сложных приложений, которым необходима рефлексия. Их примеры включают инструментарий анализа кода, визуализаторы классов, инспекторы объектов и т.п. Но даже этот инструментарий со временем отказывается от использования рефлексии — по мере того, как недостатки ее применения становятся все яснее. Если у вас есть сомнения, действительно ли ваше приложение не в состоянии обойтись без рефлексии, вероятно, оно вполне может без нее обойтись.

Вы можете без особых затрат воспользоваться многими преимуществами рефлексии, применяя ее в очень ограниченном виде. Во многих программах, которым нужен класс, отсутствующий во время компиляции, для ссылки на него можно использовать подходящий интерфейс или суперкласс (раздел 9.8). Если это то, что вам нужно, можно создавать экземпляры рефлексивно, а затем обращаться к ним как обычно, используя их интерфейс или суперкласс.

Например, далее приведена программа, которая создает экземпляр `Set<String>`, класс которого задается первым аргументом командной строки. Остальные аргументы командной строки программа вставляет в созданное множество, а затем распечатывает его. Независимо от первого аргумента, программа выводит оставшиеся аргументы, удаляя дубликаты. Порядок вывода аргументов зависит от класса, указанного первым аргументом. Если вы укажете `java.util.HashSet`, аргументы будут выводиться в произвольном порядке, если же `java.util.TreeSet` — в алфавитном порядке, поскольку элементы в множестве `TreeSet` упорядочены.

```

// Рефлексивное инстанцирование с доступом через интерфейс
public static void main(String[] args)
{
    // Перевод имени класса в объект Class
    Class <? extends Set<String >> cl = null;

    try
    {
        cl = (Class <? extends Set<String >>) // Непроверяемое
            Class.forName(args[0]);         // приведение!
    }
    catch (ClassNotFoundException e)
    {
        fatalError("Класс не найден.");
    }

    // Получение конструктора
    Constructor <? extends Set<String >> cons = null;

    try
    {
        cons = cl.getDeclaredConstructor();
    }
    catch (NoSuchMethodException e)
    {
        fatalError("Нет конструктора без параметров");
    }

    // Инстанцирование множества
    Set<String> s = null;

    try
    {
        s = cons.newInstance();
    }
    catch (IllegalAccessException e)
    {
        fatalError("Конструктор недоступен");
    }
    catch (InstantiationException e)
    {
        fatalError("Класс не инстанцируем.");
    }
    catch (InvocationTargetException e)
    {
        fatalError("Конструктор генерирует " + e.getCause());
    }
    catch (ClassCastException e)

```

```

{
    fatalError("Класс реализует Set");
}

// Работа с множеством
s.addAll(Arrays.asList(args).subList(1, args.length));
System.out.println(s);
}
private static void fatalError(String msg)
{
    System.err.println(msg);
    System.exit(1);
}

```

Хотя эта программа — не более чем игрушка, продемонстрированная методика оказывается достаточно мощной. Эту игрушечную программу легко превратить в тестирующую программу для обобщенных множеств, которая проверяет правильность определенной реализации `Set`, активно работая с одним или несколькими экземплярами и проверяя, подчиняются ли они контракту `Set`. Точно так же ее можно превратить в инструментарий анализа производительности обобщенных множеств. Фактически эта технология достаточно мощная для реализации полнофункционального *каркаса провайдера службы* (*service provider framework*) (раздел 2.1). Обычно эта методика — все, что вам нужно от рефлексии.

Этот пример демонстрирует два недостатка рефлексии. Во-первых, пример может генерировать шесть различных исключений во время выполнения, которые, если бы мы не использовали рефлексию, были бы выявлены еще во время компиляции. (Кстати, забавно, что можно заставить программу сгенерировать все шесть исключений с помощью соответствующих аргументов командной строки.) Во-вторых, для генерации экземпляра класса по его имени потребовалось несколько десятков строк кода, в то время как вызов конструктора уложился бы ровно в одну строку. Уменьшить длину программы можно было бы путем перехвата `ReflectiveOperationException`, суперкласса для различных рефлексивных исключений, который появился в версии Java 7. Оба недостатка ограничиваются частью программы, которая инстанцирует объект. Будучи созданным, экземпляр множества не отличается от любого другого экземпляра `Set`. Благодаря этому значительная часть кода в реальном приложении не зависит от такого локального применения рефлексии.

Если вы скомпилируете эту программу, то получите предупреждение о непроверяемом приведении. Это вполне разумное предупреждение в том плане, что приведение к `Class<? extends Set<String>>` будет успешным, даже если именованный класс не является реализацией `Set`, и в этом случае программа сгенерирует исключение `ClassCastException` при инстанцировании

класса. О том, как подавить выдачу таких предупреждений, рассказывается в разделе 5.2.

Приемлемым, хотя и редким, применением рефлексии является управление зависимостями класса от других классов, методов и полей, которые могут отсутствовать во время выполнения. Это может оказаться полезным при написании пакета, который должен работать с различными версиями некоторого другого пакета. Идея заключается в том, чтобы скомпилировать ваш пакет с минимальными требованиями к окружению (обычно это поддержка самой старой версии), а доступ ко всем новым классам и методам осуществлять через рефлексия. Чтобы этот способ работал, необходимо предпринимать правильные действия, когда в ходе работы программы обнаружится, что тот или иной новый класс или метод, к которому вы пытаетесь обратиться, не существует. Соответствующие действия могут заключаться в использовании некоторых альтернативных средств, позволяющих достичь той же цели, или в работе с усеченными функциональными возможностями.

Таким образом, рефлексия представляет собой мощный инструмент, который необходим для решения некоторых сложных задач системного программирования, однако обладающий множеством недостатков. Если вы пишете программу, которая должна работать с классами, неизвестными во время компиляции, то по возможности используйте рефлексия только для инстанцирования объектов, а для доступа к созданным объектам используйте некоторый интерфейс или суперкласс, который известен во время компиляции.

9.10. Пользуйтесь машинно-зависимыми методами осторожно

Интерфейс Java Native Interface (JNI) позволяет программам Java вызывать *машинно-зависимые методы*² (native methods), которые представляют собой методы, написанные на языках программирования, дающих машинно-зависимый код (native programming languages), таких как C или C++. Исторически такие методы имели три основных применения. Они обеспечивали доступ к средствам платформы, таким, например, как реестр Windows. Они также обеспечивали доступ к существующим библиотекам с машинным кодом, включая устаревшие библиотеки, предоставляющие доступ к устаревшим данным. Наконец, эти методы использовались при написании критических в смысле

² В русскоязычной среде встречаются также переводы *native methods* как *родные* или *нативные методы*. — Примеч. пер.

производительности фрагментов приложений на соответствующих языках программирования.

Использовать машинно-зависимые методы для доступа к возможностям конкретной платформы вполне законно, но редко необходимо: по мере развития платформа Java предоставляет доступ ко многим функциям, которые ранее обеспечивались только в рамках принимающих платформ. Например, API процессов, добавленный в Java 9, обеспечивает доступ к процессам операционной системы. Вполне допустимо также применение таких методов при использовании машинно-зависимых библиотек, если эквивалентные библиотеки в Java недоступны.

Использовать машинно-зависимые методы для повышения производительности редко целесообразно. В ранних версиях (до Java 3) это было необходимо довольно часто, но виртуальная машина JVM стала с тех пор гораздо быстрее. Для большинства задач теперь можно получить вполне сопоставимые по производительности результаты в Java. Например, когда в версии 1.1 была добавлена библиотека `java.math`, класс `BigInteger` был основан на вспомогательной арифметической библиотеке, написанной на языке C. В Java 3 `BigInteger` перепроектирован на Java и тщательно настроен таким образом, что теперь он быстрее, чем исходная машинно-зависимая реализация.

Грустным в этой истории является то, что с тех пор в реализации `BigInteger` мало что изменилось, за исключением более быстрого умножения больших чисел в Java 8, при том что работы над машинно-зависимыми библиотеками в этой области ведутся постоянно, в особенности над библиотекой вычислений произвольной точности GNU (GMP). Программисты на Java, реально нуждающиеся в высокой производительности арифметических вычислений с произвольной точностью, в настоящее время вынуждены использовать GMP посредством машинно-зависимых методов [5].

Применение машинно-зависимых методов имеет *серьезные* недостатки. Поскольку такие методы *небезопасны* (раздел 8.2), использующие их приложения становятся восприимчивыми к ошибкам, связанным с памятью. Для каждой новой платформы машинно-зависимый программный код необходимо компилировать заново, что снижает его переносимость. Такие методы труднее отлаживать. При отсутствии должного опыта и внимательности применение этих методов может даже привести к *снижению* производительности из-за того, что сборщик мусора не может автоматизировать, или даже отслеживать, использование памяти такими методами (раздел 2.8), а также из-за накладных расходов на их вызов и возврат из них. Наконец, машинно-зависимые методы требуют дополнительного “склеивающего кода”, который сложно писать и трудно читать.

Поэтому подумайте дважды, прежде чем воспользоваться машинно-зависимыми методами. Они редко требуются для повышения производительности.

Если же вы вынуждены прибегать к ним для доступа к низкоуровневым ресурсам или машинно-зависимым библиотекам, старайтесь использовать как можно меньше машинно-зависимого кода и как можно тщательнее его тестируйте. Единственная ошибка в таком коде может испортить все приложение.

9.11. Оптимизируйте осторожно

Есть три афоризма, посвященных оптимизации, которые следует знать каждому программисту.

Во имя эффективности (без необходимости ее достижения) делается больше вычислительных грехов, чем по любой иной причине, включая непроходимую тупость.

— Уильям Вульф (William A. Wulf) [53]

Мы *должны* не вспоминать о мелких усовершенствованиях около 97% времени. Преждевременная оптимизация — корень всех зол.

— Дональд Кнут (Donald E. Knuth) [29]

Следуйте двум правилам оптимизации.

1. Не оптимизируйте.
2. (Только для экспертов.) Не оптимизируйте, пока не получите идеально ясное неоптимизированное решение.

— М.А. Джексон (M.A. Jackson) [18]

Эти афоризмы на два десятилетия опередили язык программирования Java. В них отражена сущая правда об оптимизации: легче причинить вред, чем сделать благо, особенно если вы взялись за оптимизацию преждевременно. В процессе оптимизации вы можете получить программный код, который не будет ни быстрым, ни правильным и который будет нелегко исправить.

Не жертвуйте надежными архитектурными принципами во имя производительности. **Старайтесь писать хорошие, а не быстрые программы.** Если хорошая программа работает недостаточно быстро, правильная архитектура позволит ее оптимизировать. Хорошие программы воплощают принцип *сокрытия информации* (information hiding): они по возможности локализуют проектные решения в отдельных модулях, а потому эти решения можно менять, не затрагивая остальные части системы (раздел 4.1).

Это *не* означает, что вы должны игнорировать проблемы производительности до тех пор, пока ваша программа не будет завершена. Проблемы реализации могут быть решены путем последующей оптимизации, однако глубинные архитектурные пороки, которые ограничивают производительность, практически невозможно устранить, не переписав заново всю систему. Изменение задним числом фундаментальных основ вашего проекта может породить систему с “больной” структурой, которую сложно сопровождать и развивать. Поэтому думать о производительности следует уже в процессе проектирования приложения.

Старайтесь избегать проектных решений, ограничивающих производительность. Труднее всего менять те компоненты, которые обеспечивают взаимодействие программы с окружающим миром. Главными среди таких компонентов являются API, протоколы физического уровня и форматы данных. Мало того что эти компоненты впоследствии сложно или невозможно менять, любой из них способен существенно ограничить производительность, которую можно получить от системы.

Рассмотрите влияние на производительность проектных решений, на которых основан ваш API. Изменяемый открытый тип может потребовать создания множества ненужных защитных копий (раздел 8.2). Аналогично использование наследования в открытом классе, в котором уместнее была бы композиция, навсегда привязывает класс к его суперклассу, что может искусственно ограничивать производительность подкласса (раздел 4.4). В качестве последнего примера: используя в API тип реализации, а не интерфейс, вы оказываетесь привязанными к определенной реализации, несмотря на то, что в будущем, возможно, будут написаны более быстрые реализации (раздел 9.8).

Влияние дизайна API на производительность действительно велико. Рассмотрим метод `getSize` класса `java.awt.Component`. Критичным для производительности является решение возвращать экземпляр `Dimension`, а также то, что экземпляры `Dimension` являются изменяемыми. Это приводит к тому, что любая реализация этого метода при каждом вызове создает новый экземпляр `Dimension`. И хотя в современных виртуальных машинах создание небольших объектов относительно дешево, излишнее создание миллионов объектов может нанести производительности приложения серьезный ущерб.

В данном случае имеется несколько альтернатив. В идеале класс `Dimension` должен стать неизменяемым (раздел 4.3). В качестве альтернативы метод `getSize` можно заменить двумя методами, возвращающими отдельные примитивные компоненты объекта `Dimension`. И действительно, с целью повышения производительности в версии Java 2 два таких метода были добавлены в класс `Component`. Однако уже существовавший к тому времени клиентский код продолжает использовать метод `getSize`, так что его производительность по-прежнему страдает от первоначального неверного проектного решения для API.

К счастью, в общем случае качество схемы API коррелирует с производительностью. **Не следует изменять и искажать API ради достижения высокой производительности.** Проблемы с производительностью, которые заставляют вас переделать API, могут исчезнуть в будущих версиях платформы или иного программного обеспечения, а вот искаженный API и связанная с ним головная боль останутся с вами навсегда.

После того как вы тщательно спроектировали программу и создали четкую, краткую и хорошо структурированную ее реализацию, — *только тогда* можно подумать об оптимизации, если, конечно, вы еще не удовлетворены производительностью программы.

Вспомните два правила Джексона. К ним можно добавить еще одно: **измеряйте производительность до и после каждой попытки оптимизации.** Вы можете быть удивлены тем, что обнаружите. Зачастую попытка оптимизации не оказывает измеримого влияния на производительность, а иногда она только ухудшает положение. Основная причина этого заключается в том, что очень трудно угадать, где ваша программа впустую тратит процессорное время. Часть программы, которую вы считаете медленной, может не быть виновата в низкой производительности, и в этом случае вы будете напрасно тратить время, пытаясь ее оптимизировать. Общее правило заключается в том, что программы тратят 90% времени работы, выполняя 10% кода.

Средства профилирования помогут вам определить, где именно следует сосредоточить усилия по оптимизации. Подобные инструменты предоставляют вам информацию о ходе выполнения программы, например сколько времени требуется для работы каждому методу и сколько раз он был вызван. Это позволит вам найти узкие места, которые нужно оптимизировать, и даже может указать вам на необходимость замены самого алгоритма. Если в вашей программе скрыт алгоритм с квадратичным (или еще худшим) временем работы, никакие улучшения проблему не решат. Вам придется заменить алгоритм более эффективным. Чем больше код программы, тем большее значение имеет профилирование. Это все равно что искать иголку в стоге сена: чем больше стог, тем больше пользы от металлоискателя. Инструмент, который заслуживает особого упоминания, — `jmh`, который является не профайлером, а *каркасом тонкой проверки производительности* (`microbenchmarking framework`), который предоставляет беспрецедентно подробную информацию о производительности Java-кода [26].

Необходимость выяснения влияния попыток оптимизации на производительность для платформы Java стоит острее, чем для традиционных языков программирования наподобие C или C++, потому что язык программирования Java имеет слабую *модель производительности* (`performance model`): относительная стоимость различных примитивных операций не определена. Разрыв между

тем, что пишет программист, и тем, что выполняется процессором, здесь гораздо значительнее, чем у традиционных компилируемых языков, и это сильно усложняет надежное предсказание того, как будет влиять на производительность та или иная оптимизация. Существует множество мифов о производительности, которые на поверку оказываются полуправдой, а то и просто ложью.

Модель производительности Java не только плохо определена, но и варьирует от реализации к реализации, от выпуска к выпуску и от процессора к процессору. Запуская свои программы с различными реализациями или на разных аппаратных платформах, важно измерить воздействие вашей оптимизации на каждой из них. Возможно, вам придется идти на компромиссы, чтобы учесть производительность при использовании различных реализаций или аппаратных платформ.

В течение почти двух десятилетий с того момента, как был впервые написан этот раздел, выросла сложность каждого компонента стека программного обеспечения Java (от процессоров до виртуальных машин и библиотек) и значительно увеличилось разнообразие оборудования, на котором работает Java. Все это делает производительность программ Java еще менее предсказуемой, чем она была в 2001 году, и вопрос измерений еще более важным.

Мораль такова: не следует стараться писать быстрые программы — старайтесь писать хорошие программы; скорость придет сама. Но не забывайте о производительности, когда проектируете систему, особенно когда проектируете API, протоколы физического уровня и форматы данных. Закончив построение системы, замерьте ее производительность. Если система достаточно быстрая, вы сделали свою работу. Если нет — найдите источник проблемы с помощью профайлера и приступайте к оптимизации соответствующих частей системы. Первый шаг заключается в изучении выбора алгоритмов: никакая низкоуровневая оптимизация не сможет компенсировать плохой выбор алгоритма. При необходимости повторяйте процесс, измеряя производительность после каждого внесенного изменения, пока результат вас не удовлетворит.

9.12. Придерживайтесь общепринятых соглашений по именованию

Платформа Java имеет хорошо устоявшийся набор *соглашений по именованию*, многие из которых содержатся в спецификации языка программирования Java [25, 6.1]. Можно сказать, что эти соглашения делятся на две группы — типографские и грамматические.

Имеется только небольшой набор типографских соглашений, касающихся выбора имен для пакетов, классов, интерфейсов и полей. Никогда не

нарушайте их, не имея на то веской причины. API, не соблюдающий эти соглашения, будет трудно использовать. Если реализация нарушает эти соглашения, ее будет сложно сопровождать. В обоих случаях нарушение соглашений может запутывать и раздражать программистов, работающих с вашим кодом, а также способствовать появлению ложных предположений, приводящих к ошибкам. Эти соглашения кратко изложены в данном разделе.

Имена пакетов и модулей должны представлять собой иерархию, отдельные части которой разделены точкой. Эти части должны состоять из строчных букв и изредка цифр. Название любого пакета, который будет использоваться за пределами организации, обязано начинаться с доменного имени вашей организации, с частями в обратном порядке, например `edu.cmu.com.google.org.eff`. Исключение из этого правила составляют стандартные библиотеки и необязательные пакеты, названия которых начинаются со слов `java` и `javax`. Пользователи не должны создавать пакетов с именами, начинающимися с `java` или `javax`. Детальное описание правил, касающихся преобразования имен доменов в префиксы названий пакетов, можно найти в спецификации языка программирования Java [25, 6.1].

Оставшаяся часть названия пакета должна состоять из одного или нескольких компонентов, описывающих этот пакет. Компоненты должны быть короткими, обычно не длиннее восьми символов. Поощряются выразительные сокращения, например `util` вместо `utilities`. Допустимы сокращения, например, `awt`. Такие компоненты, как правило, должны состоять из единственного слова или аббревиатуры.

Многие пакеты имеют имена, в которых, помимо названия домена, присутствует только один компонент. Дополнительные компоненты в имени пакета нужны лишь для больших систем, размер которых настолько велик, что требует создания неформальной иерархии. Например, в пакете `javax.util.concurrent.atomic`. Подобные пакеты часто называют *подпакетами*, хотя лингвистической поддержки иерархий пакетов не существует.

Имена классов и интерфейсов, включая имена перечислений и типов аннотаций, должны состоять из одного или нескольких слов, причем в каждом слове первая буква должна быть заглавной, например `List` или `FutureTask`. Необходимо избегать сокращений, за исключением некоторых общепринятых (таких, как `max` и `min`) и аббревиатур. Нет полного единодушия по поводу того, должны ли аббревиатуры писаться полностью прописными буквами или же заглавной у них должна быть только первая буква. Хотя некоторые программисты пишут все в верхнем регистре, есть один сильный аргумент в пользу только первой заглавной буквы — в этом случае всегда ясно, где заканчивается одно слово и начинается другое. Какое имя класса вы предпочли бы увидеть: `HTTPURL` или `HttpUrl`?

Имена методов и полей подчиняются тем же самым типографским соглашениям, что и имена классов и интерфейсов, за исключением того, что первая буква имени всегда должна быть строчной, например `remove` или `ensureCapacity`. Если первым словом в имени метода или поля оказывается аббревиатура, она вся пишется строчными буквами.

Единственное исключение из этого правила касается “константных полей” (`constant field`), имена которых должны состоять из одного или нескольких слов в верхнем регистре и разделенных символом подчеркивания, например `VALUES` или `NEGATIVE_INFINITY`. Константное поле — это поле, объявленное как `static final`, значение которого не меняется. Если такое поле имеет примитивный или неизменяемый ссылочный тип (раздел 4.3), то это константное поле. Например, константы перечислений являются константными полями. Если поле `static final` имеет изменяемый ссылочный тип, оно все еще может быть константным, если объект, на который оно ссылается, является неизменяемым. Обратите внимание, что константные поля — *единственное* место, где допустимо использование символа подчеркивания.

Имена локальных переменных подчиняются тем же типографским соглашениям, что и имена членов классов, но в них можно использовать аббревиатуры, отдельные символы, а также короткие последовательности символов, смысл которых зависит от контекста, в котором находятся локальные переменные, например `i`, `denom`, `houseNumber`. Входные параметры являются частным случаем локальных переменных. Они должны именоваться более аккуратно, чем обычные локальные переменные, так как их имена являются неотъемлемой частью документации метода.

Имена параметров типа обычно состоят из одной буквы. Чаще всего это один из следующих пяти вариантов: `T` — для произвольного типа, `E` — для типа элемента коллекции, `K` и `V` — для типов ключа и значения отображения и `X` — для исключения. Тип возвращаемого значения функции, как правило, именуется как `R`. Последовательность произвольных типов может быть `T`, `U`, `V` или `T1`, `T2`, `T3`.

В качестве быстрой справки в следующей таблице приведены примеры типографских соглашений.

Тип идентификатора	Пример
Пакет или модуль	<code>org.junit.jupiter.api, com.google.common.collect</code>
Класс или интерфейс	<code>Stream, FutureTask, LinkedHashMap, HttpClient</code>
Метод или поле	<code>remove, groupingBy, getCrc</code>
Константное поле	<code>MIN_VALUE, NEGATIVE_INFINITY</code>
Локальная переменная	<code>i, denom, houseNum</code>
Параметр типа	<code>T, E, K, V, X, R, U, V, T1, T2</code>

По сравнению с типографскими грамматические соглашения, касающиеся именованья, более гибкие и противоречивые. Для пакетов практически нет грамматических соглашений. Для именованья инстанцируемых классов, включая типы перечислений, обычно используются существительные в единственном числе или именные группы, например `Thread`, `PriorityQueue` или `ChessPiece`. Неинстанцируемые вспомогательные классы (раздел 2.4) часто именуется существительным во множественном числе, как, например, `Collectors` или `Collections`. Интерфейсы именуется так же, как классы, например `Collection` или `Comparator`, либо с окончаниями, образованными от прилагательных, — `able` или `ible`, например `Runnable`, `Iterable` или `Accessible`. Поскольку у типов аннотаций имеется много вариантов использования, здесь нет преобладания каких-либо частей речи. Существительные, глаголы, предлоги и прилагательные распространены одинаково, например `BindingAnnotation`, `Inject`, `ImplementedBy` или `Singleton`.

Для методов, выполняющих какое-либо действие, в качестве имени используются глаголы или глагольные конструкции, например `append` или `drawImage`. Для методов, возвращающих булево значение, обычно применяются имена, в которых сначала идет слово `is` (реже `has`), а потом существительное, именная группа или любое слово или фраза, играющее роль прилагательного, например `isDigit`, `isProbablePrime`, `isEmpty`, `isEnabled` или `hasSiblings`.

Для именованья методов, не связанных с булевыми операциями, а также методов, возвращающих атрибут объекта, для которого они были вызваны, обычно используется существительное, именная группа либо глагольная группа, начинающаяся с глагола `get`, например `size`, `hashCode` или `getTime`. Отдельные программисты требуют, чтобы применялась лишь третья форма (начинающаяся с `get`), но для подобных претензий нет никаких оснований. Первые две формы обычно делают текст программы более удобочитаемым, например:

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Следи за гаишниками!");
```

Форма, начинающаяся с `get`, происходит от в основном устаревшей спецификации *Java Beans*, которая сформировала базис архитектуры повторно используемых компонентов. Есть и современные инструменты, которые продолжают именоваться *Beans*, и вы вполне можете прибегнуть к ним в любом коде, который должен использоваться в сочетании с этим инструментарием. Имеются также серьезные основания для следования этому именованию, если класс содержит методы доступа для того же самого атрибута. В этом случае методы доступа обычно именуется `getAttribute` и `setAttribute`.

Несколько имен методов заслуживают отдельного упоминания. Методы экземпляров, которые преобразуют тип объекта и возвращают независимый объект другого типа, часто называются *toType*, например *toString* или *toArray*. Методы, которые возвращают *представление* (view, раздел 2.6), имеющее тип, отличный от типа объекта, обычно называются *asType*, например *asList*. Методы, возвращающие примитивный тип с тем же значением, что и у объекта, для которого они были вызваны, называются *typeValue*, например *intValue*. Распространенные имена для статических фабрик включают *from*, *of*, *valueOf*, *instance*, *getInstance*, *newInstance*, *getType* и *newType* (раздел 2.1).

Грамматические соглашения для названий полей формализованы в меньшей степени и не играют такой большой роли, как в случае с классами, интерфейсами и методами, поскольку хорошо спроектированный API если и предоставляет какие-либо поля, то очень редко. Поля типа `boolean` обычно именуется так же, как и логические методы доступа, но с опущенным префиксом *is*, например *initialized*, *composite*. Поля других типов, как правило, именуется с помощью существительных или именных групп, таких как *height*, *digits* или *bodyStyle*. Грамматические соглашения для локальных переменных аналогичны соглашениям для полей, но их соблюдение еще менее обязательно.

Изучите стандартные соглашения по именованию и доведите их использование до автоматизма. Типографские соглашения простые и практически однозначные; грамматические соглашения более сложные и менее строгие. Как сказано в спецификации [25, 6.1], “не следует рабски следовать соглашениям, если длительная практика диктует иное решение”. Руководствуйтесь здравым смыслом.

Исключения

При использовании надлежащим образом исключения могут повысить удобочитаемость исходного текста программ и их надежность, а также облегчить сопровождение. При некорректном применении можно получить обратный эффект. В этой главе приводятся рекомендации по эффективному использованию исключений.

10.1. Используйте исключения только в исключительных ситуациях

Однажды, если вам не повезет, вы можете сделать ошибку в коде программы, например такую:

```
// Неверное применение исключений. Никогда так не делайте!  
try {  
    int i = 0;  
    while(true)  
        range[i++].climb();  
} catch (ArrayIndexOutOfBoundsException e) {}
```

Что делает этот код? Его изучение не дает полной ясности, и это уже достаточная причина, чтобы им не пользоваться (раздел 9.11). Здесь приведена плохо продуманная идиома для циклического перебора элементов в массиве. Бесконечный цикл завершается генерацией исключения `ArrayIndexOutOfBoundsException` при обращении к первому элементу за пределами массива, после чего выполняется перехват с последующим игнорированием этого исключения. Предполагается, что этот подход эквивалентен стандартной идиоме цикла по массиву, которую узнает любой программист на Java:

```
for (Mountain m : range)  
    m.climb();
```

Так почему же кто-то выбрал идиому, использующую исключения, вместо другой — испытанной и правильной? Это вводящая в заблуждение попытка повысить производительность, исходящая из ложного умозаключения о том, что, поскольку виртуальная машина проверяет границы при всех обращениях к массиву, обычная проверка завершения цикла `for` (скрытая компилятором, но присутствующая в выполнении цикла) избыточна и ее следует устранить. В этом рассуждении неверны три момента.

- Поскольку исключения создавались для применения в исключительных условиях, лишь очень немногие реализации JVM делают их такими же быстрыми, как и явные тесты.
- Размещение кода внутри блока `try-catch` не дает возможности JVM выполнить ряд оптимизаций, которые в противном случае были бы ею выполнены.
- Стандартная идиома цикла по массиву вовсе не обязательно приводит к избыточным проверкам. В процессе оптимизации ряд реализаций JVM их выбрасывает.

Практически идиома, использующая исключения, работает гораздо медленнее стандартной идиомы. На моей машине идиома, использующая исключения, для массива из 100 элементов более чем в два раза медленнее стандартной.

Идиома цикла, использующая исключения, не только снижает производительность и делает код неудобочитаемым и непонятным — нет гарантии, что она вообще будет работать. При наличии ошибки в теле цикла применение исключений для управления потоком выполнения может замаскировать ошибку, тем самым значительно усложнив процесс отладки. Предположим, что вычисления в теле цикла вызывают метод, который приводит к выходу за границу при доступе к какому-либо другому массиву. При применении правильной идиомы цикла эта ошибка привела бы к необработанному исключению, которое вызвало бы немедленное завершение потока с трассировкой стека. В случае же идиомы с использованием исключения будет перехвачено исключение, вызванное ошибкой, и окажется неверно интерпретировано как обычное завершение цикла.

Мораль проста: **исключения, как и подразумевает их название, должны применяться лишь для исключительных ситуаций; для обычного управления потоком выполнения их не следует использовать никогда.** Вообще говоря, вы всегда должны предпочитать стандартные, легко распознаваемые идиомы всяческим ухищрениям для получения лучшей производительности. Даже если реальный выигрыш в производительности имеет место, он может быть сведен на нет постоянным совершенствованием реализаций платформ.

А вот тонкие ошибки и сложность сопровождения, вызываемые слишком хитроумными идиомами, наверняка останутся.

Этот принцип относится и к проектированию API. **Хорошо спроектированный API не должен заставлять своих клиентов использовать исключения для обычного управления потоком выполнения.** Если в классе есть метод, зависящий от состояния, который может быть вызван лишь при выполнении определенных непредсказуемых условий, то в этом же классе, как правило, должен присутствовать отдельный проверяющий состояние метод, который показывает, можно ли вызывать первый метод. Например, интерфейс `Iterator` имеет зависящий от состояния метод `next` и соответствующий метод проверки состояния `hasNext`. Это позволяет применять для обхода коллекции в цикле стандартную идиому с традиционным циклом `for` (или циклом по коллекции, который использует метод `hasNext` внутренне):

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext();)
{
    Foo foo = i.next();
    ...
}
```

Если у `Iterator` метод `hasNext` отсутствует, клиенты будут вынуждены прибегать к следующему способу:

```
// Не используйте эту мерзость для обхода коллекции!
try
{
    Iterator<Foo> i = collection.iterator();

    while (true)
    {
        Foo foo = i.next();
        ...
    }
}
catch (NoSuchElementException e) {}
```

Этот пример очень похож на пример обхода массива, приведенный в начале раздела. Кроме многословности и запутанности, цикл с использованием исключений обычно имеет худшую производительность и может маскировать ошибки, возникающие в других, не связанных с ним частях системы.

