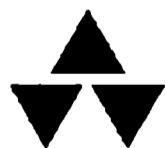


Effective JavaTM

Programming Language Guide

Second Edition

Joshua Bloch



ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Effective Java™ Programming Language Guide
Second Edition
Joshua Bloch

Java™ Эффективное программирование
Второе издание
Джошуа Блох

Переводчик Е. Коротылев
Верстка Т. Середа

Copyright © 2008 Sun Microsystems, Inc.
4150 Network Circle,
Santa Clara, California 95054, U.S.A.

Library of Congress control Number: 2008926278
ISBN-13: 978-0-321-3568-0
ISBN-10: 0-321-3568-3

© Издательство «Лори», 2014
Изд. № : OAI (03)
ЛР № : 070612 30.09.97 г.
ISBN 978-5-85582-348-6

Подписано в печать 05.01.2014. Формат 70x100/16
Бумага офсет № 1. Гарнитура Академическая. Печать офсетная
Печ. л. 29. Тираж 200

Благодарности ко второй редакции

 благодарю читателей первой редакции за то, что приняли ее с энтузиазмом, за то, что близко к сердцу приняли мои идеи, за то, что сообщили мне о положительном влиянии на них и их работу. Я благодарю профессоров, использующих эту книгу в преподавании, а также инженеров, взявших ее на вооружение.

Я благодарю команду из Addison—Wesley за доброту, профессионализм, терпение и сохранение достоинства, несмотря на давление. Всегда сохранял спокойствие Грег Доунч — замечательный редактор и джентльмен. Руководитель производства Джули Нахил олицетворяла в себе все то, что должен представлять собой руководитель производства, — она была прилежна, быстра, организована и дружелюбна. Литературный редактор Барбара Вуд была достаточно щепетильна и с должным чувством вкуса.

Я вновь должен благодарить судьбу за то, что у меня была лучшая команда рецензентов, которую только можно представить, и я искренне благодарю каждого из них. Основная команда, люди, которые просмотрели каждую главу, — это Лекси Боер (Lexi Bouger), Синди Блох (Cindy Bloch), Бет Боттос (Beth Bottos), Джо Бой-

бир (Joe Bowbeer), Брайан Гоутс (Brian Goetz), Тим Хэлоран (Tim Halloran), Брайан Кернхэм (Brian Kernigham), Роб Конингсберг (Rob Koningsberg), Тим Пирлс (Tim Peierls), Билл Пу (Bill Pough), Йошики Шибата (Yoshiki Shibata), Питер Старт (Peter Stout), Питер Вайнбергер (Peter Weinberger) и Фрэнк Яеллин (Frank Yellin). Другие рецензенты — Пабло Беллер (Pablo Bellver), Дэн Блох (Dan Bloch), Дэн Борнстайн (Dan Bornstein), Кевин Бурриллион (Kevin Bourrillion), Мартин Буххольц (Martin Buchholz), Джо Дарси (Joe Darcy), Нил Гафтер (Neil Gafter), Лоренс Гонсалвес (Laurence Gonsalves), Аарон Гринхаус (Aaron Greenhouse), Барри Хейс (Barry Hayes), Питер Джоунс (Peter Jones), Анджелика Лангер (Angelika Langer), Даг Ли (Doug Lee), Боб Ли (Bob Lee), Джерими Менсон (Jeremy Manson), Том Мэй (Tom May), Майк Мак Клоски (Mike McCloskey), Андрей Терещенко (Andriy Tereshchenko) и Пол Тима (Paul Tuma). Эти рецензенты внесли много предложений, приведших к улучшению этой книги, и избавили меня от многих неловких ситуаций. Тем не менее, если я и оказываюсь где-то сейчас в неловком положении, то это исключительно моя ответственность.

Особая благодарность Дагу Ли и Тому Пирлсу, которые озвучили многие идеи, отраженные в этой книге. Даг и Тим щедро и безоговорочно жертвовали своим временем и делились своими знаниями.

Я благодарю менеджера Google Прабху Кришну (Prabha Krishna) за ее бесконечную поддержку и содействие.

И наконец я благодарю свою жену, Синди Блох, за то, что она вдохновила меня на написание этой книги и прочитала каждую статью в необработанном виде, за помощь мне в работе с программой FrameMaker, за написание предметного указателя и просто за то, что терпела меня в процессе написания всей книги.

Благодарности к первой редакции

Я благодарю Патрика Чана (Patrick Chan) за то, что он посоветовал мне написать эту книгу, а также подбросил идею Лайзе Френдли (Lisa Friendly), главному редактору серии, Тима Линдхолма (Tim Lindholm), технического редактора серии, и Майка Хендрикsonа (Mike Hendrickson), исполнительного редактора издательства Addison—Wesley Professional. Благодарю Лайзу, Тима и Майка за их поддержку при реализации этого проекта, за сверхчеловеческое терпение и несгибаемую веру в то, что когда-нибудь я напишу эту книгу.

Я благодарю Джеймса Гослинга (James Gosling) и его незаурядную команду за то, что они дали мне нечто значительное, о чем можно написать, а также многих разработчиков платформы Java, последовавших стопами Джеймса. В особенности я благодарен моим коллегам по работе в компании Sun из Java Platform Tools and Libraries Group за понимание, одобрение и поддержку. В эту группу входят Эндрю Беннетт (Andrew Bennett), Джо Дарси (Joe Darcy), Нил Гафтер (Neal Gafter), Айрис Гарсия (Iris Garcia), Константин Кладко (Konstantin Kladko), Иена Литтл (Ian Little), Майк Макло-

ски (Mike McCloskey) и Марк Рейнхольд (Mark Reinhold). Среди бывших членов группы: Дзенгуа Ли (Zhenghua Li), Билл Мэддоx (Bill Maddox) и Нейвин Санджиева (Naveen Sanjeeva).

Я благодарю моего руководителя Эндрю Беннетт (Andrew Bennett) и директора Ларри Абрахамса (Larry Abrahams) за полную и страстную поддержку этого проекта. Благодарю Рича Грина (Rich Green), вице-президента компании Java Software, за создание условий, когда разработчики имеют возможность творить и публиковать свои работы.

Мне чрезвычайно повезло с самой лучшей, какую только можно вообразить, группой рецензентов, и я приношу мои самые искренние благодарности каждому из них: Эндрю Беннетту (Andrew Bennett), Синди Блох (Cindy Bloch), Дэну Блох (Dan Bloch), Бет Ботос (Beth Bottos), Джо Баубиеру (Joe Bowbeer), Джиладу Браче (Gilad Bracha), Мэри Кампьон (Mary Campione), Джо Дарси (Joe Darcy), Дэвиду Экхардту (David Eckhardt), Джо Фиалли (Joe Fialli), Лайзе Френдли (Lisa Friendly), Джеймсу Гослингу (James Gosling), Петеру Хаггеру (Peter Haggar), Брайену Кернигану (Brian Kernighan), Константину Кладко (Konstantin Kladko), Дагу Ли (Doug Lea), Дзенгуа Ли (Zhenghua Li), Тиму Линдхолму (Tim Lindholm), Майку Маклоски (Mike McCloskey), Тиму Пейерлсу (Tim Peierls), Марку Рейнхольду (Mark Reinhold), Кену Расселу (Ken Russell), Биллу Шэннону (Bill Shannon), Петеру Статуту (Peter Stout), Филу Уодлеру (Phil Wadler), Давиду Холмсу (David Holmes) и двум анонимным рецензентам. Они внесли множество предложений, которые позволили существенно улучшить эту книгу и избавили меня от многих затруднений. Все оставшиеся недочеты полностью лежат на моей совести.

Многие мои коллеги, работающие в компании Sun и вне ее, участвовали в технических дискуссиях, которые улучшили качество этой книги. Среди прочих: Бен Гомес (Ben Gomes), Стеффен Граруп (Steffen Grarup), Петер Кесслер (Peter Kessler), Ричард Рода (Richard Roda), Джон Роуз (John Rose) и Дэвид Стартэмайер (David Stoutamire), давшие полезные разъяснения. Особая благо-

дарность Дагу Ли (Doug Lea), озвучившему многие идеи в этой книге. Даг неизменно щедро делился своим временем и знаниями.

Я благодарен Джули Дайниколе (Julie Dinicola), Джекки Дусетт (Jacqui Doucette), Майку Хендриксону (Mike Hendrickson), Хизер Ольщик (Heather Olszyk), Трейси Расс (Tracy Russ) и всем сотрудникам Addison—Wesley за их поддержку и профессионализм. Даже будучи до невозможности занятыми, они всегда были дружелюбны и учтивы.

Я благодарю Гая Стила (Guy Steele), написавшего Предисловие. Его участие в этом проекте — большая честь для меня.

Наконец, я благодарен моей жене Синди Блох (Cindy Bloch), которая своим ободрением, а подчас и угрозами помогла мне написать эту книгу. Благодарю за чтение каждой статьи в необработанном виде, за помощь при работе с программой Framemaker, написание предметного указателя и за то, что терпела меня, пока я писал эту книгу.

Содержание

<i>Предисловие</i>	xv
<i>Предисловие автора ко второй редакции</i>	xviii
<i>Предисловие автора ко второй редакции</i>	xx
1 Введение	1
2 Создание и уничтожение объектов	6
1 <i>Рассмотрите возможность замены конструкторов статическими методами генерации.....</i>	<i>6</i>
2 <i>Используйте шаблон <i>Builder</i>, когда приходится иметь дело с большим количеством параметров конструктора.....</i>	<i>5</i>
3 <i>Свойство синглтон обеспечивайте закрытым конструктором или типом перечислений.....</i>	<i>24</i>
4 <i>Отсутствие экземпляров обеспечивает закрытый конструктор.....</i>	<i>27</i>
5 <i>Избегайте ненужных объектов.....</i>	<i>28</i>

6	Уничтожайте устаревшие ссылки (на объекты)	34
7	Остерегайтесь методов <i>finalize</i>	38
3	Методы, общие для всех объектов	46
8	Переопределяя метод <i>equals</i> , соблюдайте общие соглашения	46
9	Переопределяя метод <i>equals</i> , всегда переопределяйте <i>hashCode</i>	62
10	Всегда переопределяйте метод <i>toString</i>	71
11	Соблюдайте осторожность при переопределении метода <i>clone</i>	74
12	Подумайте над реализацией интерфейса <i>Comparable</i>	86
4	Классы и интерфейсы	94
13	Сводите к минимуму доступность классов и членов	94
14	В открытых классах используйте методы доступа, а не открытые поля	100
15	Предпочитайте постоянство	102
16	Предпочитайте компоновку наследованию	114
17	Проектируйте и документируйте наследование либо запрещайте его	122
18	Предпочитайте интерфейсы абстрактным классам .	130
19	Используйте интерфейсы только для определения типов	138
20	Объединение заменяйте иерархией классов	140
21	Используйте объект функции для выполнения сравнения	144

22	Предпочитайте статические классы-члены нестатическим	148
5	Средства обобщенного программирования (Generics)	154
23	Не используйте необработанные типы в новом коде	155
24	Избегайте предупреждений о непроверенном коде	163
25	Предпочитайте списки массивам	166
26	Поддерживайте обобщенные типы	173
27	Поддерживайте обобщенные методы	179
28	Используйте ограниченные групповые символы для увеличения гибкости API	186
29	Использование неоднородных контейнеров	195
6	Перечислимые типы и аннотации	203
30	Используйте перечислимые типы вместо констант <i>int</i>	203
31	Используйте поля экземпляра вместо числовых значений	218
32	Используйте <i>EnumSet</i> вместо битовых полей	219
33	Используйте <i>EnumMap</i> вместо порядкового индексирования	221
34	Имитируйте расширяемые перечислимые типы с помощью интерфейсов	227
35	Предпочитайте аннотации шаблонам присвоения имен	232

36	<i>Используйте аннотацию <code>Override</code> последовательно</i>	241
37	<i>Используйте маркерные интерфейсы для определения типов</i>	244
7	Методы	248
38	<i>Проверяйте достоверность параметров</i>	248
39	<i>При необходимости создавайте резервные копии</i>	252
40	<i>Тщательно проектируйте сигнатуру метода</i>	258
41	<i>Перезагружая методы, соблюдайте осторожность</i> ..	261
42	<i>Используйте <code>varargs</code> с осторожностью</i>	269
43	<i>Возвращайте массив нулевой длины, а не <code>null</code></i> ..	274
44	<i>Для всех открытых элементов API пишите doc-комментарии</i>	277
8	Общие вопросы программирования	286
45	<i>Сводите к минимуму область видимости локальных переменных</i>	286
46	<i>Предпочитайте использование цикла <code>for-each</code></i> ..	290
47	<i>Изучите библиотеки и пользуйтесь ими</i>	294
48	<i>Если требуются точные ответы, избегайте использования типов <code>float</code> и <code>double</code></i>	298
49	<i>Отдавайте предпочтение использованию обычных примитивных типов, а не упакованных примитивных типов</i>	301
49	<i>Не используйте строку там, где более уместен иной тип</i>	306
50	<i>При конкатенации строк опасайтесь потери производительности</i>	310

51 Для ссылки на объект используйте его интерфейс.....	311
52 Предпочитайте интерфейс отражению класса....	314
53 Соблюдайте осторожность при использовании машинозависимых методов.....	319
54 Соблюдайте осторожность при оптимизации....	321
55 При выборе имен придерживайтесь общепринятых соглашений.....	325
9 Исключения	331
56 Используйте исключения лишь в исключительных ситуациях.....	331
57 Применяйте обрабатываемые исключения для восстановления, для программных ошибок используйте исключения времени выполнения...	335
58 Избегайте ненужных обрабатываемых исключений ..	338
59 Предпочитайте стандартные исключения.....	341
60 Инициируйте исключения, соответствующие абстракции.....	344
61 Для каждого метода документируйте все инициируемые исключения.....	347
62 В описание исключения добавляйте информацию о сбое.....	350
63 Добивайтесь атомарности методов по отношению к сбоям.....	353
64 Не игнорируйте исключений.....	356
10 Потоки	358
65 Синхронизируйте доступ потоков к совместно используемым изменяемым данным ..	358

66	<i>Избегайте избыточной синхронизации</i>	365
67	<i>Предпочитайте использование экзекуторов и заданий вместо потоков</i>	373
68	<i>Предпочитайте использовать утилиты параллельности, нежели wait и notify</i>	376
69	<i>При работе с потоками документируйте уровень безопасности</i>	383
70	<i>С осторожностью используйте отложенную инициализацию</i>	388
71	<i>Не попадайте в зависимость от планировщика потоков</i>	393
72	<i>Избегайте группировки потоков</i>	396
11	Сериализация	399
73	<i>Соблюдайте осторожность при реализации интерфейса Serializable</i>	399
74	<i>Рассмотрите возможность использования специализированной сериализованной формы</i>	407
75	<i>Метод readObject должен создаваться с защитой</i>	416
76	<i>Для контроля над экземплярами предпочтайте использование перечислимых типов методу readResolve</i>	424
77	<i>Рассмотрите использование агентов сериализации вместо сериализованных экземпляров</i>	430
	Список литературы	435

Предисловие

Если бы сослуживец сказал вам: «Моя супруга сегодня вечером производит дома необычный обед. Придешь?», то вам в голову, вероятно, придут сразу три мысли: вас уже пригласили на обед, во-вторых, ваш сослуживец явно иностранец, ну и, прежде всего, вы будете крайне озадачены.

Если вы сами когда-нибудь изучали второй язык, а затем пробовали им пользоваться за пределами аудитории, то уже знаете, что есть три вещи, которые необходимо знать: каким образом структурирован изучаемый язык (грамматика), как называются вещи, о которых вы хотите сказать (словарь), а также общепринятые и эффективные варианты разговора о повседневных вещах (лексические обороты). В аудитории слишком часто ограничиваются изучением лишь первых двух из этих вещей, и вы обнаруживаете, что носители языка постоянно давятся от смеха, выслушивая, как вы пытаетесь, чтобы вас поняли.

С языком программирования дело обстоит практически так же. Вам необходимо понимать суть языка: является он алгоритмическим, функциональным или объектно-ориентированным. Вам нужно знать словарь языка: какие структуры данных, операции и возможности предоставляют стандартные библиотеки. Вам необходимо также ознакомиться с общепринятыми и эффективными способами структурирования вашего кода. В книгах, посвященных языкам про-

граммирования, часто освещаются лишь первые два вопроса, приемы работы с языком если и обсуждаются, то лишь кратко. Возможно, это происходит потому, что о первых двух вещах писать несколько проще. Грамматика и словарь — это свойства самого языка, тогда как способ его использования характеризует группу людей, которая этим языком пользуется.

Например, язык программирования Java — это объектно-ориентированный язык с единичным наследованием, обеспечивающим для каждого метода императивный (ориентированный на действия) стиль программирования. Его библиотеки ориентированы на поддержку графических дисплеев, работу с сетью, распределенные вычисления и безопасность. Однако как наилучшим образом использовать этот язык на практике?

Есть и другой аспект. Программы, в отличие от произнесенных фраз, а также большинства книг и журналов, имеют возможность меняться со временем. Обычно недостаточно создать программный код, который эффективно работает и без труда может быть понят другими людьми. Нужно еще организовать этот код таким образом, чтобы его можно было легко модифицировать. Для некоторой задачи A может быть десяток вариантов написания программного кода. Из этих десяти семь окажутся неуклюжими, неэффективными или запутывающими читателя. Какой же из оставшихся трех вариантов вероятнее всего будет похож на программный код, который потребуется в следующем году для новой версии программы, решающей задачу A'?

Есть много книг, по которым можно изучать грамматику языка программирования Java, в том числе книги «*The Java™ Programming Language*» авторов Arnold, Gosling и Holmes [Arnold05] или «*The Java™ Language Specification*» авторов Gosling, Joy, Bracha и вашего покорного слуги [JLS]. Точно так же есть множество книг, посвященных библиотекам и прикладным интерфейсам, которые связаны с языком Java.

Эта книга посвящена третьей теме: общепринятым и эффективным приемам работы с языком Java. Джошуа Блох (Joshua Bloch)

провел несколько лет в компании Sun Microsystems, работая с языком программирования Java, занимаясь расширением и реализацией программного кода. Он также прочел большое количество программного кода, написанного многими людьми, в том числе и мной. Здесь же, приведя в некую систему, он дает дальние советы о том, каким образом структурировать ваш код, чтобы он работал хорошо, чтобы его смогли понять другие люди, чтобы последующие модификации и усовершенствования доставляли меньше головной боли и даже, возможно, чтобы ваши программы были приятными, элегантными и красивыми.

Гай Л. Стил-младший (Guy L. Steele Jr.)

Берлингтон, шт. Массачусетс

Апрель 2001 г.

Предисловие автора ко второй редакции

С тех пор как я написал первую редакцию этой книги, произошло много изменений в платформе Java, и я решил, что давно пора уже написать вторую редакцию. Наиболее значимыми изменениями стали добавление обобщений, типов перечислений, комментариев, автоматического создания контейнеров, циклов типа `for-each` в Java 5. Также новшеством стало добавление новой библиотеки совместимости `java.util.concurrent`, также появившейся в Java 5. Мне повезло, что вместе с Джиладом Брачей я смог возглавить команду, разработавшую новые особенности языка. Мне также повезло в том, что удалось работать в команде, возглавляемой Дагом Ли и разработавшей библиотеку совместимости.

Другим значительным изменением в платформе стало ее усвоение современными интегрированными средами разработки, такими как Eclipse, IntelliJ IDEA и NetBeans, и инструментами статического анализа, как, например, FindBugs. Я не принимал участия в этом процессе, однако смог извлечь из этого огромную выгоду и узнал, как они влияют на опыт разработки.

В 2004 году я перешел из компании Sun в компанию Google, однако продолжал принимать участие в разработке платформы Java

в течение последних четырех лет, помогая в разработке взаимосовместимости и коллекций программных интерфейсов, используя офисы Google, а также Java Community Process. Я также имел удовольствие разрабатывать библиотеки для использования в Google. Теперь я знаю, что такое быть пользователем.

Когда в 2001 году я писал первую редакцию книги, моей основной целью было поделиться с вами моим опытом, чтобы вы смогли повторить мои успехи и избежать моих неудач. Новый материал также приводит реальные примеры из библиотек платформы Java.

Успех первой редакции превзошел все мои ожидания, и я сделал все возможное, чтобы сохранить дух предыдущей редакции, освещая новый материал, что требовалось для обновления данной книги. Невозможно было избежать того, что книга стала больше, — и она действительно стала. Вместо 57 в ней 78 статей. И я не просто добавил 23 новые статьи, а тщательно переработал весь оригинальный вариант — и удалил некоторые статьи, которые уже просто неактуальны. В приложении вы можете увидеть, как соотносится материал этой редакции с материалом первой редакции.

В предисловии к первой редакции я написал, что язык программирования Java и его библиотеки очень способствуют качеству и производительности и как с ними здорово работать. Изменения в 5-м и 6-м выпусках сделали их еще лучше. Сейчас платформа гораздо больше и сложнее, чем в 2001 году, но, когда вы познакомитесь с ее идиоматикой и примерами использования новых возможностей, это позволит сделать ваши программы лучше и сделает вашу жизнь легче. Надеюсь, что эта редакция передаст вам мой энтузиазм и поможет вам более эффективно и с большим удовольствием использовать платформу и ее новые возможности.

Сан-Хосе, шт. Калифорния
Апрель 2008 г.

Предисловие автора к первой редакции

В 1996 году я направился на запад работать в компании JavaSoft, как она тогда называлась, поскольку было очевидно, что именно там происходят главные события. На протяжении пяти лет я работал архитектором библиотек для платформы Java. Я занимался проектированием и разработкой множества таких библиотек, занимался их обслуживанием, а также давал консультации по многим другим библиотекам. Контроль над этими библиотеками в ходе становления платформы языка Java — такая возможность предоставляется только раз в жизни. Не будет преувеличением сказать, что я имел честь работать с некоторыми великими разработчиками нашего времени. В процессе работы я многое узнал о языке программирования Java: что там работает, а что нет, как пользоваться языком и его библиотеками для получения наилучшего результата.

Эта книга является попыткой поделиться с вами моим опытом, чтобы вы смогли повторить мои успехи и избежать моих неудач. Оформление книги я позаимствовал из руководства Скотта Мейерса (Scott Meyers) «*Effective C++*» [Meyers98], которое состоит из 50 статей, каждая из которых посвящена одному конкретному правилу, позволяющему улучшить ваши программы и проекты. Я на-

шёл такое оформление необычайно эффективным, и, надеюсь, вы тоже его оцените.

Во многих случаях я осмелился иллюстрировать статьи реальными примерами из библиотек для платформы Java. Говоря, что нечто можно сделать лучше, я старался брать программный код, который я писал сам, однако иногда я брал разработанное коллегами. Приношу мои искренние извинения, если, несмотря на все старания, кого-либо при этом обидел. Негативные примеры приведены не для того, чтобы кого-то опорочить, а с целью сотрудничества, чтобы все мы могли извлечь пользу из опыта тех, кто уже прошел этот путь.

Хотя эта книга предназначена не только для людей, занимающихся разработкой повторно используемых компонент, она неизбежно отражает мой опыт в написании таких, накопленный за последние два десятилетия. Я привык думать в терминах прикладных интерфейсов (API) и предлагаю вам делать то же. Даже если вы не занимаетесь разработкой повторно используемых компонент, если вы будете пользоваться этими терминами, это может повысить качество написанных вами программ. Более того, нередко случается писать многократно используемые компоненты, даже не подозревая об этом: вы написали нечто полезное, поделились своим результатом с приятелем, и вскоре у вас будет уже с полдюжины пользователей. С этого момента вы лишаетесь возможности свободно менять этот API и получаете благодарности за все те усилия, которые вы потратили на его разработку, когда писали эту программу в первый раз.

Мое особое внимание к разработке API может показаться несколько противоестественным для ярых приверженцев новых облегченных методик разработки программного обеспечения, таких как «Экстремальное программирование» [Beck99]. В этих методиках особое значение придается написанию самой простой программы, которая только сможет работать. Если вы пользуетесь одной из этих методик, то обнаружите, что внимание к разработке API сослужит вам добрую службу в процессе последующей перестройки программы

(*refactoring*). Основной задачей перестроения является усовершенствование структуры системы, а также исключение дублирующего программного кода. Этой цели невозможно достичь, если у компонент системы нет хорошо спроектированного API.

Ни один язык не идеален, но некоторые — великолепны. Я обнаружил, что язык программирования Java и его библиотеки в огромной степени способствуют повышению качества и производительности труда, а также доставляют радость при работе с ними. Надеюсь, эта книга отражает мой энтузиазм и способна сделать вашу работу с языком Java более эффективной и приятной.

Купертино, шт. Калифорния
Апрель 2001 г.

1

Г л а в а

Введение

Эта книга писалась с тем, чтобы помочь вам наиболее эффективно использовать язык программирования Java и его основные библиотеки `java.lang`, `java.util` и, в меньшей степени, `java.util.concurrent` и `java.io`. Время от времени в книге затрагиваются и другие библиотеки, но мы не касаемся программирования графического интерфейса пользователя, специализированных API или мобильных устройств.

Книга состоит из семидесяти восьми статей, каждая из которых описывает одно правило. В этих статьях собран опыт, который самые лучшие и опытные программисты обычно считают полезным. Статьи произвольно разбиты на десять глав, каждая из которых касается того или иного обширного аспекта проектирования программного обеспечения. Нет необходимости читать эту книгу от корки до корки: каждая статья в той или иной степени самостоятельна. Статьи имеют множество перекрестных ссылок, поэтому вы можете с легкостью построить по этой книге ваш собственный учебный курс.

Многие новые возможности были добавлены в платформу Java 5 (релиз 1.5). Большинство статей этой книги в той или иной степени используют эти новые возможности. Следующая таблица иллюстрирует, где и какие новые возможности были освещены в данной книге:

Большинство статей иллюстрируются примерами программ. Главной особенностью этой книги является наличие в ней примеров

Новая возможность	Глава или статья, где она описана
Общности	Глава 5
Перечисления	Статьи 30–34
Аннотации	Статьи 35–37
Циклы for-each	Статья 46
Автоупаковка	Статьи 40, 49
Аргументы переменной длины Varargs	Статья 42
Статический импорт	Статья 19
Java.util.concurrent	Статьи 68, 69

программного кода, которые иллюстрируют многие шаблоны (*design pattern*) и идиомы. Где это необходимо, шаблоны и идиомы имеют ссылки на основные работы в этой области [Gamma95].

Многие статьи содержат один или несколько примеров программ, иллюстрирующих приемы, которых следует избегать. Подобные примеры, иногда называемые «антишаблонами», четко обозначе-

ны комментарием, таким как «Никогда так не делайте!». В каждом таком случае в статье дается объяснение, почему этот пример плох, и предлагается альтернатива.

Эта книга не предназначена для начинающих: предполагается, что вы уже хорошо владеете языком программирования Java. Если же это не так, обратитесь к одному из множества замечательных вводных текстов [Arnold05, Sestoft05]. Хотя эта книга построена так, чтобы она была доступна для любого, кто работает с этим языком, она должна давать пищу для размышлений даже опытным программистам.

Большинство правил этой книги берут начало от нескольких фундаментальных принципов. Ясность и простота имеют первостепенное значение. Функционирование модуля не должно вызывать удивление у его пользователя. Модули должны быть настолько компактны, насколько это возможно, но не более того. (В этой книге термин «модуль» относится к любому программному компоненту, который используется много раз, от отдельного метода до сложной системы, состоящей из нескольких пакетов.) Программный код следует использовать повторно, а не копировать. Взаимозависимость между модулями должна быть сведена к минимуму. Ошибку нужно выяв-

лять как можно ближе к тому месту, где она возникла, в идеале — уже на стадии компиляции.

Правила, изложенные в этой книге, не охватывают все 100% практики, в подавляющем большинстве случаев они описывают самые лучшие приемы программирования. Вам не следует покорно следовать этим правилам, но и нарушать их нужно нечасто, имея на то вескую причину. Как и для большинства других дисциплин, изучение искусства программирования заключается сначала в заучивании правил, а затем в изучении условий, когда они нарушаются.

Большая часть этой книги посвящена отнюдь не производительности программ. Речь идет о написании понятных, правильных, полезных, надежных, гибких программ, которые удобно сопровождать. Если вы сможете сделать это, то добиться необходимой производительности программ будет относительно просто (статья 55). В некоторых статьях обсуждаются вопросы производительности, а в нескольких даже приведены показатели производительности. Эти данные, предваряемые выражением «на моей машине», в лучшем случае следует рассматривать как приблизительные.

Для справки, моя машина — это старый компьютер домашней сборки с процессором 2.2 ГГц двухъядерный Opteron 170 с 2 Гбайт оперативной памяти под управлением Microsoft Windows XP Professional SP2, на котором установлен Java 1.6_05 Standard Edition Software Development Kit (SDK) компании Sun. В состав этого SDK входят две виртуальные машины — Java HotSpot Client VM (клиентская виртуальная машина) и Server VM (серверная виртуальная машина). Производительность измерялась на серверной машине.

При обсуждении особенностей языка программирования Java и его библиотек иногда возникает необходимость сослаться на конкретные версии. Для краткости в этой книге используются «рабочие», а не официальные номера версий. В таблице 1.1 показано соответствие между названиями версий и их рабочими номерами.

Данные примеры по возможности являются полными, однако предпочтение отдается не завершенности, а удобству чтения. В примерах широко используются классы пакетов `java.util` и `java.io`. Соот-

Таблица 1.1

Официальное название версии	Рабочий номер версии
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4
Java 2 Platform, Standard Edition, v 5.0	1.5
Java 2 Platform, Standard Edition, v 6.0	1.6

ветственно, чтобы скомпилировать пример, вам потребуется добавить один или более операторов `import`:

```
import java.util.*;
import java.util.concurrent.*;
import java.io.*;
```

В примерах опущены другие детали. На веб-сайте этой книги (<http://java.sun.com/docs/books/effective>) содержится полная версия каждого примера, которую можно откомпилировать и запустить.

Технические термины в этой книге большей частью используются в том виде, как они были определены в *The Java Language Specification [JLS]*. Однако некоторые термины заслуживают отдельного упоминания. Язык Java поддерживает четыре группы типов: интерфейсы (*interface*) (в том числе и аннотации (*annotations*)), классы (*class*) (в том числе и перечисления (*enums*)), массивы (*array*) и простые типы (*primitive*). Первые три группы называются ссылочными типами (*reference type*). Экземплярами классов и массивов являются объекты, значения простых типов таковыми не являются. Членами класса (*members*) являются его поля (*fields*), методы (*methods*), а также классы-члены (*member classes*) и интерфейсы-члены (*member interfaces*). Сигнатура метода (*signature*) состоит из его названия и типов, которые имеют его формальные параметры. Тип значения, которое возвращается этим методом, в сигнатуру не входит.

Некоторые термины в этой книге используются в ином значении, чем в *The Java Language Specification*.

В отличие от указанной спецификации наследование (*inheritance*) в этой книге используется как синоним образования подклассов (*subclassing*). Вместо того чтобы использовать для интерфейсов термин «наследование», в этой книге просто констатируется, что некий класс реализует (*implement*) интерфейс или что один интерфейс является расширением другого (*extend*). Чтобы описать уровень доступа, который используется, когда ничего больше не указано, в книге используется описательный термин «доступ только в пределах пакета» (*package-private*) вместо формально правильного термина «доступ по умолчанию» (*default access*) [JLS, 6.6.1].

В этой книге используются несколько технических терминов, которых нет в *The Java Language Specification*. Термин «внешний API» (*exported API*), или просто API, относится к классам, интерфейсам, конструкторам, членам и серийным формам, с помощью которых программист получает доступ к классу, интерфейсу или пакету. (Термин API, являющийся сокращением от *application programming interface* — программный интерфейс приложения, используется вместо термина «интерфейс» (*interface*), который следовало бы использовать в противном случае. Это позволяет избежать путаницы с одноименной конструкцией языка Java.) Программист, который пишет программу, использующую некий API, называется здесь пользователем (*user*) указанного API. Класс, в реализации которого используется некий API, называется клиентом (*client*) этого API.

Классы, интерфейсы, конструкторы, члены и серийные формы все вместе называются элементами API (*API element*). Внешний API образуется из элементов API, которые доступны за пределами пакета, где этот API был определен. Указанные элементы может использовать любой клиент, а автор этого API берет на себя их поддержку. Не случайно документацию именно к этим элементам генерирует утилита Javadoc, будучи запущена в режиме по умолчанию. Грубо говоря, внешний API пакета состоит из открытых (*public*) и защищенных (*protected*) членов, а также конструкторов всех открытых классов и интерфейсов в пакете.

Г л а в а 2

Создание и уничтожение объектов

В этой главе речь идет о создании и уничтожении объектов: как и когда создавать объекты, как и когда этого не делать, как сделать так, чтобы объекты гарантированно уничтожались своевременно, а также как управлять всеми операциями по очистке, которые должны предшествовать уничтожению объекта.

Статья 1

Рассмотрите возможность замены конструкторов статическими методами генерации

Обычно, чтобы разрешить клиенту получать экземпляр класса, ему предоставляется открытый (*public*) конструктор. Есть и другой, менее известный прием, который должен быть в арсенале любого программиста. Класс может иметь открытый *статический метод генерации* (*static factory method*), который является просто статическим методом, возвращающим экземпляр класса. Простой пример такого метода возьмем из класса Boolean (класса, являющего оболочкой для простого типа *boolean*). Этот метод преобразует простое значение *boolean* в ссылку на объект Boolean:

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

Обратите внимание, что статический метод генерации и порождающий шаблон из Шаблонов проектирования (Gamma95, с. 107) не есть одно и то же.

Статические методы генерации могут быть предоставлены клиентам класса не только вместо конструкторов, но и в дополнение к ним. Замена открытого конструктора статическим методом генерации имеет как достоинства, так и недостатки.

Первое преимущество статического метода генерации состоит в том, что, в отличие от конструкторов, он имеет название. Тогда как параметры конструктора сами по себе не дают описания возвращаемого объекта, статический метод генерации с хорошо подобранным названием может упростить работу с классом и, как следствие, сделать соответствующий программный код клиента более понятным. Например, конструктор BigInteger(int, int, Random), который возвращает BigInteger, вероятно являющийся простым числом (*prime*), лучше было бы представить как статический метод генерации с названием BigInteger.probablePrime. (В конечном счете этот статический метод был добавлен в версии 1.4.)

Класс может иметь только один конструктор с заданной сигнатурой. Известно, что программисты обходят это ограничение, создавая конструкторы, чьи списки параметров отличаются лишь порядком следования типов. Это плохая идея. Человек, использующий подобный API, не сможет запомнить, для чего нужен один конструктор, а для чего другой, и в конце концов по ошибке вызовет не тот конструктор. Люди, читающие программный код, в котором используются такие конструкторы, не смогут понять, что же он делает, если не будут сверяться с сопроводительной документацией к этому классу.

Поскольку статические методы генерации имеют имена, к ним не относится ограничение конструкторов, запрещающее иметь в классе более одного метода с заданной сигнатурой. Соответственно, в ситуациях, когда очевидно, что в классе нужно иметь несколько кон-

структур с одной и той же сигнатурой, вам следует рассмотреть возможность замены одного или нескольких конструкторов статическими методами генерации. Тщательно выбранные названия будут подчеркивать их различия.

Второе преимущество статических методов генерации заключается в том, что, в отличие от конструкторов, они не обязаны при каждом вызове создавать новый объект. Это позволяет использовать для неизменяемого класса (статья 15) предварительно созданные экземпляры либо кэшировать экземпляры класса по мере их создания, а затем раздавать их повторно, избегая создания ненужных дублирующих объектов. Подобный прием иллюстрирует метод Boolean.valueOf(boolean): он не создает объектов. Эта методика схожа с шаблоном Flyweight (Gamma95, с. 195). Она может значительно повысить производительность программы, если в ней часто возникает необходимость в создании одинаковых объектов, особенно в тех случаях, когда создание этих объектов требует больших затрат.

Способность статических методов генерации при повторных вызовах возвращать тот же самый объект позволяет классам в любой момент времени четко контролировать, какие экземпляры объекта еще существуют. Классы, которые делают это, называются классами контроля экземпляров (*instance-controlled*). Есть несколько причин для написания таких классов. Во-первых, контроль над экземплярами позволяет дать гарантию, что некий класс является синглтоном (статья 3) или что он является абстрактным (статья 4). Во-вторых, это позволяет убедиться в том, что у неизменяемого класса (статья 5) не появилось двух одинаковых экземпляров: `a.equals(b)` тогда и только тогда, когда `a==b`. Если класс дает такую гарантию, его клиенты вместо метода `equals(Object)` могут использовать оператор `==`, что может привести к существенному повышению производительности программы. Перечисления типов (статья 30) также дают такую гарантию.

Третье преимущество статического метода генерации заключается в том, что, в отличие от конструктора, он может возвращать

тить объект, который соответствует не только заявленному типу возвращаемого значения, но и любому его подтипу. Это дает вам значительную гибкость в выборе класса для возвращаемого объекта. Например, благодаря такой гибкости интерфейс API может возвращать объект, не декларируя его класс как public. Скрытие реализации классов может привести к созданию очень компактного API. Этот прием идеально подходит для конструкций, построенных на интерфейсах (статья 18), когда эти интерфейсы для статических методов генерации задают собственный тип возвращаемого значения. У интерфейсов не может быть статических методов, так что статические методы интерфейса с именем *Type* помещаются в абстрактный класс (статья 4) с именем *Types*, для которого нельзя создать экземпляр.

Например, архитектура Collections Framework имеет тридцать две полезные реализации интерфейсов коллекции: неизменяемые коллекции, синхронизированные коллекции и т.д. Большинство этих реализаций с помощью статических методов генерации сводятся в единственный класс (`java.util.Collections`), для которого невозможно создать экземпляр. Все классы, соответствующие возвращаемым объектам, не являются открытыми.

API Collections Framework имеет гораздо меньшие размеры, чем это было бы, если бы в нем были представлены тридцать два отдельных открытых класса для всех возможных реализаций. Сократился не просто объем этого API, но и его «концептуальная нагрузка». Пользователь знает, что возвращаемый объект имеет в точности тот API, который указан в соответствующем интерфейсе, и ему нет необходимости читать дополнительные документы к этому классу. Более того, использование такого статического метода генерации дает клиенту право обращаться к возвращаемому объекту, используя его собственный интерфейс, а не через интерфейс класса реализации, что обычно является хорошим приемом (статья 52).

Скрытым может быть не только класс объекта, возвращаемого открытым статическим методом генерации. Сам этот класс может меняться от вызова к вызову, в зависимости от того, какие значения па-

раметров переданы статическому методу генерации. Это может быть любой класс, который является подтипов по отношению к возвращаемому типу, заявленному в интерфейсе. Класс возвращаемого объекта может также меняться от версии к версии, что повышает удобство сопровождения программы и повышает ее производительность.

В момент, когда пишется класс, содержащий статический метод генерации, класс, соответствующий возвращаемому объекту, может даже не существовать. Подобные гибкие статические методы генерации лежат в основе систем с предоставлением услуг (*service provider frameworks*), таких как Java Cryptography Extension (JCE). Система с предоставлением услуг — это такая система, где поставщик может создавать различные реализации интерфейса API, доступные пользователям этой системы. Чтобы сделать эти реализации доступными для использования, предусмотрен механизм регистрации (*register*). Клиенты могут пользоваться указанным API, не беспокоясь о том, с какой из его реализаций они имеют дело.

У класса `java.util.EnumSet` (статья 32), представленного в версии 1.5, нет открытых конструкторов, только статические методы. Они возвращают одну из двух реализаций в зависимости от размера типа перечисления: если значение равно 64 и менее элементов (как у большей части типов перечислений), то статический метод возвращает экземпляр `RegularEnumSet`, подкрепленный единичным значением `long`. Если же тип перечислений содержит 65 и более элементов, то метод возвращает экземпляр `JumboEnumSet`, подкрепленный массивом `long`.

Существование двух реализаций классов невидимо для клиентов. Если экземпляр `RegularEnumSet` перестанет давать преимущество в производительности небольшому количеству типов перечислений, то его можно избежать в дальнейшем без каких-либо вредных последствий. Таким же образом, при будущем выполнении могут добавиться третья и четвертая реализации `EnumSet`, если это улучшит производительность. Клиенты не знают и не должны беспокоиться о классах объектов, возвращаемых им методами, — для них важны только некоторые подклассы `EnumSet`.

Классу объекта, возвращаемого статическим методом, нет необходимости существовать на момент написания класса, содержащего метод. Подобная гибкость методов генерации создает основу для систем предоставления услуг (*service provider frameworks*), таких как Java Database Connectivity API (JDBC). Система предоставления услуг — это система, в которой несколько провайдеров реализуют службы, и система делает эти реализации доступными для своих клиентов, разъединяя их с реализацией.

Имеется три основных компонента системы предоставления услуг: интерфейс службы (*service interface*), который предоставляется поставщиком, интерфейс регистрации поставщика (*provider registration API*), который использует система для регистрации реализации, и интерфейс доступа к службе (*service access API*), который используется клиентом для получения экземпляра службы. Интерфейс доступа к службе обычно позволяет определить некоторые критерии для выбора поставщика, которые тем не менее не являются обязательными. При отсутствии таковых он возвращает экземпляр реализации по умолчанию. Интерфейс доступа к службе — это «гибкий производственный метод», составляющий основу системы предоставления услуг.

Есть еще не обязательный четвертый компонент службы предоставления услуг — интерфейс поставщика службы (*service provider interface*), который внедряется провайдером для создания экземпляров реализации их служб. При отсутствии этого интерфейса реализации регистрируются по имени класса, а их экземпляры создаются рефлексивным образом (статья 53). В случае с JDBC Connection здесь играет роль интерфейса службы, `DriverManager.registerDriver` — интерфейс регистрации провайдера, `DriverManager.getConnection` — интерфейс доступа к службе, и `Driver` — интерфейс поставщика службы.

Может быть несколько вариантов шаблона службы предоставления услуг. Например, интерфейс доступа к службе может возвратить более развернутый интерфейс службы, чем требуемый провайдером,

при использовании паттерна «Адаптер» [Gamma 95, с. 139], Здесь приведена простая реализация с интерфейсом службы поставщика и поставщиком по умолчанию:

```
// Service provider framework sketch
// Service interface
public interface Service {
    // Service-specific methods go here
}

// Service provider interface
public interface Provider {
    Service newService();
}

// Noninstantiable class for service registration and access
public class Services {
    private Services() { } // Prevents instantiation (Item 4)

    // Maps service names to services
    private static final Map<String, Provider> providers =
        new ConcurrentHashMap<String, Provider>();
    public static final String DEFAULT_PROVIDER_NAME = "<def>";

    // Provider registration API
    public static void registerDefaultProvider(Provider p) {
        registerProvider(DEFAULT_PROVIDER_NAME, p);
    }

    public static void registerProvider(String name, Provider p){
        providers.put(name, p);
    }

    // Service access API
    public static Service newInstance() {
        return newInstance(DEFAULT_PROVIDER_NAME);
    }
```

```

public static Service newInstance(String name) {
    Provider p = providers.get(name);
    if (p == null)
        throw new IllegalArgumentException(
            "No provider registered with name: " + name);
    return p.newService();
}
}

```

Четвертое преимущество статических шаблонов проектирования заключается в том, что они уменьшают многословие при создании экземпляров типов с параметрами. К сожалению, вам необходимо определить параметры типа при запуске конструктора классов с параметрами, даже если они понятны из контекста. Поэтому приходится обозначать параметры типа дважды:

```

Map<String, List<String>> m =
new HashMap<String, List<String>>();

```

Эта излишняя спецификация становится проблемой по мере увеличения сложности параметров типов. При использовании же статических методов компилятор сможет за вас создать параметры. Это еще называется (*type inference*). Например, предположим, что реализация `HashMap` дала нам следующий метод (статья 27):

```

public static <K, V> HashMap<K, V> newInstance() {
    return new HashMap<K, V>();
}

```

В данном случае многословное выражение может быть заменено следующей краткой альтернативой:

```

Map<String, List<String>> m = HashMap.newInstance();

```

Когда-нибудь вывод типа, сделанный таким образом, будет возможно применять при вызове конструкторов, а не только при вызове статических методов, но в релизе 1.6 платформы такое пока невозможно.

К сожалению, у стандартного набора реализаций, таких как `HashMap`, нет своих статических методов в версии 1.6, но вы можете доба-

вить эти методы в собственный класс параметров. Теперь вы сможете предоставлять статические методы генерации в собственных классах с параметрами.

Основной недостаток использования только статических методов генерации заключается в том, что классы, не имеющие открытых или защищенных конструкторов, не могут иметь подклассов. Это же верно и в отношении классов, которые возвращаются открытыми статическими методами генерации, но сами открытыми не являются. Например, в архитектуре Collections Framework невозможно создать подкласс ни для одного из классов реализаций. Сомнительно, что в такой маскировке может быть благо, поскольку поощряет программистов использовать не наследование, а композицию (статья 14).

Второй недостаток статических методов генерации состоит в том, что их трудно отличить от других статических методов. В документации API они не выделены так, как это было бы сделано для конструкторов. Поэтому иногда из документации к классу сложно понять, как создать экземпляр класса, в котором вместо конструкторов клиенту предоставлены статические методы генерации. Возможно, когда-нибудь в официальной документации по Java будет уделено должное внимание статическим методам. Указанный недостаток может быть смягчен, если придерживаться стандартных соглашений, касающихся именования. Эти названия статических методов генерации становятся общепринятыми:

- `valueOf` — возвращает экземпляр, который, грубо говоря, имеет то же значение, что и его параметры. Статические методы генерации с таким названием фактически являются операторами преобразования типов.
- `of` — более краткая альтернатива для `valueOf`, распространенная при использовании `EnumSet` (статья 32).
- `getInstance` — возвращает экземпляр, который описан параметрами, однако говорить о том, что он будет иметь то же

значение, нельзя. В случае с синглтоном этот метод возвращает единственный экземпляр данного класса. Это название является общепринятым в системах с предоставлением услуг.

- newInstance — то же, что и getInstance, только newInstance дает гарантию, что каждый экземпляр отличается от всех остальных.
- getType — то же, что и getInstance, но используется, когда метод генерации находится в другом классе. Type обозначает тип объекта, возвращенного методом генерации.
- newType — то же, что и newInstance, но используется, когда метод генерации находится в другом классе. Type обозначает тип объекта, возвращенного методом генерации.

Подведем итоги. И статические методы генерации, и открытые конструкторы имеют свою область применения, имеет смысл разобраться, какие они имеют достоинства друг перед другом. Обычно статические методы предпочтительнее, поэтому не надо бросаться создавать конструкторы, не рассмотрев сначала возможность использования статических методов генерации, поскольку последние часто оказываются лучше. Если вы взвесили обе возможности и не нашли достаточных доводов ни в чью пользу, вероятно, лучше всего создать конструктор, просто потому, что такой подход является нормой.

*Статья
2*

Используйте шаблон Builder, когда приходится иметь дело с большим количеством параметров конструктора

У конструкторов и статических методов есть одно общее ограничение: они плохо масштабируют большое количество необязательных параметров. Рассмотрим такой случай: класс, представляющий собой этикетку с информацией о питательности на упаковке с продуктами питания. На этих этикетках есть несколько обязательных полей —

размер порции, количество порций в упаковке, калорийность, а также ряд необязательных параметров — общее содержание жиров, содержание насыщенных жиров, содержание жиров с трансизомерами жирных кислот, содержание холестерина, натрия и т.д. У большинства продуктов не нулевыми будут только несколько из этих необязательных значений.

Какой конструктор или какие методы нужно использовать для написания данного класса? Традиционно, программисты будут использовать шаблоны с телескопическими конструкторами (*Telescoping Constructor Pattern*), при использовании которых вы выдаете набор конструкторов: конструктор с одними обязательными параметрами, конструктор с одним необязательным параметром, конструктор с двумя обязательными параметрами и т.д., до тех пор пока не будет конструктора со всеми необязательными параметрами. Вот как это выглядит на практике. Для краткости мы будем использовать только 4 не обязательных параметра:

```
// Telescoping constructor pattern - does not scale well!
public class NutritionFacts {
    private final int servingSize; // (mL) required
    private final int servings; // (per container) required
    private final int calories; // optional
    private final int fat; // (g) optional
    private final int sodium; // (mg) optional
    private final int carbohydrate; // (g) optional
    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;
}

}
```

Если вы хотите создать экземпляр данного класса, то вы будете использовать конструктор с минимальным набором параметров, который бы содержал все параметры, которые вы хотите установить:

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

Обычно для вызова конструктора потребуется передавать множество параметров, которые вы не хотите устанавливать, но вы в любом случае вынуждены передать для них значение. В нашем случае мы установили значение 0 для поля fat. Поскольку мы имеем только шесть параметров, может показаться, что это не так уж и плохо, но ситуация выходит из-под контроля, когда число параметров увеличивается.

Короче говоря, **шаблоны телескопических конструкторов нормально работают, но становится трудно писать код программы-клиента, когда имеется много параметров, а еще труднее этот код читать**. Читателю остается только гадать, что означают все эти значения, и нужно тщательно высчитывать позицию параметра, чтобы выяснить, к какому полю он относится. Длинные последова-

тельности одинаково типизированных параметров могут приводить к тонким ошибкам. Если клиент случайно перепутает два из таких параметров, то компиляция будет успешной, но программа будет неправильно работать при выполнении (статья 40).

Второй вариант, когда вы столкнулись с конструктором со многими параметрами, — это использование шаблонов JavaBeans, где вы вызываете конструктор без параметров, чтобы создать объект, а затем вызываете сеттеры для установки обязательных и всех интересующих необязательных параметров:

```
// JavaBeans Pattern – allows inconsistency, mandates mutability
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize = -1; // Required; no default value
    private int servings = -1; // " " "
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;
    public NutritionFacts() { }
    // Setters
    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; }
}
```

Данный шаблон лишен недостатков шаблона телескопических конструкторов. Он прост, хотя и содержит большое количество слов, но получившийся код легко читается.

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
```

```
cocaCola.setSodium(35);  
cocaCola.setCarbohydrate(27);
```

К сожалению, шаблон JavaBeans не лишен серьезных недостатков. Поскольку его конструкция разделена между несколькими вызовами, **JavaBean может находиться в неустойчивом состоянии частично из-за такой конструкции.** У класса нет возможности принудительно обеспечить стабильность простой проверкой действительности параметров конструктора. Попытка использования объекта, если он находится в неустойчивом состоянии, может привести к ошибкам выполнения даже после удаления ошибки из кода, что создает трудности при отладке. Схожим недостатком является то, что **шаблон JavaBeans исключает возможность сделать класс неизменным** (статья 15), что требует дополнительных усилий со стороны программиста для обеспечения безопасности в многопоточной среде.

Действие этого недостатка можно уменьшить, вручную «замораживая» объект после того, как его создание завершено, и запретив его использование, пока он заморожен, но этот вариант редко используется на практике. Более того, он может привести к ошибкам выполнения, так как компилятор не может удостовериться в том, вызывает ли программист метод заморозки для объекта до того, как он будет использоваться.

К счастью, есть и третья альтернатива, которая сочетает в себе безопасность шаблона телескопических конструкций и читаемость шаблона JavaBeans. Она является одной из форм шаблона «конструктор» [Gamma 95, с. 97]. Вместо непосредственного создания желаемого объекта клиент вызывает конструктор (или статический метод) со всеми необходимыми параметрами и получает объект *Builder* (*builder object*). Затем клиент вызывает сеттеры на этом объекте для установки всех интересующих параметров. Наконец, клиент вызывает метод *build* для генерации объекта, который будет являться неизменным. «Конструктор» является статическим внутренним классом в классе (статья 22), который он создает. Вот как это выглядит на практике:

```
// Builder Pattern
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 {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 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 carbohydrate(int val)
            { carbohydrate = val; return this; }
        public Builder sodium(int val)
            { sodium = 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;
}
}

```

Обратите внимание, что NutritionFacts является неизменным и что все значения параметров по умолчанию находятся в одном месте. Сеттеры объекта «конструктор» возвращают сам этот «конструктор». Поэтому вызовы можно объединять в цепочку. Вот как выглядит код клиента:

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

Этот клиентский код легко писать и, что еще важнее, легко читать. **Шаблон «конструктора» имитирует именные дополнительные параметры**, так же как в языках Ada и Python.

Как и конструктор, «конструктор» может навязывать инварианты на свои параметры. Метод `build` позволяет проверить эти инварианты. Очень важно, чтобы они были проверены после копирования параметров из «конструктора» на объект и чтобы они были проверены на полях объекта, а не на полях «конструктора» (статья 39). Если хоть один инвариант нарушается, то метод `build` должен вывести сообщение `IllegalStateException` (статья 60). Детали ошибки в сообщении содержат информацию, какие инварианты были нарушены (статья 63).

Другой способ навязывания инвариантов на большое количество параметров — это заставить сеттеры охватить группу параметров, на которых должны применяться инварианты. Если условия инвариантов не соблюдаются, то сеттеры выводят ошибку `IllegalArgumentException`. В данном случае преимущество в том, что ошибка обнаруживается сразу, как только переданы неверные параметры, вместо того чтобы ждать запуска метода `build`.

Другое небольшое преимущество использования «конструктора» вместо конструкторов заключается в том, что у «конструктора»

может быть несколько параметров varargs. У конструкторов, как и у методов, может быть только один параметр varargs. Поскольку «конструкторы» используют отдельные методы для установки каждого параметра, они могут иметь сколько угодно параметров varargs до ограничения в один параметр на один сеттер.

Шаблон «конструктора» обладает гибкостью. Один «конструктор» может использоваться для создания нескольких объектов. Параметры «конструктора» могут быть изменены для того, чтобы создавать различные объекты. «Конструктор» может автоматически заполнить некоторые поля, например серийный номер, который автоматически увеличивается каждый раз, когда создается объект.

«Конструктор», параметры которого заданы, создает отличный шаблон проектирования (*Abstract factory*) [Gamma95, с. 87]. Другими словами, клиент может передать такой «конструктор» методу, чтобы метод мог создавать один или более объектов для клиента. Чтобы сделать возможным такое использование, вам необходим тип для представления построения. Если вы используете релиз 1.5 или более поздний, то будет достаточно одного родового типа (статья 26) для всех «конструкторов», вне зависимости от того, какой тип объекта они создают:

```
// A builder for objects of type T public interface Builder<T> {  
    public T build();  
}
```

Обратите внимание, что наш класс NutritionFacts.Builder может быть объявлен для реализации «конструктора» Builder<NutritionFacts>.

Методы, которые работают с экземпляром «конструктора», обычно накладывают ограничения на его параметры, используя связанные типы групповых символов (*bounded wildcard type*) (статья 28). Например, вот метод, который строит дерево, используя предоставленный клиентом экземпляр «конструктора», для построения каждого узла.