В качестве альтернативы отдельному методу проверки состояния можно использовать зависящий от состояния метод, который будет возвращать пустой `Optional` (раздел 8.7) или некоторое особое значение (такое, как `null`) в случае, когда необходимые вычисления не могут быть выполнены.

Приведем некоторые рекомендации, которые помогут вам сделать выбор между методом проверки состояния и методом с возвратом пустого `Optional` или особого значения. Если к объекту возможен параллельный доступ без внешней синхронизации или если смена его состояния возможна извне, может потребоваться прием с особым возвращаемым значением, поскольку состояние объекта может измениться между вызовом метода проверки состояния и вызовом метода, зависящего от состояния объекта. Пустой `Optional` или особое возвращаемое значение может потребоваться с точки зрения повышения производительности, если отдельный метод проверки состояния дублирует работу метода, зависящего от состояния объекта. Однако при прочих равных условиях метод проверки состояния предпочтительнее особого возвращаемого значения. При его использовании исходный текст более удобочитаем, а неправильное использование проще обнаруживать и исправлять: если вы забудете вызвать метод тестирования состояния, то зависимый от состояния метод приведет к генерации исключения, делая ошибку очевидной; если же вы забудете проверить специальное возвращаемое значение, ошибка может оказаться слишком тонкой (это не проблема для возвращаемых значений `Optional`).

Исключения созданы для использования в исключительных ситуациях. Не используйте их для обычного управления потоком выполнения и не пишите такие API, которые будут заставлять делать это других.

10.2. Используйте для восстановления проверяемые исключения, а для программных ошибок — исключения времени выполнения

В языке программирования Java предусмотрены три типа генерируемых объектов исключений: *проверяемые исключения* (`checked exception`), *исключения времени выполнения* (`run-time exception`) и *ошибки* (`error`). Программисты обычно путают, когда следует использовать каждый из этих типов. Решение не всегда очевидно, но есть несколько общих правил, в значительной мере упрощающих выбор.

Основное правило при выборе между проверяемым и непроверяемым исключениями гласит: **используйте проверяемые исключения для ситуаций, когда есть основания полагать, что вызывающий код способен выполнить восстановление.** Генерируя проверяемое исключение, вы заставляете вызывающий код обработать его в конструкции `catch` или передать дальше. Каждое проверяемое исключение, которое метод объявил как могущее быть сгенерированным, является, таким образом, серьезным указанием для пользователя API о том, что при вызове данного метода может возникнуть соответствующая ситуация.

Предоставляя пользователю проверяемое исключение, разработчик API передает ему право осуществить восстановление из соответствующей ситуации. Пользователь может пренебречь этим правом, перехватив исключение и проигнорировав его, но, как правило, это оказывается плохим решением (раздел 10.9).

Есть два типа непроверяемых генерируемых исключений: исключения времени выполнения и ошибки. Поведение у них одинаковое: оба типа не требуется (и, в общем случае, нельзя) перехватывать. Если программа генерирует непроверяемое исключение или ошибку, то, как правило, это означает, что восстановление невозможно и дальнейшее выполнение программы принесет больше вреда, чем пользы. Если программа не перехватывает такое исключение, его генерация вызовет остановку текущего потока с соответствующим сообщением об ошибке.

Используйте исключения времени выполнения для указания на программные ошибки. Подавляющее большинство исключений времени выполнения сообщает о *нарушении предусловий*. Нарушение предусловия означает, что клиент API не смог выполнить контракт, установленный спецификацией этого API. Например, в контракте о доступе к массиву указано, что индекс массива должен попадать в интервал от нуля до значения, на единицу меньшего длины массива. Исключение `ArrayIndexOutOfBoundsException` указывает, что это предусловие было нарушено.

С этим советом связана одна проблема — не всегда ясно, с чем вы имеете дело: с восстановимой ситуацией или с ошибкой программирования. Например, рассмотрим случай исчерпания ресурсов, которое может быть вызвано ошибкой программирования, такой как выделение неоправданно большого массива, или истинной нехваткой ресурсов. Если исчерпание ресурсов временное — ситуация поддается восстановлению. Решение о возможности восстановления принимается проектировщиком API. Если вы считаете, что ситуация, вероятно, поддается восстановлению, используйте проверяемое исключение; если нет — используйте исключение времени выполнения. Если не ясно, возможно ли восстановление, вероятно, лучше использовать непроверяемое исключение (по причинам, рассмотренным в разделе 10.3).

Хотя в спецификации языка Java это не требуется, существует строго соблюдаемое соглашение о том, что *ошибки* зарезервированы при использовании JVM для того, чтобы указывать на дефицит ресурсов, нарушение инвариантов и другие ситуации, делающие дальнейшее выполнение программы невозможным. Поскольку эти соглашения признаны практически повсеместно, лучше всего не создавать новых подклассов `Error`. Таким образом, **все реализуемые вами непроверяемые исключения должны прямо или косвенно наследовать класс `RuntimeException`**. Вы не только не должны определять

подклассы `Error`, но и, за исключением `AssertionError`, вы не должны их генерировать.

Для исключительной ситуации можно определить класс, который не является подклассом `Exception`, `RuntimeException` или `Error`. В спецификации языка Java это непосредственно не оговаривается, однако неявно подразумевается, что такие классы будут вести себя так же, как и обычные проверяемые исключения (которые являются подклассами `Exception`, но не `RuntimeException`). Когда же вы должны использовать такой класс? Если одним словом — никогда. Не имея никаких преимуществ перед обычным проверяемым исключением, он будет только запутывать пользователей вашего API.

Разработчики API часто забывают, что исключения — это полноценные объекты, для которых можно определять любые методы. Основное назначение таких методов — создание кода, который снабжает исключение дополнительной информацией о ситуации, вызвавшей генерацию данного исключения. Если таких методов нет, программистам придется разбираться со строковым представлением этого исключения, выуживая из него дополнительную информацию. Это крайне плохая практика (раздел 3.3). Классы редко указывают какие-либо детали в своем строковом представлении, да и само строковое представление может меняться от реализации к реализации, от версии к версии. Следовательно, программный код, который анализирует строковое представление исключения, скорее всего, окажется непереносимым и ненадежным.

Поскольку проверяемые исключения обычно указывают на ситуации, когда возможно восстановление и продолжение выполнения, для такого типа исключения важно создать методы, которые предоставляли бы клиенту информацию, помогающую возобновить работу. Предположим, что проверяемое исключение инициируется при неудачной попытке покупки с помощью подарочной карты из-за недостаточной суммы денег. Для этого исключения должен быть реализован метод доступа, который запрашивает недостающую сумму с тем, чтобы можно было сообщить о ней покупателю. (См. дополнительную информацию в разделе 10.7.)

Подытожим советы этого раздела. Генерируйте проверяемые исключения для восстанавливаемых ситуаций и непроверяемые исключения — для ошибок программирования. Если у вас возникают сомнения, генерируйте непроверяемые исключения. Не определяйте классы исключений, которые не являются ни проверяемыми, ни исключениями времени выполнения. Для ваших проверяемых исключений предоставляйте методы для оказания помощи в восстановлении.

10.3. Избегайте ненужных проверяемых исключений

Многим программистам Java не нравятся проверяемые исключения; однако при надлежащем применении они могут улучшить API и программы. В отличие от возвращаемых кодов и непроверяемых исключений они *заставляют* программиста работать с возникшей проблемой, что значительно повышает надежность приложения. Это означает, что злоупотребление проверяемыми исключениями может сделать API менее удобным для использования. Если метод генерирует одно или несколько проверяемых исключений, то в программном коде, который вызывает этот метод, должна присутствовать обработка этих исключений в виде одного или нескольких блоков `catch` либо должно быть объявлено, что код сам может генерировать такие исключения, и должна быть выполнена их передача дальше. В любом случае перед пользователем API стоит нелегкая задача, которая усложнилась в Java 8, так как методы, генерирующие проверяемые исключения, не могут быть использованы в потоках непосредственно (разделы 7.4–7.7).

Такое решение оправдано, если предотвратить возникновение исключительной ситуации невозможно даже при надлежащем применении API, и программист, пользующийся данным API, может предпринять некоторые полезные действия, столкнувшись с этим исключением. Если хоть одно из этих условий не выполняется, лучше пользоваться непроверяемым исключением. Роль лакмусовой бумажки в данном случае играет вопрос “Как программист будет обрабатывать исключение?” Является ли лучшим, что он может сделать, это:

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Не может быть!
}
```

Или это:

```
} catch (TheCheckedException e) {
    e.printStackTrace(); // Ладно, закончим работу.
    System.exit(1);
}
```

Если программист не может сделать ничего лучшего, используйте непроверяемое исключение.

Дополнительные заботы программиста, связанные с обработкой проверяемого исключения, значительно увеличиваются, если это *единственное* исключение, генерируемое данным методом. Если есть и другие исключения, метод должен уже находиться в блоке `try`, так что для конкретного исключения понадобится всего лишь еще один блок `catch`. Если же метод генерирует единственное проверяемое исключение, то будет требоваться, чтобы вызов

соответствующего метода был помещен в блок `try` и не был использован в потоках непосредственно. В таких условиях имеет смысл подумать, нет ли какого-либо способа избежать проверяемого исключения.

Самый простой способ устранить проверяемое исключение — использовать возврат в качестве возвращаемого типа `Optional` (раздел 8.7). Вместо генерации проверяемого исключения метод просто возвращает пустой `Optional`. Недостатком этого подхода является то, что метод не может вернуть дополнительную информацию, детализирующую его неспособность выполнить требуемые вычисления. Исключения, напротив, имеют описательные типы и могут экспортировать методы для предоставления дополнительной информации (раздел 10.2).

Можно также превратить проверяемое исключение в непроверяемое путем разбиения метода, генерирующего исключение, на два метода, первый из которых будет возвращать значение типа `boolean`, указывающее, должно ли генерироваться исключение. Таким образом, в результате преобразования API последовательность вызовов

```
// Применение проверяемого исключения
try
{
    obj.action(args);
}
catch (TheCheckedException e)
{
    ... // Обработка исключительного состояния
}
```

превращается в следующую:

```
// Применение метода проверки состояния и непроверяемого исключения
if (obj.actionPermitted(args))
{
    obj.action(args);
} else {
    ... // Обработка исключительного состояния
}
```

Такое преобразование возможно не всегда, но если оно приемлемо, то может сделать работу с API более удобной. Хотя второй вариант последовательности вызовов выглядит не лучше первого, этот вариант API более гибкий. Если программист знает, что вызов будет успешным, или согласен на завершение потока выполнения в случае неудачного вызова, новая версия API позволяет использовать следующую упрощенную последовательность вызовов:

```
obj.action(args);
```

Если вы предполагаете, что тривиальная последовательность вызовов будет нормой, то преобразование API может быть приемлемым. Полученный в результате этого преобразования API, в сущности, тот же, что и API с методом проверки состояния (раздел 10.1), а потому к нему относятся те же самые предупреждения: если к объекту одновременно без внешней синхронизации могут иметь доступ сразу несколько потоков или если смена его состояния возможна извне, указанное преобразование использовать не рекомендуется, поскольку в промежутке между вызовами `actionPermitted` и вызовом `action` состояние объекта может успеть измениться. Если отдельный метод `actionPermitted` может дублировать работу метода `action`, то от преобразования, вероятно, стоит отказаться по соображениям производительности.

Таким образом, при нечастом использовании проверяемые исключения могут повысить надежность программ. При злоупотреблении они делают использование API болезненным. Если вызывающий код не в состоянии восстановиться после сбоя, генерируйте непроверяемые исключения. Если восстановление возможно и вы хотите *заставить* пользователей обрабатывать исключительные ситуации, рассмотрите сначала возможность возврата объекта `Optional`. Только если этот способ не обеспечивает достаточного количества информации о происшедшем сбое, следует генерировать проверяемое исключение.

10.4. Предпочитайте использовать стандартные исключения

Одной из сильных сторон экспертов, отличающей их от менее опытных программистов, является то, что эксперты борются за высокую степень повторного использования программного кода (и обычно добиваются ее). Исключения, несмотря на название, не являются исключением из общего правила, гласящего, что повторно используемый код — это хорошо. В библиотеках Java имеется множество исключений, охватывающих большую часть потребностей большинства API в исключениях.

Повторное использование имеющихся исключений имеет ряд преимуществ. Главное среди них то, что они упрощают изучение и применение ваших API, поскольку соответствуют установленным соглашениям, с которыми хорошо знакомы программисты. С этим же связано второе преимущество, заключающееся в том, что программы, использующие ваш API, проще для чтения и понимания, поскольку в них нет незнакомых, сбивающих с толку исключений. Наконец, чем меньше классов исключений, тем меньше требуется места в памяти и времени на их загрузку.

Чаще всего используется исключение `IllegalArgumentException` (раздел 8.1). Обычно оно генерируется, когда вызываемому методу передается аргумент с неправильным значением. Например, `IllegalArgumentException` может генерироваться в случае, когда в качестве аргумента, указывающего количество повторов некоторого действия, передается отрицательное значение.

Другое часто используемое исключение — `IllegalStateException`. Обычно оно генерируется, когда вызов является некорректным из-за состояния объекта. Например, это исключение может генерироваться, когда делается попытка использовать некий объект до его надлежащей инициализации.

Возможно, каждый неправильный вызов метода сводится к неверным аргументам или некорректному состоянию, но для определенных типов неправильных аргументов и состояний стандартно используются другие исключения. Если вызывающий код передал в качестве аргумента `null`, тогда как значение `null` для этого параметра запрещено, то в этом случае в соответствии с соглашениями должно генерироваться исключение `NullPointerException`, а не `IllegalArgumentException`. Аналогично, если при вызове передано значение, выходящее за границы допустимого диапазона, должно генерироваться исключение `IndexOutOfBoundsException`, а не `IllegalArgumentException`.

Еще одно универсальное исключение, о котором необходимо знать, — `ConcurrentModificationException`. Оно должно генерироваться, когда объект, предназначенный для работы в одном потоке (или с внешней синхронизацией), обнаруживает изменение из параллельного потока. Это исключение является в лучшем случае намеком, потому что надежно определить параллельное внесение изменений невозможно.

Последнее универсальное исключение, заслуживающее упоминания, — `UnsupportedOperationException`. Оно генерируется, если объект не поддерживает запрошенную операцию. По сравнению с другими исключениями, рассмотренными ранее в этом разделе, `UnsupportedOperationException` применяется довольно редко, поскольку большинство объектов обеспечивает поддержку всех своих методов. Это исключение используется классами, которые не поддерживают одну или несколько *необязательных операций*, определяемых в реализуемом ими интерфейсе. Например, реализация интерфейса `List`, имеющая только функцию добавления элементов, будет инициировать это исключение при попытке удаления элемента.

Не используйте `Exception`, `RuntimeException`, `Throwable` и `Error` непосредственно. Рассматривайте эти классы как если бы они были абстрактными. Нельзя надежно протестировать эти исключения, поскольку они являются суперклассами для других исключений, которые может генерировать метод.

В приведенной далее таблице представлены наиболее распространенные из повторно используемых исключений.

Исключение	Причина применения
<code>IllegalArgumentException</code>	Неверное ненулевое значение параметра
<code>IllegalStateException</code>	Неверное состояние объекта для вызова метода
<code>NullPointerException</code>	Неразрешенное нулевое значение параметра
<code>IndexOutOfBoundsException</code>	Индексный параметр за границей допустимого диапазона
<code>ConcurrentModificationException</code>	Обнаружено запрещенное параллельное изменение объекта
<code>UnsupportedOperationException</code>	Объект не поддерживает метод

Помимо перечисленных исключений, при определенных обстоятельствах могут применяться и другие исключения. Например, при реализации таких арифметических объектов, как комплексные числа и матрицы, возможно использование исключений `ArithmeticException` и `NumberFormatException`. Если исключение отвечает вашим потребностям, пользуйтесь им, но только так, чтобы условия, при которых оно будет генерироваться, не вступали в противоречие с документацией к этому исключению. Выбирая исключение, следует исходить из его семантики, а не только из имени. Кроме того, вы можете создавать подклассы исключений, если хотите дополнить имеющиеся исключения информацией (раздел 10.7), но помните, что исключения сериализуемы (глава 12, “Сериализация”). Одного этого достаточно, чтобы не прибегать к написанию собственного класса исключений без достаточно уважительной причины.

Выбор, какое исключение повторно использовать, может быть сложным, поскольку причины применения из приведенной выше таблицы не являются взаимоисключающими. Рассмотрим, например, объект, представляющий колоду карт, и предположим, что у него есть метод, осуществляющий раздачу карт из колоды, причем в качестве аргумента ему передается количество запрошенных карт. Допустим, что при вызове в качестве параметра было передано значение, превышающее количество оставшихся в колоде карт. Эту ситуацию можно толковать и как `IllegalArgumentException` (значение параметра слишком велико), и как `IllegalStateException` (объект колоды содержит слишком мало карт для обработки запроса). В данном случае следует использовать такое правило: **генерируйте `IllegalStateException`, если не работает ни одно значение аргумента; в противном случае генерируйте `IllegalArgumentException`.**

10.5. Генерируйте исключения, соответствующие абстракции

Если метод инициирует исключение, не имеющее видимой связи с решаемой задачей, это сбивает с толку. Часто это происходит, когда метод передает дальше исключение, инициированное абстракцией более низкого уровня. Это не только приводит в замешательство программиста, но и засоряет API верхнего уровня деталями реализации. Если в следующей версии реализация верхнего уровня поменяется, то могут поменяться и инициируемые им исключения, в результате чего могут перестать работать имеющиеся клиентские программы.

Во избежание этой проблемы **верхние уровни приложения должны перехватывать исключения нижних уровней и, в свою очередь, генерировать исключения, которые можно пояснить в терминах абстракции верхнего уровня.** Эта идиома известна как *трансляция исключений* (exception translation):

```
// Трансляция исключений
try
{
    ... // Применение низкоуровневой абстракции
}
catch (LowerLevelException e)
{
    throw new HigherLevelException(...);
}
```

Приведем конкретный пример трансляции исключения, взятый из класса `AbstractSequentialList`, который представляет собой *скелетную реализацию* (раздел 4.6) интерфейса `List`. В этом примере трансляция исключения продиктована спецификацией метода `get` в интерфейсе `List<E>`:

```
/**
 * Возвращает элемент в указанной позиции списка.
 * @throws IndexOutOfBoundsException если index за пределами
 * диапазона ({@code index < 0 || index >= size()}).
 */
public E get(int index)
{
    ListIterator<E> i = listIterator(index);

    try
    {
        return i.next();
    }
}
```

```

catch (NoSuchElementException e)
{
    throw new IndexOutOfBoundsException("Index: " + index);
}
}

```

В тех случаях, когда исключение нижнего уровня может быть полезно для анализа ситуации, вызвавшей исключение, лучше использовать особый вид трансляции исключений, называемый *цепочкой (сцеплением) исключений* (exception chaining). При этом исключение нижнего уровня (в приведенном коде — `cause`) передается исключению верхнего уровня, которое, в свою очередь, предоставляет метод доступа (метод `getCause` в `Throwable`), позволяющий получить исключение нижнего уровня:

```

// Цепочка исключений
try
{
    ... // Применение низкоуровневой абстракции
}
catch (LowerLevelException cause)
{
    throw new HigherLevelException(cause);
}

```

Конструктор исключения верхнего уровня передает `cause` конструктору суперкласса, способному выполнять сцепление, так что оно в конечном итоге передается одному из конструкторов `Throwable`, умеющему работать со сцеплением, такому как `Throwable(Throwable)`:

```

// Исключение с конструктором, способным выполнять сцепление
class HigherLevelException extends Exception
{
    HigherLevelException(Throwable cause)
    {
        super(cause);
    }
}

```

У большинства стандартных исключений имеются такие конструкторы. Для исключений, у которых нет соответствующих конструкторов, можно воспользоваться методом `initCause` из `Throwable`. Цепочка исключений не только обеспечивает программный доступ к `cause` (с помощью метода `getCause`), но и интегрирует его трассировку стека вызовов в трассировку верхнего уровня.

Хотя трансляция исключений лучше бессмысленной передачи исключений нижнего уровня, злоупотреблять ею не следует. Где это возможно, наилучший способ обработки исключений нижнего уровня — полное исключение их возможности путем обеспечения гарантированной успешности

вызова метода нижнего уровня. Иногда этого можно добиться путем проверки корректности аргументов метода верхнего уровня перед их передачей на нижний уровень.

Если предупредить появление исключений на нижних уровнях невозможно, то лучшее решение состоит в том, чтобы верхний уровень молча обрабатывал эти исключения, изолируя клиента от проблем нижнего уровня. В таких условиях чаще всего достаточно протоколировать исключения, используя соответствующий механизм регистрации, такой, например, как `java.util.logging`. Это позволяет программисту изучить проблему и в то же время изолирует от нее клиентский код и пользователей.

Когда невозможно предотвратить или обработать исключения с нижних уровней, используйте трансляцию исключений (если только исключения методов нижнего уровня не оказываются гарантированно приемлемыми на высоком уровне). Цепочки исключений предоставляют лучшее из обоих миров: позволяют генерировать соответствующие высокоуровневые исключения и при этом захватывать низкоуровневые исключения для анализа сбоев (раздел 10.7).

10.6. Документируйте все исключения, которые может генерировать метод

Описание генерируемых методом исключений составляет важную часть документации, которая необходима для правильного применения метода. Поэтому крайне важно, чтобы вы уделите время тщательному описанию всех исключений, генерируемых каждым методом (раздел 8.8).

Всегда объявляйте проверяемые исключения индивидуально, и точно документируйте условия, при которых каждое из них генерируется, для чего используйте дескриптор Javadoc `@throws`. Не пытайтесь сократить описание, объявляя, что метод генерирует некий суперкласс исключений, вместо того чтобы описать несколько классов возможных исключений. Например, никогда не объявляйте, что метод генерирует `Exception` или, что еще хуже, `Throwable`. Помимо того, что такая формулировка не дает программисту никакой информации о том, какие исключения могут быть сгенерированы данным методом, она значительно затрудняет работу с методом, поскольку надежно скрывает любые другие исключения, которые могут быть сгенерированы в этом же контексте. Единственным исключением из этого совета является метод `main`, который может быть безопасно объявлен как генерирующий `Exception`, потому что он вызывается только виртуальной машиной.

Хотя язык Java не требует от программистов объявления непроверяемых исключений, которые могут генерироваться методом, имеет смысл

документировать их так же тщательно, как и проверяемые исключения. Непроверяемые исключения обычно представляют программные ошибки (раздел 10.2), так что ознакомление программиста с этими ошибками может помочь ему их избежать. Хорошо составленный список непроверяемых исключений, которые может генерировать метод, фактически описывает *предусловия* для его успешного выполнения. Важно, чтобы в документации к каждому открытому методу были описаны его предусловия (раздел 8.8), а документирование непроверяемых исключений как раз и является наилучшим способом удовлетворения этому требованию.

Особенно важна документация непроверяемых исключений, которые могут быть сгенерированы методами интерфейсов. Такая документация является частью *общего контракта* интерфейса и обеспечивает единообразное поведение различных его реализаций.

Используйте дескриптор Javadoc @throws для описания каждого исключения, которое может быть сгенерировано методом, но не используйте ключевое слово throws для непроверяемых исключений. Программист, пользующийся вашим API, должен знать, какие исключения проверяемые, а какие — нет, поскольку в первом и втором случаях на него возлагается различная ответственность. Документация, созданная с помощью дескриптора Javadoc @throws, без соответствующей конструкции в объявлении метода предоставляет программисту визуальный сигнал о том, что это исключение непроверяемое.

Следует отметить, что документирование всех непроверяемых исключений, которые могут быть сгенерированы каждым методом, — это не всегда достижимый в реальности идеал. Когда выполняется пересмотр класса и предоставляемый пользователю метод изменяется таким образом, что начинает генерировать новые непроверяемые исключения, это не является нарушением совместимости ни на уровне исходных текстов, ни на бинарном уровне. Предположим, что некоторый класс вызывает метод другого, независимо написанного класса. Авторы первого класса могут тщательно документировать все непроверяемые исключения, генерируемые каждым методом. Однако, если второй класс был изменен так, что теперь он генерирует дополнительные непроверяемые исключения, первый класс (не претерпевший изменений) будет передавать эти новые непроверяемые исключения дальше, хотя они для него и не документированы.

Если одно и то же исключение генерируется несколькими методами класса по одной и той же причине, его можно документировать в общей документации всего класса, а не описывать отдельно для каждого метода. Распространенным примером является исключение NullPointerException. В документации к классу можно написать что-то вроде “все методы этого класса генерируют исключение NullPointerException, если в качестве какого-либо параметра передается нулевая ссылка на объект”.

Итак, необходимо документировать все исключения, которые могут быть сгенерированы каждым из созданных вами методов. Это относится как к проверяемым, так и к непроверяемым исключениям, и как к абстрактным, так и к конкретным методам. Эта документация должна иметь вид дескрипторов `@throws` в документирующем комментарии. Объявляйте все проверяемые исключения отдельно в блоках `throws`, для непроверяемых исключений эту конструкцию использовать не следует. Если вы не будете документировать исключения, которые генерируют ваши методы, другим программистам будет сложно или даже невозможно использовать написанные вами классы и интерфейсы.

10.7. Включайте в сообщения информация о сбое

Когда выполнение программы аварийно завершается из-за неперехваченного исключения, система автоматически распечатывает трассировку стека (`stack trace`) для этого исключения. Трассировка стека содержит *строковое представление* данного исключения, результат вызова его метода `toString`. Обычно это представление состоит из названия класса исключения и *описания исключения*. Часто это единственная информация, с которой приходится иметь дело программистам, исследующим сбой программы. И если воспроизвести такой сбой нелегко, то получить какую-либо дополнительную информацию еще сложнее или вообще невозможно. Поэтому крайне важно, чтобы метод `toString` класса исключения возвращал как можно больше информации о причинах сбоя. Иными словами, строковое представление исключения должно сохранить информацию об отказе для последующего анализа.

Для сохранения информации о сбое строковое представление исключения должно содержать значения всех параметров и полей, приведших к генерации исключения. Например, подробное описание исключения `IndexOutOfBoundsException` должно содержать нижнюю границу, верхнюю границу и значение индекса, вышедшее за указанные границы. Такая информация говорит об отказе очень многое. Любое из трех значений (или все они) могут быть неправильным. Представленный индекс может оказаться на единицу меньше нижней границы или быть равным верхней границе либо иметь некоторое несуразное значение — как слишком маленькое, так и слишком большое. Нижняя граница может оказаться больше верхней (серьезная ошибка нарушения внутреннего инварианта). Каждая из указанных ситуаций указывает на отдельную проблему, и, если программист знает, какого рода ошибку следует искать, это в огромной степени облегчает поиск.

Одно предостережение, касающееся информации, критичной для безопасности: поскольку трассировка стека в процессе диагностики и устранения ошибок программного обеспечения видима многим людям, **не включайте пароли, ключи шифрования и прочую подобную информацию в детальные сообщения о сбоях.**

Хотя наличие в строковом представлении исключения всей относящейся к делу информации является критическим, в общем случае оно не должно быть пространным. Трассировка стека предназначается для анализа вместе с документацией и исходными текстами приложения и, как правило, содержит имя файла и номер строки, в которой сгенерировано исключение, а также файлы и номера строк, соответствующие остальным вызовам в стеке. Многословные пространные описания сбоя, как правило, излишни — необходимую информацию можно получить, читая документацию и исходный текст программы.

Не следует путать строковое представление исключения и сообщение об ошибке на пользовательском уровне, которое должно быть понятно конечным пользователям. В отличие от сообщения об ошибке для пользователя, описание исключения нужно главным образом специалистам для анализа причин сбоя. Поэтому содержащаяся в строковом представлении информация гораздо важнее его удобочитаемости. Кроме того, сообщения для пользователей обычно *локализованы*, в то время как локализация детальных сообщений об ошибке — явление редкое.

Один из способов гарантировать, что строковое представление исключения будет содержать информацию, достаточную для описания сбоя, состоит в запросе в конструкторах исключений не строки описания, а информации о сбое. Само описание исключения для представления этой информации может быть затем сгенерировано автоматически. Например, вместо конструктора с параметром `String` исключение `IndexOutOfBoundsException` может иметь конструктор следующего вида:

```
/**
 * Constructs an IndexOutOfBoundsException.
 *
 * @param lowerBound - наименьший разрешенный индекс
 * @param upperBound - наибольший разрешенный индекс
 * @param index - фактическое значение индекса
 */
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                int index)
{
    // Генерация детального сообщения о сбое
    super(String.format(
        "Lower bound: %d, Upper bound: %d, Index: %d",
        lowerBound, upperBound, index));
}
```

```
// Сохранение информации о сбое для программного доступа
this.lowerBound = lowerBound;
this.upperBound = upperBound;
this.index = index;
}
```

По состоянию на Java 9 исключение `IndexOutOfBoundsException` получило конструктор, принимающий значение параметра `index` типа `int`, но, увы, не параметры `lowerBound` и `upperBound`. Вообще говоря, эта идиома в библиотеках платформы Java используется не слишком активно, хотя и крайне рекомендуется к применению, так как позволяет программисту, генерирующему исключение, легко зафиксировать обстоятельства сбоя. Фактически эта идиома просто не позволяет программисту поступить иначе! Вместо того чтобы заставлять каждого пользующегося классом генерировать свое строковое представление, в этой идиоме собран весь код, необходимый для того, чтобы качественное строковое представление генерировал сам класс исключения.

Как говорилось в разделе 10.2, возможно, имеет смысл, чтобы исключение предоставляло методы доступа к информации об обстоятельствах сбоя (в представленном выше примере это `lowerBound`, `upperBound` и `index`). Наличие таких методов доступа для проверяемых исключений важнее, чем для непроверяемых, поскольку информация об обстоятельствах сбоя может быть полезна для восстановления работоспособности программы. Программный доступ к деталям непроверяемого исключения редко интересует программистов (хотя это и не исключено). Однако согласно общему принципу (раздел 3.3) такие методы доступа имеет смысл создавать даже для непроверяемых исключений.

10.8. Добивайтесь атомарности сбоев

После того как объект генерирует исключение, в общем случае желательно, чтобы он оставался во вполне определенном, пригодном для дальнейшего использования состоянии, даже если сбой произошел посреди выполнения операции. Особенно это касается проверяемых исключений, когда предполагается, что вызывающий код будет восстанавливать работоспособность программы. **Вообще говоря, вызов метода, завершившийся сбоем, должен оставлять проверяемый объект в том же состоянии, в каком тот был перед вызовом.** Метод, обладающий таким свойством, называют *атомарным по отношению к сбою* (*failure atomic*).

Добиться этого можно несколькими способами. Простейший способ заключается в разработке неизменяемых объектов (раздел 4.3). Если объект неизменяемый, получение атомарности не требует усилий. Если операция

заканчивается сбоем, это может помешать созданию нового объекта, но никогда не оставит уже имеющийся объект в несогласованном состоянии, поскольку состояние каждого неизменяемого объекта является согласованным в момент его создания и после этого уже не меняется.

Для методов, работающих с изменяемыми объектами, атомарность по отношению к сбою чаще всего достигается путем проверки правильности параметров перед выполнением операции (раздел 8.1). Благодаря этому любое исключение будет генерироваться до того, как начнется модификация объекта. В качестве примера рассмотрим метод `Stack.pop` из раздела 2.7:

```
public Object pop()
{
    if (size == 0)
        throw new EmptyStackException();

    Object result = elements[--size];
    elements[size] = null; // Удаление устаревшей ссылки
    return result;
}
```

Если убрать начальную проверку размера, метод все равно будет генерировать исключение при попытке получить элемент из пустого стека. Однако при этом он будет оставлять поле `size` в несогласованном (отрицательном) состоянии, приводя к тому, что сбоем будет завершаться вызов любого метода этого объекта. Кроме того, исключение `ArrayIndexOutOfBoundsException`, генерируемое методом `pop`, не будет соответствовать нужной абстракции (раздел 10.5).

Прием, тесно связанный с предыдущим и позволяющий добиться атомарности по отношению к сбоям, заключается в упорядочении вычислений таким образом, чтобы все части кода, способные повлечь сбой, предшествовали первой модификации объекта. Такой прием является естественным расширением предыдущего в случаях, когда невозможно произвести проверку аргументов, не выполнив хотя бы части вычислений. Рассмотрим, например, класс `TreeMap`, элементы которого сортируются согласно некоторому правилу. Для того чтобы в экземпляр `TreeMap` можно было добавить элемент, последний должен иметь такой тип, который допускал бы сравнение с помощью процедур, обеспечивающих упорядочение `TreeMap`. Попытка добавить элемент неправильного типа, естественно, закончится исключением `ClassCastException`, которое будет сгенерировано в процессе поиска элемента в дереве, но до того, как в этом дереве что-либо будет изменено.

Третий подход состоит в выполнении операции над временной копией объекта и замене содержимого объекта содержимым копии по завершении операции. Этот подход возникает естественным образом, когда вычисление может

выполняться более быстро после того, как данные сохранены во временной структуре данных. Например, некоторые функции сортировки перед сортировкой копируют свои входные списки в массив для снижения стоимости доступа к элементам во внутреннем цикле сортировки. Это делается для повышения производительности, но в качестве дополнительного преимущества гарантируется, что исходный список останется нетронутым, если произойдет сбой сортировки.

Последний, очень редко встречающийся прием заключается в написании специального *кода восстановления* (recovery code), который перехватывает сбой, возникающий в ходе выполнения операции, и заставляет объект вернуться в состояние, в котором он находился до начала операции. Этот прием используется главным образом для долговременных структур, располагающихся на диске.

К сожалению, при всей желаемости атомарности по отношению к отказам достичь ее можно не всегда. Например, если два потока одновременно, без должной синхронизации пытаются модифицировать некий объект, последний может оказаться в несогласованном состоянии. А потому после перехвата исключения `ConcurrentModificationException` нельзя полагаться на то, что объект все еще пригоден для использования. Ошибки невосстановимы, и потому не нужно даже пытаться сохранять атомарность в случае генерации `AssertionError`.

Даже там, где атомарность по отношению к сбоям возможна, она не всегда желательна. Для некоторых операций она существенно увеличивает стоимость или сложность вычислений. Вместе с тем очень часто это свойство достигается без особого труда, если хорошо разобраться в проблеме.

Резюме: как правило, любое исключение, являющееся частью спецификации метода, должно оставлять объект в том же состоянии, в котором он находился до вызова метода. В случае нарушения этого правила в документации API должно быть четко указано, в каком состоянии будет находиться объект. К сожалению, множество имеющейся документации API далеко от этого идеала.

10.9. Не игнорируйте исключения

Хотя этот совет кажется очевидным, он нарушается настолько часто, что заслуживает повторения. Когда разработчики API объявляют, что некоторый метод может генерировать исключение, они пытаются этим что-то вам сказать. Не игнорируйте сказанное! Игнорировать исключения легко: необходимо всего лишь окружить вызов метода конструкцией `try` с пустым блоком `catch`:

```
// Пустой блок catch игнорирует исключения - очень подозрительно!
try
{
    ...
}
catch (SomeException e) {}
```

Пустой блок catch лишает исключение смысла, который состоит в том, чтобы вы обработали исключительную ситуацию. Игнорировать исключение — все равно что игнорировать пожарную тревогу: выключить сирену, чтобы больше ни у кого не было возможности узнать, нет ли здесь настоящего пожара. Возможно, вам удастся всех обмануть, но результаты окажутся катастрофичными. Где бы вы ни увидели пустой блок `catch`, в вашей голове тут же должна включаться сирена.

Существуют ситуации, когда игнорирование исключений может оказаться целесообразным. Например, это может оказаться разумно при закрытии `FileInputStream`. Вы не изменили состояние файла, так что нет необходимости выполнять какие-либо действия по восстановлению, и вы уже прочли информацию из файла, так что нет причины прекращать текущую операцию. Но даже в этом случае разумно записать исключение в журнал, чтобы позже можно было изучить проблему, если такие исключения будут нередкими. **Если вы решаете игнорировать исключение, блок catch должен содержать комментарий, поясняющий, почему такое решение приемлемо, а переменная исключения должна иметь имя `ignored`:**

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // По умолчанию; гарантированно достаточно
                  // для любого отображения

try
{
    numColors = f.get(1L, TimeUnit.SECONDS);
}
catch (TimeoutException | ExecutionException ignored)
{
    // Используем значение по умолчанию: минимальная раскраска
    // желательна, но не является необходимой
}
```

Представленная в этом разделе рекомендация в равной степени относится как к проверяемым, так и к непроверяемым исключениям. Вне зависимости от того, представляет ли исключение предсказуемое условие или программную ошибку, если оно игнорируется и используется пустой блок `catch`, то в результате программа, столкнувшись с ошибкой, продолжит работу, никак на нее не реагируя. В таком случае в любой произвольный момент времени

программа может завершиться с ошибкой, и программный код, в котором это произойдет, не будет иметь никакого отношения к действительному источнику проблемы. Должным образом обработав исключение, вы можете избежать неприятностей. Даже простая передача необрабатываемого исключения далее вызовет по крайней мере быстрый останов программы, при котором будет сохранена информация, полезная для отладки сбоя.

Параллельные вычисления

Потоки позволяют выполнять несколько действий параллельно. Многопоточное программирование сложнее однопоточного, потому что больше вещей могут пойти не так, а воспроизвести сбой может быть довольно сложно. Но избежать его невозможно. Оно изначально присуще платформе Java; кроме того, оно требуется для получения высокой производительности при использовании многопроцессорных систем (которые в настоящее время широко распространены). В этой главе содержатся советы, которые помогут вам создавать понятные, правильные и хорошо документированные многопоточные программы.

11.1. Синхронизируйте доступ к совместно используемым изменяемым данным

Ключевое слово `synchronized` гарантирует, что в любой момент времени метод или блок будет выполняться только одним потоком. Многие программисты рассматривают синхронизацию исключительно как средство *взаимосключений* (*mutual exclusion*), которое не позволяет одному потоку наблюдать объект в несогласованном состоянии, пока его модифицирует другой поток. С этой точки зрения объект создается в согласованном состоянии (раздел 4.3), а затем блокируется методами, обращающимися к нему. Эти методы следят за состоянием объекта и (необязательно) могут вызывать *смену состояний* (*state transition*), преобразуя объект из одного согласованного состояния в другое. Правильное применение синхронизации гарантирует, что ни один метод никогда не сможет наблюдать этот объект в несогласованном, промежуточном состоянии.

Такая точка зрения верна, но не отражает всей картины. Без синхронизации изменения в одном потоке могут быть не видны в другом. Синхронизация не только не дает потоку возможности наблюдать объект в несогласованном состоянии, но также гарантирует, что каждый поток, входя в синхронизированный

метод или блок, будет видеть результаты выполнения всех предыдущих изменений, которые были защищены той же блокировкой.

Спецификация языка Java гарантирует, что чтение и запись переменной *атомарна*, если только это не переменная типа `long` или `double` [25, 17.4, 17.7]. Иными словами, гарантируется, что при чтении переменной (отличной от `long` и `double`) будет возвращаться значение, которое было записано в эту переменную одним из потоков, даже если новые значения в эту переменную одновременно без какой-либо синхронизации записывают несколько потоков.

Возможно, вы слышали, что для повышения производительности при чтении и записи атомарных данных нужно избегать синхронизации. Это опасный совет. Хотя свойство атомарности гарантирует, что при чтении поля поток не увидит случайного значения, нет гарантии, что значение, записанное одним потоком, будет видимо другим. **Синхронизация необходима как для взаимного исключения потоков, так и для надежного взаимодействия между ними.** Это является следствием той части спецификации языка программирования Java, которая называется *моделью памяти* (`memory model`) и указывает, когда и как изменения, сделанные одним потоком, становятся видимыми для других потоков [25, 17.4; 13, 16].

Отсутствие синхронизации для доступа к совместно используемым данным может иметь серьезные последствия, даже если данные читаются и записываются атомарно. Рассмотрим задачу остановки одного потока из другого. В библиотеках имеется метод `Thread.stop`, но он давно устарел и является *небезопасным* — работа с ним может привести к повреждению данных. **Не используйте `Thread.stop`.** Для остановки одного потока из другого рекомендуется использовать прием, который заключается в том, что первый поток опрашивает поле типа `boolean`, начальное значение которого — `false`, но для указания, что первый поток должен остановить сам себя, может быть установлено равным `true` вторым потоком. Поскольку чтение и запись поля типа `boolean` атомарны, у некоторых программистов появляется соблазн обращаться к нему без синхронизации:

```
// Неверно! Как вы думаете, сколько времени
// будет работать эта программа?
public class StopThread
{
    private static boolean stopRequested;
    public static void main(String[] args)
        throws InterruptedException
    {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
```

```
        while (!stopRequested)
            i++;
    });
    backgroundThread.start();
    TimeUnit.SECONDS.sleep(1);
    stopRequested = true;
}
}
```

Вы, возможно, ожидаете, что такая программа будет работать секунду, после чего основной поток поменяет значение `stopRequested` на `true`, что приведет к тому, что цикл фонового потока завершится. Однако на моей машине программа *никогда* не завершается: цикл фонового потока выполняется вечно!

Проблема в данном коде заключается в том, что в отсутствие синхронизации нет гарантированного времени, когда поток, подлежащий остановке, увидит (если вообще увидит!) измененное в основном потоке значение `stopRequested`. При отсутствии синхронизации виртуальная машина может вполне законно преобразовать код

```
while (!stopRequested)
    i++;
```

в следующий:

```
if (!stopRequested)
    while (true)
        i++;
```

Эта оптимизация называется поднятием (hoisting), и это именно то, что делает OpenJDK Server VM. Результат — *падение живучести* (liveness failure): программе не удастся продолжить успешно работать. Один из способов решить эту проблему — синхронизировать доступ к полю `stopRequested`. Приведенная далее программа работает, как и ожидается, около одной секунды:

```
// Завершение потока с использованием синхронизации
public class StopThread
{
    private static boolean stopRequested;
    private static synchronized void requestStop()
    {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested()
    {
        return stopRequested;
    }
}
```

```

public static void main(String[] args)
throws InterruptedException
{
    Thread backgroundThread = new Thread(() -> {
        int i = 0;

        while (!stopRequested())
            i++;
    });
    backgroundThread.start();
    TimeUnit.SECONDS.sleep(1);
    requestStop();
}
}

```

Обратите внимание, что и метод записи (`requestStop`), и метод чтения (`stopRequested`) синхронизированы. Синхронизировать один только метод записи *недостаточно!* Работа синхронизации не гарантируется, если не синхронизированы обе операции — и чтения, и записи. Иногда программа, которая синхронизирует только записи (или чтения), может *казаться* работающей на некоторых машинах, но это случай обманчивой внешности.

Действия синхронизированных методов в `StopThread` были бы атомарными даже без синхронизации. Другими словами, синхронизация этих методов используется *исключительно* для коммуникации, а не для взаимного исключения. Хотя затраты на синхронизацию каждой итерации цикла невелики, есть корректная, менее многословная альтернатива с более высокой производительностью. Блокировку во второй версии `StopThread` можно опустить, если `stopRequested` объявить как `volatile`. Хотя модификатор `volatile` не обеспечивает взаимоисключения, он гарантирует, что любой поток, который читает поле, увидит последнее записанное в поле значение:

```

// Завершение потока с помощью volatile-поля
public class StopThread
{
    private static volatile boolean stopRequested;
    public static void main(String[] args)
    throws InterruptedException
    {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;

            while (!stopRequested)
                i++;
        });
        backgroundThread.start();
        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

Использовать поле `volatile` нужно с осторожностью. Рассмотрим следующий метод, который должен генерировать серийный номер:

```
// Неверно — требуется синхронизация!  
private static volatile int nextSerialNumber = 0;  
  
public static int generateSerialNumber()  
{  
    return nextSerialNumber++;  
}
```

Предназначение этого метода — гарантировать, что при каждом вызове будет возвращаться уникальное значение номера (пока не будет выполнено 2^{32} вызова). Состояние метода включает лишь одно атомарно доступное поле (`nextSerialNumber`), и все его значения являются допустимыми. Следовательно, для защиты инвариантов данного генератора синхронизация не нужна. Тем не менее без синхронизации этот метод не работает корректно.

Проблема в том, что оператор инкремента (`++`) атомарным не является. Он выполняет *две* операции над полем `nextSerialNumber`: сначала читает его значение, потом записывает новое значение, равное увеличенному на один старому. Если второй поток читает значение между чтением первым потоком старого и записью нового значения, то второй поток будет видеть те же значения, что и первый, и возвращать тот же серийный номер. Это *сбой безопасности* (*safety failure*): программа вычисляет неверные результаты.

Одним из способов исправления метода `generateSerialNumber` является простое добавление в его объявление модификатора `synchronized`. Это гарантирует, что различные вызовы не будут перемешиваться и каждый новый вызов будет видеть результат работы всех предыдущих вызовов. После того как вы сделаете это, вам необходимо удалить модификатор `volatile` у `nextSerialNumber`. Чтобы сделать этот метод “пуленепробиваемым”, возможно, имеет смысл заменить `int` на `long` или генерировать какое-либо исключение, когда поле `nextSerialNumber` будет близко к переполнению.

Все же лучше последовать совету из раздела 9.3 и использовать класс `AtomicLong`, являющийся частью `java.concurrent.atomic`. Этот пакет предоставляет примитивы для безопасного с точки зрения параллельности, свободного от блокировок программирования отдельных переменных. В то время как `volatile` обеспечивает только коммуникационный эффект синхронизации, этот пакет предоставляет также атомарность. Это именно то, что нам нужно для `generateSerialNumber`, и, скорее всего, его производительность будет выше, чем у синхронизированной версии:

```
// Синхронизация без блокировок из java.util.concurrent.atomic  
private static final AtomicLong nextSerialNum = new AtomicLong();  
public static long generateSerialNumber()
```

```
{
    return nextSerialNum.getAndIncrement();
}
```

Наилучший способ избежать проблем, описанных в этом разделе, — не использовать совместно изменяемые данные. Либо совместно используйте неизменяемые данные (раздел 4.3), либо вообще не используйте совместно никакие данные. Другими словами, **ограничивайте изменяемые данные одним потоком**. Если вы примете данную стратегию, документируйте ее, чтобы она использовалась и при дальнейшем развитии программы. Необходимо также глубокое понимание каркасов и библиотек, которыми вы пользуетесь, так как они могут использовать потоки без вашего ведома.

Вполне допустимо, чтобы один поток изменял объект данных на некоторое время, а затем совместно использовал его с другими потоками, синхронизируя только предоставление для общего использования ссылки на объект. Тогда другие потоки смогут читать объект без дальнейшей синхронизации, пока он не будет изменен снова. Такие объекты называются *фактически неизменяемыми* (effectively immutable) [13, 3.5.4]. Передача такой ссылки на объект из одного потока другому называется *безопасной публикацией* (safe publication) [13, 3.5.3]. Существует много способов безопасной публикации ссылки на объект: вы можете хранить ее в статическом поле как часть инициализации класса, в поле `volatile`, поле `final` или в поле, доступ к которому обеспечивается с помощью обычной блокировки; можно также поместить ее в параллельную коллекцию (раздел 11.4).

Таким образом, когда несколько потоков совместно используют изменяемые данные, каждый поток, который читает или записывает эти данные, должен выполнять синхронизацию. Без синхронизации невозможно гарантировать, что изменения в объекте, сделанные одним потоком, будут видны для другого потока. Несинхронизированный доступ к данным может привести к отказам, затрагивающим живучесть и безопасность системы. Отлаживать такие отказы крайне сложно. Они могут зависеть от времени, а поведение программы может радикально меняться от одной виртуальной машины к другой. Для только межпоточных коммуникаций подходящей формой синхронизации является модификатор `volatile`, но использовать его корректно может оказаться непростой задачей.

11.2. Избегайте излишней синхронизации

Раздел 11.1 предупреждает вас об опасностях недостаточной синхронизации. Данный раздел посвящен обратной проблеме. В зависимости от ситуации

излишняя синхронизация может приводить к снижению производительности приложения, взаимной блокировке потоков и даже к непредсказуемому поведению программы.

Для исключения проблем с живучестью и безопасностью никогда не передавайте управление клиенту из синхронизированного метода или блока. Иными словами, из синхронизируемой области не следует вызывать методы, которые предназначены для перекрытия, или метод, предоставленный клиентом в форме функционального объекта (раздел 4.10). С точки зрения класса, содержащего синхронизируемую область, такой метод является *чужим* (alien). У класса нет сведений о том, что этот метод делает, и нет никакого контроля над ним. В зависимости от действий чужого метода его вызов из синхронизируемой области может привести к исключениям, взаимным блокировкам или повреждению данных.

Рассмотрим класс, который реализует оболочку *наблюдателя* множества. Он позволяет клиентам подписываться на уведомления о добавлении элементов в множество. Это реализация проектного шаблона *Наблюдатель* (Observer) [12]. Для краткости класс не обеспечивает уведомления об удалении элементов из множества, но настроить такие уведомления самостоятельно не составит труда. Этот класс реализован поверх `ForwardingSet` из раздела 4.4:

// Неверно — вызов чужого метода из синхронизируемого блока!

```
public class ObservableSet<E> extends ForwardingSet<E>
{
    public ObservableSet(Set<E> set)
    {
        super(set);
    }
    private final List<SetObserver<E>> observers
        = new ArrayList<>();
    public void addObserver(SetObserver<E> observer)
    {
        synchronized(observers)
        {
            observers.add(observer);
        }
    }
    public boolean removeObserver(SetObserver<E> observer)
    {
        synchronized(observers)
        {
            return observers.remove(observer);
        }
    }
    private void notifyElementAdded(E element)
    {
        synchronized(observers)

```

```

        {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }
    @Override public boolean add(E element)
    {
        boolean added = super.add(element);

        if (added)
            notifyElementAdded(element);

        return added;
    }
    @Override public boolean addAll(Collection <? extends E > c)
    {
        boolean result = false;

        for (E element : c)
            result |= add(element); // Вызов notifyElementAdded

        return result;
    }
}

```

Наблюдатели подписываются на уведомления, вызывая метод `addObserver`, и отказываются от подписки с помощью метода `removeObserver`. В обоих случаях методу передается экземпляр интерфейса *обратного вызова* (callback):

```

@FunctionalInterface public interface SetObserver<E>
{
    // Вызывается при добавлении элемента к множеству
    void added(ObservableSet<E> set, E element);
}

```

Этот интерфейс является структурно идентичным `BiConsumer<ObservableSet<E>, E>`. Мы решили определить пользовательский функциональный интерфейс как потому, что имена интерфейса и метода делают код более удобочитаемым, так и потому, что интерфейс может развиваться и включать множественные обратные вызовы. Все это — аргументы в пользу применения `BiConsumer` (раздел 7.3).

При беглом взгляде `ObserverSet` работает. Например, следующая программа выводит числа от 0 до 99:

```

public static void main(String[] args)
{
    ObservableSet<Integer> set =

```

```

    new ObservableSet<>(new HashSet<>());
    set.addObserver((s, e) -> System.out.println(e));

    for (int i = 0; i < 100; i++)
        set.add(i);
}

```

Теперь попробуем нечто более необычное. Предположим, мы заменили вызов `addObserver` другим методом, который выводит добавленное в множество значение `Integer` и выполняет отказ от подписки, если значение равно 23:

```

set.addObserver(new SetObserver<>() {
    public void added(ObservableSet<Integer> s, Integer e)
    {
        System.out.println(e);

        if (e == 23)
            s.removeObserver(this);
    }
});

```

Обратите внимание, что этот вызов использует экземпляр анонимного класса вместо лямбда-выражения, примененного в предыдущем вызове. Дело в том, что функциональный объект должен передать себя в `s.removeObserver`, а лямбда-выражение не может получить доступ к самому себе (раздел 7.1).

Вы можете ожидать, что программа напечатает числа от 0 до 23, после чего наблюдатель отказывается от подписки и программа завершает работу. Но на самом деле она выводит значения от 0 до 23, после чего генерирует исключение `ConcurrentModificationException`. Проблема в том, что `notifyElementAdded` находится в процессе итерации по списку `observers`, когда вызывается метод наблюдателя `added`. Метод `added` вызывает метод `removeObserver` множества, который, в свою очередь, вызывает `observer.remove`. И теперь у нас проблема. Мы пытаемся удалить элемент из списка во время обхода последнего, что является некорректным действием. Итерация в методе `notifyElementAdded` находится в синхронизируемом блоке, чтобы воспрепятствовать параллельному изменению множества, но не препятствует обратному вызову наблюдаемого множества и изменению списка `observers`.

Вновь попробуем нечто странное: напишем попытку наблюдателя выполнить отказ от подписки, но вместо непосредственного вызова `removeObserver` воспользуемся для этого услугами другого потока. Этот наблюдатель использует *службу выполнения* (`executor service`) (раздел 11.3).