```
Tree buildTree(Builder<? extends Node> nodeBuilder) { }
```

Традиционным применением шаблонов проектирования в Java являются классы, с методом newInstance, являющимся частью ме-

тода `build`. При таком использовании очень много проблем. Метод `newInstance` всегда пытается запустить конструктор класса без параметров, который может даже и не существовать. И вы не получите сообщение об ошибке на этапе компиляции, если у класса нет доступа к конструктору без параметров. Вместо этого клиентский код натолкнется на ошибку `InstantiationException` или `IllegalAccessException` в процессе выполнения, что ужасно неприятно. Также метод `newInstance` распространяет любое сообщение об исключении (ошибке), выведенное конструктором без параметров, даже если в тексте метода нет соответствующих выражений `throws`. **Другими словами, `Class.newInstance` прерывает проверку ошибок на этапе компиляции.** Интерфейс `Builder`, рассмотренный выше, исправляет данный недостаток.

У шаблона «конструктор» есть свои недостатки. Для создания объекта вам надо создать сначала его «конструктор». Затраты на создание «конструктора» малозаметны на практике на самом деле, но в некоторых ситуациях, где производительность является важным моментом, это может создать проблемы. Кроме того, шаблоны «конструктора» более длинные, нежели шаблоны телескопических конструкций, поэтому использовать их нужно при наличии достаточного количества параметров, например четырех и более. Но имейте в виду, что в будущем вы можете захотеть добавить параметры. Если вы начнете использовать конструкторы или статические методы, а затем добавите «конструктор», когда количество параметров в классе выйдет из-под контроля, то уже ненужные конструкторы или статические методы будут для вас словно заноза в пальце. Поэтому изначально желательно начинать именно с «конструктора».

В общем, шаблон «конструктора» — это хороший выбор при проектировании классов, у чьих конструкторов и статических методов имеется много параметров, особенно если большинство из них не являются обязательными. Клиентский код легче читать и писать при использовании «конструкторов», чем при использовании традиционных шаблонов телескопических конструкторов. Кроме того, «конструкторы» гораздо безопаснее, чем `JavaBeans`.

Слайдер

3

Свойство синглтон обеспечивайте закрытым конструктором или типом перечислений

Синглтон (*singleton*) — это просто класс, для которого экземпляр создается только один раз [Gamma95, с. 127]. Синглтоны обычно представляют некоторые компоненты системы, которые действительно являются уникальными, например видеодисплей или файловая система. Превращение класса в синглтон может создать сложности для тестирования его клиентов, так как невозможно заменить ложную реализацию синглтоном, если только он не реализует интерфейс, который одновременно служит его типом.

До релиза 1.5 для реализации синглтонов использовалось два подхода. Оба они основаны на создании закрытого (*private*) конструктора и открытого (*public*) статического члена, который позволяет клиентам иметь доступ к единственному экземпляру этого класса. В первом варианте открытый статический член является полем типа *final*:

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

Закрытый конструктор вызывается только один раз, чтобы инициализировать поле *Elvis.INSTANCE*. Отсутствие открытых или защищенных конструкторов гарантирует «вселенную с одним *Elvis*»: после инициализации класса *Elvis* будет существовать ровно один экземпляр *Elvis* — не больше и не меньше. И ничего клиент с этим поделать не может, за одним исключением: клиент с расширенными правами может в свою очередь запустить частный конструктор с помощью метода *AccessibleObject.setAccessible*. Если вы хотите защиту от такого рода атаки, необходимо изменить конструктор так,

чтобы он выводил сообщение об ошибке, если поступит запрос на создание второго экземпляра.

Во втором варианте вместо открытого статического поля типа `final` создается открытый статический метод генерации:

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

Все вызовы статического метода `Elvis.getInstance` возвращают ссылку на один и тот же объект, и никакие другие экземпляры `Elvis` никогда созданы не будут (за исключением того же вышеупомянутого недостатка).

Основное преимущество первого подхода заключается в том, что из декларации членов, составляющих класс, понятно, что этот класс является синглтоном: открытое статическое поле имеет тип `final`, а потому это поле всегда будет содержать ссылку на один и тот же объект. Первый вариант по сравнению со вторым более не имеет прежнего преимущества в производительности: современная реализация JVM встраивает вызовы для статического метода генерации во втором варианте.

Одно из преимуществ второго подхода (использование статического метода генерации) заключается в том, что он дает вам возможность отказаться от решения сделать класс синглтоном, не меняя при этом его API. Статический метод генерации для синглтона возвращает единственный экземпляр этого класса, однако это можно легко поменять и возвращать, скажем, свой уникальный экземпляр для каждого потока, обращающегося к этому методу. Второе преимущество, которое касается родовых типов, подробно описано в статье 27.

Зачастую ни одно из этих преимуществ не имеет значение и подход с использованием открытого поля.

Чтобы класс синглтона был сериализуемым (глава 11), надо просто добавить к его декларации `implements Serializable` будет недостаточно. Чтобы дать синглтону нужные гарантии, вам необходимо объявить все экземпляры полей как временные (`transient`), а также создать метод `readResolve` (статья 77). В противном случае каждая десериализация сериализованного экземпляра будет приводить к созданию нового экземпляра, что в нашем примере приведет к обнаружению ложных `Elvis`. Чтобы предотвратить это, добавьте в класс `Elvis` следующий метод `readResolve`:

```
// Метод readResolve для сохранения свойств синглтона
private Object readResolve() {
    // Возвращает один истинный Elvis и дает возможность *сборщику
    // мусора*
    // избавиться от самозванца Elvis
    return INSTANCE;
}
```

В релизе 1.5 также имеется третий подход к реализации синглтонов. Просто создать тип перечислений одним элементом:

```
// Перечисление синглтона - более предпочтительный подход.
public enum Elvis {
    INSTANCE;
    public void leaveTheBuilding() { ... }
}
```

Данный подход функционально эквивалентен подходу использования открытого поля, за исключением того факта, что он более точен, бесплатно предоставляет механизм сериализации и дает железную гарантию, что не будет создано большого количества экземпляров, даже перед лицом сложной сериализации или отражающих атак. В то время как этот подход еще только должен быть усвоен, тем не менее **наилучшим является реализация синглтона через тип перечисления с одним элементом.**

Отсутствие экземпляров обеспечивает закрытый конструктор

Время от времени вам будет необходимо написать класс, который является всего лишь собранием статических методов и статических полей. Такие классы приобрели дурную репутацию, поскольку некоторые люди неправильно пользуются ими, чтобы с помощью объектно-ориентированных языков писать процедурные программы. Подобные классы требуют правильного использования. Их можно использовать для того, чтобы собирать вместе связанные друг с другом методы обработки простых значений или массивов, как это сделано в библиотеках `java.lang.Math` или `java.util.Arrays`, либо чтобы собирать вместе статические методы объектов, в том числе методов статической генерации (статья 1) для объектов, которые реализуют определенный интерфейс, как это сделано в `java.util.Collections`. Можно также собрать методы в неком окончательном (*final*) классе, вместо того, чтобы заниматься расширением класса.

Подобные классы утилит (*utility class*) разрабатываются не для того, чтобы для них создавать экземпляры — такой экземпляр был бы абсурдом. Однако, если у класса нет явных конструкторов, компилятор по умолчанию сам создает для него открытый конструктор (*default constructor*), не имеющий параметров. Для пользователя этот конструктор ничем не будет отличаться от любого другого. В опубликованных API нередко можно встретить классы, непреднамеренно наделенные способностью порождать экземпляры.

Попытки запретить классу создавать экземпляры, объявив его абстрактным, не работают. Такой класс может иметь подкласс, для которого можно создавать экземпляры. Более того, это вводит пользователя в заблуждение, заставляя думать, что данный класс был разработан именно для наследования (статья 17). Есть, однако, простая идиома, гарантирующая отсутствие экземпляров. Конструктор по умолчанию создается только тогда, когда у класса нет явных конструкторов, и потому **запретить создание экземпляров можно, поместив в класс единственный явный закрытый конструктор:**

```
// Класс утилит, не имеющий экземпляров
public class UtilityClass {
    // Подавляет появление конструктора по умолчанию, а заодно
    // и создание экземпляров класса
    private UtilityClass() {
        throw new AssertionError();
    }
    ... // Остальное опущено
}
```

Поскольку явный конструктор заявлен как закрытый (*private*), то за пределами класса он будет недоступен. Не обязательна строка с `AssertionError`, но является подстраховкой на случай, если конструктор будет случайно вызван в самом классе. Она гарантирует, что для класса никогда не будет создано никаких экземпляров. Эта идиома несколько алогична, поскольку конструктор создается здесь именно для того, чтобы им нельзя было пользоваться. Соответственно, есть смысл поместить в текст программы комментарий, как описано выше.

Побочным эффектом является то, что данная идиома не позволяет создавать подклассы для этого класса. Явно или неявно, все конструкторы должны вызывать доступный им конструктор суперкласса. Здесь же подкласс лишен доступа к конструктору, к которому можно было бы обратиться.

Статья
5

Избегайте ненужных объектов

Вместо того чтобы создавать новый функционально эквивалентный объект всякий раз, когда в нем возникает необходимость, часто можно просто еще раз использовать тот же объект. Использовать что-либо снова и изящнее, и быстрее. Если объект является неизменяемым (*immutable*), его всегда можно использовать повторно (статья 15).

Рассмотрим следующий оператор, демонстрирующий, как делать не надо:

```
String s = new String("stringette"); // Никогда так не делайте!
```

При каждом проходе этот оператор создает новый экземпляр `String`, но ни одна из процедур создания объектов не является необходимой. Аргумент конструктора `String` — «`stringette`» — сам является экземпляром класса `String` и функционально равнозначен всем объектам, создаваемым этим конструктором. Если этот оператор попадает в цикл или часто вызываемый метод, без всякой надобности могут создаваться миллионы экземпляров `String`.

Исправленная версия выглядит просто:

```
String s = "stringette";
```

В этом варианте используется единственный экземпляр `String` вместо того, чтобы при каждом проходе создавать новые. Более того, дается гарантия того, что этот объект будет повторно использовать любой другой программный код, который выполняется на той же виртуальной машине, где содержится эта строка-константа [JLS, 3.10.5].

Создания дублирующих объектов часто можно избежать, если в неизменяемом классе, имеющем и конструкторы, и статические методы генерации (статья 1), вторые предпочесть первым. Например, статический метод генерации `Boolean.valueOf(String)` почти всегда предпочтительнее, чем конструктор `Boolean(String)`. При каждом вызове конструктор создает новый объект, тогда как от статического метода генерации этого не требуется.

Вы можете повторно использовать не только неизменяемые объекты, но и изменяемые, если знаете, что последние меняться уже не будут. Рассмотрим более тонкий и гораздо более распространенный пример того, как не надо поступать. В нем участвуют изменяемые объекты `Date`, которые более не меняются, после того как их значение вычислено. Этот класс моделирует человека и имеет метод `isBabyBoomer` который говорит, является ли человек рожденным

вследствие «беби-буна», другими словами, родился ли он между 1946 и 1964 годами:

```
public class Person {  
    private final Date birthDate;  
    // Прочие поля, методы и конструкторы опущены  
    // Никогда так не делайте!  
    public boolean isBabyBoomer() {  
        // Не нужно размещение затратных объектов  
        Calendar gmtCal =  
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);  
        Date boomStart = gmtCal.getTime();  
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);  
        Date boomEnd = gmtCal.getTime();  
        return birthDate.compareTo(boomStart) >= 0 &&  
            birthDate.compareTo(boomEnd) < 0;  
    }  
}
```

Метод `isBabyBoomer` при каждом вызове без всякой надобности создает новые экземпляры `Calendar`, `TimeZone` и два экземпляра `Date`. В следующей версии подобная расточительность пресекается с помощью статического инициализатора:

```
public class Person {  
    private final Date birthDate;  
    // Другие поля, методы и конструкторы опущены  
    /**  
     * Даты начала и конца демографического взрыва  
     */  
    private static final Date BOOM_START;  
    private static final Date BOOM_END;  
    static {  
        Calendar gmtCal =  
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));  
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);  
    }  
}
```

```
    BOOM_START = gmtCal.getTime();
    gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
    BOOM_END = gmtCal.getTime();
}
public boolean isBabyBoomer() {
    return birthDate.compareTo(BOOM_START) >= 0 &&
birthDate.compareTo(BOOM_END) < 0;
}
}
```

В исправленной версии класса Person экземпляры Calendar, TimeZone и Date создаются только один раз в ходе инициализации вместо того, чтобы создавать их при каждом вызове метода isBabyBoomer. Если данный метод вызывается достаточно часто, это приводит к значительному выигрышу в производительности. На моей машине исходная версия программы тратит на 10 миллионов вызовов 32 000 мс, улучшенная – 130 мс, что в 250 раз быстрее. Причем улучшается не только производительность программы, но и наглядность. Замена локальных переменных boomStart и boomEnd статическими полями типа final показывает, что эти даты рассматриваются как константы, соответственно, программный код становится более понятным. Для полной ясности заметим, что экономия от подобной оптимизации не всегда будет столь впечатляющей, просто здесь особенно много ресурсов требует создание экземпляров Calendar.

Если метод isBabyBoomer вызываться не будет, инициализация полей BOOM_START и BOOM_END в улучшенной версии класса Person окажется напрасной. Ненужных действий можно было бы избежать, использовав для этих полей отложенную инициализацию (*lazily initializing*) (статья 71), которая бы выполнялась при первом вызове метода isBabyBoomer, однако делать это не рекомендуется. Как часто бывает в случаях с отложенной инициализацией, это усложнит реализацию и вряд ли приведет к заметному повышению производительности (статья 55).

Во всех примерах, ранее приведенных в этой статье, было очевидно, что рассматриваемые объекты можно использовать повторно,

поскольку они были неизменяемыми. Есть, однако, другие ситуации, когда это не столь очевидно. Рассмотрим случай с адаптерами (*adapter*) [Gamma95, с. 139], которые известны также как представления (*view*). Адаптер – это объект, который делегирован нижележащим объектом, создавая для него альтернативный интерфейс. Поскольку адаптер не имеет иных состояний, помимо состояния нижележащего объекта, то для адаптера, представляющего данный объект, более одного экземпляра создавать не надо.

Например, в интерфейсе `Map` метод `keySet` возвращает для объекта `Map` представление `Set`, которое содержит все ключи данной схемы. По незнанию можно подумать, что каждый вызов метода `keySet` должен создавать новый экземпляр `Set`. Однако в действительности для некоего объекта `Map` любые вызовы `keySet` могут возвращать один и тот же экземпляр `Set`. И хотя обычно возвращаемый экземпляр `Set` является изменяемым, все возвращаемые объекты функционально идентичны: когда меняется один из них, то же самое происходит и со всеми остальными экземплярами `Set`, поскольку за всеми ними стоит один и тот же экземпляр `Map`. Нет необходимости создавать несколько экземпляров объекта представления `KeySet`, хотя их создание и безвредно.

В релизе 1.5 есть и другой способ создания ненужных объектов. Он называется автоупаковка (*autoboxing*) и позволяет программисту использовать и примитивные им упакованные примитивные типы, выполнять упаковку и распаковку автоматически в случае необходимости. Имеются тонкие семантические различия и не тонкие отличия в производительности (статья 49). Рассмотрим следующую программу, которая рассчитывает сумму всех положительных значений `int`. Для этого используется арифметическая функция `long`, так как `int` недостаточно большая, чтобы хранить сумму всех положительных значений `int`.

```
// Ужасно медленная программа! Можете определить, на каком этапе
// 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 дает правильный ответ, но выполняется она намного медленнее, чем должна, из-за ошибки только в одном символе. Переменная `sum` объявлена как `Long` вместо `long`, что означает, что программа создает 2^{31} ненужных экземпляров `Long` (грубо говоря, каждый раз, когда `long` добавляется к сумме `Long`). Изменение декларирования `sum` на `long` вместо `Long` уменьшает время выполнения программы на моей машине с 43 до 6,8 секунды. Урок понятен: предпочтение надо отдавать примитивным и упакованным примитивным типам и избегать непреднамеренного автоупаковывания.

В этой статье отнюдь не утверждается, что создание объектов требует много ресурсов и его нужно избегать. Наоборот, создание и повторное использование небольших объектов, чьи конструкторы выполняют несложную и понятную работу, необременительно, особенно для современных реализаций JVM. Создание дополнительных объектов ради большей наглядности, упрощения и расширения возможностей программы — это обычно хорошая практика.

И наоборот, отказ от создания объектов и поддержка собственного пула объектов (*object pool*) — плохая идея, если только объекты в этом пуле не будут крайне ресурсоемкими. Основной пример объекта, для которого оправданно создание пула, — это может быть соединение с базой данных (*database connection*). Затраты на установление такого соединения достаточно высоки, и потому многократное использование такого объекта оправданно. Также лицензионная политика вашей БД может накладывать ограничения на фиксированное количество соединений. Однако в общем случае создание собственного пула объектов загромождает вашу программу, увеличивает расход памяти и снижает производительность программы. Современные реализации JVM имеют хорошо оптимизированные сборщики мусора, которые при

работе с небольшими объектами с легкостью превосходят подобные пулы объектов.

В противовес этой статье можно привести статью 39, посвященную резервному копированию (*defensive copying*). Если в этой статье говорится: «Не создавайте новый объект, если вы обязаны использовать имеющийся еще раз», то статья 39 гласит: «Не надо использовать имеющийся объект еще раз, если вы обязаны создать новый». Заметим, что ущерб от повторного использования объекта, когда требуется резервное копирование, значительно превосходит ущерб от бесполезного создания дублирующего объекта. Отсутствие резервных копий там, где они необходимы, может привести к коварным ошибкам и дырам в системе безопасности, создание же ненужных объектов всего лишь влияет на стиль и производительность программы.

Статья 6

Уничтожайте устаревшие ссылки (на объекты)

Когда вы переходите с языка программирования с ручным управлением памятью, такого как Си или С++, на язык с автоматической очисткой памяти (*garbage-collect* — «сборка мусора»), ваша работа как программиста существенно упрощается благодаря тому обстоятельству, что ваши объекты автоматически утилизируются, как только вы перестаете с ними работать. Когда вы впервые сталкиваетесь с этим, это производит впечатление почти что волшебства. Легко может создаться впечатление, что вам больше не надо думать об управлении памятью, но это не совсем так.

Рассмотрим следующую реализацию простого стека:

```
// Можете ли вы заметить «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];
}
/**
 * убедиться в том, что в стеке есть место хотя бы еще для
 * одного элемента, каждый раз, когда нужно увеличить массив,
 * просто удваивать его емкость
 */
private void ensureCapacity() {
    if (elements.length == size)
        elements = Arrays.copyOf(elements, 2 * size + 1);
}
```

В этой программе нет погрешностей, которые бросались бы в глаза (но посмотрите в статье 26 оригинальную версию). Вы можете тщательно ее тестировать, любое испытание она пройдет с успехом, но в ней все же скрыта одна проблема. Грубо говоря, в этой программе имеется «утечка памяти», которая может тихо проявляться в виде снижения производительности, в связи с усиленной работой сборщика мусора либо увеличения размера используемой памяти. В крайнем случае подобная утечка памяти может привести к началу подкачки страниц с диска и даже аварийному завершению программы с диагностикой OutOfMemoryError, хотя подобные отказы встречаются относительно редко.

Так где же происходит утечка? Если стек растет, а затем уменьшается, то объекты, которые были вытолкнуты из стека, не могут быть удалены, даже если программа, пользующаяся этим стеком, уже не имеет ссылок на них. Все дело в том, что этот стек сохраняет устаревшие ссылки (*obsolete reference*) на эти объекты. Устаревшая ссылка — это такая ссылка, которая уже никогда не будет разыменована. В данном случае устаревшими являются любые ссылки, оказавшиеся за пределами активной части этого массива элементов. Активная же часть стека включает элементы, чей индекс меньше значения переменной `size`.

Утечка памяти в языках с автоматической сборкой мусора (или, точнее, непреднамеренное сохранение объектов — *unintentional object retention*) очень коварна. Если ссылка на объект была непреднамеренно сохранена, сборщик мусора не сможет удалить не только этот объект, но и все объекты, на которые он ссылается, и т.д. Если даже непреднамеренно было сохранено всего несколько объектов, многие и многие объекты могут стать недоступны сборщику мусора, а это может оказать большое влияние на производительность программы.

Решаются проблемы такого типа очень просто: как только ссылки устаревают, их нужно обнулять. В случае с нашим классом `Stack` ссылка становится устаревшей, как только ее объект был вытолкнут из стека. Исправленный вариант метода `pop` выглядит следующим образом:

```
public Object pop() {  
    if (size == 0)  
        throw new EmptyStackException();  
    Object result = elements[- size];  
    elements[size] = null; // Убираем устаревшую ссылку  
    return result;  
}
```

Обнуление устаревших ссылок дает и другое преимущество: если впоследствии кто-то по ошибке попытается разыменовать какую-либо из этих ссылок, программа незамедлительно завершится

с диагностикой `NullPointerException` вместо того, чтобы спокойно выполнять неправильную работу. Всегда выгодно обнаруживать ошибки программирования настолько быстро, насколько это возможно.

Когда программисты впервые сталкиваются с подобной проблемой, они начинают перестраховываться, обнуляя все ссылки на объекты, лишь только программа заканчивает работу с ними. Это нежелательно и не необходимо, поскольку это без необходимости загромождает программу и, по-видимому, снизит ее производительность. **Обнуление ссылок на объект должно быть не нормой, а исключением.** Лучший способ избавиться от устаревшей ссылки — вновь использовать переменную, в которой она находилась, либо выйти из области видимости этой переменной. Это происходит естественным образом, если для каждой переменной вы задаете самую ограниченную область видимости (статья 45).

Так когда же вы должны обнулять ссылку? Какая особенность класса `Stack` сделала его восприимчивым к утечке памяти? Просто класс `Stack` управляет своей памятью. Пул хранения состоит из массива элементов (причем его ячейками являются ссылки на объекты, а не сами объекты). Как было указано выше, элементы из активной части массива считаются занятыми, в остальной — свободными. Сборщик мусора этого знать никак не может, и для него все ссылки на объекты, хранящиеся в массиве, в равной степени действительны. Только программист знает, что неактивная часть массива не нужна. Реально сообщить этот факт сборщику мусора программист может, лишь вручную обнуляя элементы массива по мере того, как они переходят в неактивную часть массива.

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

Другим распространенным источником утечек памяти являются кэши. Поместив однажды в кэш ссылку на некий объект, легко можно забыть о том, что она там есть, и держать ссылку в кэше еще долгое время после того, как она стала недействительной. У этой проблемы возможны несколько решений. Если вам посчастливилось создавать кэш, в котором запись остается значимой ровно до тех пор, пока за пределами кэша остаются ссылки на ее ключ, представьте этот кэш как WeakHashMap: когда записи устареют, они будут удалены автоматически. Не забывайте, что использование WeakHashMap имеет смысл, только если желаемое время жизни записей кэша определено внешними ссылками на ключ, а не на значение.

В более общем случае время, в течение которого запись в кэше остается значимой, четко не оговаривается. Записи просто теряют свою значимость с течением времени. В таких обстоятельствах кэш следует время от времени очищать от записей, которыми уже никто не пользуется. Подобную чистку может выполнять фоновый поток (возможно, Timer или ScheduledThreadPoolExecutor) либо побочный эффект от добавления в кэш новых записей. Реализации второго подхода помогает метод removeEldestEntry из класса java.util.LinkedHashMap.

Третий распространенный путь утечки памяти — это приложения в режиме ожидания и прочие обратные вызовы. Если вы реализуете API, в котором клиенты регистрируют обратные вызовы, но не отменяют свою регистрацию, то они накапливаются, если вы ничего не предпримете. Лучший способ убедиться, что обратные вызовы не обошли вниманием программы «сборщики мусора», — это хранить только слабые ссылки на них, 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, обнаруживают ее лишь в результате тщательной инспекции программного кода или с помощью инструмента отладки, известного

как профилировщик (*heap profiler*). Поэтому очень важно научиться предвидеть проблемы, похожие на эту, до того, как они возникнут, и предупреждать их появление.



Остерегайтесь методов `finalize`

Методы `finalize` непредсказуемы, часто опасны и, как правило, не нужны. Их использование может привести к странному поведению программы, низкой производительности и проблемам с переносимостью. У методов `finalize` есть лишь несколько областей применения, о которых мы поговорим в этой статье позднее, а главное правило таково: следует избегать методов `finalize`.

Программистов, пишущих на C++, следует предостеречь о том, что думать о методах `finalize` как об аналоге деструкторов в C++, нельзя. В C++ деструктор — это нормальный способ утилизации ресурсов, связанных с объектом, обязательное дополнение к конструктору. В языке программирования Java, когда объект становится недоступен, очистку связанной с ним памяти осуществляет сборщик мусора. Со стороны же программиста никаких специальных действий не требуется. В C++ деструкторы используются для освобождения не только памяти, но и других ресурсов системы. В языке программирования Java для этого обычно используется блок `try--finally`.

Нет гарантии, что метод `finalize` будут вызван немедленно [JLS,12.6]. С момента, когда объект становится недоступен, и до момента выполнения метода `finalize` может пройти сколь угодно длительное время. Это означает, что **с помощью метода `finalize` нельзя выполнять никаких операций, критичных по времени**. Например, будет серьезной ошибкой ставить процедуру закрытия открытых файлов в зависимость от метода `finalize`, поскольку дескрипторы открытых файлов — ресурс ограниченный. Если из-за того, что JVM медлит с запуском методов `finalize`, открытыми бу-

дут оставаться много файлов, программа может завершиться с ошибкой, поскольку не сможет открывать новые файлы.

Частота, с которой запускаются методы `finalize`, 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 методов `finalize`. Вполне возможно, что такая программа будет превосходно работать с JVM, на которой вы проводите ее тестирование, а затем позорно даст сбой на JVM, которую предпочитает ваш самый важный заказчик.

Запоздалый вызов методов `finalize` — не только теоретическая проблема. Создав для какого-либо класса метод `finalize`, в некоторых редких случаях можно спровоцировать произвольную задержку при удалении его экземпляров. Один мой коллега недавно отлаживал приложение GUI, которое было рассчитано на долгое 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 лишь вызова методов `finalize` и утилизации. К несчастью, поток утилизации выполнялся с меньшим приоритетом, чем другой поток того же приложения, а потому удаление объектов не могло осуществляться в том же темпе, в каком они становились доступны для удаления. Спецификация языка Java не дает поручительства, в каком из потоков будут выполняться методы `finalize`. Поэтому нет иного универсального способа предотвратить проблемы такого сорта, кроме как просто воздерживаться от использования методов `finalize`.

Спецификация языка Java не только не дает поручительства, что методы `finalize` будут вызваны быстро, она не дает гарантии, что они вообще будут вызваны. Вполне возможно и даже очень вероятно, что программа завершится, так и не вызвав для некоторых объектов, ставших недоступными, метода `finalize`. Как следствие, **вы никогда не должны ставить обновление критического фиксируемого (*persistent*) состояния в зависимость от метода `finalize`.**

Например, ставить освобождение фиксируемой блокировки разделяемого ресурса, такого как база данных, в зависимость от метода `finalize` — верный способ привести всю вашу распределенную систему к сокрушительному краху.

Не соблазняйтесь методами `System.gc` и `System.runFinalization`. Они могут увеличить вероятность запуска утилизации, но не гарантируют ее. Единственные методы, которые требуют гарантированного удаления, — это `System.runFinalizersOnExit` и его вредный близнец `Runtime.runFinalizersOnExit`. Эти методы некорректны и признаны устаревшими.

В том случае, если вы до сих пор не убедились, что методов `finalize` следует избегать, вот еще одна пикантная новость, стоящая упоминания: если в ходе утилизации возникает необработанная исключительная ситуация (*exception*), она игнорируется, а утилизация этого объекта прекращается [JLS, 12.6]. Необработанная исключительная ситуация может оставить объект в испорченном состоянии. И если другой поток попытается воспользоваться таким испорченным объектом, результат в определенной мере может быть непредсказуем. Обычно необработанная исключительная ситуация завершает поток и выдает распечатку стека, однако в методе `finalize` этого не происходит, он даже не дает предупреждений.

И еще одна деталь: производительность просто ужасающим образом понижается при использовании методов `finalize`. На моей машине время на создание и удаление простого объекта составляет 5,6 нс. При добавлении метода `finalize` оно увеличилось до 2400 нс. Другими словами, это в 430 раз замедлило создание и уничтожение объектов.

Так что же вам делать, вместо того чтобы писать метод `finalize` для класса, объекты которого инкапсулируют ресурсы, требующие завершения, такие как файлы или потоки? Просто создайте метод для прямого завершения и потребуйте, чтобы клиенты этого класса вызывали этот метод для каждого экземпляра, когда он им больше не нужен. Стоит упомянуть об одной детали: экземпляр сам должен

следить за тем, был ли он завершен. Метод прямого завершения должен делать запись в неком закрытом поле о том, что этот объект более не является действительным. Остальные методы класса должны проверять это поле и инициировать исключительную ситуацию `IllegalStateException`, если их вызывают после того, как данный объект был завершен.

Типичный пример метода прямого завершения — это метод `close` в `InputStream` и `OutputStream` и `java.sql.Connection`. Другой пример: метод `cancel` из `java.util.Timer`, который нужным образом меняет состояние объекта, заставляя поток (*thread*), который связан с экземпляром `Timer`, аккуратно завершить свою работу. Среди примеров из пакета `java.awt` — `Graphics.dispose` и `Window.dispose`. На эти методы часто не обращают внимания, и неизбежно это приводит к ужасным последствиям для производительности программы. Проблема касается также метода `Image.flush`, который освобождает все ресурсы, связанные с экземпляром `Image`, но оставляет при этом последний в таком состоянии, что его еще можно использовать, выделив вновь необходимые ресурсы.

Методы прямого завершения часто используются в сочетании с конструкцией try-finally, чтобы обеспечить гарантированное завершение. Вызов метода прямого завершения из оператора `finally` гарантирует, что он будет выполнен, если даже при работе с объектом возникнет исключительная ситуация:

```
// Блок try-finally гарантирует вызов метода завершения
Foo foo = new Foo(...);
try {
    // Сделать то, что необходимо сделать с foo
    ...
} finally {
    foo.terminate(); // Метод прямого завершения
}
```

Зачем же тогда вообще нужны методы `finalize`? Есть два приемлемых применения. Первое — выступать в роли «страховочной

сетки», на тот случай, если владелец объекта забудет вызвать метод прямого завершения, который вы создаете по совету, данному в предыдущем абзаце. Хотя нет гарантии, что метод `finalize` будет вызван своевременно, в тех случаях (будем надеяться, редких), когда клиент не вызовет метод прямого завершения. **Но метод `finalize` обязательно должен записать предупреждение, если он обнаружит, что ресурсы не высвобождены**, так как это является ошибкой в клиентском коде, которая должна быть устранена. Если вы хотите написать такую страховочную сеть, хорошо подумайте о том, стоит ли излишняя безопасность излишней ресурсоемкости.

Четыре класса, представленные как пример использования метода прямого завершения (`FileInputStream`, `FileOutputStream`, `Timer` и `Connection`), тоже имеют методы `finalize`, которые используются в качестве страховочной сетки на тот случай, если соответствующие методы завершения не были вызваны. К сожалению, в данном случае метод `finalize` не записывает предупреждений. Такие предупреждения невозможно добавить после опубликования API, так как имеющийся клиент будет сломан.

Другое приемлемое применение методов `finalize` связано с объектами, имеющими «родные узлы» (*native peers*). Родной узел — это родной объект (*native object*), к которому обычный объект обращается через машинозависимые методы. Поскольку родной узел не является обычным объектом, сборщик мусора о нем не знает, и, соответственно, когда утилизируется обычный узел, утилизировать родной узел он не может. Метод `finalize` является приемлемым механизмом для решения этой задачи при условии, что родной узел не содержит критических ресурсов. Если же родной узел содержит ресурсы, которые необходимо освободить немедленно, данный класс должен иметь, как было описано ранее, метод прямого завершения. Этот метод завершения должен делать все, что необходимо для освобождения соответствующего критического ресурса. Метод завершения может быть машинозависимым методом либо вызывать таковой.

Важно отметить, что здесь нет автоматического связывания методов `finalize` («*finalizer chaining*»). Если в классе (за исключением `Object`) есть метод `finalize`, но в подклассе он был переопределен, то метод `finalize` в подклассе должен вызывать метод `finalize` из суперкласса. Вы должны завершить подкласс в блоке `try`, а затем в соответствующем блоке `finally` вызвать метод `finalize` суперкласса. Тем самым гарантируется, что метод `finalize` суперкласса будет вызван, даже при завершении подкласса инициируется исключительная ситуация, и наоборот. Вот как это выглядит. Обратите внимание, что в этом примере используется аннотация `Override` (`@Override`), которая появилась в релизе 1.5. Вы можете игнорировать сейчас аннотацию `Override` или посмотреть в статье 36, что она означает:

```
// Ручное связывание метода finalize
@Override protected void finalize() throws Throwable {
    try {
        // Ликвидируем состояние подкласса
    } finally {
        super.finalize();
    }
}
```

Если разработчик подкласса переопределяет метод `finalize` суперкласса, но забывает вызвать его, метод `finalize` суперкласса вызван так и не будет. Защититься от такого беспечного или вредного подкласса можно ценой создания некоего дополнительного объекта для каждого объекта, подлежащего утилизации. Вместо того чтобы помещать метод `finalize` в класс, требующий утилизации, поместите его в анонимный класс (статья 22), единственным назначением которого будет утилизация соответствующего экземпляра. Для каждого экземпляра контролируемого класса создается единственный экземпляр анонимного класса, называемый хранителем утилизации (*finalizer guardian*). Контролируемый экземпляр содержит в закрытом экземпляре поля единственную в системе ссылку на хранитель утилизации. Таким образом, хранитель утилизации становится до-

ступен для удаления в момент утилизации контролируемого им экземпляра. Когда хранитель утилизируется, он выполняет процедуры, необходимые для ликвидации контролируемого им экземпляра, как если бы его метод `finalize` был методом контролируемого класса:

```
// Идиома хранителя утилизации (Finalizer Guardian)
public class Foo {
    // Единственная задача этого объекта – утилизировать внешний объект Foo
    private final Object finalizerGuardian = new Object() {
        @Override protected void finalize() throws Throwable {
            // Утилизирует внешний объект Foo
        }
    };
    ... // Остальное опущено
}
```

Заметим, что открытый класс `Foo` не имеет метода `finalize` (за исключением тривиального, унаследованного от класса `Object`), а потому не имеет значения, был ли в методе `finalize` у подкласса вызов метода `super.finalize` или нет. Возможность использования этой методики следует рассмотреть для каждого открытого расширяемого класса, имеющего метод `finalize`.

Подведем итоги. Не надо использовать методы `finalize`, кроме как в качестве страховочной сетки или для освобождения некритических местных ресурсов. В тех редких случаях, когда вы должны использовать метод `finalize`, не забывайте делать вызов `super.finalize`. Если вы используете метод `finalize` в качестве страховочной сетки, не забывайте записывать случаи, когда метод некорректно используется. И последнее: если вам необходимо связать метод `finalize` с открытым классом без модификатора `final`, подумайте об использовании хранителя утилизации, чтобы быть уверенным в том, что утилизация будет выполнена, даже если в подклассе в методе `finalize` не будет вызова `super.finalize`.

Г л а в а 3

Методы, общие для всех объектов

Хотя класс `Object` и является цельным классом, прежде всего он предназначен для расширения. Поскольку все его методы без модификатора `final` — `equals`, `hashCode`, `toString` и `finalize` — предназначены для переопределения, для них есть общие соглашения (*general contracts*). Любой класс, в котором эти методы переопределяются, обязан подчиняться соответствующим соглашениям. Если он этого не делает, то это препятствует правильному функционированию других взаимодействующих с ним классов, работа которых зависит от выполнения указанных соглашений.

В этой главе рассказывается, как и когда следует переопределять методы класса `Object`, не имеющие модификатора `final`. Метод `finalize` в этой главе не рассматривается, поскольку он обсуждался в статье 7. В этой главе также обсуждается метод `Comparable.compareTo`, который не принадлежит классу `Object`, однако имеет схожие свойства.

Спайсер

**Переопределяя метод `equals`,
соблюдайте общие соглашения**

Переопределение метода `equals` кажется простой операцией, однако есть множество способов выполнить это неправильно, и по-

следствия этого могут быть ужасны. Простейший способ избежать проблем: вообще не переопределять метод `equals`. В этом случае каждый экземпляр класса будет равен только самому себе. Это решение будет правильным, если выполняется какое-либо из следующих условий:

- **Каждый экземпляр класса внутренне уникален.** Это утверждение справедливо для таких классов, как `Thread`, которые представляют не величины, а активные сущности. Реализация метода `equals`, предоставленная в классе `Object`, для этих классов работает совершенно правильно.
- **Вас не интересует, предусмотрена ли в классе проверка «логического равенства».** Например, в классе `java.util.Random` можно было бы переопределить метод `equals` с тем, чтобы проверять, будут ли два экземпляра `Random` генерировать одну и ту же последовательность случайных чисел, однако разработчики посчитали, что клиенты не должны знать о такой возможности и она им не понадобится. В таком случае тот вариант метода `equals`, который наследуется от класса `Object`, вполне приемлем.
- **Метод `equals` уже переопределен в суперклассе, и функционал, унаследованный от суперкласса, для данного класса вполне приемлем.** Например, большинство реализаций интерфейса `Set` наследует реализацию метода `equals` от класса `AbstractSet`, `List` наследует реализацию от `AbstractList`, а `Map` — от `AbstractMap`.
- **Класс является закрытым или доступен только в пределах пакета, и вы уверены, что его метод `equals` никогда не будет вызван.** Сомнительно, что в такой ситуации метод `equals` следует переопределять, разве что на тот случай, если его однажды случайно вызовут:

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

Так когда же имеет смысл переопределять `Object.equals()`? Когда для класса определено понятие логической эквивалентности (*logical equality*), которая не совпадает с тождественностью объектов, а метод `equals` в суперклассе не был переопределен с тем, чтобы реализовать требуемый функционал. Обычно это случается с классами значений, такими как `Integer` или `Date`. Программист, сравнивающий ссылки на объекты значений с помощью метода `equals`, скорее желает выяснить, являются ли они логически эквивалентными, а не просто узнать, указывают ли эти ссылки на один и тот же объект. Переопределение метода `equals` необходимо не только для того, чтобы удовлетворить ожидания программистов, оно позволяет использовать экземпляры класса в качестве ключей в некой схеме или элементов в неком наборе, имеющих необходимое и предсказуемое поведение.

Есть один вид классов значений, которым не нужно переопределение метода `equals`, — это класс, использующий управление экземплярами (статья 1), чтобы убедиться, что один объект существует с одним значением. Типы перечислений (статья 30) также попадают под эту категорию. Поскольку для классов этого типа гарантируется, что каждому значению соответствует не больше одного объекта, то метод `equals` из `Object` для этих классов будет равнозначен методу логического сравнения.

Когда вы переопределяете метод `equals`, вам необходимо твердо придерживаться принятых для него общих соглашений. Воспроизведем эти соглашения по тексту спецификации `Object` [JavaSE6]:

Метод `equals` реализует отношение эквивалентности:

- **Рефлексивность:** Для любых ненулевых ссылок 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`, то и выражение $x.equals(z)$ должно возвращать `true`.
- **Непротиворечивость:** Для любых ссылок на значения x и y , если несколько раз вызвать $x.equals(y)$, постоянно будет возвращаться значение `true` либо постоянно будет возвращаться значение `false` при условии, что никакая информация, которая используется при сравнении объектов, не поменялась.
- 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 немного ужасным, однако игнорировать это нельзя! Если вы нарушили эту условия, то вполне рискуете обнаружить, как ваша программа работает неустойчиво или заканчивается с ошибкой, а установить источник ошибок крайне сложно. Перефразируя Джона Донна (John Donne), можно сказать: ни один класс — не остров. («Нет человека, что был бы сам по себе, как остров...» — Донн Дж. Взываю на краю. — Прим. пер.) Экземпляры одного класса часто передаются другому классу. Работа многих классов, в том числе всех классов коллекции, зависит от того, соблюдают ли передаваемые им объекты соглашения для метода `equals`.

Теперь, когда вы знаете, с каким ущербом связано нарушение соглашений для метода `equals`, давайте рассмотрим их внимательнее. Хорошая новость заключается в том, что, вопреки внешнему виду, соглашения не так сложны. Как только вы их поймете, придерживаться их будет совсем не сложно. Давайте по очереди рассмотрим все пять соглашений.

Рефлексивность. Первое требование говорит просто о том, что объект должен быть равен самому себе. Трудно представить себе непреднамеренное нарушение этого требования. Если вы нарушили это требование, а затем добавили экземпляр в ваш класс коллекции, то в этой коллекции метод `contains` почти наверняка сообщит вам, что в коллекции нет экземпляра, которой вы только что добавили.

Симметрия. Второе требование гласит, что любые два объекта должны сходиться во мнении, равны ли они между собой. В отличие от предыдущего представить непреднамеренное нарушение этого требования несложно. Например, рассмотрим следующий класс:

```
//Ошибка: нарушение симметрии!
public final class CaseInsensitiveString {
    private String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Ошибка: нарушение симметрии!
    @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;
    }

    ... // Остальное опущено
}
```

Действуя из лучших побуждений, метод equals в этом классе наивно пытается взаимодействовать с обычными строками. Предположим, что у нас есть одна строка, независимая от регистра, и вторая — обычная.

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

Как и предполагалось, выражение cis.equals(s) возвращает true. Проблема заключается в том, что, хотя метод equals в классе CaseInsensitiveString знает о существовании обычных строк,

метод `equals` в классе `String` о строках, нечувствительных к регистру, не догадывается. Поэтому выражение `s.equals(cis)` возвращает `false`, явно нарушая симметрию. Предположим, вы помещаете в коллекцию строку, нечувствительную к регистру:

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

Какое значение после этого возвратит выражение `list.contains(s)`? Кто знает. В текущей версии JDK от компании Sun выяснилось, что он возвращает `false`, но это всего лишь особенность реализации. В другой реализации с легкостью может быть возвращено `true` или во время выполнения будет инициирована исключительная ситуация. **Как только вы нарушили соглашение для `equals`, вы просто не можете знать, как поведут себя другие объекты, столкнувшись с вашим объектом.**

Чтобы устранить эту проблему, просто удалите из метода `equals` неудавшуюся попытку взаимодействовать с классом `String`. Как только вы сделаете это, вы сможете перестроить этот метод так, чтобы он содержал один оператор возврата:

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

Транзитивность. Третье требование в соглашениях для метода `equals` гласит, что если один объект равен второму, а второй объект равен третьему, то и первый объект должен быть равен третьему объекту. И вновь несложно представить непреднамеренное нарушение этого требования. Рассмотрим случай с программистом, который создает подкласс, придающий своему суперклассу новый аспект. Иными словами, подкласс привносит немного информации, оказывающей влияние на процедуру сравнения. Начнем с простого неизменяемого класса, соответствующего точке в двухмерном 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;  
    }  
  
    ... // Остальное опущено  
}
```

Предположим, что вы хотите расширить этот класс, добавив 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;  
    }  
    ... // Остальное опущено  
}
```

Как должен выглядеть метод `equals`? Если вы оставите его как есть, реализация метода будет наследоваться от класса `Point`, и информация о цвете при сравнении с помощью методов `equals` будет игнорироваться. Хотя такое решение и не нарушает общих соглашений для метода `equals`, очевидно, что оно неприемлемо. Допустим, вы пишете метод `equals`, который возвращает значение `true`, только если его аргументом является цветная точка, имеющая то же положение и тот же цвет:

```
// Ошибка -- нарушение симметрии!
@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 вы можете получить разные результаты, сравнивая обычную точку с цветной и наоборот. Прежняя процедура сравнения игнорирует цвет, а новая всегда возвращает `false` из-за того, что указан неправильный тип 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);
```

После этого выражение `p.equals(cp)` возвратит `true`, а `cp.equals(p)` возвратит `false`. Вы можете попытаться решить эту проблему, заставив метод `ColorPoint.equals` игнорировать цвет при выполнении «смешанных сравнений»:

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

    // Если о – обычная точка, выполнить сравнение без проверки цвета
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // Если о – цветная точка, выполнить полное сравнение
    return super.equals(o) && cp.color == color;
}
```

Такой подход обеспечивает симметрию, но за счет транзитивности:

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

В этот момент выражения `p1.equals(p2)` и `p2.equals(p3)` возвращают значение `true`, а `p1.equals(p3)` возвращает `false` — прямое нарушение транзитивности. Первые два сравнения игнорируют цвет, в третьем цвет учитывается.

Так где же решение? Оказывается, это фундаментальная проблема эквивалентных отношений в объектно ориентированных языках. **Не существует способа расширить класс, порождающий экземпляры, и добавить к нему компонент значения, сохранив при этом соглашения для метода equals**, если только вы не желаете воздержаться от использования преимуществ объектно ориентированной абстракции.

Вы можете услышать мнение, что можно расширить порождающий экземпляры класс и добавить компонент значения с сохранением соглашений, используя проверку `getClass` вместо проверки `instanceof` в методе `equals`:

```
// Ошибка – нарушен принцип подстановки Барбары Лисков (с. 40)
@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;
}
```

Это имеет воздействие на объекты сравнения, только если у них одна и та же реализация класса. Хотя это, может, и выглядит неплохо, но результат неприемлем.

Предположим, что мы хотим написать метод определяющий, является ли точка целого числа частью единичной окружности. Есть только один способ сделать это:

```
// Инициализируем единичную окружность, содержащую все точки
// этой окружности.
private static final Set<Point> unitCircle;
static {
    unitCircle = new HashSet<Point>();
    unitCircle.add(new Point(1, 0));
```

```

        unitCircle.add(new Point( 0, 1));
        unitCircle.add(new Point(-1, 0));
        unitCircle.add(new Point( 0, -1));
    }
    public static boolean onUnitCircle(Point p) {
        return unitCircle.contains(p);
    }
}

```

Может, это и не самый быстрый способ применения данной функции, но работает он превосходно. Предположим, вы хотите расширить Point неким тривиально простым способом, который не добавляет значение компоненту, скажем, используя его конструктор для отслеживания, сколько экземпляров 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 int numberCreated() { return counter.get(); }
}

```

Принцип подстановки Барбары Лисков утверждает, что любое важное свойство типа должно содержаться также в его подтипе. Таким образом, любой метод, написанный для типа, должен также работать и на его подтипах [Liskov87]. Но предположим, мы передаем экземпляр CounterPoint методу onUnitCircle. Если класс Point использует метод equals, основанный на getClass, то метод onUnitCircle возвратит значение false независимо от значений x и y экземпляра CounterPoint. Это происходит потому, что коллекция, такая как HashSet, используемая методом onUnitCircle, использует метод equals для проверки содержимого, и ни один экземпляр CounterPoint не равен ни одному экземпляру Point. Если тем не менее вы корректным образом используете метод equals на ос-

нове instanceof на экземпляре Point, тот же самый метод onUnitCircle будет работать хорошо, если будет представлен экземпляром CounterPoint.

В то время как нет удовлетворительного способа расширить порождающий экземпляры класс и добавить компоненты значений, есть замечательный обходной путь. Следуйте рекомендациям из статьи 16, «Наследованию предпочтайте компоновку». Вместо того чтобы экземпляром ColorPoint расширять экземпляр Point, создайте в ColorPoint закрытое поле Point и открытый метод view (статья 5), который возвратил бы обычную точку в ту же самую позицию, что и цветную точку

```
// Добавляет компонент значения, не нарушая соглашения equals.  
public class ColorPoint {  
    private final Point point;  
    private final Color color;  
    public ColorPoint(int x, int y, Color color) {  
        if (color == null)  
            throw new NullPointerException();  
        point = new Point(x, y);  
        this.color = color;  
    }  
    /**  
     * Возвращает вид этой цветной точки.  
     */  
    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);  
    }  
    ... // Остальное опущено  
}
```

В библиотеках для платформы Java содержатся некоторые классы, которые являются подклассами для класса, создающего экземпляры, и при этом придают ему новый аспект. Например, `java.sql.Timestamp` является подклассом класса `java.util.Date` и добавляет поле для наносекунд. Реализация метода `equals` в `Timestamp` нарушает правило симметрии, и это может привести к странному поведению программы, если объекты `Timestamp` и `Date` использовать в одной коллекции или смешивать еще как-нибудь иначе. В документации к классу `Timestamp` есть предупреждение, предостерегающее программиста от смешивания объектов `Date` и `Timestamp`. Пока вы не смешиваете их, у вас проблем не будет, однако ничто не помешает вам сделать это, и устранение возникших в результате ошибок может быть непростым. Такое поведение класса `Timestamp` не является правильным, и подражать ему не надо.

Заметим, что вы можете добавить аспект в подклассе абстрактного класса, не нарушая при этом соглашений для метода `equals`. Это важно для тех разновидностей иерархии классов, которые вы получите, следуя совету из статьи 20: «Объединение заменяйте иерархией классов». Например, вы можете иметь простой абстрактный класс `Shape`, а также подклассы `Circle`, добавляющий поле радиуса, и `Rectangle`, добавляющий поля длины и ширины. И только что продемонстрированные проблемы не будут возникать до тех пор, пока нет возможности создавать экземпляры суперкласса.

Непротиворечивость. Четвертое требование в соглашениях для метода `equals` гласит, что если два объекта равны, они должны быть равны все время, пока один из них (или оба) не будет изменен. Это не столько настоящее требование, сколько напоминание о том, что изменяемые объекты в разное время могут быть равны разным объектам, а неизменяемые объекты — не могут. Когда вы пишете класс, хорошо подумайте, не следует ли его сделать неизменяемым (статья 15). Если вы решите, что это необходимо, позаботьтесь о том, чтобы ваш метод `equals` выполнял это ограничение: равные объекты должны оставаться все время равными, а неравные объекты — соответственно, неравными.

Вне зависимости от того, является ли класс неизменяемым или нет, **не ставьте метод `equals` в зависимость от ненадежных ресурс-**

сов. Очень трудно соблюдать требование непротиворечивости, если вы нарушаете его запреты. Например, метод `equals`, принадлежащий `java.net.URL`, полагается на сравнение IP адресов для хоста, ассоциирующегося с этим 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 метод `URL.equals` нарушит соглашения `equals` и на практике будут наблюдаться проблемы. (К сожалению, такое поведение невозможно изменить в связи с требованиями совместимости.) За очень небольшим исключением, методы `equals` должны выполнять детерминистские расчеты на находящихся в памяти объектах.