```

// Наблюдатель, использующий фоновый поток без необходимости
set.addObserver(new SetObserver<>()

```

```

{
    public void added(ObservableSet<Integer> s, Integer e)
    {
        System.out.println(e);

        if (e == 23)
        {
            ExecutorService exec =
                Executors.newSingleThreadExecutor();

            try
            {
                exec.submit(() -> s.removeObserver(this)).get();
            }
            catch (ExecutionException | InterruptedException ex)
            {
                throw new AssertionError(ex);
            }

            finally
            {
                exec.shutdown();
            }
        }
    }
});

```

Кстати, обратите внимание, что эта программа перехватывает два различных типа исключений в одной конструкции `catch`. Эта возможность, неофициально известная как *мультиперехват* (*multi-catch*), была добавлена в версии языка Java 7. Она может значительно повысить ясность и уменьшить размер программы, которая одинаково ведет себя в ответ на несколько типов исключений.

При выполнении этой программы исключение не генерируется — мы получаем взаимоблокировку. Фоновый поток вызывает `s.removeObserver`, который пытается заблокировать `observers`, но не может захватить блокировку, потому что основной поток уже ее захватил. В то же время основной поток ждет, когда фоновый поток завершит удаление наблюдателя, чем и объясняется блокировка.

Этот пример надуманный, так как у наблюдателя нет причин для использования фонового потока, но проблема реальна. Запуск чужого метода из синхронизируемой области вызывает много взаимоблокировок в реальных системах, таких как инструментариум GUI.

В обоих предыдущих примерах (исключение и взаимоблокировка) нам повезло. Ресурсы, которые защищены синхронизируемой областью (`observers`), находились в согласованном состоянии в момент запуска чужого метода

(added). Предположим, что чужой метод был вызван из синхронизируемой области в момент, когда инвариант, защищаемый синхронизируемой областью, был временно нарушен. Поскольку блокировки в языке Java являются *реентерабельными* (reentrant), такие вызовы не ведут к взаимоблокировке. Как и в первом примере, который завершился исключением, вызываемый поток уже захватил блокировку, так что поток при попытке повторно захватить блокировку завершится успешно, даже если над данными, защищенными блокировкой, будет выполняться другая, концептуально несвязанная операция. Последствия такого сбоя могут быть катастрофическими. По сути, блокировка не в состоянии выполнить свою работу. Реентерабельные блокировки позволяют упростить создание многопоточных объектно-ориентированных программ, но могут превратить сбой живучести в сбой безопасности.

К счастью, решить такую проблему обычно не слишком сложно — путем перемещения вызова чужого метода за пределы синхронизированных блоков. Для метода `notifyElementAdded` это включает получение “снимка” списка `observers`, который затем можно безопасно обойти без блокировки. При таком изменении оба предыдущих примера будут выполняться без исключений и взаимоблокировок :

```
// Чужой метод вынесен за пределы синхронизируемого
// блока — открытые вызовы
private void notifyElementAdded(E element)
{
    List<SetObserver<E>> snapshot = null;
    synchronized(observers)
    {
        snapshot = new ArrayList<>(observers);
    }

    for (SetObserver<E> observer : snapshot)
        observer.added(this, element);
}
```

На самом деле есть лучший способ переместить вызов чужого метода за рамки синхронизируемого блока. Библиотеки предоставляют *параллельную коллекцию* (раздел 11.4), известную как `CopyOnWriteArrayList`, созданную специально для этой цели. Эта реализация `List` представляет собой вариант `ArrayList`, в котором все модифицирующие операции реализуются с помощью создания копии всего базового массива. Поскольку внутренний массив никогда не изменяется, итерации не требуют блокировки и выполняются очень быстро. Для большинства применений производительность `CopyOnWriteArrayList` будет ужасной, но для списка наблюдателей он подойдет идеально, так как список редко изменяется и очень часто выполняется его обход.

Изменять методы `add` и `addAll` множества `ObservableSet` для применения `CopyOnWriteArrayList` не нужно. Вот как выглядит оставшаяся часть класса. Обратите внимание, что явной синхронизации здесь нет:

```
// Безопасное с точки зрения параллельности множество
// с применением CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
    new CopyOnWriteArrayList<>();
public void addObserver(SetObserver<E> observer)
{
    observers.add(observer);
}
public boolean removeObserver(SetObserver<E> observer)
{
    return observers.remove(observer);
}
private void notifyElementAdded(E element)
{
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

Чужой метод, вызываемый за пределами синхронизируемой области, известен как *открытый вызов* (open call) [13, 10.1.4]. Кроме предотвращения сбоев, открытые вызовы могут существенно увеличить степень параллельности. Чужой метод может выполняться в течение произвольно длительного времени. Если чужой метод вызван из синхронизируемой области, другие потоки не получат доступ к излишне, без надобности защищенным ресурсам.

Как правило, нужно выполнять как можно меньше действий внутри синхронизируемой области. Получить блокировку, изучить совместно используемые данные, преобразовать их нужным образом, а затем снять блокировку. Если вам требуется выполнение некоторых длительных действий, найдите способ вынести их за пределы синхронизируемой области, не нарушая рекомендаций из раздела 11.1.

Первая часть этого раздела была посвящена корректности. Теперь вкратце рассмотрим вопрос производительности. Хотя затраты на синхронизацию существенно уменьшились со времен появления языка Java, тем не менее с синхронизацией важно не переусердствовать. В мультипроцессорном мире реальная стоимость излишней синхронизации — это не время, затрачиваемое процессором на захват блокировок, а утерянные возможности параллельного выполнения и задержки, вызванные необходимостью гарантировать, что каждое ядро имеет согласованное представление памяти. Другие скрытые затраты излишней синхронизации заключаются в том, что она может ограничить возможность виртуальной машины оптимизировать выполнение кода.

Если вы создаете изменяемый класс, у вас есть два варианта: вы можете опустить все синхронизации и разрешить клиенту выполнять внешнюю синхронизацию, если желательно одновременное использование. Вы также можете выполнять синхронизацию внутренне, делая класс *безопасным с точки зрения потоков* (раздел 11.5). Выбирать последний вариант следует, только если вы можете добиться значительно более высокой степени параллелизма, используя внутреннюю синхронизацию, а не позволяя клиенту блокировать весь объект извне. В коллекциях `java.util` (за исключением устаревших `Vector` и `Hashtable`) принят первый подход, в то время как в `java.util.concurrent` — последний (раздел 11.4).

На заре развития Java многие классы нарушили эти рекомендации. Например, экземпляры `StringBuffer` почти всегда используются одним потоком, но в них выполняется внутренняя синхронизация. Именно по этой причине `StringBuffer` был вытеснен `StringBuilder`, который представляет собой просто несинхронизированный `StringBuffer`. То же самое является основной причиной, по которой потокобезопасный генератор псевдослучайных чисел в `java.util.Random` был впоследствии вытеснен посредством несинхронизируемой реализации в `java.util.concurrent.ThreadLocalRandom`. Если вы сомневаетесь, *не* синхронизируйте свой класс, но обязательно документируйте, что он не является потокобезопасным.

Если вы синхронизируете класс внутренне, то можете воспользоваться различными приемами достижения высокой степени параллельности, такими как разделение блокировок, распределение блокировок и неблокирующее управление параллельностью. Эти приемы выходят за рамки этой книги, но вы можете найти подробную информацию о них в других книгах [13, 17].

Если метод изменяет статическое поле и существует возможность вызова метода из нескольких потоков, вы *должны* синхронизировать доступ к полю внутренне (если только класс не допускает недетерминированное поведение). Многопоточный клиент не в состоянии выполнить внешнюю синхронизацию такого метода, поскольку несвязанные клиенты могут вызывать метод без синхронизации. Это поле, по существу, является глобальной переменной, даже если оно закрытое, потому что его могут считывать и изменять несвязанные клиенты. Поле `nextSerialNumber`, используемое методом `generateSerialNumber` в разделе 11.1, является примером такой ситуации.

В силу всего сказанного во избежание взаимной блокировки и повреждения данных никогда не вызывайте чужие методы из синхронизируемой области. В общем случае постарайтесь ограничить объем работы, выполняемой в синхронизируемых областях, до минимума. Проектируя изменяемый класс, подумайте о том, должен ли он иметь собственную синхронизацию. В современном многопроцессорном мире как никогда важно не злоупотреблять

синхронизацией. Синхронизируйте класс внутренне, только если для этого есть важные причины, и четко документируйте свое решение (раздел 11.5).

11.3. Предпочитайте исполнителей, задания и потоки данных потокам исполнения

В первом издании данной книги содержался код для простой *рабочей очереди* [3, рекомендация 49]. Этот класс позволял клиентам ставить в очередь рабочие задачи для асинхронной обработки фоновым потоком. Когда рабочая очередь становится ненужной, клиент может вызвать метод, запрашивающий фоновый поток о завершении после выполнения задач в очереди. Приведенная реализация была не более чем игрушкой, но она состояла из целой страницы очень тонкого кода, подверженного при любой неосторожности сбоям безопасности и живучести. К счастью, больше нет причин для написания такого кода.

Ко времени написания второго издания книги в Java был добавлен пакет `java.util.concurrent`. Этот пакет содержит *Executor Framework* (каркас исполнителей), обеспечивающий гибкую функциональную возможность выполнения задач на основе интерфейсов. Создание рабочей очереди, которая в результате оказывается реализованной лучше, чем в первой редакции книги, требует всего лишь одной строки кода:

```
ExecutorService exec = Executors.newSingleThreadExecutor();
```

Вот как ей передается задание `Runnable` для выполнения:

```
exec.execute(runnable);
```

А вот как корректно завершить работу исполнителя (если вы этого не сделаете, то, скорее всего, ваша виртуальная машина не завершит работу):

```
exec.shutdown();
```

С помощью службы исполнителей можно делать *гораздо больше* вещей. Например, вы можете подождать, пока не завершится определенная задача (с помощью метода `get`, как рассказывается в разделе 11.2), или подождать, пока одна или все задачи из коллекции задач не завершатся (используя методы `invokeAny` и `invokeAll`). Можно подождать, пока служба не завершится (используя метод `awaitTermination`), можно поочередно выводить результаты выполненных задач по мере их завершения (используя `ExecutorCompletionService`) и т.д.

Если вы хотите обрабатывать запросы из очереди более чем одним потоком, просто вызовите другую статическую фабрику, которая создает другую

разновидность службы исполнителей, именуемую *пулом потоков выполнения* (thread pool). Вы можете создать пул потоков с фиксированным или переменным количеством потоков. Класс `java.util.concurrent.Executors` содержит статические фабрики, предоставляющие большинство исполнителей, которые вам когда-либо могут понадобиться. Если же вы захотите что-то необычное, то можете использовать класс `ThreadPoolExecutor` непосредственно. Этот класс позволит вам управлять почти всеми аспектами работы пула потоков.

Выбор службы исполнителей для определенного приложения может быть довольно сложным. Для небольших программ или мало нагруженных серверов хорошим выбором обычно является `Executor.newCachedThreadPool`, поскольку он не требует настройки и в общем случае “все делает правильно”. Но выбор кешированного пула потоков для сильно нагруженных промышленных серверов будет неудачным решением! В кешированном пуле потоков передаваемые задачи не ставятся в очередь, а сразу отправляются на выполнение. Если доступных потоков нет, то создается новый поток. Если сервер нагружен настолько, что использует все процессоры на полную мощность, создание новых потоков по мере поступления задач только ухудшит ситуацию. Следовательно, для сильно нагруженного сервера лучше использовать `Executor.newFixedThreadPool`, который обеспечивает пул с фиксированным количеством потоков исполнения, или использовать класс `ThreadPoolExecutor` непосредственно для получения максимальных возможностей контроля.

Вам следует воздержаться не только от написания собственных рабочих очередей, но в общем случае и от непосредственной работы с потоками. Когда вы работаете непосредственно с потоками исполнения, класс `Thread` служит как в качестве единицы работы, так и в качестве механизма ее выполнения. В каркасе исполнителей единица работы и исполнительный механизм разделены. Ключевой абстракцией является единица работы — *задание* (task). Есть два вида заданий: `Runnable` и близкое ему `Callable` (похожее на `Runnable` с тем отличием, что оно возвращает значение). Общий механизм выполнения заданий называется *службой исполнителей* (executor service). Если вы мыслите категориями заданий и позволяете службе исполнителей выполнять их для вас, то получаете гибкость в выборе подходящей стратегии, отвечающей вашим требованиям, и при необходимости возможность изменения стратегии. По сути, каркас исполнителей служит для выполнения тем же, чем каркас коллекций служит для агрегации.

В Java 7 каркас исполнителей был расширен для поддержки заданий с ветвлением (fork-join task), которые запускаются специальной разновидностью службы исполнителей. Задание с ветвлением, представленное экземпляром `ForkJoinTask`, может быть разделено на меньшие подзадачи, и потоки,

составляющие ForkJoinPool, не только обрабатывают эти задачи, но и “воруют” задачи один у другого, чтобы гарантировать занятость всех потоков (что приводит к более высокой утилизации процессора, более высокой пропускной способности и низкой задержке). Написание и настройка таких заданий оказывается сложным делом. Параллельные потоки данных (stream) из раздела 7.7 пишутся поверх пулов с ветвлением и позволяют использовать преимущества высокой производительности ценой небольших усилий в предположении, что они подходят для выполнения ваших текущих заданий.

Полное описание каркаса исполнителей выходит за рамки данной книги, но заинтересованный читатель может найти его в книге *Java Concurrency in Practice* [13].

11.4. Предпочитайте утилиты параллельности методам `wait` и `notify`

В первом издании данной книги был раздел, посвященный корректному использованию `wait` и `notify` [3, рекомендация 50]. Содержащиеся в нем советы все еще актуальны и подытожены в конце этого раздела, но сейчас они имеют намного меньшее значение. Дело в том, что сейчас намного меньше причин для использования `wait` и `notify`. Начиная с Java 5 в платформу включены высокоуровневые утилиты параллельности, которые делают то, что раньше приходилось писать вручную поверх `wait` и `notify`. **Исходя из трудности корректного использования `wait` и `notify`, следует использовать высокоуровневые утилиты параллельности.**

Высокоуровневые утилиты из `java.util.concurrent` делятся на три категории: каркас исполнителей, вкратце упомянутый в разделе 11.3, параллельные коллекции и синхронизаторы. В этом разделе бегло описаны параллельные коллекции и синхронизаторы.

Параллельные коллекции представляют собой высокопроизводительные параллельные реализации стандартных интерфейсов коллекций, таких как `List`, `Queue` и `Map`. Для обеспечения высокого уровня параллельности эти реализации сами внутренне управляют своей синхронизацией (раздел 11.2). Таким образом, **исключить параллельную деятельность из параллельной коллекции невозможно, ее блокировка будет только тормозить выполнение программы.**

Поскольку нельзя исключить параллельную деятельность параллельных коллекций, вы не можете получить атомарные составные вызовы методов для них. Поэтому интерфейсы параллельных коллекций были оснащены *модифицирующими операциями, зависящими от состояния* (state-dependent modify

operations), объединяющими несколько примитивов в одну атомарную операцию. Такие операции оказались настолько полезными для параллельных коллекций, что были добавлены в соответствующие интерфейсы коллекций в Java 8 с использованием методов по умолчанию (раздел 4.7).

Например, метод `putIfAbsent(key, value)` из `Map` вставляет отображение ключа, если оно отсутствовало, и возвращает предыдущее значение, связанное с ключом, или `null`, если такового не было. Это облегчает реализацию безопасных с точки зрения потоков традиционных отображений. Следующий метод имитирует поведение `String.intern`:

```
// Параллельное традиционное отображение
// поверх ConcurrentMap - неоптимальный код
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<>();
public static String intern(String s)
{
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

В действительности можно сделать лучше. `ConcurrentHashMap` оптимизирован для выполнения операций извлечения, таких как `get`. Следовательно, имеет смысл вначале вызывать `get`, а `putIfAbsent` вызывать, только если `get` покажет, что это необходимо:

```
// Параллельное традиционное отображение
// поверх ConcurrentMap - более быстрый код!
public static String intern(String s)
{
    String result = map.get(s);

    if (result == null)
    {
        result = map.putIfAbsent(s, s);

        if (result == null)
            result = s;
    }
    return result;
}
```

Кроме обеспечения отличной параллельности, `ConcurrentHashMap` работает очень быстро. На моей машине метод `intern` работает в шесть раз быстрее, чем `String.intern` (но имейте в виду, что `String.intern` должен иметь определенную стратегию во избежание утечки памяти в приложении с высокой продолжительностью времени жизни). Параллельные коллекции

делают синхронизируемые коллекции в основном устаревшими. Например, используйте `ConcurrentHashMap` вместо `Collections.synchronizedMap`. Простая замена старых синхронизируемых отображений параллельными отображениями может существенно увеличить производительность параллельных приложений.

Некоторые интерфейсы коллекций были расширены с применением *блокирующих операций* (blocking operations), которые ожидают (*блокируются*) до тех пор, пока не смогут успешно выполняться. Например, `BlockingQueue` расширяет `Queue` и добавляет несколько методов, в том числе `take`, которые возвращают (с удалением) головной элемент из очереди, переходя в состояние ожидания, если очередь пуста. Это позволяет использовать блокирующие очереди для *рабочих очередей* (также известных как *очереди “производитель–потребитель”* (producer-consumer queue)), в которые добавляют рабочие задания один или несколько *потоков-производителей* и из которых один или несколько *потоков-потребителей* изымают задания для выполнения, как только они становятся доступными. Как можно ожидать, большинство реализаций `ExecutorService`, в том числе `ThreadPoolExecutor`, используют `BlockingQueue` (раздел 11.3).

Синхронизаторы (synchronizer) представляют собой объекты, которые дают возможность потокам ожидать один другого, позволяя тем самым координировать их деятельность. Наиболее часто используемые синхронизаторы — `CountDownLatch` и `Semaphore`. Менее часто используются `CyclicBarrier` и `Exchanger`. Наиболее мощным синхронизатором является `Phaser`.

Синхронизаторы `CountDownLatch` представляют собой разовые барьеры, которые позволяют одному или нескольким потокам ждать, пока один или несколько потоков не выполнят некоторые действия. Единственный конструктор `CountDownLatch` принимает значение типа `int`, которое является количеством вызовов метода `countDown`, которое должно быть выполнено до того, как все ожидающие потоки смогут продолжить работу.

Удивительно легко создавать полезные вещи поверх простых примитивов. Например, предположим, что вы хотите создать простой каркас для фиксации времени параллельного выполнения действия. Этот каркас состоит из единственного метода, который принимает исполнитель, выполняющий действие, уровень параллельности, представляющий собой количество действий, которые должны выполняться параллельно, и выполняемый объект, представляющий собой действие. Все рабочие потоки к моменту запуска таймера должны быть готовы к выполнению действия. Когда последний рабочий поток будет готов к выполнению действия, поток таймера “нажимает на спусковой крючок”, позволяя рабочим потокам выполнять действия. Как только последний рабочий поток завершит выполнение действия, поток таймера останавливает часы.

Реализация этой логики непосредственно поверх wait или notify будет запутанной, но поверх CountdownLatch она оказывается удивительно простой:

```
// Простой каркас для измерения времени параллельного выполнения
public static long time(Executor executor, int concurrency,
                       Runnable action) throws InterruptedException
{
    CountdownLatch ready = new CountdownLatch(concurrency);
    CountdownLatch start = new CountdownLatch(1);
    CountdownLatch done = new CountdownLatch(concurrency);

    for (int i = 0; i < concurrency; i++)
    {
        executor.execute(() ->
        {
            ready.countDown(); // Сообщаем таймеру о готовности

            try
            {
                start.await(); // Ждем готовности остальных
                action.run();
            }
            catch (InterruptedException e)
            {
                Thread.currentThread().interrupt();
            }
            finally {
                done.countDown(); // Сообщаем таймеру
            } // о завершении работы
        });
    }

    ready.await(); // Ждем готовности всех рабочих потоков
    long startNanos = System.nanoTime();
    start.countDown(); // Запускаем их!
    done.await(); // Ждем завершения всех рабочих потоков
    return System.nanoTime() - startNanos;
}
```

Обратите внимание, что метод использует три объекта CountdownLatch. Первый, ready, используется рабочими потоками, чтобы сообщить потоку таймера о готовности к работе. Затем рабочие потоки ждут второго объекта, start. Когда последний рабочий поток вызывает ready.countDown, поток таймера записывает момент старта и вызывает start.countDown, позволяя всем рабочим потокам продолжать работу. Поток таймера ожидает третий объект — done, пока последние рабочие потоки не закончат выполнение действия и не вызовут done.countDown. Как только это произойдет, поток таймера пробуждается и фиксирует время окончания работы.

Стоит упомянуть еще несколько деталей. Исполнитель, который передается методу `time`, должен позволять создание по крайней мере указанное количество потоков, определяемое уровнем параллельности, иначе тест никогда не завершится. Эта ситуация называется *блокировкой голодания потока* (`thread starvation deadlock`) [13, 8.1.1]. Если рабочий поток перехватывает `InterruptedException`, он использует идиому `Thread.currentThread().interrupt()`, и возвращается из своего метода `run`. Это позволяет исполнителю обрабатывать прерывание так, как он считает нужным. Наконец, обратите внимание, что для фиксации времени используется метод `System.nanoTime`. **Для определения продолжительности интервалов всегда используйте `System.nanoTime`, а не `System.currentTimeMillis`.** Метод `System.nanoTime` более аккуратен и точен, и на него не влияют корректировки системных часов. Наконец, обратите внимание, что код в этом примере не дает точной информации, если только `action` не выполняет большое количество работы, скажем, не менее секунды. Точное измерение времени — трудная задача, решать которую лучше всего с помощью специализированных каркасов, таких как `jmh` [26].

Этот раздел лишь поверхностно освещает возможности работы с утилитами параллельности. Например, три объекта `CountDownLatch` из предыдущего примера можно заменить единственным экземпляром `CyclicBarrier` или `Phaser`. Получающийся код еще короче, но и труднее для понимания.

Хотя всегда следует предпочитать использовать утилиты параллельности вместо `wait` и `notify`, вам может встретиться устаревший код, который использует `wait` и `notify`. Метод `wait` используется, чтобы заставить поток ожидать наступления определенного условия. Он должен вызываться из синхронизированной области, которая блокирует объект, для которого он вызывается. Вот стандартная идиома использования метода `wait`:

```
// Стандартная идиома использования метода wait
synchronized(obj)
{
    while (<условие не выполнено>)
        obj.wait(); // (Освобождение блокировки, повторный захват)

    ... // Выполнение действий, соответствующих условию
}
```

Всегда используйте идиому цикла ожидания для вызова метода `wait`; никогда не вызывайте его вне цикла. Цикл служит для проверки условия до и после ожидания.

Проверка условия перед ожиданием и пропуск ожидания, если условие уже выполнено, необходимы для обеспечения живучести приложения. Если

условие уже выполнено и метод `notify` (или `notifyAll`) уже был вызван перед ожиданием потока, нет гарантии, что поток *вообще* когда-либо активируется.

Проверка условия после ожидания и повторное ожидание, если условие не выполняется, необходимы для гарантии безопасности. Если поток продолжает работать, когда условие не выполняется, он может уничтожить инвариант, защищаемый блокировкой. Есть несколько причин, по которым поток может проснуться даже при невыполненном условии.

- Другой поток может получить блокировку и изменить защищенное состояние между моментом времени, когда поток вызвал `notify`, и моментом пробуждения ожидающего потока.
- Другой поток может вызвать `notify` случайно или злоумышленно при несоблюдении условия. Классы демонстрируют такие недостатки при ожидании объекта с открытым (`public`) доступом. Любой вызов метода `wait`, содержащийся в синхронизированном методе открытого объекта, подвержен этой проблеме.
- Уведомляющий поток может быть слишком “щедрым” на пробуждение ожидающих потоков. Например, уведомляющий поток может вызвать `notifyAll`, даже если только некоторые из ожидающих потоков удовлетворяют условиям.
- Ожидающий поток может (изредка) проснуться при отсутствии уведомления. Это явление известно под названием *ложное пробуждение* (*spurious wakeup*) [36, 11.4.3.6.1; 22].

Связанный вопрос — что именно использовать для пробуждения ожидающих потоков: `notify` или `notifyAll`. (Вспомните, что `notify` пробуждает единственный ожидающий поток в предположении, что поток существует, а `notifyAll` будит все ожидающие потоки.) Иногда говорят, что следует *всегда* использовать `notifyAll`. Это разумный, консервативный совет. Он всегда дает корректный результат, потому что гарантирует, что вы разбудите потоки, которые должны быть разбужены. Вы можете также разбудить некоторые другие потоки, но это не повлияет на правильность работы программы. Эти потоки проверят условия, которых они ожидали, и, если найдут их ложными, продолжат ожидание.

В качестве оптимизации можно выбрать вызов `notify` вместо `notifyAll`, если все потоки из множества ожидающих пробуждения ждут выполнения одного и того же условия и в данный момент времени только один поток может извлечь выгоду из истинности ожидаемого условия.

Даже если упомянутые предусловия окажутся истинными, могут быть и другие причины для использования `notifyAll` вместо `notify`. Так же, как размещение вызова `wait` в цикле защищает от случайных и злонамеренных уведомлений общедоступного объекта, использование `notifyAll` вместо `notify` защищает от случайных или злонамеренных ожиданий несвязанного потока. Такие ожидания могут “проглотить” критичные уведомления, оставив их потенциальных получателей в бесконечном состоянии ожидания.

Таким образом, непосредственное применение методов `wait` и `notify` представляет собой аналог программирования на “параллельном языке ассемблера” по сравнению с высокоуровневым языком, предоставленным `java.util.concurrent`. **Причина использовать `wait` и `notify` в новом коде встречается редко (если встречается вообще)**. Если вы поддерживаете код, использующий вызовы `wait` и `notify`, убедитесь, что `wait` всегда вызывается в цикле `while` с использованием стандартной идиомы. В общем случае следует предпочитать вызов метода `notifyAll` методу `notify`. Если вы используете метод `notify`, примите меры для обеспечения его живучести.

11.5. Документируйте безопасность с точки зрения потоков

То, как класс ведет себя при параллельном использовании его методов, является важной частью его контракта со своими клиентами. Если вы не документируете эту сторону поведения класса, использующие его программисты будут вынуждены делать допущения. И если эти допущения окажутся неверными, получившаяся в результате программа может иметь либо недостаточную синхронизацию (раздел 11.1), либо избыточную (раздел 11.2). В любом случае это способно привести к серьезным ошибкам.

Вы могли слышать, что можно определить, безопасен ли метод при работе с несколькими потоками, по наличию модификатора `synchronized` в документации. Это неверно по нескольким причинам. `Javadoc` не включает модификатор `synchronized` в свой вывод, и тому есть причины. **Наличие в объявлении метода модификатора `synchronized` — это деталь реализации, а не часть API**. Наличие модификатора не является надежной гарантией того, что метод безопасен с точки зрения потоков.

Более того, само утверждение о том, что наличия ключевого слова `synchronized` достаточно для утверждения о безопасности метода с точки зрения потоков, содержит неверную посылку о категоричности безопасности с точки зрения потоков — либо все, либо ничего. На самом деле класс может иметь несколько уровней безопасности с точки зрения потоков. **Чтобы класс**

можно было безопасно использовать в параллельной среде, в документации к нему должно быть четко указано, какой уровень безопасности он поддерживает. В приведенном далее списке перечислены уровни безопасности с точки зрения потоков. Это не исчерпывающий список, однако распространенные случаи в нем представлены.

- **Неизменяемый (immutable)**. Экземпляры такого класса выглядят константами. Никакой внешней синхронизации не требуется. Примеры включают `String`, `Long` и `BigInteger` (раздел 4.3).
- **Безусловная безопасность с точки зрения потоков (unconditionally thread-safe)**. Экземпляры такого класса могут изменяться, однако класс имеет достаточную внутреннюю синхронизацию, чтобы его экземпляры могли использоваться параллельно без необходимости во внешней синхронизации. Примеры включают `AtomicLong` и `ConcurrentHashMap`.
- **Условная безопасность с точки зрения потоков (conditionally thread-safe)**. Похожа на безусловную безопасность с точки зрения потоков, отличаясь требованием внешней синхронизации для безопасной работы в параллельной среде. Примеры включают коллекции, возвращаемые оболочками `Collections.synchronized`, итераторы которых требуют внешней синхронизации.
- **Небезопасный с точки зрения потоков (not thread-safe)**. Экземпляры такого класса изменяемы. Для их параллельного использования клиенты должны окружить каждый вызов метода (или последовательность вызовов) внешней синхронизацией по выбору клиента. Примеры включают реализации коллекций общего назначения, такие как `ArrayList` и `HashMap`.
- **Несовместимый с многопоточностью (thread-hostile)**. Такой класс небезопасен при параллельной работе, даже если вызовы всех методов окружены внешней синхронизацией. Обычно несовместимость связана с тем обстоятельством, что эти методы без синхронизации меняют некоторые статические данные. Никто не пишет такие классы целенаправленно; обычно это результат неспособности понять параллелизм. Если выясняется, что класс или метод несовместим с многопоточностью, он обычно исправляется или объявляется устаревшим. Метод `generateSerialNumber` из раздела 11.1 в отсутствие внутренней реализации является несовместимым с многопоточностью.

Эти категории (кроме несовместимости с многопоточностью) примерно соответствуют *аннотациям безопасности потоков (thread safety annotations)* `Immutable`, `ThreadSafe` и `NotThreadSafe` из книги *Java Concurrency in*

Practice [13, приложение A]. Условная и безусловная безопасность с точки зрения потоков охватываются аннотацией `ThreadSafe`.

Документирование класса с условной безопасностью с точки зрения потоков требует особого внимания. Вы должны указать, какие последовательности вызовов требуют внешней синхронизации и какую блокировку (в редких случаях — блокировки) необходимо захватывать для выполнения этих последовательностей. Обычно это блокировка самого экземпляра, но могут иметься и исключения. Например, в документации к методу `Collection.synchronizedMap` говорится следующее:

Обязательно следует использовать ручную синхронизацию возвращаемого отображения при обходе любого из представлений коллекции:

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<>());
Set<K> s = m.keySet(); // Не обязано быть в синхронизируемом блоке
...
synchronized(m)          // Синхронизация m, не s!
{
    for (K key : s)
        key.f();
}
```

Если не следовать этой рекомендации, может наблюдаться недетерминистическое поведение.

Описание потоковой безопасности класса в общем случае содержится в документирующем комментарии к нему, но в случае методов с особыми свойствами безопасности с точки зрения потоков следует указывать эти свойства в документирующих комментариях соответствующих методов. Документировать неизменяемость типов перечислений нет необходимости. Статические фабрики должны документировать безопасность с точки зрения потоков возвращаемых объектов, если только она не очевидна из возвращаемого типа (как продемонстрировано выше, в примере с `Collections.synchronizedMap`).

Если класс использует блокировку с открытым доступом, он позволяет клиентам выполнять последовательность вызовов методов атомарно — ценой потери гибкости. Это несовместимо с высокопроизводительным внутренним управлением параллельностью, как, например, используемым параллельными коллекциями наподобие `ConcurrentHashMap`. К тому же, случайно либо намеренно захватывая блокировку с открытым доступом на длительное время, клиент может вызвать атаку отказа в обслуживании (`denial-of-service` — DoS).

Для предотвращения такой DoS-атаки можно использовать *закрытый объект блокировки* (`private lock object`) вместо синхронизированных методов (которые подразумевают применение блокировки с открытым доступом):

```
// Идиома закрытого объекта блокировки – предотвращение DoS-атаки
private final Object lock = new Object();
public void foo()
{
    synchronized(lock)
    {
        ...
    }
}
```

Поскольку закрытый объект блокировки недоступен клиентам класса, они не смогут вмешиваться в синхронизацию объекта. По сути, мы применяем совет из раздела 4.1, инкапсулируя объект блокировки в синхронизируемом объекте.

Обратите внимание, что поле `lock` объявлено как `final`. Это не позволяет вам непреднамеренно изменить его содержимое, что могло бы привести к катастрофическому несинхронизированному доступу (раздел 11.1). Применим совет из раздела 4.3, сводя к минимуму изменяемость поля `lock`. **Поля блокировок всегда должны быть объявлены как `final`**. Этот совет актуален как при применении обычных блокировок-мониторов, рассмотренных выше, так и при использовании блокировок из пакета `java.util.concurrent.locks`.

Идиома закрытого объекта блокировки может использоваться только с классами, *безусловно* безопасными с точки зрения потоков. Условно безопасные с точки зрения потоков классы не могут использовать эту идиому, поскольку должны документировать, какую блокировку должны захватить их клиенты для выполнения определенных последовательностей вызовов методов.

Идиома закрытого объекта блокировки особенно хорошо подходит для классов, спроектированных для наследования (раздел 4.5). Если бы такой класс использовал свои экземпляры для блокировки, то его подкласс мог бы легко и непреднамеренно вмешаться в работу основного класса и наоборот. Используя одну и ту же блокировку для разных целей, подкласс и базовый класс стали бы в конечном итоге “наступать друг другу на пятки”. И это не просто теоретическая проблема; это происходит с классом `Thread` [4, задача 77].

Резюме: для каждого класса необходимо четко документировать свойства безопасности с точки зрения потоков с помощью аккуратно составленного текстового описания или аннотации потоковой безопасности. Модификатор `synchronized` к этой документации не имеет никакого отношения. Для классов с условной безопасностью с точки зрения потоков в документации необходимо указывать, какая последовательность вызовов методов требует внешней синхронизации и какая блокировка должна быть захвачена при выполнении такой последовательности. Если вы пишете класс с безусловной безопасностью с точки зрения потоков, то рассмотрите использование закрытого объекта блокировки вместо синхронизированных методов. Это защитит вас от

вмешательства в синхронизацию со стороны клиентов и подклассов и обеспечит гибкость в применении более сложных подходов к управлению параллельностью в последующих версиях.

11.6. Аккуратно применяйте отложенную инициализацию

Отложенная (“ленивая”) *инициализация* (lazy initialization) представляет собой задержку инициализации поля до тех пор, пока его значение не потребуется для вычислений. Если значение не потребуется вовсе, поле никогда не будет инициализировано. Этот прием применяется как для статических полей, так и для полей экземпляров. Хотя в основном отложенная инициализация применяется для оптимизации, она может разрушить вредоносную зацикленность при инициализации класса или экземпляра [4, задача 51].

Как и в случае большинства оптимизаций, лучшим советом для отложенной инициализации будет “не используйте ее до тех пор, пока она вам действительно не понадобится” (раздел 9.11). Отложенная инициализация — обоюдоострый меч. Она уменьшает затраты на инициализацию класса или создание экземпляра за счет увеличения стоимости обращения к полю, инициализация которого отложена. В зависимости от того, какая часть этих полей в конечном итоге потребует инициализации, насколько затратной будет их инициализация и как часто будет выполняться обращение к каждому полю, отложенная инициализация может (подобно многим иным “оптимизациям”) навредить производительности.

Отложенная инициализация имеет свою область применения. Если к полю обращается только часть экземпляров и инициализация поля требует существенных затрат, то, возможно, стоит использовать отложенную инициализацию. Единственный способ убедиться в этом — замерить производительность класса с отложенной инициализацией и без нее.

При наличии нескольких потоков отложенная инициализация становится достаточно сложной. Если два (или более) потока используют общее поле с отложенной инициализацией, то применение некоторой разновидности синхронизации становится критичным, иначе могут произойти различные ошибки (раздел 11.1). Все методы инициализации, рассматриваемые в этом разделе, являются безопасными с точки зрения потоков.

В большинстве случаев обычная инициализация предпочтительнее отложенной. Вот типичное объявление инициализируемого обычно поля экземпляра. Обратите внимание на использование модификатора `final` (раздел 4.3):

```
// Обычная инициализация поля экземпляра
private final FieldType field = computeFieldValue();
```

Если вы используете отложенную инициализацию во избежание зацикленности инициализации, используйте синхронизированный метод доступа, так как он является наиболее простой и понятной альтернативой:

```
// Отложенная инициализация поля экземпляра —
// синхронизированный метод доступа
private FieldType field;
private synchronized FieldType getField()
{
    if (field == null)
        field = computeFieldValue();

    return field;
}
```

Обе эти идиомы (*обычная инициализация* и *отложенная инициализация с синхронизированным методом доступа*) точно так же применимы и к статическим полям, отличаясь только добавлением модификатора `static` к объявлениям поля и метода доступа.

Если вам требуется отложенная инициализация статического поля для повышения производительности, используйте идиому класса отложенной инициализации. Эта идиома использует гарантию того, что класс не инициализируется до использования [25, 12.4.1]. Вот как она выглядит:

```
// Идиома класса отложенной инициализации для статических полей
private static class FieldHolder
{
    static final FieldType field = computeFieldValue();
}
private static FieldType getField() { return FieldHolder.field; }
```

Когда метод `getField` вызывается в первый раз, он первый раз читает поле `FieldHolder.field`, запуская инициализацию класса `FieldHolder`. Красота данной идиомы заключается в том, что метод `getField` не синхронизирован и выполняет только доступ к полю, так что отложенная инициализация практически ничего не добавляет к стоимости доступа. Типичная виртуальная машина синхронизирует доступ к полям только для инициализации класса. Как только класс инициализирован, виртуальная машина изменяет код таким образом, чтобы последующие обращения к полю не содержали никаких проверок или синхронизации.

Если вам требуется отложенная инициализация поля экземпляра для повышения производительности, используйте идиому двойной проверки. Эта идиома избавляет от затрат на блокировку при доступе к полю после инициализации (раздел 11.2). Идея данной идиомы заключается в том, чтобы проверять значение поля дважды (отсюда и название): один раз — без блокировки

и в случае, если поле окажется неинициализированным, второй раз — с блокировкой. И только если вторая проверка покажет, что поле не инициализировано, выполняется его инициализация. Поскольку блокировки, когда поле уже инициализировано, нет — *критично* объявление поля как `volatile` (раздел 11.1). Вот как выглядит эта идиома:

```
// Идиома двойной проверки для
// отложенной инициализации полей экземпляров
private volatile FieldType field;
private FieldType getField()
{
    FieldType result = field;

    if (result == null)           // Первая проверка (без блокировки)
    {
        synchronized(this)
        {
            if (field == null) // Вторая проверка (с блокировкой)
                field = result = computeFieldValue();
        }
    }

    return result;
}
```

Этот код может показаться несколько запутанным. В частности, может быть непонятной необходимость локальной переменной `result`. Эта переменная нужна для гарантии однократного чтения `field` в распространенном случае, когда она уже инициализирована. Хотя в ней нет острой необходимости, она может улучшить производительность и повысить элегантность с точки зрения стандартов низкоуровневого параллельного программирования. На моей машине этот метод работал на 25% быстрее, чем обычная версия без локальной переменной.

Хотя идиома двойной проверки может быть применена и к статическим полям, для этого нет никаких причин: идиома класса отложенной инициализации является лучшим выбором.

Заслуживают внимания два варианта идиомы двойной проверки. В некоторых случаях может понадобиться отложенная инициализация поля экземпляра, которая допускает повторную инициализацию. Если вы окажетесь в такой ситуации, можете использовать вариант идиомы двойной проверки, которая обходится без второй проверки и называется *идиомой однократной проверки* (правда, не удивительно?). Вот как она выглядит (обратите внимание, что поле `field` снова объявлено как `volatile`):

```
// Идиома однократной проверки – может
// приводить к повторной инициализации!
private volatile FieldType field;
private FieldType getField()
{
    FieldType result = field;

    if (result == null)
        field = result = computeFieldValue();

    return result;
}
```

Все рассмотренные в этом разделе методы инициализации относятся к примитивным полям, а также к полям, представляющим собой ссылки на объекты. Когда идиома двойной или однократной проверки применяется к примитивному числовому полю, значение поля сравнивается с 0 (значение по умолчанию для примитивных числовых переменных), а не с null.

Если для вас не важно, чтобы *каждый* поток заново вычислял значение поля, а тип поля — примитивный, но не long или double, то модификатор volatile из объявления поля в идиоме однократной проверки можно убрать. Этот вариант известен как *специфичная идиома однократной проверки* (race single-check idiom). В некоторых архитектурах она ускоряет доступ к полю за счет дополнительных инициализаций (до одной на поток, получающий доступ к полю). Это, конечно, экзотический прием, не предназначенный для ежедневного использования.

Мораль такова: большинство полей следует инициализировать с помощью обычной, не отложенной инициализации. Если же вы вынуждены прибегнуть к отложенной инициализации поля с целью повышения производительности или для решения проблемы циклических инициализаций, то используйте одну из подходящих методик отложенной инициализации. Для полей экземпляров это идиома двойной проверки; для статических полей — идиома класса отложенной инициализации. Для полей экземпляров, которые допускают повторную инициализацию, можно также рассмотреть применение идиомы однократной проверки.

11.7. Избегайте зависимости от планировщика потоков

Когда в системе выполняется несколько потоков, планировщик определяет, какие из них будут выполняться и в течение какого времени. Любая корректная операционная система будет пытаться сделать это распределение времени

работы потоков справедливым, однако конкретная стратегия может быть разной. Соответственно, хорошо написанные программы не должны зависеть от деталей применяемой стратегии. **Любая программа, корректность или производительность которой зависит от планировщика потоков, скорее всего, переносимой не будет.**

Лучший способ написать надежную, с малым временем отклика, переносимую многопоточную программу состоит в том, чтобы в любой момент времени среднее количество *работающих* потоков не было значительно большим количества процессоров. В этом случае планировщику потоков остается небогатый выбор: он просто передает управление выполняемым потокам, пока они не завершатся. Поведение программы при этом не будет сильно меняться даже при выборе других алгоритмов диспетчеризации потоков. Обратите внимание, что количество работающих потоков не является тем же, что и общее количество потоков (которое может быть гораздо большим). Потоки, которые находятся в состоянии ожидания, не являются работающими.

Основной прием, позволяющий поддерживать небольшое количество запущенных потоков, — когда каждый поток выполняет небольшую порцию работы, а затем ожидает следующей. **Потоки не должны запускаться, если они не выполняют полезную работу.** В терминах каркаса исполнителей (раздел 11.3) это означает правильный размер пула потоков [13, 8.2] и небольшие задания (но *не слишком* маленькие, иначе накладные расходы диспетчеризации отрицательно скажутся на производительности).

Потоки не должны быть в *активном ожидании* (busy-wait) с регулярной проверкой состояния совместно используемого объекта в ожидании изменения этого состояния. Помимо того, что программа при этом становится зависимой от работы планировщика, активное ожидание может значительно повысить нагрузку процессора, уменьшая количество полезной работы, которая могла бы быть выполнена. В качестве примера, как *не* надо поступать, рассмотрим следующую неправильную реализацию CountdownLatch:

```
// Неверная реализация CountdownLatch — бесконечный опрос!
public class SlowCountDownLatch
{
    private int count;
    public SlowCountDownLatch(int count)
    {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }
}
```

```
public void await()
{
    while (true)
    {
        synchronized(this)
        {
            if (count == 0)
                return;
        }
    }
}

public synchronized void countDown()
{
    if (count != 0)
        count--;
}
}
```

На моей машине `SlowCountDownLatch` примерно в 10 раз медленнее `CountDownLatch` из Java, когда в ожидании находятся 1000 потоков. Хотя данный пример выглядит нереалистичным, не так уж редко можно встретить системы, в которых потоки запускаются без надобности. Результат может быть не настолько критичным, как в случае `SlowCountDownLatch`, но производительность и переносимость, скорее всего, пострадают.

Столкнувшись с программой, которая едва работает из-за того, что некоторые потоки не получают достаточного времени процессора, **не поддавайтесь искушению “исправить” программу добавлением вызовов `Thread.yield`**. Вам, возможно, удастся заставить программу работать, но она не будет переносимой. Те же вызовы `yield`, которые повысят производительность на одной реализации виртуальной машины Java, на второй дадут результат похуже, а на третьей вообще не приведут ни к какому эффекту. **`Thread.yield` не имеет проверяемой семантики**. Лучшим подходом будет изменение структуры приложения для снижения количества параллельно работающих потоков.

Связанный прием, к которому также относится подобное предупреждение, — это настройка приоритетов потоков. **Приоритеты потоков находятся среди наименее переносимых возможностей Java**. Не так уж неразумно улучшить время отклика приложения, “поиграв” с приоритетами потоков, но это редко бывает необходимо и к тому же делает приложения непереносимыми. Совсем неразумно использовать приоритетность потоков для решения серьезных проблем с живучестью. Проблема, скорее всего, останется до тех пор, пока вы не найдете и не устраните фундаментальные ошибки.

Одним словом, правильность вашего приложения не должна зависеть от планировщика потоков, иначе полученное приложение не будет ни надежным, ни переносимым. Как следствие не надо полагаться на метод `Thread.yield` и приоритеты потоков. Эти возможности представляют собой не более чем подсказки для планировщика. Их можно изредка использовать для улучшения качества уже работающей реализации, но ими никогда нельзя пользоваться для “исправления” программы, которая едва работает.

Сериализация

Эта глава посвящена *сериализации объектов* (object serialization), которая представляет собой каркас Java для кодирования объектов в виде потока байтов (*сериализации*) и восстановления объектов из закодированного представления (*десериализации*). После сериализации объекта его закодированное представление может быть передано от одной виртуальной машины другой или сохранено на диске для последующей десериализации. В этой главе особое внимание уделяется вопросам безопасности сериализации и способам ее минимизации.

12.1. Предпочитайте альтернативы сериализации Java

Когда сериализация была добавлена в Java в 1997 году, было известно о том, что данная технология несколько рискованная. Этот подход был испытан в исследовательском языке программирования Modula-3, но никогда не был реализован ни в одном производственном языке. Хотя обещание получения распределенных объектов ценой минимальных усилий со стороны программиста было очень привлекательным, эта цена включала невидимые конструкторы и лишние строки между API и реализацией с потенциальными проблемами корректности, производительности, безопасности и обслуживания. Сторонники сериализации считали, что выгоды перевешивают риски, но история показала иное.

Проблемы безопасности, описанные в предыдущих изданиях этой книги, оказались и в самом деле такими серьезными, как многие опасались. Уязвимости, теоретически обсуждавшиеся в начале 2000-х годов, превратились в серьезный вредоносный код в следующем десятилетии, включая атаку на агентство железнодорожных перевозок Сан-Франциско (SFMTA Muni), которая отключила всю систему тарифных сборов на два дня в ноябре 2016 года [11].

Фундаментальная проблема сериализации состоит в том, что область, в которой она может быть атакована¹ (в смысле проблем безопасности) слишком велика для защиты и постоянно растет: графы объектов десериализуются с помощью вызовов метода `readObject` над `ObjectInputStream`. Этот метод, по существу, представляет собой волшебный конструктор, который может инстанцировать объекты практически любого типа на пути класса, при условии, что тип реализует интерфейс `Serializable`. В процессе десериализации потока байтов этот метод может выполнять код любого из этих типов, поэтому код *всех* этих типов может быть атакован.

Поверхность атаки включает классы библиотек платформы Java, сторонних библиотек, таких как Apache Commons Collections, и самого приложения. Даже если вы придерживаетесь всех наилучших практик и преуспеваете в написании неуязвимых для атак сериализуемых классов, ваше приложение все равно может быть уязвимым. По словам Роберта Сикорда (Robert Seacord), технического руководителя координационного центра CERT:

Десериализация Java — прямая и явная угроза, поскольку она широко используется как непосредственно в приложениях, так и опосредованно подсистемами Java, такими как RMI (удаленный вызов методов), JMX (расширения управления Java) и JMS (системы обмена сообщениями Java). Десериализация ненадежных потоков может привести к удаленному выполнению кода (RCE), отказу в обслуживании (DoS) и целому ряду других нападений. Приложения могут быть уязвимыми для этих атак, даже если они не делают ничего неверного. [39]

И взломщики, и исследователи в области безопасности изучают сериализуемые типы в Java-библиотеках и распространенных сторонних библиотеках в поисках методов, вызываемых в процессе десериализации, которые могут выполнять потенциально опасные действия. Такие методы называются *гаджетами* (gadget). Несколько гаджетов могут использоваться совместно, формируя *цепочку гаджетов* (gadget chain). Время от времени обнаруживается цепочка гаджетов, достаточно мощная, чтобы позволить злоумышленнику выполнить произвольный машинный код, обладая только лишь возможностью представить тщательно сформированный байтовый поток для десериализации. Именно это произошло при нападении на SFMTA Muni. Это нападение не было изолированным — были (и будут) и другие атаки.

Без использования каких-либо гаджетов вы можете легко провести атаку отказа в обслуживании, заставляя десериализовывать короткий поток, требующий много времени для десериализации. Такие потоки известны как *бомбы*

¹ В оригинале используется термин *attack surface*: поверхность атаки. — *Примеч. пер.*

десериализации [48]. Вот пример от Вутера Кекертса (Wouter Coekaerts), который использует только хеш-множества и строку [8]:

```
// Бомба десериализации – десериализация такого потока длится вечно
static byte[] bomb()
{
    Set<Object> root = new HashSet<>();
    Set<Object> s1 = root;
    Set<Object> s2 = new HashSet<>();

    for (int i = 0; i < 100; i++)
    {
        Set<Object> t1 = new HashSet<>();
        Set<Object> t2 = new HashSet<>();
        t1.add("foo");          // Делаем t1 неравным t2
        s1.add(t1);
        s1.add(t2);
        s2.add(t1);
        s2.add(t2);
        s1 = t1;
        s2 = t2;
    }

    return serialize(root); // Для краткости метод опущен
}
```

Граф объекта состоит из 201 экземпляра `HashSet`, каждый из которых содержит 3 или меньше ссылок на объекты. Весь поток имеет длину 5744 байта, но Солнце погаснет задолго до того, как вы его десериализуете. Проблема заключается в том, что десериализация экземпляра `HashSet` требует вычисления хеш-кодов его элементов. Два элемента корневого множества являются хеш-множествами, каждое из которых содержит два хеш-множества элементов, каждый из которых содержит два хеш-множества элементов... и так далее до ста уровней в глубину. Таким образом, десериализация множества вызывает метод `hashCode` более 2^{100} раз. Помимо того, что десериализация длится вечно, десериализатор не видит никаких признаков того, что что-то неладно. Создается несколько объектов, а глубина стека ограничена.

Так как же можно защититься от этих проблем? Вы открываете себя для атаки всякий раз, когда десериализуете поток байтов, который не можете проверить. **Лучший способ избежать проблем, связанных с сериализацией, — никогда ничего не десериализовать.** По словам компьютера по имени Джошуа из фильма 1983 года *WarGames*, “единственный ход, обеспечивающий победу, — не играть”. **Нет никаких оснований для использования сериализации Java в любой новой системе, которую вы пишете.** Существуют другие механизмы преобразования между объектами и последовательностями байтов,

которые позволяют избежать множества опасностей сериализации Java, в то же время предлагая многочисленные преимущества, такие как кросс-платформенная поддержка, высокая производительность, большее количество инструментария и опыт большого сообщества программистов. В этой книге мы ссылаемся на эти механизмы как на *кроссплатформенные представления структурированных данных*. В то время как другие иногда именуют их системами сериализации, в этой книге мы не используем такое название, чтобы избежать путаницы с сериализацией Java.

Общее у этих представлений то, что они *гораздо* проще сериализации Java. Они не поддерживают автоматическую сериализацию и десериализацию произвольных графов объектов. Вместо этого они поддерживают простые, структурированные объекты данных, состоящие из коллекции пар “атрибут/значение”. Поддерживаются только несколько примитивных типов данных и массивы. Этой простой абстракции оказывается достаточно для построения распределенных систем — чрезвычайно мощных и достаточно простых, чтобы избежать серьезных проблем, которые преследуют сериализацию Java с момента ее создания.

Ведущими кросс-платформенными представлениями структурированных данных являются JSON [27] и Protocol Buffers (буфера протоколов), известные также как *protobuf* [37]. JSON был разработан Дугласом Крокфордом (Douglas Crockford) для взаимодействия браузеров с серверами, а буфера протоколов созданы компанией Google для хранения и обмена структурированными данными между своими серверами. Несмотря на то что эти представления иногда называют *нейтральными по отношению к языкам* (*language-neutral*), JSON изначально разрабатывался для JavaScript, а *protobuf* — для C++; оба представления сохраняют следы своего происхождения.

Наиболее существенные различия между JSON и *protobuf* заключаются в том, что JSON основан на текстовом представлении и может быть прочитан человеком, в то время как *protobuf* — представление бинарное (и значительно более эффективное); JSON также является исключительно представлением данных, в то время как *protobuf* предлагает *схемы* (типы) для документирования и обеспечения надлежащего использования информации. Хотя *protobuf* более эффективен, чем JSON, JSON исключительно эффективен для текстового представления. И хотя *protobuf* — бинарное представление, он обеспечивает и альтернативное текстовое представление для использования там, где требуется удобочитаемость для человека (*pbtxt*).

Если же вы не можете полностью избежать сериализации Java, например, потому что работаете в рамках устаревшей системы, которая требует ее применения, следующей альтернативой является **никогда не десериализовать**

непроверенные данные. В частности, вы никогда не должны принимать RMI-трафик из недоверенных источников. Официальное руководство по безопасному кодированию Java гласит: “*Десериализация ненадежных данных опасна, и ее следует избегать*”. Это предложение выделено полужирным курсивом, и это единственный текст в документе, выделенный таким образом [19].

Если вы не можете избежать сериализации и не уверены абсолютно в безопасности десериализуемых данных, используйте фильтрацию десериализуемых объектов, добавленную в Java 9 и перенесенную в более ранние версии (`java.io.ObjectInputFilter`). Она позволяет указать фильтр, который будет применяться к потокам данных, прежде чем они будут десериализованы. Фильтр работает на уровне классов, позволяя вам принимать или отклонять определенные классы. Когда фильтр по умолчанию принимает классы и отвергает только потенциально опасные классы из определенного списка, такой подход известен как *черный список*; если по умолчанию фильтр отвергает классы и принимает только те, которые внесены в список считающихся безопасными, он известен как *белый список*. **Предпочитайте белые списки черным**, так как черный список защищает вас только от известных угроз. Для автоматической подготовки белого списка для вашего приложения может использоваться инструментарий под названием Serial Whitelist Application Trainer (SWAT) [38]. Фильтрация способна также защитить вас от чрезмерного потребления памяти и чрезмерно глубоких графов объектов, но не способна защитить от бомб сериализации наподобие показанных выше.

К сожалению, сериализация все еще широко распространена в экосистеме Java. Если вы поддерживаете систему, которая основана на сериализации Java, серьезно подумайте о возможности перехода на кроссплатформенное представление структурированных данных, несмотря на возможную трудоемкость этого перехода. Но может быть так, что вы будете вынуждены писать или обслуживать сериализуемые классы. Требуется большая осторожность, чтобы написать сериализуемый класс, который является одновременно правильным, безопасным и эффективным. Оставшаяся часть этой главы поможет вам рекомендациями, когда и как это делать.

Сериализация опасна, и ее следует избегать. Если вы проектируете систему “с нуля”, используйте кроссплатформенное представление структурированных данных, например JSON или protobuf. Не десериализуйте ненадежные данные. Если же вы вынуждены это делать, то используйте фильтрацию десериализации (но учтите, что она не гарантирует предотвращение всех атак). Избегайте написания сериализуемых классов. Если же вы вынуждены это делать, будьте осторожны!