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 нуля (*null*). Хотя трудно себе представить, чтобы в ответ на вызов `o.equals(null)` будет случайно возвращено значение `true`, все же нетрудно представить случайное инициирование исключительной ситуации `NullPointerException`. Общие соглашения этого не допускают. Во многих классах методы `equals` имеют защиту в виде явной проверки аргумента на `null`:

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

Такая проверка не является обязательной. Чтобы проверить равенство аргумента, метод `equals` должен сначала привести аргумент к нужному типу, чтобы затем можно было воспользоваться соответствующими механизмами доступа или напрямую обращаться к его полям. Перед приведением типа метод `equals` должен воспользоваться оператором `instanceof`, чтобы проверить, что аргумент имеет правильный тип:

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof MyType))
```

```
    return false;  
  
}
```

Если бы эта проверка типа отсутствовала, а метод equals получил аргумент неправильного типа, то он бы инициировал исключительную ситуацию ClassCastException, что нарушает соглашения для метода equals. Однако здесь представлен оператор instanceof, и если его первый operand равен null, то, вне зависимости от типа второго операнда, он возвратит false [JLS, 15.20.2]. Поэтому, если был передан null, проверка типа возвратит false и, соответственно, вам нет необходимости делать отдельную проверку для null. Собрав все это вместе, получаем рецепт для создания высококачественного метода equals:

- 1. Используйте оператор == для проверки, является ли аргумент ссылкой на указанный объект.** Если это так, возвращайте true. Это всего лишь способ повысить производительность программы, которая будет низкой, если процедура сравнения может быть трудоемкой.
- 2. Используйте оператор instanceof для проверки, имеет ли аргумент правильный тип.** Если это не так, возвращайте false. Обычно правильный тип — это тип того класса, которому принадлежит данный метод. В некоторых случаях это может быть какой-либо интерфейс, реализуемый этим классом. Если класс реализует интерфейс, который уточняет соглашения для метода equals, то в качестве типа указывайте этот интерфейс, что позволит выполнять сравнение классов, реализующих этот интерфейс. Подобным свойством обладают интерфейсы коллекций Set, List, Map и Map.Entry.
- 3. Приводите аргумент к правильному типу.** Поскольку эта операция следует за проверкой instanceof, она гарантированно будет выполнена.
- 4. Пройдитесь по всем «значимым» полям класса и убедитесь в том, что значение такого поля в аргументе и значение того же поля в объекте соответствуют друг другу.**

Если проверки для всех полей прошли успешно, возвращайте результат `true`, в противном случае возвращайте `false`. Если на шаге 2 тип был определен как интерфейс, вы должны получать доступ к значимым полям аргумента, используя методы самого интерфейса. Если же тип аргумента определен как класс, то, в зависимости от условий, вам, возможно, удастся получить прямой доступ к полям аргумента.

Для простых полей, за исключением типов `float` и `double`, для сравнения используйте оператор `==`. Для полей со ссылкой на объекты рекурсивно вызывайте метод `equals`. Для поля `float` используйте метод `Float.compare`. Для полей `double` используйте метод `Double.compare`. Особая процедура обработки полей `float` и `double` нужна потому, что существуют особые значения `Float.NaN`, `-0.0f`, а также аналогичные значения для типа `double`. Подробности см. в документации к `Float.equals`. При работе с полями массивов применяйте методы `Array.equals`, появившиеся в версии 1.5. Некоторые поля, предназначенные для ссылки на объекты, вполне оправданно могут иметь значение `null`. Чтобы не допустить возникновения исключительной ситуации `NullPointerException`, для сравнения подобных полей используйте следующую идиому:

```
(field == null ? o.field == null : field.equals(o.field))
```

Если `field` и `o.field` часто ссылаются на один и тот же объект, следующий альтернативный вариант может оказаться быстрее:

```
(field == o.field || (field != null && field.equals(o.field)))
```

Для некоторых классов, таких как представленный ранее `CaseInsensitiveString`, сравнение полей оказывается гораздо более сложным, чем простая проверка равенства. Если это так, то вам, возможно, потребуется каждому объекту придать некую каноническую форму. Таким образом, метод `equals` сможет выполнять простое и точное сравнение этих канонических форм вместо того, чтобы пользоваться более трудоемким и неточным вариантом сравнения. Описанный прием более подходит для неизменяемых классов (статья 15),

поскольку, когда объект меняется, приходится приводить и его каноническую форму в соответствие последним изменениям.

На производительность метода `equals` может оказывать влияние очередность сравнения полей. Чтобы добиться наилучшей производительности, вы должны в первую очередь сравнивать те поля, которые будут различаться с большей вероятностью, либо те, которые сравнивать проще. В идеале оба этих качества должны совпадать. Не следует сравнивать поля, которые не являются частью логического состояния объекта, например поля `Object`, которые используются для синхронизации операций. Вам нет необходимости сравнивать избыточные поля, значение которых можно вычислить, отталкиваясь от «значащих полей» объекта, однако сравнение этих полей может повысить производительность метода `equals`. И если значение избыточного поля равнозначно суммарному описанию объекта в целом, то сравнение подобных полей позволит вам сэкономить на сравнении действительных данных, если будет выявлено расхождение.

5. Закончив написание собственного метода `equals`, задайте себе три вопроса: является ли он симметричным, является ли транзитивным и является ли непротиворечивым? И не просто задайте себе вопрос, напишите тест для проверки соблюдений этих условий. Если ответ отрицательный, разберитесь, почему не удалось реализовать эти свойства, и подправьте метод соответствующим образом. Конечно же, ваш метод `equals` должен отвечать и двум другим свойствам (рефлексивности и отличию от нулевого значения).

В качестве конкретного примера метода `equals`, который был выстроен по приведенному выше рецепту, можно посмотреть `PhoneNumber.equals` из статьи 9. Несколько заключительных предостережений:

- **Переопределяя метод `equals`, всегда переопределяйте метод `hashCode` (статья 9).**
- **Не старайтесь быть слишком умными.** Если вы просто проверяете равенство полей, соблюдать условия соглашений для

метода `equals` совсем не трудно. Если же в поисках равенства вы излишне агрессивны, можно легко нарваться на неприятности. Так, использование синонимов в каком бы то ни было обличье обычно оказывается плохим решением. Например, класс `File` не должен пытаться считать равными символьные связи, которые относятся к одному и тому же файлу. К счастью, он этого и не делает.

- **Декларируя метод `equals`, не надо указывать вместо `Object` другие типы объектов.** Нередко программисты пишут метод `equals` следующим образом, а потом часами ломают голову над тем, почему он не работает правильно:

```
public boolean equals(MyClass o) {  
}
```

Проблема заключается в том, что этот метод не переопределяет (*override*) метод `Object.equals`, чей аргумент имеет тип `Object`, а перегружает его (*overload*, статья 41). Подобный «строго типизированный» метод `equals` можно создать в дополнение к обычному методу `equals`, однако, поскольку оба метода возвращают один и тот же результат, нет никакой причины это делать. Хотя при определенных условиях это может дать минимальный выигрыш в производительности, но это не оправдывает дополнительного усложнения программы (статья 55).

Слайд 9

Переопределяя метод `equals`, всегда переопределяйте `hashCode`

Распространенным источником ошибок является то, что нет переопределения метода `hashCode`. **Вы должны переопределять метод `hashCode` в каждом классе, где переопределен метод `equals`.** Невыполнение этого условия приведет к нарушению общих согла-

шений для метода `Object.hashCode`, а это не позволит вашему классу правильно работать в сочетании с любыми коллекциями, построенными на использовании хэш-таблиц, в том числе с `HashMap`, `HashSet` и `HashTable`.

Приведем текст соглашений, представленных в спецификации `Object` [JavaSE6]:

- Если во время работы приложения несколько раз обратиться к одному и тому же объекту, метод `hashCode` должен постоянно возвращать одно и то же целое число, показывая тем самым, что информация, которая используется при сравнении этого объекта с другими (метод `equals`), не поменялась. Однако, если приложение остановить и запустить снова, это число может стать другим.
- Если метод `equals(Object)` показывает, что два объекта равны друг другу, то, вызвав для каждого из них метод `hashCode`, вы должны получить в обоих случаях одно и то же целое число.
- Если метод `equals(Object)` показывает, что два объекта друг другу не равны, вовсе не обязательно, что метод `hashCode` возвратит для них разные числа. Между тем программист должен понимать, что генерация разных чисел для неравных объектов может повысить эффективность хэш-таблиц.

Главным условием является второе: равные объекты должны иметь одинаковый хэш-код. Если вы не переопределите метод `hashCode`, оно будет нарушено: два различных экземпляра с точки зрения метода `equals` могут быть логически равны, однако для метода `hashCode` из класса `Object` это всего лишь два объекта, не имеющие между собой ничего общего. Поэтому метод `hashCode`, скорее всего, возвратит для этих объектов два случайных числа, а не одинаковых, как того требует соглашение.

В качестве примера рассмотрим следующий упрощенный класс `PhoneNumber`, в котором метод `equals` построен по рецепту из статьи 8:

```
public final class PhoneNumber {  
    private final short areaCode;  
    private final short prefix;  
    private final short lineNumber;  
  
    public PhoneNumber(int areaCode, int prefix,  
                      int lineNumber) {  
        rangeCheck(areaCode, 999, "area code");  
        rangeCheck(exchange, 999, "prefix");  
        rangeCheck(extension, 9999, "line number");  
        this.areaCode = (short) areaCode;  
        this.prefix = (short) prefix;  
        this.lineNumber = (short) lineNumber;  
    }  
    private static void rangeCheck(int arg, int max,  
                                   String name) {  
        if (arg < 0 || arg > max)  
            throw new IllegalArgumentException(name + ": " + arg);  
    }  
  
    @Override public boolean equals(Object o) {  
        if (o == this)  
            return true;  
        if (!(o instanceof PhoneNumber))  
            return false;  
        PhoneNumber pn = (PhoneNumber)o;  
        return pn.lineNumber == lineNumber &&  
               pn.prefix == prefix &&  
               pn.areaCode == areaCode;  
    }  
  
    // Ошибка Нет метода hashCode!  
  
    ... // Остальное опущено  
}
```

Предположим, вы попытались использовать этот класс с `HashMap`:

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

После этого вы вправе ожидать, что `m.get(new PhoneNumber(707, 867, 5309))` возвратит строку «`Jenny`», однако он возвращает `null`. Заметим, что здесь задействованы два экземпляра класса `PhoneNumber`: один используется для вставки в таблицу `HashMap`, другой, равный ему экземпляр — для поиска. Отсутствие в классе `PhoneNumber` переопределенного метода `hashCode` приводит к тому, что двум равным экземплярам соответствует разный хэш-код, т.е. имеем нарушение соглашений для этого метода. Как следствие, метод `get` ищет указанный телефонный номер в другом сегменте хэш-таблицы, а не там, где была сделана запись с помощью метода `put`. Даже если два экземпляра попадут в один и тот же сегмент, метод `get` однозначно выдаст результат `null`, поскольку у `HashMap` есть оптимизация, позволяющая кэшировать хэш-код, связанный с каждой записью, и `HashMap` не озадачивается проверкой равенства объектов, если хэш-код не совпадает.

Разрешить эту проблему можно, просто поместив в класс `PhoneNumber` правильный метод `hashCode`. Так как же должен выглядеть метод `hashCode`? Написать действующий, но не слишком хороший метод очень просто. Например, следующий метод всегда приемлем, но пользоваться им не надо никогда:

```
// Самая плохая из допустимых хэш-функций - никогда ею не пользуйтесь!  
@Override public int hashCode() { return 42; }
```

Этот метод приемлем, поскольку для равных объектов он гарантирует возврат одного и того же хэш-кода. Это ужасно, поскольку он гарантирует получение одного и того же хэш-кода для любого объекта. Соответственно, любой объект будет привязан к одному и тому же сегменту хэш-таблицы, а сами хэш-таблицы вырождаются в связные списки. Для программ, время работы которых с увеличением

нием хэш-таблиц должно увеличиваться линейно, таковое увеличивается в квадратичной зависимости. Для больших хэш-таблиц такое отличие равносильно переходу от работоспособного к неработоспособному варианту.

Хорошая хэш-функция для неравных объектов стремится генерировать различные хэш-коды. И это именно то, что подразумевает третье условия в соглашения для `hashCode`. В идеале хэш-функция должна равномерно распределять любое возможное множество неравных экземпляров класса по всем возможным значениям хэш-кода. Достичь этого идеала может быть чрезвычайно сложно. К счастью, не так трудно получить для него хорошее приближение. Представим простой рецепт:

1. Присвойте переменной `result` (тип `int`) некоторое ненулевое число, скажем, 17.
2. Для каждого значимого поля `f` в вашем объекте (т.е. поля, значение которого принимается в расчет методом `equals`) выполните следующее:
 - a. Вычислите для этого поля хэш-код с (тип `int`):
 - i. Если поле имеет тип `boolean`, вычислите `(f ? 1 : 0)`;
 - ii. Если поле имеет тип `byte`, `char`, `short` или `int`, вычислите `(int)f`;
 - iii. Если поле имеет тип `long`, вычислите `(int)(f ^ (f >> 32))`;
 - iv. Если поле имеет тип `float`, вычислите `Float.floatToIntBits(f)`;
 - v. Если поле имеет тип `double`, вычислите `Double.doubleToLongBits(f)`, а затем преобразуйте полученное значение, как указано в 2.a.iii;
 - vi. Если поле является ссылкой на объект, а метод `equals` данного класса сравнивает это поле, рекурсивно вызывая

вая другие методы `equals`, так же рекурсивно вызывайте для этого поля метод `hashCode`. Если требуется более сложное сравнение, вычислите для данного поля каноническое представление (*canonical representation*), а затем вызовите метод `hashCode` уже для него. Если значение поля равно `null`, возвращайте значение 0 (можно любую другую константу, но традиционно используется 0);

vii. Если поле является массивом, обрабатываете его так, как если бы каждый его элемент был отдельным полем. Иными словами, вычислите хэш-код для каждого значимого элемента, рекурсивно применяя данные правила, а затем объединяя полученные значения так, как описано в 2.b.

b. Объедините хэш-код с, вычисленный на этапе a, с текущим значением поля `result` следующим образом:

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

3. Верните значение `result`.

4. Когда вы закончили писать метод `hashCode`, спросите себя, имеют ли равные экземпляры одинаковый хэш-код. Если нет, выясните, в чем причина, и устраните эту проблему.

Из процедуры получения хэш-кода можно исключить избыточные поля. Иными словами, можно исключить любое поле, чье значение можно вычислить исходя из значений полей, которые уже включены в рассматриваемую процедуру. Вы *обязаны* исключать из процедуры все поля, которые не используются в ходе проверки равенства. Если эти поля не исключить, это может привести к нарушению второго правила в соглашениях для `hashCode`.

На этапе 1 используется ненулевое начальное значение. Благодаря этому не будут игнорироваться те обрабатываемые в первую очередь поля, у которых значение хэш-кода, полученное на этапе 2.a,

оказалось нулевым. Если же на этапе 1 в качестве начального значения использовался нуль, то ни одно из этих обрабатываемых в первую очередь полей не сможет повлиять на общее значение хэш-кода, что может привести к увеличению числа коллизий. Число 17 выбрано произвольно.

Умножение в шаге 2.b создает зависимость значения хэш-кода от очередности обработки полей, а это дает гораздо лучшую хэш-функцию в случае, когда в классе много одинаковых полей. Например, если из хэш-функции для класса `String`, построенной по этому рецепту, исключить умножение, то все анаграммы (слова, полученные от некоего исходного слова путем перестановки букв) будут иметь один и тот же хэш-код. Множитель 31 выбран потому, что является простым нечетным числом. Если бы это было четное число и при умножении произошло переполнение, информация была бы потеряна, поскольку умножение числа на 2 равнозначно его арифметическому сдвигу. Хотя преимущества от использования простых чисел не столь очевидны, именно их принято использовать для этой цели. Замечательное свойство числа 31 заключается в том, что умножение может быть заменено сдвигом и вычитанием для лучшей производительности: `31 * I == (i<<5) - i`. Современные виртуальные машины автоматически выполняют эту небольшую оптимизацию.

Давайте используем этот рецепт для класса `PhoneNumber`. В нем есть три значимых поля, все имеют тип `short`:

```
@Override public int hashCode() {  
    int result = 17;  
    result = 31* result + areaCode;  
    result = 31* result + prefix;  
    result = 31* result + lineNumber;  
    return result;  
}
```

Поскольку этот метод возвращает результат простого детерминированного вычисления, исходными данными для которого являются три значащих поля в экземпляре `PhoneNumber`, то должно быть

понятно, что равные экземпляры PhoneNumber будут иметь равный хэш-код. Фактически, этот метод является абсолютно правильной реализацией hashCode для класса PhoneNumber наравне с методами из библиотек платформ Java. Он прост, достаточно быстр и правильно разносит неравные телефонные номера по разным сегментам хэш-таблицы.

Если класс является неизменным и при этом важны затраты на вычисление хэш-кода, вы можете сохранять хэш-код в самом этом объекте вместо того, чтобы вычислять его всякий раз заново, как только в нем появится необходимость. Если вы полагаете, что большинство объектов данного типа будут использоваться как ключи в хэш-таблице, то вам следует вычислять соответствующий хэш-код уже в момент создания соответствующего экземпляра. В противном случае вы можете выбрать инициализацию, отложенную до первого обращения к методу hashCode (статья 71). Хотя достоинства подобного режима для нашего класса PhoneNumbers не очевидны, давайте покажем, как это делается:

```
// Отложенная инициализация, кэшируемый hashCode
private volatile int hashCode; // (см. статью 71)

@Override public int hashCode() {
    int result = hashCode;
    if (hashCode == 0) {
        int result = 17;
        result = 31 * result + areaCode;
        result = 31 * result + prefix;
        result = 31 * result + lineNumber;
        hashCode = result;
    }
    return result;
}
```

Хотя рецепт, изложенный в этой статье, и позволяет создавать довольно хорошие хэш-функции, он не соответствует ни современным хэш-функциям, ни хэш-функциям из библиотек для платфор-

мы Java, которые реализованы в версии 1.6. Разворотка подобных хэш-функций является предметом активных исследований, которые лучше оставить математикам и ученым, работающим в области теории вычислительных машин. Возможно, в последней версии платформы Java для библиотечных классов будут представлены современные хэш-функции, а также методы-утилиты, которые позволят рядовым программистам самим создавать такие хэш-функции. Пока же приемы, которые описаны в этой статье, приемлемы для большинства приложений.

Повышение производительности не стоит того, чтобы при вычислении хэш-кода игнорировать значимые части объекта. Хотя полученная хэш-функция и может работать быстрее, ее качество может ухудшиться до такой степени, что обработка хэш-таблицы будет производиться слишком медленно. В частности, не исключено, что хэш-функция столкнется с большим количеством экземпляров, которые существенно разнятся как раз в тех частях, которые вы решили игнорировать. Если это произойдет, то хэш-функция сопоставит всем этим экземплярам всего лишь несколько значений хэш-кода. Соответственно, коллекция, основанная на хэш-функциях, будет демонстрировать падение производительности в квадратичной зависимости от числа элементов. Это не просто теоретическая проблема. Хэш-функция класса `String`, реализованная во всех версиях платформы Java до номера 1.2, проверялась самое большое для строк с 16 символами и равномерным распределением пробелов по всей строке, начиная с первого символа. Для больших коллекций иерархических имен, таких как `URL`, эта хэш-функция демонстрировала как раз то патологическое поведение, о котором здесь говорилось.

У многих классов в библиотеках для платформы Java, таких как `String`, `Integer` или `Date`, конкретное значение, возвращаемое методом `hashCode`, определяется как функция от значения экземпляра. Вообще говоря, это не слишком хорошая идея, поскольку она серьезно ограничивает ваши возможности по улучшению хэш-функций в будущих версиях. Если бы вы оставили детали реализации хэш-функции

неконкретизированными и в ней обнаружился изъян, вы бы могли исправить эту хэш-функцию в следующей версии, не опасаясь потерять совместимость с теми клиентами, работа которых зависит от того, какое конкретное значение возвращает хэш-функция.

Статья 10

Всегда переопределяйте метод `toString`

Хотя в классе `java.lang.Object` предусмотрена реализация метода `toString`, строка, которую он возвращает, как правило, совсем не та, которую желает видеть пользователь вашего класса. Она состоит из названия класса, за которым следует символ «коммерческого at» (@) и его хэш-код в виде беззнакового шестнадцатеричного числа, например «`PhoneNumber@163b91`». Общее соглашение для метода `toString` гласит, что возвращаемая строка должна быть «лаконичным, но информативным, легко читаемым представлением объекта» [JavaSE6]. Хотя можно поспорить, является ли строка «`PhoneNumber@163b91`» лаконичной и легко читаемой, она не столь информативна, как, например, «(707) 867–5309». Далее в соглашении для метода `toString` говорится: «Рекомендуется во всех подклассах переопределять этот метод». Хороший совет, ничего не скажешь.

Хотя эти соглашения можно соблюдать не столь строго, как соглашения для методов `equals` и `hashCode` (статьи 8 и 9), однако, **качественно реализовав метод `toString`, вы сделаете свой класс гораздо более приятным в использовании**. Метод `toString` вызывается автоматически, когда ваш объект передается методам `println`, `printf`, оператору сцепления строк или `assert` или выводится отладчиком. (Метод `printf` появился в версии 1.5, как и схожие методы, в том числе `String.format`, который, грубо говоря, равен методу C `sprint`.)

Если вы создали хороший метод `toString`, для `PhoneNumber` получить удобное диагностическое сообщение можно простым способом:

```
System.out.println("Failed to connect: " + phoneNumber);
```

Программисты все равно будут строить такие диагностические сообщения, переопределите вы метод `toString` или нет, и, если этого не сделать, сообщения понятнее не станут. Преимущества от реализации удачного метода `toString` передаются не только на экземпляры этого класса, но и на объекты, которые содержат ссылки на эти экземпляры, особенно это касается коллекций. Что бы вы предпочли увидеть: «`{Jenny=PhoneNumber@163b91}`» или же «`{Jenny=(707) 867-5309}`»?

Будучи переопределен, метод `toString` должен передавать всю полезную информацию, которая содержится в объекте, как это было только что показано на примере с телефонными номерами. Однако это не годится для больших объектов или объектов, состояние которых трудно представить в виде строки. В таких случаях метод `toString` должен возвращать такие резюме, как «`Manhattan white pages (1487536 listings)`» или «`Thread [main, 5, main]`». В идеале полученная строка не должна требовать разъяснений. (Последний пример со `Thread` этому требованию не отвечает.)

При реализации метода `toString` вы должны принять одно важное решение: будете ли вы в документации описывать формат возвращаемого значения. Это желательно делать для классов-значений (*value class*), таких как телефонные номера и таблицы. Задав определенный формат, вы получите то преимущество, что он будет стандартным, однозначным и удобным для чтения представлением соответствующего объекта. Это представление можно использовать для ввода, вывода, а также для создания удобных для прочтения записей в фиксируемых объектах, таких как документы XML. При задании определенного формата, как правило, полезно бывает создать соответствующий конструктор объектов типа `String` или статический метод генерации, что позволит программисту с легкостью осуществлять преобразование в обе стороны между объектом и его строковым представлением. Такой подход используется в библиотеках платформы Java для многих классов-значений, включая `BigInteger`, `BigDecimal` и большинства упакованных примитивных классов.

Неудобство от конкретизации формата значения, возвращаемого методом `toString`, заключается в том, что если ваш класс используется широко, то, задав этот формат один раз, вы оказываетесь привязаны к нему навсегда. Другие программисты будут писать код, который анализирует данное представление, генерируют и используют его при записи объектов в базу данных (*persistent data*). Если в будущих версиях вы поменяете формат представления, они взовут, поскольку вы разрушаете созданные ими код и данные. Отказавшись от спецификации формата, вы сохраняете возможность вносить в него новую информацию, совершенствовать этот формат в последующих версиях.

Будете вы или нет объявлять формат, вы должны четко обозначить ваши намерения. Если вы описываете формат, вы должны сделать это пунктуально. В качестве примера представим метод `toString`, который должен сопровождать класс `PhoneNumber` из статьи 9:

```
/**  
 * Возвращает представление данного телефонного номера в виде строки.  
 * Стока состоит из четырнадцати символов, имеющих формат  
 * «(XXX) YYY-ZZZZ», где XXX – код зоны, YYY – префикс,  
 * ZZZZ – номер линии. (Каждая прописная буква представляет  
 * одну десятичную цифру.)  
 *  
 * Если какая-либо из трех частей телефонного номера слишком мала и  
 * не заполняет полностью свое поле, последнее дополняется ведущими нулями.  
 * Например, если значение номера абонента в АТС равно 123, то последними  
 * четырьмя символами в строковом представлении будут «0123».  
 *  
 * Заметим, что закрывающую скобку, следующую за кодом зоны, и первую  
 * цифру префикса разделяет один пробел.  
 */  
@Override public String toString() {  
    return String.format("(%03d) %03d-%04d",  
        areaCode, prefix, lineNumber);  
}
```

Если вы решили не конкретизировать формат, соответствующий комментарий к документации должен выглядеть примерно так:

```
/**  
 * Возвращает краткое описание этого зелья. Точные детали представления  
 * не конкретизированы и могут меняться, однако следующее представление  
 * может рассматриваться в качестве типичного:  
 *  
 * «[Зелье №9: тип=любовный, аромат=скипидар, вид=тушь]»  
 */  
@Override public String toString() { ... }
```

Прочитав этот комментарий, программисты, разрабатывающие программный код или постоянные данные, которые зависят от особенностей формата, уже не смогут винить никого, кроме самих себя, если формат однажды поменяется.

Вне зависимости от того, описываете вы формат или нет, всегда полезно предоставлять альтернативный программный доступ ко всей информации, которая содержится в значении, возвращаемом методом `toString`. Например, класс `PhoneNumber` должен включать методы доступа к коду зоны, префиксу и номеру линии. Если этого не сделано, то вы вынуждаете программистов, которым нужна эта информация, делать разбор данной строки. Помимо того, что вы снижаете производительность приложения и заставляете программистов делать ненужную работу, это чревато ошибками и приводит к созданию ненадежной системы, которая перестает работать, как только вы поменяете формат. Не предоставив альтернативных методов доступа, вы де-факто превращаете формат строки в элемент API, даже если в документации и указали, что он может быть изменен.

Слайд 71

Соблюдайте осторожность при переопределении метода `clone`

Интерфейс `Cloneable` проектировался в качестве дополнительного интерфейса (*mixin*, статья 18), который позволяет объектам обь-

являть о том, что они могут быть клонированы. К сожалению, он не может использоваться для этой цели. Его основной недостаток — отсутствие метода `clone`: в самом классе `Object` метод `clone` является закрытым. Вы не можете, не обращаясь к механизму отражения свойств (*reflection*) (статья 53), вызывать для объекта метод `clone` лишь на том основании, что он реализует интерфейс `Cloneable`. Даже отражение может закончиться неудачей, поскольку нет гарантии, что у данного объекта есть доступный метод `clone`. Несмотря на этот и остальные недочеты, этот механизм используется настолько широко, что имеет смысл с ним разобраться. В этой статье рассказывается, каким образом создать хороший метод `clone`, обсуждается, когда имеет смысл это делать, а также кратко описываются альтернативные подходы.

Что же делает интерфейс `Cloneable`, который, как оказалось, не имеет методов? Он определяет поведение закрытого метода `clone` в классе `Object`: если какой-либо класс реализует интерфейс `Cloneable`, то метод `clone`, реализованный в классе `Object`, возвратит его копию с копированием всех полей, в противном случае будет инициирована исключительная ситуация `CloneNotSupportedException`. Это совершенно нетипичный способ использования интерфейсов, не из тех, которым следует подражать. Обычно факт реализации некоего интерфейса говорит кое-что о том, что этот класс может делать для своих клиентов. В случае же с интерфейсом `Cloneable` он просто меняет поведение некоего защищенного метода в суперклассе.

Для того чтобы реализация интерфейса `Cloneable` могла оказывать какое-либо воздействие на класс, он сам и все его суперклассы должны следовать довольно сложному, трудно выполнимому и в значительной степени недокументированному протоколу. Получающийся механизм не укладывается в рамки языка Java: объект создается без использования конструктора.

Общие соглашения для метода `clone` достаточно свободны. Они описаны в спецификации класса `java.lang.Object` [Java SE6]:

Метод создает и возвращает копию объекта. Точное значение термина «копия» может зависеть от класса этого объекта. Общая задача ставится так, чтобы для любого объекта `x` оба выражения

`x.clone() != x`

и

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

возвращали `true`, однако эти требования не являются безусловными. Обычно условие состоит в том, чтобы выражение

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

возвращало `true`, однако и это требование не является безусловным. Копирование объекта обычно приводит к созданию нового экземпляра соответствующего класса, при этом может потребоваться также копирование внутренних структур данных. Никакие конструкторы не вызываются.

Такое соглашение создает множество проблем. Условие о том, что «никакие конструкторы не вызываются», является слишком строгим. Правильно работающий метод `clone` может воспользоваться конструкторами для создания внутренних объектов создаваемого клона. Если же класс является окончательным (*final*), метод `clone` может просто вернуть объект, созданный конструктором.

Условие, что `x.clone().getClass()` обычно должно быть тождественно `x.getClass()`, является слишком слабым. Как правило, программисты полагают, что если они расширяют класс и вызывают в полученном подклассе метод `super.clone`, то полученный в результате объект будет экземпляром этого подкласса. Реализовать такую схему, суперкласса можно только одним способом — вернуть объект, полученный в результате вызова метода `super.clone`. Если метод `clone` возвращает объект, созданный конструктором, это будет не верный класс. Поэтому **если в расширяемом классе вы переопределяете метод `clone`, то возвращаемый объект вы должны получать, вызвав `super.clone`.** Если у данного класса все суперклассы выполняют это условие, то рекурсивный вызов метода `super.clone` в конечном счете приведет к вызову метода `clone` из класса `Object` и созданию экземпляра именно того класса, который нужен. Этот механизм отдаленно напоминает автоматическое сцепление конструкторов, за исключением того, что оно не является принудительным.

В версии 1.6 интерфейс Cloneable не раскрывает, какие обязанности берет на себя класс, когда реализует этот интерфейс. **На практике же требования сводятся к тому, что в классе, реализующем интерфейс Cloneable, должен быть представлен правильно работающий открытый метод clone.** Вообще же, выполнить это условие невозможно, если только все суперклассы этого класса не будут иметь правильную реализацию метода `clone`, открытую или защищенную.

Предположим, что вы хотите реализовать интерфейс `Cloneable` с помощью класса, чьи суперклассы имеют правильно работающие методы `clone`. В зависимости от природы этого класса объект, который вы получаете после вызова `super.clone()`, может быть, а может и не быть похож на тот, что вы будете иметь в конечном итоге. С точки зрения любого суперкласса этот объект будет полнофункциональным клоном исходного объекта. Поля, объявленные в вашем классе (если таковые имеются), будут иметь те же значения, что и поля в клонируемом объекте. Если все поля объекта содержат значения простого типа или ссылки на неизменяемые объекты, то возвращаться будет именно тот объект, который вам нужен, и дальнейшая обработка в этом случае не нужна. Такой вариант демонстрирует, например, класс `PhoneNumber` из статьи 9. В этом случае, все, что от вас здесь требуется, — это обеспечить в классе `Object` открытый доступ к защищенному методу `clone`:

```
@Override public PhoneNumber clone() {  
    try {  
        return (PhoneNumber) super.clone();  
    } catch(CloneNotSupportedException e) {  
        throw new AssertionError(); // Этого не может быть  
    }  
}
```

Обратите внимание, что вышеупомянутый метод `clone` возвращает `PhoneNumber`, а не `Object`. В версии 1.5 это можно и нужно делать так, потому что ковариантные возвращаемые типы появились в версии 1.5 как часть родовых типов. Другими словами, теперь можно,

чтобы типы предопределяющего метода были подклассом типов предопределенного метода. Это позволяет предопределяющему методу давать больше информации о возвращаемом объекте и позволяет избежать необходимости передавать ее клиенту. Поскольку `Object.Clone` возвращает `Object`, то `PhoneNumber.clone` должен передавать результат `super.clone()` до того, как вернуть его, но это гораздо предпочтительнее, чем заставлять каждый вызов `PhoneNumber.clone` передавать результаты. Общий принцип здесь: **никогда не заставлять клиента делать что-то, что за него может сделать библиотека.**

Однако, если ваш объект содержит поля, имеющие ссылки на изменяемые объекты, такая реализация метода `clone` может иметь катастрофические последствия. Рассмотрим, например, класс `Stack` из статьи 6:

```
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; // Убираем устаревшую ссылку  
        return result;  
    }
```

```

    }

// Убедиться в том, что в стеке есть место хотя бы еще для одного элемента
private void ensureCapacity() {
    if (elements.length == size)
        elements = Arrays.copyOf(elements, 2 * size + 1);
}
}
}

```

Предположим, вы хотите сделать этот класс клонируемым. Если его метод `clone` просто вернет результат вызова `super.clone()`, полученный экземпляр `Stack` будет иметь правильное значение в поле `size`, однако его поле `elements` будет ссылаться на тот же самый массив, что и исходный экземпляр `Stack`. Следовательно, изменение в оригинале будет нарушать инварианты клона и наоборот. Вы быстро обнаружите, что ваша программа дает бессмысленные результаты либо инициирует исключительную ситуацию `NullPointerException`.

Подобная ситуация не могла бы возникнуть, если бы использовался основной конструктор класса `Stack`. **Метод `clone` фактически работает как еще один конструктор, и вам необходимо убедиться в том, что он не вредит оригинальному объекту и правильно устанавливает инварианты клона.** Чтобы метод `clone` в классе `Stack` работал правильно, он должен копировать содержимое стека. Проще всего это сделать путем рекурсивного вызова метода `clone` для массива `elements`:

```

@Override public Object clone() throws CloneNotSupportedException {
    Stack result = (Stack) super.clone();
    result.elements = (Object[]) elements.clone();
    return result;
}

```

Обратите внимание, что нам нет необходимости передавать результат `elements.clone()` экземпляру `Object[]`. В версии 1.5 вызов метода `clone` на массиве возвращает массив, у которого тип, обрабатываемый в процессе компиляции, точно такой же, как и у клонируемого массива.

Заметим, что такое решение не будет работать, если поле `elements` имеет модификатор `final`, поскольку тогда методу `clone` было бы запрещено помещать туда новое значение. Это фундаментальная проблема: **архитектура клона несовместима с обычным использованием полей final, содержащих ссылки на изменяемые объекты.** Исключение составляют случаи, когда эти изменяемые объекты могут безопасно использовать сразу и объект, и его клон. Чтобы сделать класс клонируемым, возможно потребуется убрать с некоторых полей модификатор `final`.

Не всегда бывает достаточно рекурсивного вызова метода `clone`. Например, предположим, что вы пишете метод `clone` для хэш-таблицы, состоящей из набора сегментов (*buckets*), каждый из которых содержит ссылку на первый элемент в связном списке, содержащем несколько пар `ключ/значение`, или содержит `null`, если этот сегмент пуст. Для лучшей производительности в этом классе вместо `java.util.LinkedList` используется собственный упрощенный связный список:

```
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;  
        }  
    }  
    ... // Остальное опущено  
}
```

Предположим, вы просто рекурсивно клонируете массив `buckets`, как это делалось для класса `Stack`:

```
// Ошибка: объекты будут иметь общее внутреннее состояние!
```

```

@Override public HashTable clone(){
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionException();
    }
}

```

Хотя клон и имеет собственный набор сегментов, последний ссылается на те же связные списки, что и исходный набор, а это может с легкостью привести к непредсказуемому поведению и клона, и оригинала. Для устранения проблемы вам придется отдельно копировать связный список для каждого сегмента. Представим один из распространенных приемов:

```

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;
        }

        // Рекурсивно копирует связный список, начинающийся с указанной записи
        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()
    }
}
...
// Остальное опущено
}
```

Закрытый класс `HashTable.Entry` был привнесен для реализации метода «глубокого копирования» (*deep copy*). Метод `clone` в классе `HashTable` размещает в памяти новый массив `buckets` нужного размера, а затем в цикле просматривает исходный набор `buckets`, выполняя глубокое копирование каждого непустого сегмента. Чтобы скопировать связный список, начинающийся с указанной записи, метод глубокого копирования (`deepCopy`) из класса `Entry` рекурсивно вызывает себя самого. Хотя этот прием выглядит изящно и прекрасно работает для не слишком длинных сегментов, он не слишком хорош для клонирования связных списков, поскольку для каждого элемента в списке он делает в стеке новую запись. И если список `buckets` окажется большим, это может легко вызвать переполнение стека. Чтобы помешать этому случиться, в методе `deepCopy` вы можете заменить рекурсию итерацией:

```
// Копирование в цикле связного списка, начинающегося с указанной записи
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;  
}
```

Окончательный вариант клонирования сложных объектов заключается в вызове метода `super.clone`, установке всех полей в первоначальное состояние и вызове методов более высокого уровня, окончательно определяющих состояние объекта. В нашем случае с классом `HashTable` поле `buckets` должно получить при инициализации новый массив сегментов, а затем для каждой пары ключ / значение в клонируемой хэш-таблице следует вызвать метод `put(key, value)` (в распечатке не показан). При таком подходе обычно получается простой, достаточно элегантный метод `clone`, пусть даже и не работающий столь же быстро, как при прямом манипулировании содержимым объекта и его клона.

Как и конструктор, метод `clone` не должен вызывать каких-либо переопределяемых методов, взятых из создаваемого клона (статья 17). Если метод `clone` вызывает переопределенный метод, то этот метод будет выполняться до того, как подкласс, в котором он был определен, установит для клона нужное состояние. Это вполне может привести к разрушению и клона, и самого оригинала. Поэтому метод `put(key, value)`, о котором говорилось в предыдущем абзаце, должен быть либо непреопределяемым (`final`), либо закрытым. (Если это закрытый метод, то, по-видимому, он является вспомогательным (*helper method*) для другого, открытого и переопределяемого метода.)

Метод `clone` в классе `Object` декларируется как способный инициировать исключительную ситуацию `CloneNotSupportedException`, однако в переопределенных методах `clone` эта декларация может быть опущена. Метод `clone` в окончательном классе не должен иметь такой декларации, поскольку работать с методами, не иницииирующими обрабатываемых исключений, приятнее, чем с теми, которые их инициируют (статья 59). Если же метод `clone` переопределяется в расширяемом классе, а особенно в классе, предназначенном для наследования (статья 17), новый метод `clone` подражает поведению `Object.clone`, он также должен декларироваться как закрытый (`protected`) и иметь декларацию для исключительной ситуации `CloneNot-`

SupportException, и класс не должен реализовывать интерфейс Cloneable. Это дает подклассу свободу выбора, реализовывать Cloneable или нет.

Еще одна деталь заслуживает внимания. Если вы решите сделать клонируемым потоковый класс, помните, что метод clone нужно должным образом синхронизировать, как и любой другой метод (статья 66). Метод Object's clone не синхронизирован, и, хотя это в принципе нормально, вам возможно потребуется написать синхронизированный метод clone, который бы запускал super.clone().

Подведем итоги. Все классы, реализующие интерфейс Cloneable, должны переопределять метод clone как открытый. Этот публичный метод должен сначала вызвать метод super.clone, а затем привести в порядок все поля, подлежащие восстановлению. Обычно это означает копирование всех изменяемых объектов, составляющих внутреннюю «глубинную структуру» клонируемого объекта и замену всех ссылок на эти объекты ссылками на соответствующие копии. Хотя обычно эти внутренние копии можно получить рекурсивным вызовом метода clone, такой подход не всегда является самым лучшим. Если класс содержит одни только поля простого типа и ссылки на неизменяемые объекты, то в таком случае, по-видимому, нет полей, нуждающихся в восстановлении. Из этого правила есть исключения. Например, поле, предоставляющее серийный номер или иной уникальный идентификатор, а также поле, показывающее время создания объекта, нуждаются в восстановлении, даже если они имеют простой тип или являются неизменяемыми.

Так ли нужны все эти сложности? Не всегда. Если вы расширяете класс, реализующий интерфейс Cloneable, у вас практически не остается иного выбора, кроме как реализовать правильно работающий метод clone. В противном случае **вам, по-видимому, лучше отказаться от некоторых альтернативных способов копирования объектов либо от самой этой возможности**. Например, для неизменяемых классов нет смысла поддерживать копирование объектов, поскольку копии будут фактически неотличимы от оригинала.

Изящный подход к копированию объектов – создание конструктора копий или копирование статических методов гене-

рации. Конструктор копий — это всего лишь конструктор, единственный аргумент которого имеет тип, соответствующий классу, где находится этот конструктор, например:

```
public Yum(Yum yum);
```

Небольшое изменение — и вместо конструктора имеем статический метод генерации:

```
public static Yum newInstance(Yum yum);
```

Использование конструктора копий (или, как его вариант, статического метода генерации) имеет много преимуществ перед механизмом Cloneable/clone: оно не связано с рискованным, выходящим за рамки языка Java механизмом создания объектов; не требует следования расплывчатым, плохо документированным соглашениям; не конфликтует с обычной схемой использования полей final; не требует от клиента перехвата ненужных исключений; наконец, клиент получает объект строго определенного типа. Конструктор копий или статический метод генерации невозможно поместить в интерфейс, Cloneable не может выполнять функции интерфейса, поскольку не имеет открытого метода clone. Поэтому нельзя утверждать, что, когда вместо метода clone вы используете конструктор копий, вы отказываетесь от возможностей интерфейса.

Более того, конструктор копий (или статический метод генерации) может иметь аргумент, тип которого соответствует интерфейсу, реализуемому этим классом. Например, все реализации коллекций общего назначения, по соглашению, имеют конструктор копий с аргументом типа Collection или Map. Конструкторы копий и методы статической генерации, использующие интерфейсы, позволяют клиенту выбирать для копии вариант реализации вместо того, чтобы принуждать его принять реализацию исходного класса. Например, допустим, у вас есть объект HashSet s, и вы хотите скопировать его как экземпляр TreeSet. Метод clone не предоставляет такой возможности, хотя это легко делается с помощью конструктора копий: new TreeSet(s).

Рассмотрев все проблемы, связанные с интерфейсом Cloneable, можно с уверенностью сказать, что остальные интерфейсы не должны становиться его расширением, а классы, которые предназначены для наследования (статья 17), не должны его реализовывать. Из-за множества недостатков этого интерфейса некоторые высококвалифицированные программисты просто предпочитают никогда не переопределять метод `clone` и никогда им не пользоваться за исключением, быть может, простого копирования массивов. Учтите, что, если в классе, предназначенном для наследования, вы не создадите по меньшей мере правильно работающий защищенный метод `clone`, реализация интерфейса Cloneable в подклассах станет невозможной.

Статья
12

Подумайте над реализацией интерфейса Comparable

В отличие от других обсуждавшихся в этой главе методов метод `compareTo` в классе `Object` не декларируется. Пожалуй, это единственный такой метод в интерфейсе `java.lang.Comparable`. По своим свойствам он похож на метод `equals` из класса `Object`, за исключением того, что помимо простой проверки равенства он позволяет выполнять упорядочивающее сравнение. Реализуя интерфейс `Comparable`, класс показывает, что его экземпляры обладают естественным свойством упорядочения (*natural ordering*). Сортировка массива объектов, реализующих интерфейс `Comparable`, выполняется просто:

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

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

```

public class WordList {
    public static void main(String[] args) {
        Set s = new TreeSet();
        s.addAll(Arrays.asList(args));
        System.out.println(s);
    }
}

```

Реализуя интерфейс Comparable, вы разрешаете вашему классу взаимодействовать со всем обширным набором общих алгоритмов и реализаций коллекций, которые связаны с этим интерфейсом. Приложив немного усилий, вы получаете огромное множество возможностей. Практически все классы значений в библиотеках платформы Java реализуют интерфейс Comparable. И если вы пишете класс значений с очевидным свойством естественного упорядочивания — алфавитным, числовым либо хронологическим, — вы должны хорошо подумать над реализацией этого интерфейса.

```

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

```

Общее соглашение для метода compareTo имеет тот же характер, что и соглашение для метода equals:

Выполняет сравнение текущего и указанного объекта и определяет их очередность. Возвращает отрицательное целое число, нуль или положительное целое число, в зависимости от того, меньше ли текущий объект, равен или, соответственно, больше указанного объекта. Если тип указанного объекта не позволяет сравнивать его с текущим объектом, инициируется исключительная ситуация ClassCastException.

В следующем описании запись `sgn(выражение)` обозначает математическую функцию *signum*, которая, по определению, возвращает -1 , 0 или 1 , в зависимости от того, является ли значение выражения отрицательным, равным нулю или положительным.

- Разработчик должен гарантировать тождество $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ для всех x и y . (Это подразумевает, что выражение $x.\text{compareTo}(y)$ должно инициировать исключительную ситуацию тогда и только тогда, когда $y.\text{compareTo}(x)$ инициирует исключение.)
- Разработчик должен также гарантировать транзитивность отношения: $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$ подразумевает $x.\text{compareTo}(z) > 0$.
- Наконец, разработчик должен гарантировать, что из тождества $x.\text{compareTo}(y) == 0$ вытекает тождество $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$ для всех z .
- Настоятельно (хотя и не безусловно) рекомендуется выполнять условие $(x.\text{compareTo}(y) == 0) == (x.equals(y))$. Вообще говоря, для любого класса, который реализует интерфейс Comparable, но нарушает это условие, сей факт должен быть четко оговорен. Рекомендуется использовать следующую формулировку: «Примечание: данный класс имеет естественное упорядочение, не согласующееся с условием равенства».

Пускай математическая природа этого соглашения у вас не вызывает отвращения. Как и соглашения для метода equals (статья 8), соглашения для compareTo не так сложны, как это кажется. Для одного класса любое разумное отношение упорядочения будет соответствовать соглашениям для compareTo. Для сравнения разных классов метод compareTo, в отличие от метода equals, использоваться не должен: если сравниваются две ссылки на объекты различных классов, можно инициировать исключительную ситуацию ClassCastException. В подобных случаях метод compareTo обычно *так делает и должен так делать*, если параметры класса заданы верно. И хотя представленное соглашение не исключает сравнения между классами, в библиотеках для платформы Java, в частности в версии 1.6, нет классов, которые бы такую возможность поддерживали.

Точно так же, как класс, нарушающий соглашения для метода hashCode, может испортить другие классы, работа которых зависит от хэширования, класс, нарушающий соглашения для метода compareTo, способен нарушить работу других классов, использующих сравнение. К классам, связанным со сравнением, относятся упорядоченные коллекции, TreeSet и TreeMap, а также вспомогательные классы Collections и Arrays, содержащие алгоритмы поиска и сортировки.

Рассмотрим условия соглашения для 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, 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 показывать одинаковый результат.

Из этих трех условий следует, что проверка равенства, осуществляемая с помощью метода compareTo, должна подчиняться тем же самым ограничениям, которые продиктованы соглашениями для метода equals: рефлексивность, симметрия, транзитивность и отличие от null. Следовательно, здесь можно дать то же самое предупреждение: невозможно расширить порождающий экземпляры класс, вводя новый аспект и не нарушая при этом соглашения для метода compareTo (статья 8). Возможен и обходной путь. Если вы хотите добавить важное свойство к классу, реализующему интерфейс Comparable, не расширять его, а напишите новый независимый класс, в котором для исходного класса выделено отдельное поле. Затем добавьте метод представления, возвращающий значение этого поля. Это даст вам возможность реализовать во втором классе любой метод compareTo, который вам нравится. При этом клиент при необходимости может рассматривать экземпляр второго класса как экземпляр первого класса.

Последний пункт соглашений для compareTo, являющийся скорее сильным предположением, чем настоящим условием, постулирует, что проверка равенства, осуществляемая с помощью метода compareTo, обычно должна давать те же самые результаты, что и метод equals. Если это условие выполняется, считается, что упорядочение, задаваемое методом compareTo, согласуется с проверкой равенства (*consistent with equals*). Если же оно нарушается, то упорядочение называется не согласующимся с проверкой равенства (*inconsistent with equals*). Класс, чей метод compareTo устанавливает порядок, не согласующийся с условием равенства, будет работоспособен, однако отсортированные коллекции, содержащие элементы этого класса, могут не соответствовать общим соглашениям для соответствующих интерфейсов коллекций (Collection, Set или Map). Дело в том, что общие соглашения для этих интерфейсов определяются в терминах метода equals, тогда как в отсортированных коллекциях используется проверка равенства, которая реализуется методом compareTo, а не equals. Если это произойдет, катастрофы не случится, но иногда это следует осознавать.

Например, рассмотрим класс BigDecimal, чей метод compareTo не согласуется с проверкой равенства. Если вы создадите HashSet и добавите в него новую запись BigDecimal(«1.0»), а затем BigDecimal(«1.00»), то этот набор будет содержать два элемента, поскольку два добавленных в этот набор экземпляра класса BigDecimal не будут равны, если их сравнивать с помощью метода equals. Однако, если вы выполняете ту же самую процедуру с TreeSet, а не HashSet, полученный набор будет содержать только один элемент, поскольку два представленных экземпляра BigDecimal будут равны, если их сравнивать с помощью метода compareTo. (Подробнее см. документацию на BigDecimal.)

Процедура написания метода compareTo похожа на процедуру для метода equals, но есть несколько ключевых различий. Перед преобразованием типа нет необходимости проверять тип аргумента. Если аргумент имеет неправильный тип, метод compareTo обязан инициализировать исключительную ситуацию ClassCastException. Если аргумент имеет значение null, метод compareTo обязан инициализировать

исключительную ситуацию `NullPointerException`. В точности то же самое вы получите, если просто приведете аргумент к правильному типу, а затем попытаетесь обратиться к его членам.

Сравнение полей в методе `compareTo` является упорядочивающим сравнением, а не сравнением с проверкой равенства. Сравнение полей, имеющих ссылки на объекты, осуществляйте путем рекурсивного вызова метода `compareTo`. Если поле не реализует интерфейс `Comparable` или вам необходимо нестандартное упорядочение, вы можете вместо всего этого использовать явную реализацию интерфейса `Comparator`. Либо пишите ваш собственный метод, либо воспользуйтесь уже имеющимся, как это было в случае с методом `compareTo` в классе `CaseInsensitiveString` из статьи 8:

```
public final class CaseInsensitiveString
    implements Comparable<CaseInsensitiveString> {
    public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Остальное опущено
}
```

Обратите внимание, что класс `CaseInsensitiveString` реализует интерфейс `Comparable<CaseInsensitiveString>`. Это значит, что ссылка на `CaseInsensitiveString` может сравниваться только с другими ссылками на `CaseInsensitiveString`. Это нормальный пример, которому нужно следовать, объявляя классы для реализации интерфейса `Comparable`. Обратите внимание, что параметром метода `CompareTo` является `CaseInsensitiveString`, а не `Object`. Это необходимо для ранее упомянутого декларирования классов.

Поля простого типа нужно сравнивать с помощью операторов. Для сравнения полей с плавающей точкой используйте `Double.Compare` или `Float.Compare` вместо операторов соотношения, которые не соблюдают соглашения для метода `compareTo`, если их применять для значений с плавающей точкой. Для сравнения массивов используйте данные инструкции для каждого элемента.

Если у класса есть несколько значимых полей, порядок их сравнения критически важен. Вы должны начать с самого значимого поля и затем следовать в порядке убывания значимости. Если сравнение дает что-либо помимо нуля (который означает равенство), все, что вам нужно сделать, — просто возвратить этот результат. Если самые значимые поля равны, продолжайте сравнивать следующие по значимости поля и т.д. Если все поля равны, равны и объекты, а потому возвращайте нуль. Такой прием демонстрирует метод compareTo для класса PhoneNumber из статьи 9:

```
public int compareTo(PhoneNumber pn) {  
    // Сравниваем коды зон  
    if (areaCode < pn.areaCode)  
        return -1;  
    if (areaCode > pn.areaCode)  
        return 1;  
  
    // Коды зон равны, сравниваем префиксы  
    if (prefix < pn.prefix)  
        return -1;  
    if (prefix > pn.prefix)  
        return 1;  
  
    // Коды зон и префиксы равны, сравниваем номера линий  
    if (lineNumber < pn.lineNumber)  
        return -1;  
    if (lineNumber > pn.lineNumber)  
        return 1;  
    return 0; // Все поля равны  
}
```

Этот метод работает прекрасно, его можно улучшить. Вспомните, что в соглашениях для метода compareTo величина возвращаемого значения не конкретизируется, только знак. Вы можете извлечь из этого пользу, упростив программу и, возможно, заставив ее работать немного быстрее:

```
public int compareTo(PhoneNumber pn) {  
    // Сравниваем коды зон  
    int areaCodeDiff = areaCode - pn.areaCode;  
    if (areaCodeDiff != 0)  
        return areaCodeDiff;  
  
    // Коды зон равны, сравниваем префиксы  
    int prefixDiff = prefix - pn.prefix;  
    if (prefixDiff != 0)  
        return prefixDiff;  
  