12.2. Реализуйте интерфейс `Serializable` крайне осторожно

Чтобы позволить экземпляру класса быть сериализуемым, достаточно добавить в его объявление слова `implements Serializable`. Поскольку это очень легко, получило широкое распространение неправильное представление, что сериализация требует от программиста совсем небольших усилий. На самом деле все гораздо сложнее. В то время как стоимость непосредственного превращения класса в сериализуемый мала, долговременная стоимость зачастую оказывается куда существеннее.

Основное неудобство реализации интерфейса `Serializable` заключается в том, что это уменьшает возможности изменения реализации класса в последующих версиях. Когда класс реализует интерфейс `Serializable`, его закодированный поток байтов (или *сериализованная форма* (`serialized form`)) становится частью его внешнего API. И как только ваш класс получит широкое распространение, вам придется поддерживать эту сериализованную форму точно так же, как вы обязаны поддерживать все остальные части интерфейса, предоставляемого клиентам. Если вы не приложите усилий по проектированию *пользовательской сериализованной формы* (`custom serialized form`), а просто примете форму, предлагаемую по умолчанию, то эта форма окажется навсегда связанной с исходным внутренним представлением класса. Иначе говоря, если вы принимаете сериализованную форму по умолчанию, то закрытые или доступные на уровне пакета экземпляры полей класса станут частью внешнего API и практика минимизации доступности полей (раздел 4.1) потеряет свою эффективность в качестве средства сокрытия информации.

Если вы примете сериализованную форму по умолчанию, а затем измените внутреннее представление класса, это может привести к таким изменениям в форме, что она станет несовместимой с предыдущими версиями. Клиенты, которые будут пытаться сериализовать объект с помощью старой версии класса, а десериализовать его уже с помощью новой версии, столкнутся со сбоем программы. Можно изменить внутреннее представление класса, сохраняя первоначальную сериализованную форму (с использованием методов `ObjectOutputStream.putFields` и `ObjectOutputStream.readFields`), но этот механизм довольно сложен и оставляет в исходном коде программы видимые изъяны. Поэтому следует тщательно проектировать качественную сериализованную форму, с которой вам предстоит жить долгое время (разделы 12.3, 12.6). Поступив таким образом, вы усложните создание приложения, но дело стоит затраченных усилий. Даже хорошо спроектированная сериализованная форма ограничивает дальнейшую эволюцию класса; плохо же спроектированная форма может его искалечить.

Простым примером того, какие ограничения на изменение класса накладывает сериализация, могут служить *уникальные идентификаторы потоков* (*stream unique identifiers*), более известные как *serial version UID* (идентификатор класса в языке Java, используемый при сериализации с использованием стандартного алгоритма, связанный с каждым сериализуемым классом). Если вы не укажете этот идентификатор явно, объявляя поле `static final long` с именем `serialVersionUID`, система сгенерирует его во время выполнения автоматически, применяя криптографическую хеш-функцию (SHA-1) к структуре класса. На это значение оказывают влияние имя класса, реализуемые им интерфейсы, а также большинство его членов, включая генерируемые компилятором. Если вы что-либо измените в этом наборе, например добавите простой метод, изменится и автоматически генерируемое значение идентификатора. Следовательно, если вы не объявите этот идентификатор явно, совместимость с предыдущими версиями будет потеряна, и во время выполнения вы получите исключение `InvalidClassException`.

Второе неудобство реализации интерфейса `Serializable` заключается в том, что повышается вероятность появления ошибок и брешей в системе безопасности (раздел 12.1). Объекты обычно создаются с помощью конструкторов, сериализация же представляет собой механизм создания объектов, *выходящий за рамки языка Java*. Принимаете ли вы поведение по умолчанию или перекрываете его — десериализация представляет собой “скрытый конструктор”, имеющий все те же проблемы, что и прочие конструкторы. Поскольку явного конструктора, связанного с десериализацией, нет, легко забыть, что при десериализации вы должны гарантировать все инварианты, устанавливаемые настоящими конструкторами, и исключить возможность получения злоумышленником доступа к внутреннему содержимому создаваемого объекта. Понадеявшись на механизм десериализации по умолчанию, вы можете легко оставить объекты открытыми для несанкционированного доступа и разрушения инвариантов (раздел 12.4).

Третье неудобство реализации интерфейса `Serializable` заключается в том, что выпуск новой версии класса сопряжен с большой работой по тестированию. При пересмотре сериализуемого класса важно проверить возможность сериализации экземпляра в новой версии и последующей его десериализации в старой и наоборот. Таким образом, объем необходимого тестирования оказывается пропорциональным произведению количества сериализуемых классов и количества имеющихся версий, которые могут оказаться большими величинами. Необходимо гарантировать не только успешность процесса сериализации-десериализации, но и то, что он будет создавать точную копию исходного объекта. Если при написании класса была тщательно спроектирована пользовательская сериализованная форма, необходимость в тестировании уменьшится (разделы 12.3, 12.6).

Реализация интерфейса `Serializable` должна быть тщательно продумана. Очень важно, если класс участвует в каркасе, основанном на сериализации Java для передачи или хранения. Это существенно упрощает применение класса в качестве компонента другого класса, который должен реализовывать интерфейс `Serializable`. Однако с реализацией интерфейса `Serializable` связано множество неудобств. Каждый раз, проектируя класс, сравнивайте неудобства и преимущества. Исторически классы значений, такие как `BigInteger`, реализуют `Serializable`, как и классы коллекций. Классы, представляющие активные сущности, такие как пул потоков, редко должны реализовывать интерфейс `Serializable`.

Классы, предназначенные для наследования (раздел 4.5), редко должны реализовывать `Serializable`, а интерфейсы — редко его расширять. Нарушение этого правила связано с большими затратами для любого, кто пытается расширить такой класс или реализовать интерфейс. Но иногда это правило можно нарушать. Например, если класс или интерфейс существует, в первую очередь, для участия в каркасе, требующем, чтобы все его участники реализовывали интерфейс `Serializable`, то имеет смысл, чтобы класс или интерфейс реализовывал или расширял `Serializable`.

Классы, спроектированные для наследования, которые реализуют интерфейс `Serializable`, включают `Throwable` и `Component`. `Throwable` реализует `Serializable`, с тем чтобы служба удаленного вызова методов (`remote method invocation` — `RMI`) могла передавать исключения от сервера к клиенту. `Component` реализует `Serializable`, чтобы можно было отправить, сохранить и восстановить GUI, но даже во времена расцвета `Swing` и `AWT` эта возможность на практике использовалась слабо.

Если вы реализуете класс с полем экземпляра, который одновременно сериализуем и расширяем, то вы должны знать о некоторых опасностях. Если имеются инварианты для значений полей экземпляра, критически важно предотвратить перекрытие метода `finalize` в подклассах (что класс может сделать, перекрывая метод `finalize` и объявляя его как `final`). В противном случае класс может быть открыт для *атак финализаторов* (раздел 2.8). Наконец, если класс имеет инварианты, которые оказывались бы нарушенными, если бы поля экземпляров были инициализированы значениями по умолчанию (нулем — для целочисленных типов, `false` — для типа `boolean` и `null` — для объектов ссылочных типов), то вы должны добавить следующий метод `readObjectNoData`:

```
// readObjectNoData для расширяемых
// сериализуемых классов с состояниями
private void readObjectNoData() throws InvalidObjectException
{
    throw new InvalidObjectException("Требуется данные потока");
}
```

Этот метод был добавлен в Java 4 для охвата ситуации, включающей добавление сериализуемого суперкласса к существующему сериализуемому классу [40, 3.5].

Нужно сделать одно предупреждение относительно решения *не* реализовывать интерфейс `Serializable`. Если класс, предназначенный для наследования, не является сериализуемым, могут потребоваться дополнительные усилия для написания сериализуемого подкласса. Обычная десериализация такого класса требует, чтобы у суперкласса имелся доступный конструктор без параметров [40, 1.10]. Если вы не предоставляете такой конструктор, подклассы будут вынуждены использовать проектный шаблон прокси сериализации (раздел 12.6).

Внутренние классы (раздел 4.10) не должны реализовывать `Serializable`. Они используют генерируемые компилятором *синтетические поля* (*synthetic fields*) для хранения ссылок на *охватывающие экземпляры* (*enclosing instances*) и хранения значений локальных переменных из включаемых областей видимости. Как именно эти поля соответствуют определению класса, стандартом не определено, как и имена анонимных и локальных классов. Таким образом, сериализуемая форма по умолчанию для внутреннего класса не определена. Однако *статический класс-член* может реализовывать `Serializable`.

Помните: легкость реализации интерфейса `Serializable` обманчива. Если только класс не используется лишь в защищенной среде, в которой различные версии никогда не будут взаимодействовать, а серверы никогда не будут работать с непроверенными данными, то реализация `Serializable` является серьезным обязательством, которое можно принимать только с очень большой осторожностью. И трижды требуется осторожность, когда класс допускает наследование.

12.3. Подумайте о применении пользовательской сериализованной формы

Когда вы пишете класс в условиях дефицита времени, то, вообще говоря, имеет смысл сконцентрировать свои усилия на проектировании наилучшего API. Иногда это означает создание “одноразовой” реализации, которая — как вы точно знаете — поменяется в следующей версии. Обычно это не вызывает проблем, однако, если данный класс реализует интерфейс `Serializable` и использует при этом сериализованную форму по умолчанию, вам уже никогда не удастся полностью избавиться от этой временной реализации, и она всегда будет навязывать вам именно эту сериализованную форму. Это не

теоретическая проблема — такое уже происходило с рядом классов из библиотек платформы Java, включая `BigInteger`.

Нельзя принимать сериализованную форму, предлагаемую по умолчанию, не обдумав сперва как следует, насколько она вас устраивает. Ваше решение должно быть взвешенным, чтобы данное кодирование было приемлемым с точки зрения гибкости, производительности и корректности. Вообще говоря, вы должны принимать сериализованную форму по умолчанию только тогда, когда она практически идентична той кодировке, которую вы бы выбрали, если бы проектировали сериализованную форму сами.

Сериализованная форма представления объекта по умолчанию — довольно эффективное кодирование *физического* представления графа объектов, имеющего корнем данный объект. Другими словами, эта форма описывает данные, содержащиеся в каждом объекте, доступном из данного. Она также описывает топологию взаимосвязи этих объектов. Идеальная сериализованная форма, описывающая объект, содержит только представляемые им *логические* данные и не зависит от физического представления.

Сериализованная форма по умолчанию, скорее всего, окажется приемлемой в случае, когда физическое представление объекта идентично его логическому содержанию. Например, сериализованная форма по умолчанию будет разумной для следующего класса, который упрощенно представляет имя человека:

// Хороший кандидат на сериализованную форму по умолчанию

```
public class Name implements Serializable
{
    /**
     * Фамилия. Не должно быть null.
     * @serial
     */
    private final String lastName;
    /**
     * Имя. Не должно быть null.
     * @serial
     */
    private final String firstName;
    /**
     * Второе имя, или null при отсутствии такового.
     * @serial
     */
    private final String middleName;
    ... // Остальная часть кода опущена
}
```

Логически имя человека в английском языке состоит из трех строк, представляющих фамилию, имя и второе имя. Экземпляры полей в классе Name в точности отражают это логическое содержание.

Даже если вы решите, что сериализованная форма по умолчанию для вас пригодна, имейте в виду, что зачастую для обеспечения инвариантов и безопасности требуется предоставление метода `readObject`. В случае класса Name метод `readObject` должен гарантировать, что поля `lastName` и `firstName` ненулевые. Этот вопрос подробно рассматривается в разделах 12.4 и 12.6.

Обратите внимание, что поля `lastName`, `firstName` и `middleName` документированы, несмотря на то что являются закрытыми. Дело в том, что эти закрытые поля определяют открытый API, а именно — сериализованную форму класса, а всякий открытый API должен быть документирован. Наличие дескриптора `@serial` говорит Javadoc о том, что эту информацию необходимо поместить на специальную страницу, на которой описываются сериализованные формы.

На противоположном конце спектра от Name находится другой класс, который мы сейчас рассмотрим и который представляет набор строк (забудем ненадолго о том, что для этого лучше было бы использовать одну из стандартных реализаций интерфейса `List`):

```
// Ужасный кандидат для сериализованной формы по умолчанию
public final class StringList implements Serializable
{
    private int size = 0;
    private Entry head = null;
    private static class Entry implements Serializable
    {
        String data;
        Entry next;
        Entry previous;
    }
    ... // Остальная часть кода опущена
}
```

Логически этот класс представляет последовательность строк. Физически последовательность представлена классом в виде дважды связанного списка. Если вы примете сериализованную форму по умолчанию, она старательно отразит каждый элемент в этом списке, а также все связи между элементами в обоих направлениях.

Применение сериализованной формы по умолчанию, когда физическое представление объекта существенно отличается от содержащихся в нем логических данных, имеет четыре недостатка.

- **Она навсегда связывает внешний API класса с его текущим внутренним представлением.** В приведенном примере закрытый класс `StringList.Entry` становится частью открытого API. Даже если в будущей версии внутреннее представление изменится, класс `StringList` все равно должен будет принимать в качестве входных данных представление в виде связанного списка и генерировать его же на выходе. Этот класс уже никогда не избавится от кода, необходимого для работы со связанными списками, даже если он ими уже не пользуется.
- **Она может потреблять слишком много памяти.** В приведенном выше примере в сериализованной форме безо всякой на то надобности представлены каждый элемент связанного списка и все его связи. Эти элементы и связи являются просто деталями реализации, не стоящими включения в сериализованную форму. Из-за того, что полученная форма слишком велика, ее запись на диск или передача по сети будет выполняться слишком медленно.
- **Она может требовать слишком много времени.** Логика сериализации не имеет информации о топологии графа объекта, так что ей приходится выполнять дорогостоящий обход графа. В приведенном примере достаточно было бы просто пройти по ссылкам `next`.
- **Она может вызвать переполнение стека.** Процедура сериализации по умолчанию выполняет рекурсивный обход графа объектов, что может вызвать переполнение стека даже при обработке графов среднего размера. На моей машине сериализация экземпляра `StringList` с 1000–1800 элементами приводит к генерации исключения `StackOverflowError`. Удивительно, но размер минимального списка, для которого сериализация вызывает переполнение стека, меняется от запуска к запуску (на моей машине). Размер минимального списка, демонстрирующего эту проблему, может зависеть от платформы реализации и параметров командной строки; некоторые реализации могут не сталкиваться с этой проблемой вовсе.

Разумная сериализованная форма для класса `StringList` — это просто количество строк в списке, за которым следуют сами строки. Это представление состоит из логических данных, представленных классом `StringList`. Вот исправленный вариант `StringList`, содержащий методы `writeObject` и `readObject`, которые реализуют эту сериализованную форму. Напомним, что модификатор `transient` указывает на то, что поле экземпляра в сериализованной форме по умолчанию опущено:

```
// StringList с правильной пользовательской сериализованной формой
public final class StringList implements Serializable
{
    private transient int size = 0;
    private transient Entry head = null;
    // No longer Serializable!
    private static class Entry
    {
        String data;
        Entry next;
        Entry previous;
    }
    // Добавление указанной строки в список
    public final void add(String s)
    {
        ...
    }
    /**
     * Сериализация этого экземпляра {@code StringList}.
     *
     * @serialData Размер списка (количество строк, которые он
     * содержит) ({@code int}), а после него – все элементы
     * списка (каждый {@code String}) в правильном порядке.
     */
    private void writeObject(ObjectOutputStream s)
    throws IOException
    {
        s.defaultWriteObject();
        s.writeInt(size);
        // Запись элементов в правильном порядке.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }
    private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException
    {
        s.defaultReadObject();
        int numElements = s.readInt();
        // Чтение всех элементов и вставка их в список
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }
    ... // Остальная часть кода опущена
}

```

Первое, что делает `writeObject`, — вызывает `defaultWriteObject`, а метод `readObject` вызывает `defaultReadObject`, несмотря на то что поля класса `StringList` объявлены как `transient`. Вы могли слышать, что если

все экземпляры полей имеют модификатор `transient`, то можно обойтись без вызова методов `defaultWriteObject` и `defaultReadObject`, но спецификация сериализации требует вызывать их в любом случае. Эти вызовы позволяют позже добавить поля экземпляра, не объявленные как `transient`, в последующие версии с сохранением прямой и обратной совместимости. Так, если сериализовать экземпляр класса в более поздней версии, а десериализовать в более ранней версии, то добавленные поля будут просто проигнорированы. Если бы более ранняя версия метода `readObject` не вызывала метод `defaultReadObject`, то десериализация закончилась бы генерацией исключения `StreamCorruptedException`.

Заметим также, что, хотя метод `writeObject` является закрытым, он также документирован. Объяснение здесь то же, что и в случае документации закрытых полей в классе `Name`. Этот закрытый метод определяет сериализованную форму — открытый API, а открытый API должен быть документирован. Подобно дескриптору `@serial` для полей, дескриптор `@serialData` для методов говорит Javadoc о том, что данную информацию необходимо поместить на страницу с описанием сериализованных форм.

Что касается производительности, то при средней длине строки, равной десяти символам, сериализованная форма исправленной версии `StringList` занимает примерно вдвое меньше памяти, чем в первоначальном варианте. На моей машине сериализация исправленного варианта `StringList` при длине строк в десять символов выполняется примерно в два раза быстрее, чем сериализация первоначального варианта. И наконец, у исправленного варианта не возникает проблем с переполнением стека, а потому практически нет ограничения сверху на размер `StringList`, для которого можно выполнить сериализацию.

Сериализованная форма, предлагаемая по умолчанию, плохо подходит для класса `StringList`, но есть классы, для которых она подходит еще меньше. Для `StringList` сериализованная форма по умолчанию негибкая и медленно работающая; однако она *корректна* в том смысле, что сериализация и десериализация экземпляра `StringList` приводят к созданию точной копии исходного объекта с сохранением всех его инвариантов. Но это не так для объекта, инварианты которого связаны с деталями реализации.

Например, рассмотрим случай с хеш-таблицей. Ее физическим представлением является набор сегментов, содержащих записи “ключ/значение”. Сегмент, в который будет помещена запись, определяется функцией, которая вычисляет хеш-код ключа. Вообще говоря, нельзя гарантировать, что в различных реализациях JVM этот сегмент будет одним и тем же. В действительности нельзя даже гарантировать, что он будет оставаться одним и тем же при разных выполнениях JVM. Следовательно, использование для хеш-таблицы

сериализованной формы по умолчанию может стать серьезной ошибкой: сериализация и десериализация хеш-таблицы могут привести к созданию объекта, инварианты которого будут серьезно нарушены.

Независимо от того, используете ли вы сериализованную форму по умолчанию, каждое поле экземпляра, не являющееся `transient`, будет сериализовано при вызове метода `defaultWriteObject`. Поэтому каждое поле, которое не требуется помещать в сериализованную форму, следует пометить этим модификатором. К таковым относятся производные поля, значения которых можно вычислить по “первоисточникам”, как, например, кешированное хеш-значение. Сюда также относятся поля, значения которых связаны с конкретным выполнением JVM, как, например, поле типа `long`, в котором хранится указатель на машинно-зависимую структуру данных. **Прежде чем решить сделать какое-то поле не являющимся `transient`, убедитесь в том, что его значение является частью логического состояния объекта.** В случае пользовательской сериализованной формы большинство полей экземпляров (или даже все их) нужно пометить модификатором `transient`, как в показанном выше примере с классом `StringList`.

Если вы используете сериализованную форму по умолчанию и поместили одно или несколько полей модификатором `transient`, помните, что эти поля при десериализации получают значения по умолчанию: `null` — для полей ссылок на объекты, нули — для примитивных числовых полей и `false` — для полей типа `boolean` [25, 4.12.5]. Если для какого-либо из полей указанные значения неприемлемы, необходимо предоставить метод `readObject`, который вызывает метод `defaultReadObject`, а затем восстанавливает приемлемые значения в полях, помеченных как `transient` (раздел 12.4). Альтернативный подход заключается в том, чтобы отложить инициализацию этих полей до первого их использования (раздел 11.6).

Используете вы сериализованную форму по умолчанию или нет, **вы должны использовать для сериализации объекта ту же синхронизацию, что и для любого иного метода чтения всего состояния объекта.** Так, например, если у вас есть объект, безопасный с точки зрения потоков (раздел 11.5), который достигает безопасности путем синхронизации каждого метода, и вы планируете использовать сериализованную форму по умолчанию, то используйте следующий метод `writeObject`:

```
// writeObject для синхронизированного класса
// с сериализованной формой по умолчанию
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException
{
    s.defaultWriteObject();
}
```

Если вы используете синхронизацию в методе `writeObject`, убедитесь, что она подчиняется тем же ограничениям в отношении порядка блокировок, что и другие действия, в противном случае вы рискуете получить взаимоблокировку из-за порядка получения ресурсов [13, 10.1.5].

Независимо от того, какую сериализованную форму вы выберете, в каждом сериализуемом классе, который вы пишете, явным образом объявляйте идентификатор версии сериализации. Тем самым вы исключите этот идентификатор из числа потенциальных источников несовместимости (раздел 12.2). Это также даст определенный выигрыш в производительности. Если данный идентификатор не предоставлен, для его генерации потребуется выполнить трудоемкие вычисления во время выполнения программы.

Для объявления данного идентификатора достаточно добавить в класс одну строку:

```
private static final long serialVersionUID = значение_типа_Long;
```

Если вы пишете новый класс, не важно, какое значение вы выберете для *значение_типа_Long*. Это значение можно сгенерировать, запустив утилиту `serialver` для данного класса, но точно так же можно получить число, “взяв его с потолка”. Уникальность данного идентификатора *не* требуется. Если вы изменяете существующий класс, который не имеет идентификатора версии, и хотите, чтобы новая версия принимала существующие сериализованные экземпляры, используйте значение, которое было сгенерировано для старой версии автоматически. Вы можете получить этот номер, запустив утилиту `serialver` для старой версии класса — той, для которой существуют сериализованные экземпляры.

Если вы хотите создать новую версию класса, *несовместимую* с существующими версиями, просто измените значение в объявлении идентификатора версии. Это приведет к тому, что попытка десериализации сериализованного предыдущей версией экземпляра сгенерирует исключение `InvalidClassException`. **Не изменяйте идентификатор версии, если только вы не хотите нарушить совместимость со всеми существующими сериализованными экземплярами класса.**

Резюме: если уж вы решили, что класс должен быть сериализуемым (раздел 12.2), хорошо подумайте над тем, какой должна быть его сериализованная форма. Используйте форму по умолчанию, *только* если она правильно описывает логическое состояние объекта. В противном случае создайте пользовательскую сериализованную форму, которая надлежащим образом описывает этот объект. На разработку сериализованной формы для класса вы должны выделить не меньше времени, чем на разработку его экспортируемых методов (раздел 8.3). Точно так же, как из будущих версий невозможно изъять те

методы класса, которые были доступны клиентам, нельзя изымать и поля из сериализованной формы. Чтобы обеспечить совместимость сериализации, эти поля должны оставаться в форме навсегда. Неверный выбор сериализованной формы может иметь постоянное отрицательное влияние на сложность и производительность класса.

12.4. Создавайте защищенные методы readObject

В разделе 8.2 представлен неизменяемый класс интервалов времени, который содержит изменяемые закрытые поля Date. Чтобы сохранить свои инварианты и неизменяемость, этот класс создает защитную копию объектов Date в своем конструкторе и методах доступа. Вот как выглядит этот класс:

```
// Неизменяемый класс, использующий защитное копирование
public final class Period
{
    private final Date start;
    private final Date end;
    /**
     * @param start - начало интервала
     * @param end - конец интервала; не должен предшествовать началу
     * @throws IllegalArgumentException, если start следует после end
     * @throws NullPointerException если start или end равно null
     */
    public Period(Date start, Date end)
    {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());

        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                start + " после " + end);
    }
    public Date start()
    {
        return new Date(start.getTime());
    }
    public Date end()
    {
        return new Date(end.getTime());
    }
    public String toString()
    {
        return start + " - " + end;
    }
    ... // Остальная часть кода опущена
}
```

Предположим, что вы решили сделать этот класс сериализуемым. Поскольку физическое представление объекта `Period` в точности отражает его логическое содержание, вполне можно воспользоваться сериализованной формой по умолчанию (раздел 12.3). Может показаться, что все, что вам нужно для того, чтобы сделать класс сериализуемым, — это добавить в его объявление слова `implements Serializable`. Однако если вы поступите таким образом, то класс больше не сможет гарантировать свои критически важные инварианты.

Проблема заключается в том, что метод `readObject` фактически является еще одним открытым конструктором и потому требует такого же внимания, как и любой другой конструктор. Точно так же, как конструктор, метод `readObject` должен проверять корректность своих аргументов (раздел 8.1) и при необходимости выполнять защитное копирование (раздел 8.2). Если метод `readObject` не выполнит хотя бы одно из этих условий, то злоумышленник сможет относительно легко нарушить инварианты этого класса.

Грубо говоря, метод `readObject` — это конструктор, который в качестве единственного входного параметра принимает поток байтов. В обычных условиях этот поток байтов создается в результате сериализации нормально построенного экземпляра. Проблема возникает, когда метод `readObject` сталкивается с потоком байтов, полученным искусственно, с целью генерации объекта, который нарушает инварианты класса. Такой поток байтов может использоваться для создания *невозможного объекта*, который не может быть создан с использованием обычного конструктора.

Предположим, что мы просто добавили `implements Serializable` в объявление класса `Period`. Приведенная далее уродливая программа генерирует такой экземпляр класса `Period`, в котором конец интервала предшествует его началу. Приведения значений типа `byte` с установленным старшим битом являются следствием отсутствия в Java литералов `byte` в сочетании с достойным сожаления решением сделать тип `byte` знаковым:

```
public class BogusPeriod
{
    // Поток байтов, который не может поступать
    // из реального экземпляра Period!
    private static final byte[] serializedForm =
    {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
        0x4c, 0x00, 0x03, (byte)0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
```

```

0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
(byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
0x00, 0x78
};
public static void main(String[] args)
{
    Period p = (Period) deserialize(serializedForm);
    System.out.println(p);
}
// Возвращает объект определенной сериализованной формы
static Object deserialize(byte[] sf)
{
    try
    {
        return new ObjectInputStream(
            new ByteArrayInputStream(sf)).readObject();
    }
    catch (IOException | ClassNotFoundException e)
    {
        throw new IllegalArgumentException(e);
    }
}
}
}

```

Литерал массива байтов, использованный для инициализации массива `serializedForm`, был получен путем сериализации обычного экземпляра `Period` с последующим редактированием потока байтов вручную. Детали построения потока для данного примера значения не имеют, однако если вам это любопытно, то формат потока байтов сериализации описан в *Java Object Serialization Specification* [40, 6]. Если вы запустите эту программу, она выведет

```
Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984
```

Простое объявление класса `Period` как сериализуемого позволило создать объект, который нарушает инварианты класса.