    // Коды зон и номера АТС равны, сравниваем номера линий  
    return lineNumber - pn.lineNumber;  
}
```

Такая уловка работает прекрасно, но применять ее следует крайне осторожно. Не пользуйтесь ею, если у вас нет уверенности, что рассматриваемое поле не может иметь отрицательное значение, или, что бывает еще чаще, разность между наименьшим и наибольшим возможными значениями поля меньше или равна значению `Integer.MAX_VALUE` ($2^{31} - 1$). Причина, по которой этот прием не всегда работает, обычно заключается в том, что 32-битное целое число со знаком является недостаточно большим, чтобы показать разность двух 32-битных целых чисел с произвольным знаком. Если i — большое положительное целое число, а j — большое отрицательное целое число, то при вычислении разности $(i-j)$ произойдет переполнение и будет возвращено отрицательное значение. Соответственно, полученный нами метод `compareTo` работать не будет: для некоторых аргументов будет возвращен бессмысленный результат, тем самым будут нарушены первое и второе условия соглашения для метода `compareTo`. И эта проблема не является чисто теоретической, она уже вызывала сбои в реальных системах. Выявить причину подобных отказов может быть трудно, поскольку неправильный метод `compareTo` со многими входными значениями работает правильно.

4

Г л а в а

Классы и интерфейсы

Классы и интерфейсы занимают в языке программирования Java центральное положение. Они являются основными элементами абстракции. Язык Java содержит множество мощных элементов, которые можно использовать при построении 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.

Статья
13

Сводите к минимуму доступность классов и членов

Единственный чрезвычайно важный фактор, отличающий хорошо спроектированный модуль от неудачного, — степень сокрытия от других модулей его внутренних данных и других деталей реализации. Хорошо спроектированный модуль скрывает все детали реализации, четко разделяя свой API и его реализацию. Модули взаимодействуют друг с другом только через свои API, и ни один из них не знает, какая обработка происходит внутри другого модуля. Представленная концепция, называемая сокрытием информации (*information hiding*) или инкапсуляцией (*encapsulation*), представляет собой один из фундаментальных принципов разработки программного обеспечения [Parnas72].

Сокрытие информации важно по многим причинам, большинство которых связано с тем обстоятельством, что этот механизм эффективно изолирует друг от друга модули, составляющие систему, позволяя разрабатывать, тестировать, оптимизировать, использовать, исследовать и обновлять их по отдельности. Благодаря этому ускоряется разработка системы, поскольку различные модули могут создаваться параллельно. Кроме того, уменьшаются расходы на сопровождение приложения, поскольку каждый модуль можно быстро изучить и отладить, минимально рискуя навредить остальным модулям. Само по себе сокрытие информации не может обеспечить хорошей производительности, но оно создает условия для эффективного управления производительностью. Когда разработка системы завершена и процедура ее профилирования показала, работа каких модулей вызывает падение производительности (статья 55), можно заняться их оптимизацией, не нарушая правильной работы остальных модулей. Сокрытие информации повышает возможность повторного использования программ, поскольку каждый отдельно взятый модуль независим от остальных модулей и часто оказывается полезен в иных контекстах, чем тот, для которого он разрабатывался. Наконец, сокрытие информации уменьшает риски при построении больших систем: удачными могут оказаться отдельные модули, даже если в целом система не будет пользоваться успехом.

Язык программирования Java имеет множество возможностей для сокрытия информации. Одна из таких функций — механизм управления доступом (*access control*) [JLS, 6.6], задающий степень доступности (*accessibility*) для интерфейсов, классов и членов классов. Доступность любой сущности определяется тем, в каком месте она была декларирована и какие модификаторы доступа, если такие есть, присутствуют в ее декларации (*private*, *protected* или *public*). Правильное использование этих модификаторов имеет большое значение для сокрытия информации.

Главное правило заключается в том, что вы должны сделать каждый класс или член максимально недоступным, насколько

это возможно. Другими словами, вы должны использовать самый низший из возможных уровней доступа, который еще допускает правильное функционирование создаваемой программы.

Для классов и интерфейсов верхнего уровня (не являющихся вложенными) существуют лишь два возможных уровня доступа: доступный только в пределах пакета (*package-private*) и открытый (*public*). Если вы объявляете класс или интерфейс верхнего уровня с модификатором *public*, он будет открытм, в противном случае он будет доступен только в пределах пакета. Если класс или интерфейс верхнего уровня можно сделать доступным только в пакете, он должен стать таковым. При этом класс или интерфейс становится частью реализации этого пакета, а не частью его внешнего 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 его поддерживать во имя сохранения совместимости.

Если класс или интерфейс верхнего уровня, доступный лишь в пределах пакета, используется только в одном классе, вы должны рассмотреть возможность превращения его в закрытый класс (или интерфейс), который будет вложен именно в тот класс, где он используется (статья 22). Тем самым вы еще более уменьшите его доступность. Однако это уже не так важно, как сделать необоснованно открытый класс доступным только в пределах пакета, поскольку класс, доступный лишь в пакете, уже является частью реализации этого пакета, а не его внешнего API.

Для членов класса (полей, методов, вложенных классов и вложенных интерфейсов) существует четыре возможных уровня доступа, которые перечислены здесь в порядке увеличения доступности:

- закрытый (*private*) – данный член доступен лишь в пределах того класса верхнего уровня, где он был объявлен.
- доступный лишь в пределах пакета (*package-private*) – член доступен из любого класса в пределах того пакета, где он был объявлен. Формально этот уровень называется доступом

по умолчанию (*default access*), и именно этот уровень доступа вы получаете, если не было указано модификаторов доступа.

- **зашщищенный (protected)** — член доступен для подклассов того класса, где этот член был объявлен (с небольшими ограничениями [JLS, 6.6.2]), доступ к члену есть из любого класса в пакете, где этот член был объявлен.
- **открытый (public)** — член доступен отовсюду.

После того как для вашего класса был тщательно спроектирован открытый API, 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 всего пакета. Если вы обнаружите, что таких членов слишком много, еще раз проверьте модель вашей системы и попытайтесь найти другой вариант разбиения на классы, при котором они были бы лучше изолированы друг от друга. Как было сказано, и закрытый член, и член, доступный только в пределах пакета, являются частью реализации класса и обычно не оказывают воздействия на его внешний API. Однако, тем не менее, они могут «просочиться» во внешний API, если этот класс реализует интерфейс `Serializable` (статьи 74 и 75).

Если уровень для члена открытого класса меняется с доступного в пакете на защищенный, уровень доступности этого члена резко возрастает. Для этого класса защищенный член является частью внешнего API, а потому ему навсегда должна быть обеспечена поддержка. Более того, наличие защищенного члена в классе, передаваемом за пределы пакета, представляет собой открытую передачу деталей реализации (статья 17). Потребность в использовании защищенных членов должна возникать сравнительно редко.

Существует одно правило, ограничивающее ваши возможности по уменьшению доступности методов. Если какой-либо метод переопределяет метод суперкласса, то методу в подклассе не разрешается иметь более низкий уровень доступа, чем был у метода в суперкласс-

се [JLS, 8.4.8.3]. Это необходимо для того, чтобы гарантировать, что экземпляр подкласса можно будет использовать повсюду, где можно было использовать экземпляр суперкласса. Если вы нарушите это правило, то, когда вы попытаетесь скомпилировать этот подкласс, компилятор будет генерировать сообщение об ошибке. Частный случай этого правила: если класс реализует некий интерфейс, то все методы этого класса, представленные в этом интерфейсе, должны быть объявлены как открытые (*public*). Это объясняется тем, что в интерфейсе все методы неявно подразумеваются открытыми [JLS 9.1.5].

Открытые поля (в отличие от открытых методов) в открытых классах должны появляться редко (если вообще должны появляться). Если поле не имеет модификатора *final* или имеет модификатор и ссылается на изменяемый объект, то, делая его открытым, вы упускаете возможность наложить ограничение на значения, которые могут быть записаны в это поле. Вы также упускаете возможность предпринимать какие-либо действия в ответ на изменение этого поля. Отсюда простой вывод: классы с открытыми изменяемыми полями небезопасны в системе с несколькими потоками (*not thread-safe*). Даже если поле имеет модификатор *final* и не ссылается на изменяемый объект, объявляя его открытым, вы отказываетесь от возможности гибкого перехода на новое представление внутренних данных, в котором это поле будет отсутствовать.

То же самое правило относится и к статическим полям, за одним исключением. С помощью полей *public static final* классы могут выставлять наружу константы, подразумевая, что константы образуют целую часть абстракции, предоставленной классом. Согласно договоренности, названия таких полей состоят из прописных букв, слова в названии разделены символом подчеркивания (статья 56). Крайне важно, чтобы эти поля содержали либо простые значения, либо ссылки на неизменяемые объекты (статья 15). Поле с модификатором *final*, содержащее ссылку на изменяемый объект, обладает всеми недостатками поля без модификатора *final*: хотя саму ссылку

нельзя изменить, объект, на который она указывает, может быть изменен — с нежелательными последствиями.

Заметим, что массив ненулевой длины всегда является изменяемым. Поэтому **практически никогда нельзя декларировать поле массива как** public static final. Если в классе будет такое поле, клиенты получат возможность менять содержимое этого массива. Часто это является причиной появления дыр в системе безопасности.

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

Имейте в виду, что многие среды разработки (IDE) генерируют методы доступа, возвращающие ссылки на закрытые поля массивов, приводят как раз именно к такой ошибке. Есть два способа решения проблемы. Вы можете заменить открытый массив закрытым массивом и **открытым неизменяемым списком**:

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

Другая альтернатива: если на этапе компиляции вам необходима проверка типов и вы готовы пожертвовать производительностью, вы можете заменить открытое поле массива **открытым методом**, который возвращает копию закрытого массива.

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

Подведем итоги. Всегда следует снижать уровень доступа, насколько это возможно. Тщательно разработав наименьший открытый API, вы должны не дать возможность каким-либо случайнym классам, интерфейсам и членам стать частью этого API. За исключением полей типа public static final, других открытых полей в открытом классе быть не должно. Убедитесь в том, что объекты, на которые есть ссылки в полях типа public static final, не являются изменяемыми.

Страница
74**В открытых классах используйте
методы доступа, а не открытые поля**

Иногда вы можете поддаться искушению написать вырожденные классы, служащие только для одной цели — группировать поля экземпляров.

```
// Вырожденные классы, подобные этому, не должны быть открытыми
class Point {
    public double x;
    public double y;
}
```

Поскольку доступ к таким классам осуществляется через поле данных, они лишены преимуществ инкапсуляции (статья 13). Вы не можете поменять структуру такого класса, не изменив его API. Вы не можете использовать каких-либо инвариантов. Вы также не можете предпринять каких-либо дополнительных действий, когда меняется значение поля. Для программистов, строго придерживающихся объектно ориентированного подхода, такой класс заслуживает осуждения и в любом случае его следует заменить классом с закрытыми полями и открытыми методами доступа и, для изменяемых классов, мутаторами (установщиками):

```
// Инкапсуляция данных методами доступа и мутаторами
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 совершенно правы: если класс доступен за пределами пакета, то предусмотрительный программист создает соответствующие методы доступа, оставляя возможность менять внутреннее представление этого класса. Если открытый класс показал клиенту свои поля данных, то всякая возможность менять это представление может быть потеряна, поскольку программный код клиентов открытого класса может оказаться где угодно.

Однако если класс доступен только в пределах пакета или является закрытым вложенным классом, то никакого настоящего ущерба от прямого доступа к его полям с данными не будет, при условии, что эти поля действительно описывают выстраиваемую этим классом абстракцию. По сравнению с методами доступа такой подход создает меньше визуального беспорядка и в декларации 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 этот класс. В том редком случае, 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 более: только внешним классом.

Несколько классов в библиотеках для платформы Java нарушают данный совет не предоставлять непосредственного доступа к полям открытого класса. В глаза бросаются такие примеры, как классы Point и Dimension из пакета java.awt. Вместо того чтобы соответствовать этим классам, их следует рассматривать как предупреждение. В статье 55 рассказывается, как решение раскрыть внутреннее содержание класса Dimension привело к серьезным проблемам с производительностью, которые нельзя было разрешить, не затрагивая клиентов.

Хотя и это и плохая идея, если открытый класс раскрывает поля напрямую, это все же менее опасно, чем если поля являются неизме-

няемыми. Невозможно изменить представление такого класса, не поменяв его API, и вы не можете делать вспомогательных действий во время чтения поля, но вы можете ввести инвариант. В следующем примере класс должен гарантировать, что каждый экземпляр представляет верное время:

```
// Открытый класс с раскрытым неизменяемым полем – спорно.  
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;  
    }  
    ... // Остальное опущено.  
}
```

Подведем итоги. Открытые классы никогда не должны раскрывать неизменяемые поля. Это не так опасно, но тем не менее спорно, чтобы открытые классы раскрывали неизменяемые поля. Иногда требуется классам, открытым в рамках пакета, или закрытым вложенным классам раскрывать поля вне зависимости от их изменяемости.

Статья
75

Предпочитайте постоянство

Неизменяемый класс — это просто такой класс, экземпляры которого нельзя поменять. Вся информация, которая содержится в любом его экземпляре, записывается в момент его создания и остается

неизменной в течение всего времени существования этого объекта. В библиотеках для платформы Java имеется целый ряд неизменяемых классов, в том числе `String`, простые классы-оболочки, `BigInteger` и `BigDecimal`. Для этого есть много веских причин: по сравнению с изменяемыми классами их проще проектировать, разрабатывать и использовать. Они менее подвержены ошибкам и более надежны.

Делая класс неизменяемым, выполняйте следующие пять правил:

- 1. Не создавайте каких-либо методов, которые модифицируют представленный объект.** (Эти методы называются мутаторами — *mutator*.)
- 2. Убедитесь в том, что ни один метод класса не может быть переопределён.** Это предотвратит потерю свойства неизменности данного класса в небрежном или умышленно плохо написанном подклассе. Защита методов от переопределения обычно осуществляется путем объявления класса в качестве окончательного, однако есть и другие способы (см. ниже).
- 3. Сделайте все поля окончательными (`final`).** Это ясно выражит ваши намерения, причем в некоторой степени их будет поддерживать сама система. Это может понадобиться для того, чтобы обеспечить правильное поведение программы в том случае, когда ссылка на вновь созданный экземпляр передается от одного потока в другой без выполнения синхронизации, известная как модель памяти (*memory model*) [JLS, 17.5; Goetz06, 16].
- 4. Сделайте все поля закрытыми (`private`).** Это не позволит клиентам непосредственно менять значение полей. Хотя формально неизменяемые классы и могут иметь открытые поля с модификатором `final`, которые содержат либо значения простого типа, либо ссылки на неизменяемые объекты, делать это не рекомендуется, поскольку они будут препятствовать изменению в последующих версиях внутреннего представления класса (статья 13).

5. Убедитесь в монопольном доступе ко всем изменяемым компонентам. Если в вашем классе есть какие-либо поля, содержащие ссылки на изменяемые объекты, удостоверьтесь в том, что клиенты этого класса не смогут получить ссылок на эти объекты. Никогда не инициализируйте такое поле ссылкой на объект, полученной от клиента, метод доступа не должен возвращать хранящейся в этом поле ссылки на объект. Используя конструкторы, методы доступа к полям и методы `readObject` (статья 76), создавайте резервные копии (*defensive copies*, статья 39).

В примерах из предыдущих статей многие классы были неизменяемыми. Один из таких классов — `PhoneNumber` (статья 9) имеет метод доступа для каждого атрибута, но не имеет соответствующего мутатора. Представим чуть более сложный пример:

```
public final class Complex {  
    private final double re;  
    private final double im;  
    public Complex(double re, double im) {  
        this.re = re;  
        this.im = im;  
    }  
    // Методы доступа без соответствующих мутаторов  
    public double realPart() { return re; }  
    public double imaginaryPart() { return im; }  
    public Complex add(Complex c) {  
        return new Complex(re + c.re, im + c.im);  
    }  
    public Complex subtract(Complex c) {  
        return new Complex(re - c.re, im - c.im);  
    }  
    public Complex multiply(Complex c) {  
        return new Complex(re*c.re - im*c.im,  
                           re*c.im + im*c.re);  
    }  
}
```

```

public Complex divide(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;
    return Double.compare(re, c.re) == 0 &&
        Double.compare(im, c.im) == 0;
}

@Override public int hashCode() {
    int result = 17 + hashDouble(re);
    result = 31 * result + hashDouble(im);
    return result;
}

private static int hashDouble(double val) {
    long lingBits = Double.doubleToLongBits(val);
    return (int) (lingBits ^ (lingBits >>> 32));
}

@Override public String toString() {
    return "(" + re + " + " + im + "i)";
}

```

Данный класс представляет комплексное число (число с действительной и мнимой частями). Помимо обычных 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, генерируют и передают новый экземпляр класса Complex. Такой подход используется для большинства сложных неизменяемых классов. Называется это функциональным подходом (*functional*

approach), поскольку рассматриваемые методы возвращают результат применения некой функции к своему операнду, не изменяя при этом сам операнд. Альтернативой является более распространенный процедурный, или императивный, подход (*procedural or imperative approach*), при котором метод выполняет для своего операнда некую процедуру, которая меняет его состояние.

При первом знакомстве функциональный подход может показаться искусственным, однако он создает условия для неизменяемости объектов, а это имеет множество преимуществ. **Неизменяемые объекты просты.** Неизменяемый объект может находиться только в одном состоянии — в том, с которым он был создан. Если вы удостоверитесь, что каждый конструктор класса устанавливает требуемые инварианты, то это будет гарантией того, что данные инварианты будут оставаться действительными всегда, без каких-либо дополнительных усилий с вашей стороны и со стороны программиста, использующего этот класс. Что же касается изменяемого объекта, то он может иметь относительно сложное пространство состояний. Если в документации не представлено точного описания смены состояний, осуществляемой методами-мутаторами, надежное использование изменяемого класса может оказаться сложной или даже невыполнимой задачей.

Неизменяемые объекты по своей сути безопасны при работе с потоками (*thread-safe*): им не нужна синхронизация. Они не могут быть разрушены только из-за того, что одновременно к ним обращается несколько потоков. Несомненно, это самый простой способ добиться безопасности при работе в потоками. Действительно, ни один поток никогда не сможет обнаружить какого-либо воздействия со стороны другого потока через неизменяемый объект. По этой причине **неизменяемые объекты можно свободно использовать для совместного доступа.** Неизменяемые классы должны использовать это преимущество, заставляя клиентов везде, где это возможно, использовать уже существующие экземпляры. Один из простых приемов, позволяющих достичь этого, — для часто используемых значений создавать константы типа `public static final`. Например, в классе `Complex` можно представить 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. В неизменяемом классе можно предусмотреть статические методы генерации (статья 1), которые кэшируют часто запрашиваемые экземпляры вместо того, чтобы при каждом запросе создавать новые экземпляры, дублирующие уже имеющиеся. Все упаковываемые примитивные классы и классы BigInteger делают это. Использование таких статических методов генерации заставляет клиентов вместе использовать уже имеющиеся экземпляры, вместо того чтобы создавать новые. Это снижает расход памяти и сокращает работу по освобождению памяти. Выбор методов статической генерации вместо открытых конструкторов дает на этапе создания нового класса возможность добавлять кэширование позже, не изменяя при этом клиента.

Благодаря тому, что неизменяемые объекты можно свободно предоставлять для совместного доступа, не требуется создавать для них резервные копии (*defensive copies*, статья 39). В действительности вам вообще не надо делать никаких копий, поскольку они всегда будут идентичны оригиналу. Соответственно, для неизменяемого класса вам не нужно, да и не следует создавать метод `clone` и конструктор копий (*copy constructor*, статья 11). Когда платформа Java только появилась, еще не было четкого понимания этого обстоятельства, и потому класс `String` сейчас имеет конструктор копий, но он редко используется, если используется вообще (статья 5).

Можно совместно использовать не только неизменяемый объект, но и его содержимое. Например, класс BigInteger использует внутреннее представление знак/модуль (*sign/magnitude*). Знак числа представлен полем типа `int`, его модуль — массивом `int`. Метод инвертирования `negate` создает новый экземпляр BigInteger с тем же модулем и с противоположным знаком. При этом нет необходимости копировать массив, поскольку вновь созданный экземпляр BigInteger имеет внутри ссылку на тот же самый массив, что и исходный экземпляр.

Неизменяемые объекты образуют крупные строительные блоки для остальных объектов, как изменяемых, так и неизменяемых. Гораздо легче обеспечивать поддержку инвариантов сложного объекта, если вы знаете, что составляющие его объекты не будут менять его «снизу». Частный случай данного принципа: неизменяемый объект формирует большую схему соответствия между ключами и набором элементов. При этом вас не должно беспокоить то, что значения, однажды записанные в эту схему или набор, вдруг поменяются, и это приведет к разрушению инвариантов схемы или набора.

Единственный настоящий недостаток неизменяемых классов заключается в том, что для каждого уникального значения им нужен отдельный объект. Создание таких объектов может потребовать больших ресурсов, особенно если они имеют значительные размеры. Например, у вас есть объект BigInteger размером в миллион бит и хотите логически дополнить его младший бит:

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

Метод flipBit создает новый экземпляр класса BigInteger длиной также в миллион бит, который отличается от своего оригинала только одним битом. Эта операция требует времени и места, пропорциональных размеру экземпляра BigInteger. Противоположный подход использует java.util.BitSet. Как и BigInteger, BitSet представляет последовательность битов произвольной длины, однако, в отличие от BigInteger, BitSet является изменяемым классом. В классе BitSet предусмотрен метод, позволяющий в экземпляре, содержащем миллионы битов, менять значение отдельного бита в течение фиксированного времени.

Проблема производительности углубляется, когда вы выполняете многошаговую операцию, генерируя на каждом этапе новый объект, а в конце отбрасываете все эти объекты, оставляя только окончательный результат. Справиться с этой проблемой можно двумя способами. Во-первых, можно догадаться, какие многошаговые операции будут требоваться чаще всего, и представить их в качестве элемен-

тарных. Если многошаговая операция реализована как элементарная (*primitive*), неизменяемый класс уже не обязан на каждом шаге создавать отдельный объект. Извнутри неизменяемый класс может быть сколь угодно хитроумным. Например, у класса `BigInteger` есть изменяемый «класс-компаньон», который доступен только в пределах пакета и используется для ускорения многошаговых операций, таких как возведение в степень по модулю. По всем перечисленным выше причинам использовать изменяемый класс-компаньон гораздо сложнее. Однако делать это вам, к счастью, не надо. Разработчики класса `BigInteger` уже выполнили за вас всю тяжелую работу.

Описанный прием будет работать превосходно, если вам удастся точно предсказать, какие именно сложные многошаговые операции с вашим неизменяемым классом будут нужны клиентам. Если сделать это невозможно, самый лучший вариант — создание *открытого* изменяемого класса-компаньона. В библиотеках для платформы Java такой подход в основном демонстрирует класс `String`, для которого изменяемым классом-компаньоном является `StringBuffer`. В силу некоторых причин `BitSet` вряд ли играет роль изменяемого компаньона для `BigInteger`.

Теперь, когда вы знаете, как создавать неизменяемый класс и какие доводы за и против неизменяемости, обсудим несколько альтернативных вариантов. Напомним, что для гарантии неизменяемости класс должен запретить создавать подклассы. Обычно это достигается тем, что класс делается окончательным, но есть и другой, более гибкий вариант добиться этого. Альтернатива, позволяющая сделать неизменяемый класс окончательным, заключается в том, чтобы сделать все его конструкторы закрытыми либо доступными только в пакете и вместо открытых конструкторов использовать открытые *статические методы генерации* (статья 1).

Для пояснения представим, как бы выглядел класс `Complex`, если бы применялся такой подход:

```
// Неизменяемый класс со статическими методами генерации  
// 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);  
    }  
    ... // Остальное не изменилось  
}
```

Хотя этот подход не используется широко, из трех описанных альтернатив он часто оказывается наилучшим. Он самый гибкий, поскольку позволяет использовать несколько классов реализации, доступных в пределах пакета. Для клиентов за пределами пакета этот неизменяемый класс фактически является окончательным, поскольку они не могут расширить класс, взятый из другого пакета, у которого нет ни открытого, ни защищенного конструктора. Помимо того, что такой подход дает возможность гибко использовать несколько классов реализации, он дает возможность повысить производительность класса в последующих версиях путем совершенствования механизма кэширования объектов в статических методах генерации.

Как показано в статье 1, статические методы генерации объектов имеют много других преимуществ по сравнению с конструкторами. Предположим, что вы хотите создать механизм генерации комплексного числа, отталкиваясь от его полярных координат. Использовать здесь конструкторы плохо, поскольку окажется, что собственный конструктор класса Complex будет иметь ту же самую сигнатуру, которую мы только что применяли: Complex(float, float). Со статическими методами генерации все проще: достаточно просто добавить второй статический метод генерации с таким названием, которое четко обозначит его функцию:

```
public static Complex valueOfPolar(double r, double theta) {  
    return new Complex(r * Math.cos(theta)),  
    (r * Math.sin(theta));  
}
```

Когда писались классы BigInteger и BigDecimal, не было окончательного согласия в том, что неизменяемые классы должны быть фактически окончательными. Поэтому любой метод этих классов можно переопределить. К сожалению, исправить что-либо впоследствии уже было нельзя, не потеряв при этом совместимость версий снизу вверх. Поэтому, если вы пишете класс, безопасность которого зависит от неизменяемости аргумента с типом BigInteger или BigDecimal, полученного от ненадежного клиента, вы должны выполнить проверку и убедиться в том, что этот аргумент действительно является «настоящим» классом BigInteger или BigDecimal, а не экземпляром какого-либо ненадежного подкласса. Если имеет место последнее, вам необходимо создать резервную копию этого экземпляра, поскольку придется исходить из того, что он может оказаться изменяемым (статья 39):

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

Список правил для неизменяемых классов, представленный в начале статьи, гласит, что ни один метод не может модифицировать объект и все поля должны быть окончательными. Эти правила несколько строже, чем это необходимо, и их можно ослабить с целью повышения производительности программы. Действительно, ни один метод не может произвести такие изменения состояния объекта, которое можно было бы *увидеть извне*. Вместе с тем многие неизменяемые классы имеют одно или несколько избыточных полей без модификатора final, в которых они сохраняют однажды полученные результаты трудоемких вычислений. Если в дальнейшем потребуется произвести те же самые вычисления, то будет возвращено ранее

сохраненное значение, ненужные вычисления выполняться не будут. Такая уловка работает надежно именно благодаря неизменяемости объекта: неизменность его состояния является гарантией того, что если вычисления выполнять заново, то они приведут опять к тому же результату.

Например, метод `hashCode` из класса `PhoneNumber` (статья 9) вычисляет хэш-код. Получив код в первый раз, метод сохраняет его на тот случай, если хэш-код потребуется вычислять снова. Такая методика, представляющая собой классический пример отложенной инициализации (статья 71), используется также и в классе `String`.

Следует добавить одно предостережение, касающееся сериализуемости объектов. Если вы решили, чтобы ваш неизменяемый класс должен реализовывать интерфейс `Serializable` и при этом у него есть одно или несколько полей, которые ссылаются на изменяемые объекты, то вы обязаны предоставить явный метод `readObject` или `readResolve` или использовать методы `ObjectOutputStream.writeUnshared` и `ObjectInputStream.readUnshared`, даже если для этого класса можно использовать сериализуемую форму, предоставляемую по умолчанию. В противном случае может быть создан изменяемый экземпляр вашего неизменяемого класса. Эта тема детально раскрывается в статье 76.

Подведем итоги. Не стоит для каждого метода `get` писать метод `set`. **Классы должны оставаться неизменяемыми, если нет уж совсем веской причины сделать их изменяемыми.** Неизменяемые классы имеют массу преимуществ, единственный же их недостаток — возможные проблемы с производительностью при определенных условиях. Небольшие объекты значений, такие как `PhoneNumber` или `Complex`, всегда следует делать неизменяемыми. (В библиотеках для платформы Java есть несколько классов, например `java.util.Date` и `java.awt.Point`, которые должны были бы стать неизменяемыми, но таковыми не являются.) Вместе с тем вам следует серьезно подумать, прежде чем делать неизменяемыми более крупные объекты значений, такие как `String` или `BigInteger`. Создавать для вашего неизменяемого класса открытый изменяемый класс-компаньон сле-

дует только тогда, когда вы убедитесь в том, что это необходимо для получения приемлемой производительности (статья 55).