Для решения этой проблемы создадим в классе `Period` метод `readObject`, который вызывает `defaultReadObject` и проверяет правильность десериализованного объекта. Если проверка выявит ошибку, метод `readObject` генерирует исключение `InvalidObjectException`, не позволяющее завершить десериализацию:

```

// Метод readObject с проверкой корректности - недостаточен!
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException

```

```

{
    s.defaultReadObject();
    // Проверка выполнения инвариантов
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " после " + end);
}

```

Хотя это решение и не позволит злоумышленнику создать неправильный экземпляр класса `Period`, здесь притаилась еще одна, более тонкая проблема. Можно создать изменяемый экземпляр `Period`, сфабриковав поток байтов, который начинается потоком байтов, представляющим правильный экземпляр `Period`, а затем формирует дополнительные ссылки на закрытые поля `Date`, внутренние для этого экземпляра. Злоумышленник может прочесть экземпляр `Period` из `ObjectInputStream`, а затем прочесть “неконтролируемые ссылки на объекты”, добавленные к этому потоку. Эти ссылки обеспечивают злоумышленнику доступ к объектам, на которые есть ссылки в закрытых полях `Date` объекта `Period`. Меняя эти экземпляры `Date`, злоумышленник может менять и сам экземпляр `Period`. Следующий класс демонстрирует описанную атаку:

```

public class MutablePeriod
{
    // Экземпляр интервала
    public final Period period;
    // Поле start, к которому у нас не должно быть доступа
    public final Date start;
    // Поле end, к которому у нас не должно быть доступа
    public final Date end;
    public MutablePeriod()
    {
        try
        {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);
            // Сериализация корректного экземпляра Period
            out.writeObject(new Period(new Date(), new Date()));
            /*
             * Добавление неконтролируемых "ссылок на предыдущие
             * объекты" для внутренних полей Date в Period. Детали
             * см. в "Java Object Serialization Specification",
             * Раздел 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // Начальное поле
            ref[4] = 4; // Ref # 4
            bos.write(ref); // Конечное поле

```

```

    // Десериализация Period и "украденных" ссылок на Date
    Object InputStream in = new ObjectInputStream(
        new ByteArrayInputStream(bos.toByteArray()));
    period = (Period) in.readObject();
    start = (Date) in.readObject();
    end = (Date) in.readObject();
}
catch (IOException | ClassNotFoundException e)
{
    throw new AssertionError(e);
}
}
}

```

Чтобы увидеть атаку в действии, выполните следующую программу:

```

public static void main(String[] args)
{
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;
    // Переведем часы назад...
    pEnd.setYear(78);
    System.out.println(p);
    // Вернемся в благословенные 60-е!
    pEnd.setYear(69);
    System.out.println(p);
}

```

На моей машине эта программа дает следующий вывод:

```

Wed Nov 22 00:21:29 PST 2017 - Wed Nov 22 00:21:29 PST 1978
Wed Nov 22 00:21:29 PST 2017 - Sat Nov 22 00:21:29 PST 1969

```

Хотя экземпляр `Period` и создается с неповрежденными инвариантами, при желании его внутренние компоненты можно изменить извне. Завладев изменяемым экземпляром класса `Period`, злоумышленник может причинить массу вреда, передав этот экземпляр классу, безопасность которого зависит от неизменяемости класса `Period`. И это не такая уж надуманная проблема: существуют классы, безопасность которых зависит от неизменяемости класса `String`.

Источник этой проблемы в том, что метод `readObject` класса `Period` не выполняет необходимое защитное копирование. **При десериализации объекта критически важно выполнить защитное копирование всех полей, содержащих ссылки на объекты, которые не должны попасть в распоряжение клиентов.** Поэтому каждый сериализуемый неизменяемый класс, содержащий закрытые изменяемые компоненты, должен выполнить защитное

копирование этих компонентов в методе `readObject`. Приведенный далее метод `readObject` достаточен для того, чтобы гарантировать выполнение инвариантов `Period` и поддержку его неизменности:

```
// Метод readObject с защитным копированием и проверкой
корректности
private void readObject(ObjectInputStream s)
throws IOException, ClassNotFoundException
{
    s.defaultReadObject();
    // Защитное копирование изменяемых компонентов
    start = new Date(start.getTime());
    end = new Date(end.getTime());
    // Проверка выполнения инвариантов
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " после " + end);
}
```

Обратите внимание, что защитное копирование выполняется перед проверкой корректности и для защитного копирования не используется метод `clone` из класса `Date`. Эти особенности реализации необходимы для защиты объекта `Period` от атак (раздел 8.2). Заметим также, что выполнить защитное копирование полей `final` невозможно. Чтобы можно было воспользоваться методом `readObject`, мы должны сделать поля `start` и `end` нефинальными. Это грустно, но это меньшее из двух зол. Разместив в классе метод `readObject` и удалив модификатор `final` из полей `start` и `end`, мы обнаруживаем, что класс `MutablePeriod` восстанавливается неверно. Приведенная выше программа выводит теперь следующие строки:

```
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
Wed Nov 22 00:23:41 PST 2017 - Wed Nov 22 00:23:41 PST 2017
```

Есть простой безошибочный тест, показывающий, приемлем ли метод `readObject`, предлагаемый по умолчанию: будете ли вы чувствовать себя уютно, если добавите в класс открытый конструктор, который в качестве параметров принимает значения полей вашего объекта, записываемых в сериализованную форму, а затем заносит эти значения в соответствующие поля без какой-либо проверки? Если вы не можете ответить на этот вопрос утвердительно, вам нужно явным образом реализовать метод `readObject`, который должен выполнять все необходимые проверки параметров и создавать все защитные копии, как это требовалось бы от конструктора. В качестве альтернативы можно использовать *проектный шаблон прокси сериализации* (раздел 12.6). Этот шаблон настоятельно рекомендуется к применению, потому что он берет на себя большую часть усилий по безопасной десериализации.

Между конструкторами и методами `readObject` существует еще одно сходство, касающееся нефинальных сериализуемых классов: как и конструктор, метод `readObject` не должен вызывать перекрываемые методы — ни прямо, ни косвенно (раздел 4.5). Если это правило нарушено и вызываемый метод перекрыт, то перекрытый метод будет вызван до того, как состояние соответствующего подкласса будет десериализовано. Скорее всего, это приведет к сбою программы [4, Puzzle 91].

Итак, всякий раз, когда вы пишете метод `readObject`, относитесь к нему как к открытому конструктору, который должен создавать правильные экземпляры независимо от того, какой поток байтов был ему передан. Не надо исходить из того, что полученный поток байтов действительно представляет сериализованный экземпляр. Хотя все примеры в данном разделе касались классов, использующих сериализованную форму по умолчанию, все сказанное в такой же мере относится и к классам с пользовательскими сериализованными формами. Приведем в кратком изложении рекомендации по написанию методов `readObject`.

- Для классов с полями, в которых хранятся ссылки на объекты и которые должны оставаться закрытыми, выполняйте защитное копирование каждого объекта в таком поле. Изменяемые компоненты неизменяемых классов также попадают в эту категорию.
- Выполняйте проверку всех инвариантов и в случае ошибки генерируйте исключение `InvalidObjectException`. Проверки должны производиться после защитного копирования.
- Если после десериализации необходимо проверить весь граф объектов, используйте интерфейс `ObjectInputValidation` (выходит за рамки данной книги).
- Ни прямо, ни косвенно не вызывайте перекрываемые методы класса.

12.5. Для управления экземпляром предпочитайте типы перечислений методу `readResolve`

В разделе 2.3 описывается проектный шаблон *Синглтон* (Singleton) и приводится показанный далее пример класса-синглтона. В этом классе ограничен доступ к конструктору, чтобы гарантировать создание только одного экземпляра:

```
public class Elvis
{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis()
```

```

{
    ...
}
public void leaveTheBuilding()
{
    ...
}
}

```

Как отмечалось в разделе 2.3, этот класс перестает быть синглтоном, если в его объявление добавить слова `implements Serializable`. Не имеет значения, использует ли этот класс сериализованную форму по умолчанию или пользовательскую форму (раздел 12.3), а также предоставляется ли пользователю в этом классе явный метод `readObject` (раздел 12.4). Любой метод `readObject`, явный или предоставляемый по умолчанию, возвращает вновь созданный экземпляр, а не тот же экземпляр, который уже был создан в момент инициализации класса.

Метод `readResolve` позволяет нам заменить один экземпляр другим, созданным с помощью метода `readObject` [40, 3.7]. Если класс десериализуемого объекта определяет метод `readResolve` с соответствующим объявлением, то по завершении десериализации для вновь созданного объекта будет вызван этот метод. Ссылка на объект, возвращаемая этим методом, будет затем возвращена вместо вновь созданного объекта. В большинстве использований этого механизма ссылка на новый объект не сохраняется, так что объект фактически немедленно становится доступным для сборщика мусора.

Если класс `Elvis` создан для реализации интерфейса `Serializable`, то для того, чтобы обеспечить свойство синглтона, достаточно создать следующий метод `readResolve`:

```

// readResolve для управления экземпляром — но можно сделать лучше!
private Object readResolve()
{
    // Возвращает истинный объект Elvis и позволяет сборщику мусора
    // заняться самозванцем.
    return INSTANCE;
}

```

Этот метод игнорирует десериализованный объект, возвращая уникальный экземпляр `Elvis`, который был создан во время инициализации класса. По этой причине сериализованная форма экземпляра `Elvis` не обязана содержать никаких реальных данных, а все поля экземпляра должны быть помечены как `transient`. Фактически, если вы полагаетесь на метод `readResolve` для управления экземпляром, все поля со ссылками на объект *должны* быть объявлены как `transient`. В противном случае определенный

злоумышленник может получить ссылку на десериализованный объект до того, как будет выполнен метод `readResolve`, используя прием, который частично похож на атаку `MutablePeriod` из раздела 12.4.

Данная атака немного сложнее, но основная идея проста. Если синглтон содержит не-`transient`-поле ссылки на объект, содержимое этого поля будет десериализовано до вызова метода синглтона `readResolve`. Это позволяет тщательно разработанному потоку “украсть” ссылку на исходный десериализованный синглтон в момент, когда содержимое поля ссылки на объект десериализовано.

Вот как это работает. Сначала пишется класс-“вор”, у которого есть и метод `readResolve`, и поле экземпляра, ссылающееся на сериализованный синглтон, который класс-вор “скрывает”. В потоке сериализации происходит замена не-`transient`-поля синглтона экземпляром класса-вора. Теперь у вас появилась цикличность: синглтон содержит класс-вор, а класс-вор ссылается на синглтон.

Поскольку синглтон содержит класс-вор, метод этого класса `readResolve` выполняется первым, когда синглтон десериализуется. В результате, когда выполняется метод `readResolve`, его поле экземпляра все еще ссылается на частично десериализованный (и пока еще неразрешенный) синглтон.

Метод `readResolve` вора копирует ссылку из своего поля экземпляра в статическое поле, так что после выполнения метода `readResolve` она остается доступной. Затем метод возвращает значение корректного типа для поля, в котором он скрыт. Если этого не сделать, то виртуальная машина сгенерирует исключение `ClassCastException` при попытке системы сериализации сохранить ссылку на вора в этом поле.

Для конкретности рассмотрим следующий взломанный синглтон:

```
// Взломанный синглтон - имеет не-transient-поле ссылки на объект!
public class Elvis implements Serializable
{
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }
    private String[] favoriteSongs =
    { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites()
    {
        System.out.println(Arrays.toString(favoriteSongs));
    }
    private Object readResolve()
    {
        return INSTANCE;
    }
}
```

Вот класс-вор, созданный так, как описано выше:

```
public class ElvisStealer implements Serializable
{
    static Elvis impersonator;
    private Elvis payload;
    private Object readResolve()
    {
        // Сохраняет ссылку на "неразрешенный" экземпляр Elvis
        impersonator = payload;
        // Возвращает объект корректного типа для поля favoriteSongs
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
```

Наконец — уродливая программа, которая десериализует поток, созданный вручную для создания двух различных экземпляров испорченного синглтона. Метод десериализации опущен, так как он идентичен такому же из раздела 12.4:

```
public class ElvisImpersonator
{
    // Поток байтов не может поступать из реального экземпляра
    Elvis!
    private static final byte[] serializedForm =
    {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
        0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
        (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
        0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
        0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
        0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
        0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
        0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
        0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
        0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
        0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
        0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
    };
    public static void main(String[] args)
    {
        // Инициализация ElvisStealer.impersonator и возврат
        // реального Elvis (которым является Elvis.INSTANCE)
        Elvis elvis = (Elvis) deserialize(serializedForm);
        Elvis impersonator = ElvisStealer.impersonator;
        elvis.printFavorites();
        impersonator.printFavorites();
    }
}
```

Выполнение этой программы приводит к следующему результату, доказывающему, что можно создать два различных экземпляра Elvis (с различными музыкальными предпочтениями):

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

Решить эту проблему можно, объявив поле `favoriteSongs` как `transient`, но лучше сделать Elvis типом перечисления с единственным элементом (раздел 2.3). Как демонстрирует атака `ElvisStealer`, использование метода `readResolve` для предотвращения обращения взломщика ко “временно” десериализованному экземпляру — достаточно “хрупкий” способ, требующий особой осторожности.

Если вы напишете свой сериализуемый класс с управлением экземплярами как перечисление, Java гарантирует, что не появится никаких иных экземпляров класса, кроме объявленных констант, если только злоумышленник не воспользуется таким привилегированным методом, как `AccessibleObject.setAccessible` (но любой атакующий, который в состоянии это сделать, уже имеет достаточные привилегии для выполнения произвольного машинного кода). Вот как будет выглядеть пример с классом `Elvis`, когда он является перечислением:

```
// Синглтон-перечисление – предпочтительный подход
public enum Elvis
{
    INSTANCE;
    private String[] favoriteSongs =
    { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites()
    {
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

Использование `readResolve` для управления экземплярами не устарело. Если вам нужно написать сериализуемый класс с управляемыми экземплярами, которые в момент компиляции неизвестны, вы не сможете представить класс как тип перечисления.

Доступность метода `readResolve` имеет важное значение. Если вы поместите метод `readResolve` в `final`-класс, имейте в виду, что он должен быть закрытым. Если вы поместите метод `readResolve` в класс, не являющийся `final`, вам нужно внимательно отнестись к его доступности. Если он закрыт, то его действие не будет распространяться на подклассы. Если он доступен только в рамках пакета, его действие будет распространяться только

на подклассы в пределах одного и того же пакета. Если он защищенный или открытый, его действие будет распространяться на все подклассы, которые его не перекрывают. Если метод `readResolve` защищен или является открытым, а подклассы его не перекрывают, то десериализация экземпляра подкласса создаст экземпляр суперкласса, который, скорее всего, приведет к генерации исключения `ClassCastException`.

Таким образом, насколько это возможно, чтобы обеспечить инварианты управления экземплярами, используйте типы перечислений. Если это невозможно, а вам нужен класс, который был бы и сериализуемым, и с управлением экземплярами, предоставьте метод `readResolve` и гарантируйте, что все поля экземпляров класса являются либо примитивными, либо объявленными как `transient`.

12.6. Подумайте о применении прокси-агента сериализации вместо сериализованных экземпляров

Как говорилось в разделах 12.1 и 12.2 и обсуждалось на протяжении всей главы, решение реализовать `Serializable` увеличивает вероятность ошибок и проблем с безопасностью, так как позволяет создавать экземпляры с помощью механизмов за пределами языка вместо обычных конструкторов. Тем не менее имеется прием, который существенно снижает подобный риск. Этот прием известен как *шаблон прокси сериализации* (*serialization proxy pattern*).

Шаблон прокси сериализации довольно прост и прямолинеен. Во-первых, создайте закрытый статический вложенный класс, который точно представляет логическое состояние экземпляра охватываемого класса. Этот вложенный класс известен как *прокси сериализации* охватываемого класса. Он должен иметь один конструктор, типом параметра которого является охватывающий класс. Этот конструктор просто копирует данные из своего аргумента: ему не требуется никакая проверка целостности или защитное копирование. В соответствии с дизайном сериализованная форма прокси сериализации по умолчанию является идеальной сериализованной формой охватываемого класса. И охватывающий класс, и его прокси сериализации должны быть объявлены как реализующие `Serializable`.

Например, рассмотрим неизменяемый класс `Period`, написанный в разделе 8.2 и сделанный сериализуемым в разделе 12.4. Вот как выглядит прокси сериализации для этого класса (`Period` настолько прост, что его прокси сериализации содержит в точности те же поля, что и сам класс):

```
// Прокси сериализации для класса Period
private static class SerializationProxy implements Serializable
{
    private final Date start;
    private final Date end;
    SerializationProxy(Period p)
    {
        this.start = p.start;
        this.end = p.end;
    }
    private static final long serialVersionUID =
        234098243823485285L; // Любое число (раздел 12.3)
}
```

Затем добавим к охватываемому классу метод `writeReplace`. Этот метод можно дословно скопировать в любой класс с прокси сериализации:

```
// Метод writeReplace для шаблона прокси сериализации
private Object writeReplace()
{
    return new SerializationProxy(this);
}
```

Наличие этого метода в охватываемом классе заставляет систему сериализации выдавать экземпляры `SerializationProxy` вместо экземпляра охватываемого класса. Другими словами, метод `writeReplace` до самой сериализации транслирует экземпляры охватываемого класса в экземпляр его прокси.

При использовании метода `writeReplace` система сериализации никогда не сгенерирует сериализованный экземпляр охватываемого класса, но злоумышленник может сфабриковать его в попытке злонамеренно разрушить инварианты класса. Для того чтобы гарантировать, что такая атака не удастся, просто добавьте метод `readObject` к охватываемому классу:

```
// Метод readObject для шаблона прокси сериализации
private void readObject(ObjectInputStream stream)
throws InvalidObjectException
{
    throw new InvalidObjectException("Требуется прокси");
}
```

Наконец, предоставьте в классе `SerializationProxy` метод `readResolve`, который возвращает логически эквивалентный экземпляр охватываемого класса. Наличие этого метода заставляет систему сериализации транслировать при десериализации прокси сериализации обратно в экземпляр охватываемого класса.

Этот метод `readResolve` создает экземпляр охватывающего класса, используя только его открытый API, и в этом и заключается вся красота шаблона. Он максимально избегает вневязкой природы сериализации, потому что десериализованный экземпляр создается с использованием тех же конструкторов, статических фабрик и методов, что и любой иной экземпляр. Это освобождает вас от необходимости отдельно убеждаться, что десериализованные экземпляры подчиняются инвариантам класса. Если статические фабрики или конструкторы класса устанавливают эти инварианты, а методы экземпляров их поддерживают, то это гарантирует, что сериализация также будет их поддерживать.

Вот пример метода `readResolve` для вышеупомянутого `Period.SerializationProxy`:

```
// Метод readResolve для Period.SerializationProxy
private Object readResolve()
{
    return new Period(start, end); // Использует открытый конструктор
}
```

Как и подход с использованием защитного копирования, использование прокси сериализации препятствует атакам на основе потоков байтов и на основе “кражи” внутреннего поля. В отличие от этих двух подходов рассматриваемый подход позволяет полю `Period` быть объявленным как `final`, что требуется для того, чтобы класс `Period` был действительно неизменяемым (раздел 4.3). И в отличие от двух предыдущих подходов он не требует длительных размышлений. Вам не нужно думать о том, какие поля могут быть дискредитированы при атаках сериализации, вам также не нужно выполнять явную проверку корректности в качестве части процесса десериализации.

Имеется еще один подход, в котором шаблон прокси сериализации оказывается более мощным, чем защитное копирование в `readObject`. Шаблон прокси сериализации позволяет десериализованному экземпляру иметь другой класс, отличный от первоначально сериализованного экземпляра. Вы можете решить, что с практической точки зрения это бесполезно, но на самом деле это не так.

Рассмотрим случай с `EnumSet` (раздел 6.3). У этого класса нет открытых конструкторов, есть только статические фабрики. С точки зрения клиента они возвращают экземпляры `EnumSet`, но в текущей реализации OpenJDK они возвращают один или два подкласса, в зависимости от размера базового типа перечисления. Если у базового типа перечисления 64 или менее элементов, то статические фабрики возвращают `RegularEnumSet`; в противном случае они возвращают `JumboEnumSet`.

Теперь рассмотрим, что происходит при сериализации EnumSet, тип перечисления которого содержит 60 элементов, после чего к множеству добавляются еще пять элементов и выполняется десериализация. При сериализации мы имели экземпляр RegularEnumSet, но при десериализации ему лучше быть экземпляром JumboEnumSet. Фактически именно это и происходит, потому что EnumSet использует шаблон прокси сериализации. Если вам интересно, то вот как выглядит прокси сериализации EnumSet. Он действительно очень прост:

```
// Прокси сериализации EnumSet
private static class SerializationProxy <E extends Enum<E>>
    implements Serializable
{
    // Тип элемента множества.
    private final Class<E> elementType;
    // Элементы, содержащиеся в множестве.
    private final Enum<?>[] elements;
    SerializationProxy(EnumSet<E> set)
    {
        elementType = set.elementType;
        elements = set.toArray(new Enum<?>[0]);
    }
    private Object readResolve()
    {
        EnumSet<E> result = EnumSet.noneOf(elementType);

        for (Enum<?> e : elements)
            result.add((E) e);

        return result;
    }
    private static final long serialVersionUID =
        362491234563181265L;
}
```

У шаблона прокси сериализации есть два ограничения. Он не совместим с классами, расширяемыми их пользователями (раздел 4.5). Он также не совместим с некоторыми классами, графы объектов которых содержат цикличности: если вы попытаетесь вызвать метод для такого объекта из метода readResolve прокси сериализации, то получите исключение ClassCastException, так как у вас еще нет объекта, а есть только его прокси сериализации.

Наконец, дополнительные возможности и безопасность шаблона прокси сериализации не даются даром. На моей машине сериализация и десериализация экземпляров Period с помощью прокси сериализации обходятся на 14% дороже, чем с использованием защитного копирования.

Итак, когда вам приходится писать методы `readObject` или `writeObject` для класса, который не расширяется своими клиентами — рассмотрите возможность применения шаблона прокси-агента сериализации. Этот шаблон, возможно, является простейшим средством надежной сериализации объектов с нетривиальными инвариантами.

Соответствие статей второго издания разделам третьего издания

Статья во втором издании ¹	Раздел в третьем издании
1	2.1, “Рассмотрите применение статических фабричных методов вместо конструкторов”
2	2.2, “При большом количестве параметров конструктора подумайте о проектном шаблоне Строитель”
3	2.3, “Получайте синглтон с помощью закрытого конструктора или типа перечисления”
4	2.4, “Обеспечивайте неинстанцируемость с помощью закрытого конструктора”
5	2.6, “Избегайте создания излишних объектов”
6	2.7, “Избегайте устаревших ссылок на объекты”
7	2.8, “Избегайте финализаторов и очистителей”
8	3.1, “Перекрывая equals, соблюдайте общий контракт”
9	3.2, “Всегда при перекрытии equals перекрывайте hashCode”
10	3.3, “Всегда перекрывайте toString”
11	3.4, “Перекрывайте метод clone осторожно”
12	3.5, “Подумайте о реализации Comparable”
13	4.1, “Минимизируйте доступность классов и членов”
14	4.2, “Используйте в открытых классах методы доступа, а не открытые поля”
15	4.3, “Минимизируйте изменяемость”
16	4.4, “Предпочитайте композицию наследованию”
17	4.5, “Проектируйте и документируйте наследование либо запрещайте его”
18	4.6, “Предпочитайте интерфейсы абстрактным классам”
19	4.8, “Используйте интерфейсы только для определения типов”
20	4.9, “Предпочитайте иерархии классов дескрипторам классов”

¹ В переводе второго издания книги (Джошуа Блох, *Java. Эффективное программирование*. — Изд. “Лори”, 2014) “разделы” переведены как “статьи”. — *Примеч. пер.*

Статья во втором издании ¹	Раздел в третьем издании
21	7.1, “Предпочитайте лямбда-выражения анонимным классам”
22	4.10, “Предпочитайте статические классы-члены нестатическим”
23	5.1, “Не используйте несформированные типы”
24	5.2, “Устраняйте предупреждения о непроверяемом коде”
25	5.3, “Предпочитайте списки массивам”
26	5.4, “Предпочитайте обобщенные типы”
27	5.5, “Предпочитайте обобщенные методы”
28	5.6, “Используйте ограниченные символы подстановки для повышения гибкости API”
29	5.8, “Применяйте безопасные с точки зрения типов гетерогенные контейнеры”
30	6.1, “Используйте перечисления вместо констант <code>int</code> ”
31	6.2, “Используйте поля экземпляров вместо порядковых значений”
32	6.3, “Используйте <code>EnumSet</code> вместо битовых полей”
33	6.4, “Используйте <code>EnumMap</code> вместо индексирования порядковыми номерами”
34	6.5, “Имитируйте расширяемые перечисления с помощью интерфейсов”
35	6.6, “Предпочитайте аннотации схемам именования”
36	6.7, “Последовательно используйте аннотацию <code>Override</code> ”
37	6.8, “Используйте интерфейсы-маркеры для определения типов”
38	8.1, “Проверяйте корректность параметров”
39	8.2, “При необходимости создавайте защитные копии”
40	8.3, “Тщательно проектируйте сигнатуры методов”
41	8.4, “Перегружайте методы разумно”
42	8.5, “Используйте методы с переменным количеством аргументов с осторожностью”
43	8.6, “Возвращайте пустые массивы и коллекции, а не <code>null</code> ”
44	8.8, “Пишите документирующие комментарии для всех открытых элементов API”
45	9.1, “Минимизируйте область видимости локальных переменных”
46	9.2, “Предпочитайте циклы <code>for</code> для коллекции традиционным циклом <code>for</code> ”
47	9.3, “Изучите и используйте возможности библиотек”
48	9.4, “Если вам нужны точные ответы, избегайте <code>float</code> и <code>double</code> ”
49	9.5, “Предпочитайте примитивные типы упакованным примитивным типам”
50	9.6, “Избегайте применения строк там, где уместнее другой тип”
51	9.7, “Помните о проблемах производительности при конкатенации строк”
52	9.8, “Для ссылки на объекты используйте их интерфейсы”

Окончание таблицы

Статья во втором издании ¹	Раздел в третьем издании
53	9.9, “Предпочитайте интерфейсы рефлексии”
54	9.10, “Пользуйтесь машинно-зависимыми методами осторожно”
55	9.11, “Оптимизируйте осторожно”
56	9.12, “Придерживайтесь общепринятых соглашений по именованию”
57	10.1, “Используйте исключения только в исключительных ситуациях”
58	10.2, “Используйте для восстановления проверяемые исключения, а для программных ошибок — исключения времени выполнения”
59	10.3, “Избегайте ненужных проверяемых исключений”
60	10.4, “Предпочитайте использовать стандартные исключения”
61	10.5, “Генерируйте исключения, соответствующие абстракции”
62	10.6, “Документируйте все исключения, которые может генерировать метод”
63	10.7, “Включайте в сообщения информацию о сбое”
64	10.8, “Добивайтесь атомарности сбоев”
65	10.9, “Не игнорируйте исключения”
66	11.1, “Синхронизируйте доступ к совместно используемым изменяемым данным”
67	11.2, “Избегайте излишней синхронизации”
68	11.3, “Предпочитайте исполнителей, задания и потоки данных потокам исполнения”
69	11.4, “Предпочитайте утилиты параллельности методам wait и notify”
70	11.5, “Документируйте безопасность с точки зрения потоков”
71	11.6, “Аккуратно применяйте отложенную инициализацию”
72	11.7, “Избегайте зависимости от планировщика потоков”
73	Отсутствует
74	12.1, “Предпочитайте альтернативы сериализации Java” 12.2, “Реализуйте интерфейс Serializable крайне осторожно”
75	12.1, “Предпочитайте альтернативы сериализации Java” 12.3, “Подумайте о применении пользовательской сериализованной формы”
76	12.1, “Предпочитайте альтернативы сериализации Java” 12.4, “Создавайте защищенные методы readObject”
77	12.1, “Предпочитайте альтернативы сериализации Java” 12.5, “Для управления экземпляром предпочитайте типы перечислений методу readResolve”
78	12.1, “Предпочитайте альтернативы сериализации Java” 12.6, “Подумайте о применении прокси-агента сериализации вместо сериализованных экземпляров”

Список литературы

- 1 *Programming with Assertions*. 2002. Sun Microsystems. <http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
- 2 Beck, Kent. 2004. *JUnit Pocket Guide*. Sebastopol, CA: O'Reilly Media, Inc. ISBN: 0596007434.
- 3 Bloch, Joshua. 2001. *Effective Java Programming Language Guide*. Boston: Addison-Wesley. ISBN: 0201310058.
- 4 Bloch, Joshua, and Neal Gafter. 2005. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Boston: Addison-Wesley. ISBN: 032133678X.
- 5 Blum, Scott. 2014. "Faster RSA in Java with GMP." *The Square Corner* (blog). Feb. 14, 2014. <https://medium.com/square-corner-blog/faster-rsa-in-java-with-gmp-8b13c51c6ec4>
- 6 Bracha, Gilad. 2004. "Lesson: Generics" online supplement to *The Java Tutorial: A Short Course on the Basics*, 6th ed. Upper Saddle River, NJ: Addison-Wesley, 2014. <https://docs.oracle.com/javase/tutorial/extra/generics/>
- 7 Burn, Oliver. 2001–2017. *Checkstyle*. <http://checkstyle.sourceforge.net>
- 8 Coekaerts, Wouter (@WouterCoekaerts). 2015. "Billion-laughsstyle DoS for Java serialization" [https://gist.github.com/coekie/a27cc406fc9f3dc7a70d ...](https://gist.github.com/coekie/a27cc406fc9f3dc7a70d...) WONTFIX," Twitter, November 9, 2015, 9:46 a.m. <https://twitter.com/woutercoekaerts/status/663774695381078016>
- 9 Brief of Computer Scientists as Amici Curiae for the United States Court of Appeals for the Federal Circuit, Case No. 17-1118, Oracle America, Inc. v. Google, Inc. in Support of Defendant-Appellee (2017)
- 10 *Dagger*. 2013. Square, Inc. <http://square.github.io/dagger/>
- 11 Gallagher, Sean. 2016. "Muni system hacker hit others by scanning for year-old Java vulnerability." *Ars Technica*, November 29, 2016. <https://arstechnica.com/information-technology/2016/11/san-francisco-transit-ransomware-attacker-likely-used-year-old-java-exploit/>
- 12 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley. ISBN: 0201633612.
Перевод на русский язык: Гамма Э., Хелм Р., Джонсон Р., Влissидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — СПб.: Питер, 2001.
- 13 Goetz, Brian. 2006. *Java Concurrency in Practice*. With Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Boston: Addison-Wesley. ISBN: 0321349601.
- 14 Gosling, James. 1997. "The Feel of Java." *Computer* 30 no. 6 (June 1997): 53–57. <http://dx.doi.org/10.1109/2.587548>
- 15 *Guava*. 2017. Google Inc. <https://github.com/google/guava>

- 16 Guice. 2006. Google Inc. <https://github.com/google/guice>
- 17 Herlihy, Maurice, and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint*. Waltham, MA: Morgan Kaufmann Publishers. ISBN: 0123973376.
- 18 Jackson, M. A. 1975. *Principles of Program Design*. London: Academic Press. ISBN: 0123790506.
- 19 *Secure Coding Guidelines for Java SE*. 2017. Oracle. <http://www.oracle.com/technetwork/java/seccodeguide-139067.html>
- 20 *What's New in JDK 8*. 2014. Oracle. <http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html>
- 21 *Java Platform, Standard Edition What's New in Oracle JDK 9*. 2017. Oracle. <https://docs.oracle.com/javase/9/whatsnew/toc.htm>
- 22 *Java Platform, Standard Edition & Java Development Kit Version 9 API Specification*. 2017. Oracle. <https://docs.oracle.com/javase/9/docs/api/overview-summary.html>
- 23 *How to Write Doc Comments for the Javadoc Tool*. 2000–2004. Sun Microsystems. <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>
- 24 *Javadoc Reference Guide*. 2014–2017. Oracle. <https://docs.oracle.com/javase/9/javadoc/javadoc.htm>
- 25 Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2014. *The Java Language Specification, Java SE 8 Edition*. Boston: Addison-Wesley. ISBN: 013390069X. Перевод на русский язык: Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли. *Язык программирования Java SE 8. Подробное описание, 5-е изд.* — М.: “ООО И.Д. Вильямс”, 2015.
- 26 *Code Tools: jmh*. 2014. Oracle. <http://openjdk.java.net/projects/code-tools/jmh/>
- 27 *Introducing JSON*. 2013. Ecma International. <https://www.json.org>
- 28 Kahan, William, and J. W. Thomas. 1991. *Augmenting a Programming Language with Complex Arithmetic*. UCB/CSD-91-667, University of California, Berkeley.
- 29 Knuth, Donald. 1974. Structured Programming with gotoStatements. In *Computing Surveys* 6: 261–301.
- 30 Lea, Doug. 2014. *When to use parallel streams*. <http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html>
- 31 Lieberman, Henry. 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, Portland, September 1986. ACM Press.
- 32 Liskov, B. 1988. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of OOPSLA '87 and SIGPLAN Notices*, Vol. 23, No. 5: 17–34, May 1988.
- 33 Scott Meyers. 1998. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1998
Русскоязычное издание: Скотт Мейерс, *Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14*. — М.: “ООО И.Д. Вильямс”, 2016.

- 34 Naftalin, Maurice, and Philip Wadler. 2007. *Java Generics and Collections*. Sebastopol, CA: O'Reilly Media, Inc. ISBN: 0596527756.
- 35 Parnas, D. L. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. In *Communications of the ACM* 15: 1053–1058.
- 36 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) C Language] (ANSI), IEEE Standards Press, ISBN: 1559375736.
- 37 *Protocol Buffers*. 2017. Google Inc. <https://developers.google.com/protocol-buffers>
- 38 Schneider, Christian. 2016. SWAT (Serial Whitelist Application Trainer). <https://github.com/cschneider4711/SWAT/>
- 39 Seacord, Robert. 2017. *Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS)*. San Francisco: NCC Group Whitepaper. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2017/june/ncc_group_combating_java_deserialization_vulnerabilities_with_look-ahead_object_input_streams1.pdf
- 40 *Java Object Serialization Specification*. March 2005. Sun Microsystems. <http://docs.oracle.com/javase/9/docs/specs/serialization/index.html>
- 41 Sestoft, Peter. 2016. *Java Precisely*, 3rd ed. Cambridge, MA: The MIT Press. ISBN: 0262529076.
- 42 Aleksey Shipilëv. 2016. *Arrays of Wisdom of the Ancients*. <https://shipilev.net/blog/2016/arrays-wisdom-ancients/>
- 43 Smith, Robert. 1962. Algorithm 116 Complex Division. In *Communications of the ACM* 5, no. 8 (August 1962): 435.
- 44 Snyder, Alan. 1986. “Encapsulation and Inheritance in Object-Oriented Programming Languages.” In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 38–45. New York, NY: ACM Press.
- 45 *Spring Framework*. Pivotal Software, Inc. 2017. <https://projects.spring.io/spring-framework/>
- 46 Stroustrup, Bjarne. [ca. 2000]. “Is Java the language you would have designed if you didn’t have to be compatible with C?” *Bjarne Stroustrup’s FAQ*. Updated October 1, 2017. http://www.stroustrup.com/bs_faq.html#Java
- 47 Stroustrup, Bjarne. 1995. “Why C++ is not just an object-oriented programming language.” In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications*, edited by Steven Craig Bilow and Patricia S. Bilow New York, NY: ACM. <http://dx.doi.org/10.1145/260094.260207>
- 48 Svoboda, David. 2016. *Exploiting Java Serialization for Fun and Profit*. Software Engineering Institute, Carnegie Mellon University. <https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=484347>
- 49 Thomas, Jim, and Jerome T. Coonen. 1994. “Issues Regarding Imaginary Types for C and C++.” In *The Journal of C Language Translation* 5, no. 3 (March 1994): 134–138.

- 50 *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* 1999. Sun Microsystems. <https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>
- 51 Viega, John, and Gary McGraw. 2001. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston: Addison- Wesley. ISBN: 020172152X.
- 52 *W3C Markup Validation Service*. 2007. World Wide Web Consortium. <http://validator.w3.org/>
- 53 Wulf, W. A Case Against the GOTO. 1972. In *Proceedings of the 25th ACM National Conference 2*: 791–797. New York, NY: ACM Press.

Предметный указатель

A

API 18
экспортируемый 28
assert 285
AutoValue 81; 84; 90

D

double 331

E

EnumMap 221
EnumSet 218

F

float 331

G

Google 16; 21; 81; 90; 331; 416

I

instanceof 77; 164

J

JavaBean 36; 356
Javadoc 313; 319; 372
Java Native Interface 348
JSON 416

O

Optional<T> 307

S

synchronized 385; 402

T

transient 424
try-c-ресурсами 64

V

volatile 384

A

Автоматическая упаковка. См.
Упаковка автоматическая

Адаптер 52; 141; 154

Активное ожидание 410

Аннотация 27; 230

FunctionalInterface 256

Nullable 284

Override 80; 239

SafeVarargs 192

SupressWarnings 167

ThreadSafe 404

безопасности потоков 403

маркер 231; 242

метааннотация 230

процессор 231

синтетическая 237

B

Безопасная публикация 386

Библиотека 328

Битовое поле 218

Взаимоисключение 381

Внедрение зависимостей 48

Г

Гаджет 414

Голодание 400

Д

Декартово произведение 264

Делегирование 130

Десериализация 413

бомба 414

- фильтр 417
 - черный и белый списки 417
- Документирующий
 - комментарий 313
- Доступность 110
- З**
- Загрязнение кучи 177; 191
- Задание 395
- Затирание 169
- Защитное копирование 53; 434
- И**
- Идиома
 - двойной проверки 407
 - дублирования массива 94
 - закрытого объекта
 - блокировки 404; 405
 - имитация множественного
 - наследования 142
 - имитация собственного типа 40; 182
 - класса-оболочки 140
 - класса отложенной
 - инициализации 407
 - обеспечения
 - неинстанцируемости 46
 - обхода элементов 322
 - однократной проверки 408
 - трансляции исключений 286; 370
 - цикла ожидания 400
- Импорт
 - статический 148
- Инициализация
 - отложенная 85; 123; 406
- Инкапсуляция 109
- Интерфейс 27; 138; 342
 - AutoCloseable 60; 64
 - Callable 395
 - Cloneable 90
 - Comparable 100
 - Iterable 327
 - Runnable 395
 - Serializable 418
 - констант 147
 - маркер 242
 - ограниченный 243
 - обратного вызова 388
 - провайдера службы 32
 - службы 32
 - статический метод 31
 - функциональный 246; 253; 254
 - BinaryOperator 254
 - Consumer 254
 - Function 254
 - Predicate 253
 - Supplier 254
 - UnaryOperator 254
- Исключение 359
 - ConcurrentModificationException 368
 - IllegalArgumentException 368
 - IllegalStateException 368
 - IndexOutOfBoundsException 368
 - NullPointerException 368
 - UnsupportedOperationException 368
 - времени выполнения 362
 - документация 372
 - информация о сбое 374
 - мультиперехват 390
 - проверяемое 362; 365
 - трансляция 370
 - цепочка 371
- К**
- Канонический вид 78
- Каркас
 - AutoValue 87
 - Java Collections Framework 31; 127; 140
 - JUnit 229
 - внедрения зависимостей 33; 49
 - исполнителей 394

- микротестирования
 - производительности 352
 - на основе интерфейсов 31
 - на основе классов 344
 - провайдера службы 32; 347
 - с обратным вызовом 130
- Кеш 56
- Класс 27
 - BigDecimal 333
 - Error 368
 - Exception 364; 368
 - Optional<T> 307
 - RuntimeException 368
 - Throwable 368
 - абстрактный 138; 241
 - анонимный 155; 245
 - вложенный 152
 - внутренний 153
 - вспомогательный 293
 - зависимый от константы 210
 - значения 68; 88; 343
 - локальный 155
 - неизменяемый 37; 117
 - не связанные классы 301
 - оболочка 129
 - передачи 128
 - с дескрипторами 150
 - сериализованная форма 418
 - служебный 46
 - с управлением экземплярами 30
 - эквивалентности 69
- Ключ 197
- Ковариантное типизирование
 - возврата 42
- Код операции 226
- Коллектор 266
 - нисходящий 269
- Комбинаторный взрыв 140
- Композиция 128
- Компонент 26
- Конвейер потока 257
- Конструктор
 - телескопический 35
- Контейнер 197
 - гетерогенный 198
- Контракт 313; 373
- Копия
 - защитная 289
- Л**
- Ленивая инициализация 51
- Логическая эквивалентность 68
- Ложное пробуждение 401
- Локальность ссылок 279
- Лямбда-выражение 246
- М**
- Массив 27; 112; 168
- Метааннотация 230
- Метод 27; 283
 - clone 90
 - compareTo 100
 - equals 67
 - hashCode 81
 - notify 396
 - toString 87
 - wait 396
 - агрегации 33
 - атомарный по отношению
 - к себе 376
 - доступа 115; 356
 - именование 292
 - перегрузка 294
 - передачи 128
 - перекрытый 295
 - переменной арности 302
 - по умолчанию 138; 144
 - преобразования типа 33
 - проверка параметров 284
 - советы по проектированию 292
 - с переменным количеством
 - аргументов 302
 - с переменным числом
 - параметров 191

- ссылка 250
- статический фабричный 29; 50
- установки 115
- чужой 387
- Миграционная совместимость 161
- Миксин 139
- Модель
 - памяти 382
 - производительности 353
- Модуль 28; 113
- Н**
- Наследование 28; 125
 - интерфейса 125
 - реализации 125
- О**
- Обратный вызов 388
- Объект 27
 - перемещение 291
 - пул 53
 - фактически неизменяемый 386
 - функциональный 155; 245
- Окно уязвимости 289
- Оптимизация 350
- Открытость 110
- Открытый вызов 392
- Отложенная инициализация 51
- Отложенные вычисления 257
- Очиститель 57
- Ошибка 362; 363
- П**
- Падение живучести 383
- Параллельная коллекция 391; 396
- Переменная
 - локальная 321
 - область видимости 322
 - цикла 322
- Перечисление 27; 203
 - стратегия 213
- Побочный эффект 314
- Поверхность атаки 414
- Подавление исключения 65
- Постусловие 313
- Поток
 - выполнения 277
 - приоритет 411
 - пул 395
 - данных 257
 - параллелизация 277
 - конвейер 257
- Предикат 144
- Предусловие 313; 373
 - нарушение 363
- Принцип
 - атомарности сбоев 284; 286
 - взять и отдать 186
 - инкапсуляции 133
 - подстановки Лисков 74; 112
 - сокрытия информации 350
- Проблема самоидентификации 130
- Проектный шаблон
 - Адаптер 141; 154
 - Декоратор 130
 - интерфейс констант 147
 - Класс-оболочка 137
 - Мост 33
 - Наблюдатель 387
 - Обобщенная фабрика
 - синглтонов 180
 - Приспособленец 30; 31
 - Прокси сериализации 434; 440
 - Синглтон 435
 - Стратегия 246
 - Строитель 38; 42; 293
 - Фабричный метод 29; 49
 - Шаблонный метод 140; 252
- Профилирование 352
- Пул
 - объектов 53

Р

- Реализация 28
 - по умолчанию 144
 - скелетная 136; 140; 370
- Рефакторинг 18
- Рефлексивность 69; 102
- Рефлексия 344

С

- Сериализация 413
- Сигнатура 27
- Симметричность 70; 102
- Синглтон 30; 43; 435
 - обобщенная фабрика 45
 - одноэлементное перечисление 45
- Синхронизатор 398
- Синхронизация 381; 384
- Служба исполнителей 395
- Сокращенная операция 279
- Соккрытие информации 109
- Ссылка
 - на метод 250
 - неточная 300
 - слабая 57
- Строка 338
 - конкатенация 341
- Схема именования 229; 353

Т

- Тип
 - double 331
 - float 331
 - String 338
 - агрегатный 338
 - вывод 246
 - доступный при выполнении 169
 - захват 189
 - инвариантный 183
 - ковариантный 93
 - несформированный 160
 - обобщенный 159
 - ограниченный подстановочный 49
 - параметризованный 159

- перечисления 203
- примитивный 334
- упакованный 334
- рекурсивное ограничение 181
- содержащий аннотацию 236
- с рекурсивным параметром
 - типа 40
 - ссылочный 27; 334
 - функциональный 245
- Транзитивность 71; 102

У

- Упаковка
 - автоматическая 52
- Упорядочение
 - естественное 100
 - лексикографическое 106
- Управления доступом 110
- Уровни безопасности 403
- Устаревшая ссылка 55
- Утечка
 - абстракции 190
 - памяти 54; 55; 56; 154

Ф

- Фабрика 49
- Финализатор 57
 - атака 59
- Функция
 - классификатор 269
 - чистая 265

Х

- Хеш-функция 83

Ц

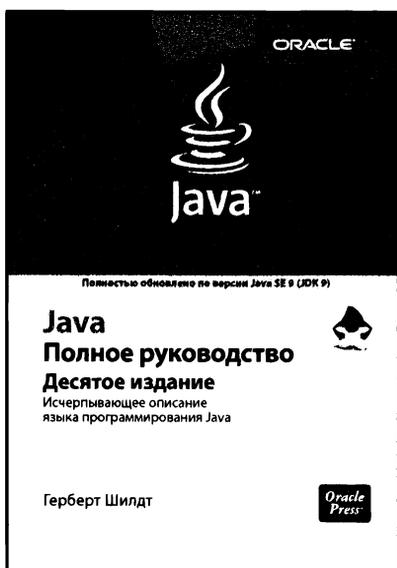
- Цикл
 - for 323
 - расширенный 325
 - по коллекции 325

Ч

- Чистая функция 265

JAVA ПОЛНОЕ РУКОВОДСТВО ДЕСЯТОЕ ИЗДАНИЕ

Герберт Шилдт



www.dialektika.com

Эта книга является исчерпывающим справочным пособием по языку программирования Java, обновленным с учетом последней версии Java SE 9. В удобной и легко доступной для изучения форме в ней подробно рассматриваются все языковые средства Java, в том числе синтаксис, ключевые слова, операции, управляющие и условные операторы, элементы объектно-ориентированного программирования (классы, объекты, методы, обобщения, интерфейсы, пакеты, коллекции), апплеты и сервлеты, библиотеки классов наряду с такими нововведениями, как модули и утилита JShell. Основные принципы и методики программирования на Java представлены на многочисленных и наглядных примерах написания программ. Книга рассчитана на широкий круг читателей, интересующихся программированием на Java.

ISBN 978-5-6040043-6-4

в продаже

JAVA: РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ 7-Е ИЗДАНИЕ

Герберт Шилдт



www.dialektika.com

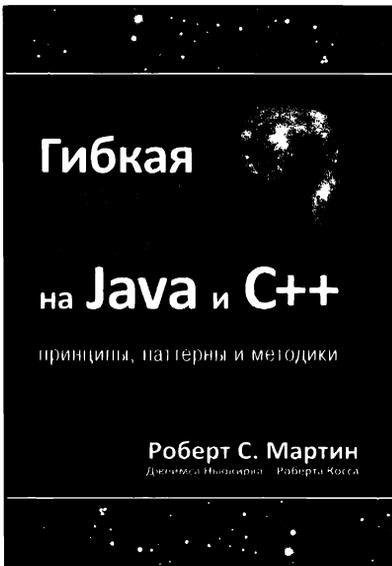
Очередное издание бестселлера, обновленное с учетом всех новинок Java Platform, Standard Edition 9 и 10 (Java SE 9 и 10), позволит читателям в кратчайшие сроки приступить к программированию на языке Java. Опытнейший автор Герберт Шилдт уже в начале книги познакомит читателей с тем, как создаются, компилируются и выполняются программы, написанные на Java. Далее обсуждаются ключевые слова, синтаксис и языковые конструкции, составляющие основу Java. Также будут рассмотрены темы повышенной сложности, включая многопоточное программирование, обобщения, лямбда-выражения, Swing, JavaFX и ключевое нововведение Java SE 9 — модули. В качестве бонуса читателей ждет знакомство с JShell — новой интерактивной оболочкой Java.

ISBN 978-5-6041394-5-5

в продаже

ГИБКАЯ РАЗРАБОТКА ПРОГРАММ НА JAVA И C++: ПРИНЦИПЫ, ПАТТЕРНЫ И МЕТОДИКИ

*Роберт Мартин, при
участии Джеймса Ньюкирка
и Роберта Косса*



www.dialektika.com

Будучи написанной разработчиками для разработчиков, книга содержит уникальный набор актуальных методов разработки программного обеспечения. В ней рассматриваются объектно-ориентированное проектирование, UML, паттерны, приемы гибкого и экстремального программирования, а также приводится детальное описание полного процесса проектирования для многократно используемых программ на C++ и Java. С применением практического подхода к решению задач в книге показано, как разрабатывать объектно-ориентированное приложение — от ранних этапов анализа и низкоуровневого проектирования до этапа реализации. Читатели ознакомятся с мыслями разработчика — здесь представлены ошибки, тупики и творческие идеи, которые возникают в процессе проектирования программного обеспечения. В книге раскрываются такие темы, как статика и динамика, принципы проектирования с использованием классов, управление сложностью, принципы проектирования с применением пакетов, анализ и проектирование, паттерны и пересечение парадигм.

ISBN 978-5-9908462-8-9

в продаже

АЛГОРИТМЫ НА JAVA

4-Е ИЗДАНИЕ

**Роберт Седжвик,
Кевин Уэйн**



www.williamspublishing.com

Последнее издание из серии бестселлеров Седжвика, содержащее самый важный объем знаний, наработанных за последние несколько десятилетий. Содержит полное описание структур данных и алгоритмов для сортировки, поиска, обработки графов и строк, включая пятьдесят алгоритмов, которые должен знать каждый программист. В книге представлены новые реализации, написанные на языке Java в доступном стиле модульного программирования — весь код доступен пользователю и готов к применению. Алгоритмы изучаются в контексте важных научных, технических и коммерческих приложений. Клиентские программы и алгоритмы записаны в реальном коде, а не на псевдокоде, как во многих других книгах. Дается вывод точных оценок производительности на основе соответствующих математических моделей и эмпирических тестов, подтверждающих эти модели.

ISBN 978-5-8459-2049-2

в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA SE 8

ПОДРОБНОЕ ОПИСАНИЕ, 5-Е ИЗДАНИЕ

*Джеймс Гослинг,
Билл Джой,
Гай Стил,
Гилад Брача,
Алекс Бакли*



Книга написана разработчиками языка Java и является полным техническим справочником по этому языку программирования. Она обеспечивает полный, точный и подробный охват всех аспектов языка программирования Java. В ней полностью описаны новые возможности, добавленные в Java SE 8, включая лямбда-выражения, ссылки на методы, методы по умолчанию, аннотации типов и повторяющиеся аннотации. В книгу также включено множество поясняющих примечаний. В ней аккуратно обозначены отличия формальных правила языка от практического поведения компиляторов.

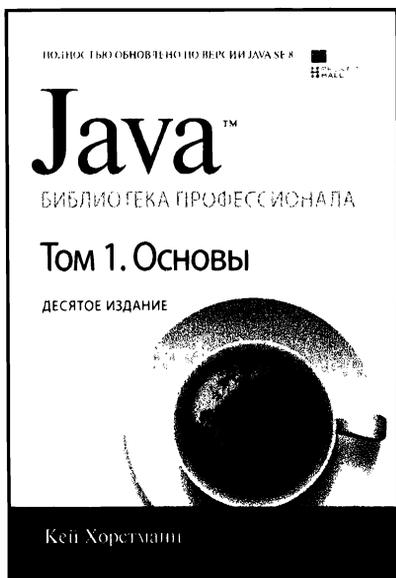
www.williamspublishing.com

ISBN 978-5-8459-1875-8

в продаже

JAVA® БИБЛИОТЕКА ПРОФЕССИОНАЛА Том 1. Основы *Десятое издание*

Кей Хорстманн



www.williamspublishing.com

Это первый том обновленного, десятого издания исчерпывающего справочного руководства по программированию на Java с учетом всех нововведений в версии Java SE 8. В нем подробно рассматриваются основы программирования на Java, в том числе основные типы и фундаментальные структуры данных, принципы объектно-ориентированного программирования и его реализация в Java, обобщения, коллекции, интерфейсы, лямбда-выражения и функциональное программирование, построение графических пользовательских интерфейсов средствами библиотеки Swing, обработка событий и исключений, развертывание приложений и апплетов, отладка программ, а также параллельное программирование. Излагаемый материал дополняется многочисленными примерами кода, которые не только иллюстрируют основные понятия, но и демонстрируют практические приемы программирования на Java.

Книга рассчитана на программистов разной квалификации и будет также полезна студентам и преподавателям дисциплин, связанных с программированием на Java.

ISBN 978-5-8459-2084-3 **в продаже**

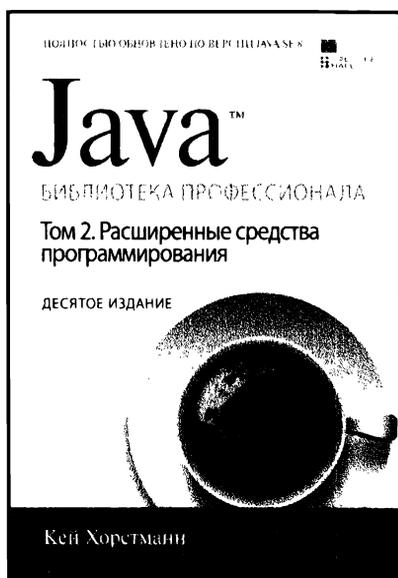
JAVA™

БИБЛИОТЕКА ПРОФЕССИОНАЛА

Том 2. РАСШИРЕННЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ

Десятое издание

Кей Хорстманн



www.williamspublishing.com

Это второй том обновленного, десятого издания исчерпывающего справочного руководства по программированию на Java с учетом всех нововведений в версии Java SE 8. В этом томе подробно рассматриваются расширенные средства программирования на Java, в том числе потоки данных, файловый ввод-вывод, XML, манипулирование датами и отметками времени, сетевое программирование и базы данных, интернационализация прикладных программ, расширенные функциональные возможности библиотек Swing и AWT, обеспечение безопасности, обработка аннотаций, оперирование платформенно-ориентированными методами. Излагаемый материал дополняется многочисленными примерами кода, которые не только иллюстрируют поясняемые понятия, но и демонстрируют практические приемы усовершенствованного программирования на Java.

Книга рассчитана на программистов разной квалификации и будет также полезна студентам и преподавателям дисциплин, связанных с программированием на Java.

ISBN 978-5-9909445-0-3

в продаже

Полное руководство по передовым методикам программирования на современных платформах Java!

Язык программирования Java существенно изменился со времени предыдущего издания книги, опубликованного вскоре после выпуска Java 6. Этот классический труд тщательно обновлен, чтобы читатели могли в полной мере воспользоваться возможностями последних версий языка и его библиотек функций. В современном Java поддерживается несколько парадигм программирования. Поэтому программисты часто испытывают потребность в конкретных рекомендациях, которые и описаны в данной книге.

Как и в предыдущих изданиях, каждая глава книги состоит из ряда разделов, в каждом из которых описаны конкретные советы, приведены тонкости платформы Java и содержатся обновленные примеры кода. Для каждой темы приводятся всеобъемлющее описание и пояснения, как следует поступить в данном случае, как не следует и почему.

Третье издание охватывает особенности языка программирования и библиотек, появившихся в Java 7, 8 и 9, в том числе конструкции функционального программирования, добавленные к своим объектно-ориентированным корням. В книгу включены также многие новые советы и глава, посвященная лямбда-выражениям и потокам.

Основные темы книги

- Функциональные интерфейсы, лямбда-выражения, ссылки на методы и потоки
- Методы по умолчанию и статические методы в интерфейсах
- Вывод типа, включая оператор "ромб" (<>) для обобщенных типов
- Аннотация @SafeVarargs
- Конструкция try-c-ресурсами
- Новые возможности библиотек, такие как интерфейс Optional<T>, java.time и удобные фабричные методы для коллекций



Джошуа Блок — профессор в Университете Карнеги-Меллона. Ранее он был главным архитектором Java в Google, заслуженным инженером в Sun Microsystems и старшим системным дизайнером в Transarc. Он возглавлял разработку и реализацию многочисленных возможностей платформы Java, включая усовершенствования в JDK 5.0 и инфраструктуру коллекций Java. Джошуа получил докторскую степень в области компьютерных наук в Университете Карнеги-Меллона и степень бакалавра наук в области компьютерных наук в Колумбийском университете.



www.williamspublishing.com

 **Pearson**
Addison-Wesley

ISBN 978-5-6041394-4-8



9 785604 139448