Есть классы, которым неизменяемость не нужна, например классы-процессы Thread и TimerTask. **Если класс невозможно сделать неизменяемым, вы должны ограничить его изменяемость, насколько это возможно.** Чем меньше число состояний, в которых может находиться объект, тем проще рассматривать этот объект, тем меньше вероятность ошибки. По этой причине **конструктор такого класса должен создавать полностью инициализированный объект, у которого все инварианты уже установлены.**

Конструктор не должен передавать другим методам класса объект, сформированный частично. Не создавайте открытый метод инициализации отдельно от конструктора, если только для этого нет чрезвычайно веской причины. Точно так же не следует создавать метод «повторной инициализации», который позволил бы использовать объект повторно, как если бы он был создан с другим исходным состоянием. Метод повторной инициализации обычно дает (если вообще дает) лишь небольшой выигрыш в производительности за счет увеличения сложности приложения.

Перечисленные правила иллюстрирует класс TimerTask. Он является изменяемым, однако пространство его состояний намеренно оставлено небольшим. Вы создаете экземпляр, задаете порядок его выполнения и, возможно, отменяете это решение. Как только задача, контролируемая таймером, была запущена на исполнение или отменена, повторно использовать его вы уже не можете.

Последнее замечание, которое нужно сделать в этой статье, касается класса Complex. Этот пример предназначался лишь для того, чтобы проиллюстрировать свойство неизменяемости. Он не обладает достоинствами промышленной реализации класса комплексных чисел. Для умножения и деления комплексных чисел он использует обычные формулы, для которых нет правильного округления, которые имеют скучную семантику для комплексных значений NaN и бесконечности [Kahan91, Smith62, Thomas94].

Предпочитайте компоновку наследованию

Наследование (*inheritance*) — это мощный способ добиться многократного использования кода, но не всегда лучший инструмент для работы. При неправильном применении наследование приводит к появлению ненадежных программ. Наследование можно безопасно использовать внутри пакета, где 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 (статья 17). Однако наследование обычновенных неабстрактных классов за пределами пакета сопряжено с риском. Напомним, что в этой книге слово «наследование» (*inheritance*) используется для обозначения *наследование реализации* (*implementation inheritance*), когда один класс расширяет другой. Проблемы, обсуждаемые в этой статье, не касаются *наследования интерфейса* (*interface inheritance*), когда класс реализует интерфейс или же один интерфейс расширяет другой.

В отличие от вызова метода наследование нарушает инкапсуляцию [Snyder86]. 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 сломаться, даже если его код и остался в неприкосновенности. Как следствие, подкласс должен развиваться вместе со своим суперклассом, если только авторы суперкласса не спроектировали и не документировали его специально для последующего расширения.

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 его 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. Переопределим оба метода:

```
// Ошибка: неправильное использование наследования!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // Число попыток вставить элемент
    private int addCount = 0;
    public InstrumentedHashSet() {
    }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    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;
    }
}
```

Представленный класс кажется правильным, но не работает. Предположим, что мы создали один экземпляр и с помощью метода addAll поместили в него три элемента:

```
InstrumentedHashSet <String> s = new InstrumentedHashSet();
s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));
```

Мы могли предположить, что после этого метод `getAddCount` должен возвратить число 3, но он возвращает 6. Что же не так? Внутри класса `HashSet` метод `addAll` реализован поверх его метода `add`, хотя в документации эта деталь реализации не отражена, что вполне оправданно. Метод `addAll` в классе `InstrumentedHashSet` добавил к значению поля `addCount` число 3. Затем с помощью `super.addAll` была вызвана реализация `addAll` в классе `HashSet`. В свою очередь это влечет вызов метода `add`, переопределенного в классе `InstrumentedHashSet` — по одному разу для каждого элемента. Каждый из этих трех вызовов добавлял к значению `addCount` еще единицу, так что и итоге общий прирост составляет шесть: добавление каждого элемента с помощью метода `addAll` засчитывалось дважды.

Мы могли бы «исправить» подкласс, отказавшись от переопределения метода `addAll`. Полученный класс и будет работать, правильность его работы зависит от того обстоятельства, что метод `addAll` в классе `HashSet` реализуется поверх метода `add`. Такое «использование самого себя» является деталью реализации, и нет гарантии, что она будет сохранена во всех реализациях платформы Java, не поменяется при переходе от одной версии к другой. Соответственно, полученный класс `InstrumentedHashSet` может быть ненадежен.

Ненамного лучшим решением будет переопределение `addAll` в качестве метода, который в цикле просматривает представленный набор и для каждого элемента один раз вызывает метод `add`. Это может гарантировать правильный результат независимо от того, реализован ли метод `addAll` в классе `HashSet` поверх метода `add`, поскольку реализация `addAll` в классе `HashSet` больше не применяется. Однако и такой прием не решает всех наших проблем. Он подразумевает повторную реализацию методов суперкласса, которые могут проводить, а могут не приводить к использованию классом самого себя. Этот вариант сложен, трудоемок и подвержен ошибкам. К тому же это не всегда возможно, поскольку некоторые методы нельзя реализовать, не имея доступа к закрытым полям, которые недоступны для подкласса.

Еще одна причина ненадежности подклассов связана с тем, что в новых версиях суперкласс может обзавестись новыми методами. Предположим, безопасность программы зависит от того, чтобы все элементы, помещенные в некую коллекцию, соответствовали некоторому утверждению. Выполнение этого условия можно гарантировать, создав для этой коллекции подкласс, переопределив в нем все методы, добавляющие элемент, таким образом, чтобы перед добавлением элемента проверялось его соответствие рассматриваемому утверждению. 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 переопределён в подклассе. Указанная проблема не является чисто теоретической. Когда производился пересмотр классов `Hashtable` и `Vector` для включения в архитектуру *Collections Framework*, пришлось закрывать несколько дыр такой природы, возникших в системе безопасности.

Обе описанные проблемы возникают из-за переопределения методов. Вы можете решить, что расширение класса окажется безопасным, если при добавлении в класс новых методов воздержитесь от переопределения уже имеющихся. Хотя расширение такого рода гораздо безопаснее, оно также не исключает риска. Если в очередной версии суперкласс получит новый метод, но окажется, что вы, к сожалению, уже имеете в подклассе метод с той же сигнатурой, но другим типом возвращаемого значения, то ваш подкласс перестанет компилироваться [JLS, 8.4.8.3]. Если же вы создали в подклассе метод с точно такой же сигнатурой, как и у нового метода в суперклассе, то переопределите последний и опять столкнетесь с обеими описанными выше проблемами. Более того, вряд ли ваш метод будет отвечать требованиям, предъявляемым к новому методу в суперклассе, так как, когда вы писали этот метод в подклассе, они еще не были сформулированы.

К счастью, есть способ устраниć все описанные проблемы. Вместо того чтобы расширять имеющийся класс, создайте в вашем новом классе закрытое поле, которое будет содержать ссылку на экземпляр прежнего класса. Такая схема называется композицией (*composition*), поскольку имеющийся класс становится частью нового класса. Каждый экземпляр метода в новом классе вызывает соответствующий метод содержащегося здесь же экземпляра прежнего класса, а затем возвращает полученный результат. Это называется передачей вызова (*forwarding*), а соответствующие методы нового класса носят название методов переадресации (*forwarding methods*). Полученный класс будет прочен, как скала: он не будет зависеть от деталей реализации прежнего класса. Даже если к имевшемуся прежде классу будут добавлены новые методы, на новый класс это не повлияет. В качестве конкретного примера использования метода компоновки/переадресации представим класс, который заменяет InstrumentedHashSet. Обратите внимание, что реализация разделена на две части: сам класс и многократно используемый класс переадресации, который содержит все методы переадресации и больше ничего:

```
// Класс-оболочка – вместо наследования используется композиция
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;
    }
}
```

```

    }

// Многократно используемый класс переадресации
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(); }
}

```

Создание класса InstrumentedSet стало возможным благодаря наличию интерфейса Set, в котором собраны функциональные возможности класса HashSet. Данная реализация не только устойчива, но и чрезвычайно гибка. Класс InstrumentedSet реализует интерфейс Set и имеет единственный конструктор, аргумент которого также имеет тип Set. В сущности, представленный класс преобразует один интерфейс Set в другой, добавляя возможность выполнять измерения. В отличие от подхода, использующего наследование, который работает только для одного конкретного класса и требует отдель-

ный конструктор для каждого конструктора в суперклассе данный класс-оболочку можно применять для расширения возможностей любой реализации интерфейса Set, он будет работать с любым ранее существовавшим конструктором. Например:

```
Set<Date> s = new InstrumentedSet<Date>(new TreeSet<Date>(cmp));  
Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));
```

Класс InstrumentedSet можно применять даже для временного оснащения экземпляра Set, который до сих пор не пользовался этими функциями:

```
static void walk(Set<Dog> dogs) {  
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<Dog>(dogs);  
    // Within this method use iDogs instead of dogs  
}
```

Класс InstrumentedSet называется классом-оболочкой (*wrapper*), поскольку каждый экземпляр InstrumentedSet является оболочкой для другого экземпляра Set. Он также известен как шаблоном *Decorator* (декоратор) [Gamma95, с. 175], класс InstrumentedSet «украшает» Set, добавляя ему новые функции. Иногда сочетание композиции и переадресации ошибочно называют делегированием (*delegation*). Однако формально назвать это делегированием нельзя, если только объект-оболочка не передает себя «обернутому» объекту [Lieberman86; Gamma95, с. 20].

Недостатков у классов-оболочек немного. Первый связан с тем, что классы-оболочки не приспособлены для использования в схемах с обратным вызовом (*callback framework*), где один объект передает другому объекту ссылку на самого себя для последующего вызова (*callback* — «обратный вызов»). Поскольку «обернутый» объект не знает о своей оболочке, он передает ссылку на самого себя (*this*), и, как следствие, обратные вызовы минуют оболочку. Это называется проблемой самоидентификации (*SELF problem*) [Lieberman86]. Некоторых разработчиков беспокоит влияние методов переадресации на производительность системы, а также влияние объектов-обо-

лочек на расход памяти. На практике же ни один из этих факторов не имеет существенного влияния. Писать методы переадресации несколько утомительно, однако это частично компенсируется тем, что вам нужно создавать лишь один конструктор.

Наследование уместно только в тех случаях, когда подкласс действительно является подтиповом (*subtype*) суперкласса. Иными словами, класс В должен расширять класс А только тогда, когда между двумя этими классами есть отношение типа «является». Если вы хотите сделать класс В расширением класса А, задайте себе вопрос: «Действительно ли каждый В является А?» Если вы не можете с уверенностью ответить на этот вопрос утвердительно, то В не должен расширять А. Если же ответ отрицательный, часто это оказывается, что В должен просто иметь закрытый от всех экземпляр А и представлять при этом меньший по объему и более простой API: А не является необходимой частью В, это просто деталь его реализации.

В библиотеках для платформы Java имеется множество очевидных нарушений этого принципа. Например, ведь стек не является вектором, соответственно класс Stack не должен быть расширением класса Vector. Точно так же список свойств не является хэш-таблицей, а потому класс Property не должен расширять 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. Более серьезно то, что, демонстрируя внутренние элементы класса, вы позволяете клиенту обращаться к ним напрямую. Это, самое меньшее, может привести к запутанной семантике. Например, если р ссылается на экземпляр класса Properties, то r.getProperty(key) может давать совсем другие результаты, чем r.get(key): старый метод учитывает значения по умолчанию, тогда как второй метод, унаследованный от класса Hashtable, этого не делает. И самое серьезное: напрямую 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 обходить это условие. Как только указанный инвариант нарушается, пользоваться другими элементами API для класса `Properties` (методами `load` и `store`) станет невозможно. Когда эта проблема была обнаружена, исправлять что-либо было слишком поздно, поскольку появились клиенты, работа которых зависит от возможности применения ключей и значений, не являющихся строками.

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

Подведем итоги. Наследование является мощным инструментом, но оно же создает и проблемы, поскольку нарушает принцип инкапсуляции. Пользоваться им можно лишь в том случае, когда между суперклассом и подклассом есть реальная связь «тип/подтип». Но даже в этом случае применение наследования может сделать программу ненадежной, особенно если подкласс и суперкласс принадлежат к разным пакетам, а сам суперкласс не предназначен для расширения. Для устранения этой ненадежности вместо наследования используйте композицию и переадресацию, особенно когда для реализации класса-оболочки есть подходящий интерфейс. Классы-оболочки не только надежнее подклассов, но и имеют большую мощность.



Проектируйте и документируйте наследование либо запрещайте его

Статья 16 предупреждает вас об опасностях создания подклассов для «чужого» класса, наследование которого не предполагалось

и не было документировано. Что же означает «класс спроектированный и документированный для наследования»?

Во-первых, требуется четко документировать последствия переопределения каждого метода в этом классе. Иными словами, для класса должно быть документировано, какие из переопределяемых методов он использует сам (*self-use*): для каждого открытого или защищенного метода, каждого конструктора в документации должно быть указано, какие переопределяемые методы он вызывает, в какой последовательности, а также каким образом результаты их вызова влияют на дальнейшую обработку. (Под переопределяемостью (*overridable*) метода здесь мы подразумеваем то, что он является неокончательным, а также что он либо открытый, либо защищенный.) В общем, для класса в документации должны быть отражены все условия, при которых он может вызвать переопределяемый метод. Например, вызов может поступать из фонового потока или от статического метода-инициализатора.

По соглашению, метод, который сам вызывают переопределяемые методы, должен содержать описание этих обращений в конце своего doc-комментария. Такое описание начинается с фразы «*This implementation*». Эту фразу не следует использовать лишь для того, чтобы показать, что поведение метода может меняться от версии к версии. Она подразумевает, что следующее описание будет касаться внутренней работы данного метода. Приведем пример, взятый из спецификации класса `java.util.AbstractCollection`:

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

Удаляет из данной коллекции один экземпляр указанного элемента, если таковой имеется (необязательная операция). Или более формально: удаляет элемент `e`, такой, что `(o == null ? e == null : o.equals(e))`, при условии, что в коллекции содержится один или несколько таких элементов. Возвращает значение `true`, если в коллекции содержался указанный элемент (или, что то же самое, если в результате этого вызова произошло изменение коллекции).

В данной реализации организуется цикл по коллекции с поиском заданного элемента. Если элемент найден, он удаляется из коллекции с помощью метода `remove`, взятого у итератора. Метод `iterator` коллекции возвращает объект итератора. Заметим, что если у итератора не был реализован метод `remove`, то данная реализация инициирует исключительную ситуацию `UnsupportedOperationException`.

Представленное описание не оставляет сомнений в том, что переопределение метода `iterator` повлияет на работу метода `remove`. Более того, в ней точно указано, каким образом работа экземпляра `Iterator`, возвращенного методом `iterator`, будет влиять на работу метода `remove`. Сравните это с ситуацией, рассматриваемой в статье 16, когда программист, создавший подкласс для `HashSet`, просто не мог знать, повлияет ли переопределение метода `add` на работу метода `addAll`.

Но разве это не нарушает авторитетное мнение, что хорошая документация API должна описывать, что делает данный метод, а не то, как он это делает? Конечно, нарушает! Это печальное следствие того обстоятельства, что наследование нарушает принцип инкапсуляции. Для того чтобы в документации к классу показать, что его можно наследовать безопасно, вы должны описать детали реализации, которые в других случаях можно было бы оставить без уточнения.

Проектирование наследования не исчерпывается описанием того, как класс использует сам себя. Для того чтобы программисты могли писать полезные подклассы, не прилагая чрезмерных усилий, от класса, возможно, потребуется **создание механизма для диагностирования своей собственной внутренней деятельности в виде правильно выбранных защищенных методов** или, в редких случаях, защищенных полей. Например, рассмотрим метод `removeRange` из класса `java.util.AbstractList`:

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

Удаляет из представленного списка все элементы, чей индекс попадает в интервал от `fromIndex` (включительно) до `toIndex` (исключая). Все последующие элементы сдвигаются влево (уменьшается их индекс). Данный вызов укорачивает список `ArrayList` на (`toIn-`

`dex - fromIndex)` элементов. (Если `toIndex == fromIndex`, то процедура ни на что не влияет.)

Этот метод используется процедурой `clear` как в самом списке, так и в его подсписках (подмножество из нескольких идущих подряд элементов — `subList.` — Прим. пер.). При переопределении этого метода, дающего доступ к деталям реализации списка, можно значительно повысить производительность операции очистки как для списка, так и его подсписков.

В данной реализации итератор списка ставится перед `fromIndex`, а затем в цикле делается вызов `ListIterator.next`, за которым следует `ListIterator.remove`. И так до тех пор, пока полностью не будет удален указанный диапазон. Примечание: если время выполнения операции `ListIterator.remove` зависит от числа элементов в списке линейным образом, то в данной реализации зависимость является квадратичной.

Параметры:

`fromIndex` индекс первого удаляемого элемента

`toIndex` индекс последнего удаляемого элемента

Описанный метод не представляет интереса для конечных пользователей реализации `List`. Он служит только для того, чтобы облегчить реализацию в подклассе быстрого метода очистки подсписков. Если бы метод `removeRange` отсутствовал, в подклассе пришлось бы довольствоваться квадратичной зависимостью для метода `clear`, вызываемого для подсписка, либо переписывать весь механизм `subList` с самого начала — задача не из легких!

Как же решить, которые из защищенных методов и полей можно раскрывать при построении класса, предназначенного для наследования? К сожалению, чудодейственного рецепта здесь не существует. Лучшее, что можно сделать, — это выбрать самую лучшую гипотезу и проверить ее на практике, написав несколько подклассов. Вы должны предоставить клиентам минимально возможное число защищенных методов и полей, поскольку каждый из них связан с деталями реализации. С другой стороны, их количество не должно быть слиш-

ком малым, поскольку отсутствие защищенного метода может сделать класс практически негодным для наследования.

Единственный способ протестировать класс, созданный для наследования, — это написать подклассы. Если при написании подкласса вы пренебрежете ключевыми защищенными членами, то данное пренебрежение негативно проявит себя. И наоборот — если несколько подклассов написаны без использования защищенных членов, то вам вероятнее всего придется сделать их закрытыми. Один или несколько таких подклассов должны быть написаны кем-либо, кто не является автором написания суперклассов.

Готовя к наследованию класс, который, по-видимому, получит широкое распространение, учтите, что вы *навсегда* задаете схему использования классом самого себя, а также реализацию, неявно представленную этими защищенными методами и полями. Такие обязательства могут усложнять или даже делать невозможным дальнейшее улучшение производительности и функциональных возможностей в будущих версиях класса. Следовательно, **вам обязательно надо протестировать ваш класс путем написания подклассов до того, как вы его выпустите.**

Заметим также, что специальные описания, обязательные для организации наследования, усложняют обычную документацию, которая предназначена для программистов, создающих экземпляры вашего класса и использующих их методы. Что же касается собственно документации, то лишь немногие инструменты и правила комментирования способны отделить документацию обычного API от той информации, которая представляет интерес лишь для тех программистов, которые создают подклассы.

Есть лишь несколько ограничений, которым обязан соответствовать класс, чтобы его наследование стало возможным. **Конструкторы класса не должны вызывать переопределяемые методы**, непосредственно или опосредованно. Нарушение этого правила может привести к аварийному завершению программы. Конструктор суперкласса выполняется прежде конструктора подкласса, а потому переопределяю-

щий метод в подклассе будет вызываться перед запуском конструктора этого подкласса. И если переопределенный метод зависит от инициализации, которую осуществляет конструктор подкласса, то этот метод будет работать совсем не так, как ожидалось. Для ясности приведем пример класса, который нарушает это правило:

```
public class Super {
    // Ошибка: конструктор вызывает переопределяемый метод
    public Super() {
        overrideMe();
    }
    public void OverrideMe() {
    }
}
```

Представим подкласс, в котором переопределяется метод `m`, неправомерно вызываемый единственным конструктором класса `Super`:

```
public final class Sub extends Super {
    private final Date date;
    // Пустое поле final, заполняется конструктором
    Sub() {
        date = new Date();
    }
    // Переопределяет метод, используемый конструктором суперкласса
    @Override public void m() {
        System.out.println(date);
    }
    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

Можно было бы ожидать, что эта программа напечатает текущую дату дважды, однако в первый раз она выводит `null`, поскольку метод `m` вызван конструктором `Super()` прежде, чем конструктор

Sub() получает возможность инициализировать поле даты. Отметим, что данная программа видит поле final в двух разных состояниях. Заметим, что если бы OverrideMe запустил любой метод на date, то запуск вывел бы сообщение NullPointerException в момент, когда конструктор Super запустил бы overrideMe. Единственная причина, почему программа не выводит сообщение об ошибке, как это должно быть, заключается в том, что метод println имеет возможность справиться с нулевым аргументом.

Реализация интерфейсов Cloneable и Serializable при проектировании наследования создает особые трудности. Вообще говоря, реализовать какой-либо из этих интерфейсов в классах, предназначенных для наследования, не очень хорошо уже потому, что они создают большие сложности для программистов, которые этот класс расширяют. Есть, однако, специальные приемы, которые можно использовать с тем, чтобы обеспечить передачу реализаций этих интерфейсов в подкласс, а не заставлять его реализовывать их заново. Эти приемы описаны в статьях 11 и 74.

Если вы решите реализовывать интерфейс Cloneable или Serializable в классе, предназначенном для наследования, то учтите, что, поскольку методы clone и readObject в значительной степени работают как конструкторы, к ним применимо то же самое ограничение: **ни методу clone, ни методу readObject не разрешается вызывать переопределяемый метод, непосредственно или опосредованно.** В случае с методом readObject переопределенный метод будет выполняться перед десериализацией состояния подкласса. Что касается метода clone, то переопределенный метод будет выполняться прежде, чем метод clone в подклассе получит возможность установить состояние клона. В обоих случаях, по-видимому, последует сбой программы. При работе с методом clone такой сбой может нанести ущерб и клонируемому объекту, и клону.

И наконец, если вы решили реализовать интерфейс Serializable в классе, предназначенном для наследования, а у этого класса есть метод readResolve или writeReplace, то вы должны делать эти методы не закрытыми, а защищенными. Если эти методы будут закрыты-

ми, то подклассы будут молча их игнорировать. Это еще один случай, когда для обеспечения наследования детали реализации класса становятся частью его API.

Таким образом, **проектирование класса для наследования накладывает на него существенные ограничения**. Это не то решение, которое должно приниматься с легкостью. В ряде ситуаций это необходимо делать, например, когда речь идет об абстрактных классах, содержащих «скелетную реализацию» интерфейса (статья 18). В других ситуациях этого делать нельзя, например, в случае с неизменяемыми классами (статья 15).

А как же обычные неабстрактные классы? По традиции они не являются окончательными, не предназначаются для порождения подклассов, не имеют соответствующего описания. Однако подобное положение дел опасно. Каждый раз, когда в такой класс вносится изменение, существует вероятность того, что перестанут работать классы клиентов, которые расширяют этот класс. Это не просто теоретическая проблема. Нередко сообщения об ошибках в подклассах возникают после того, как в неокончательном, неабстрактном классе, не предназначавшемся для наследования и не имевшем нужного описания, поменялось содержимое.

Наилучшим решением этой проблемы является запрет на создание подклассов для тех классов, которые не были специально разработаны и не имеют требуемого описания 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 методы генерации. Такая альтернатива, дающая возможность гибко использовать класс внутри подкласса, обсуждалась в статье 15. Приемлем любой из этих подходов.

Возможно, этот совет несколько сомнителен, поскольку так много программистов выросло с привычкой создавать для обычного неабстрактного класса подклассы лишь для того, чтобы добавить

ему новые возможности, например средства контроля, оповещения и синхронизации, либо наоборот, чтобы ограничить его функциональные возможности. 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 подклассов. Шаблон класса-оболочки (*wrapper class*), описанный в статье 16, создает превосходную альтернативу наследованию, используемому всего лишь для изменения функциональности.

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

Вы можете автоматически исключить использование классом собственных переопределяемых методов, оставив прежними его функции. Переместите тело каждого переопределяемого метода в закрытый вспомогательный метод (*helper method*), а затем поместите в каждый переопределяемый метод вызов своего закрытого вспомогательного метода. Наконец, каждый вызов переопределяемого метода в классе замените прямым вызовом закрытого соответствующего вспомогательного метода.

Статья
18

Предпочитайте интерфейсы абстрактным классам

В языке программирования Java предоставлено два механизма для определения типов, которые допускают множественность

реализаций: интерфейсы и абстрактные классы. Самое очевидное различие между этими двумя механизмами заключается в том, что в абстрактные классы можно включать реализацию некоторых методов, для интерфейсов это запрещено. Более важное отличие связано с тем, что для реализации типа, определенного неким абстрактным классом, класс должен стать подклассом этого абстрактного класса. С другой стороны, реализовать интерфейс может любой класс, независимо от его места в иерархии классов, если только он отвечает общепринятым соглашениям и в нем есть все необходимые для этого методы. Поскольку в языке Java не допускается множественное наследование, указанное требование для абстрактных классов серьезно ограничивает их использование для определения типов.

Имеющийся класс можно легко подогнать под реализацию нового интерфейса. Все, что для этого нужно, — добавить в класс необходимые методы, если их еще нет, и внести в декларацию класса пункт о реализации. Например, когда платформа Java была дополнена интерфейсом Comparable, многие существовавшие классы были перестроены под его реализацию. С другой стороны, уже существующие классы, вообще говоря, нельзя перестраивать 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 прародитель обоих этих классов стал его подклассом. К сожалению, это вызывает значительное нарушение иерархии типов, заставляя всех потомков этого общего предка расширять новый абстрактный класс независимо от того, целесообразно это или нет.

Интерфейсы идеально подходят для создания дополнений (mixin). Помимо своего «первоначального типа» класс может реализовать некий дополнительный тип (*mixin*), объявив о том, что в этом классе реализован дополнительный функционал. Например, Comparable является дополнительным интерфейсом, который дает классу возможность декларировать, что его экземпляры упорядочены по отношению к другим, сравнимым с ними объектам. Такой интерфейс

называется `Mixin`, поскольку позволяет к первоначальным функциям некоего типа примешивать (*mixed in*) дополнительные функциональные возможности. Абстрактные классы нельзя использовать для создания дополнений по той же причине, по которой их невозможно встроить в уже имеющиеся классы: класс не может иметь более одного родителя, и в иерархии классов нет подходящего места, куда можно поместить `Mixin`.

Интерфейсы позволяют создавать структуры типов без иерархии. Иерархии типов прекрасно подходят для организации одних сущностей, но зато другие сущности аккуратно уложить в строгую иерархию типов невозможно. Например, предположим, у нас один интерфейс представляет певца, а другой — автора песен:

```
public interface Singer {  
  
    AudioClip Sing(Song s);  
}  
  
public interface Songwriter {  
    Song compose(boolean hit);  
}
```

В жизни некоторые певцы тоже являются авторами песни. Поскольку для определения этих типов мы использовали не абстрактные классы, а интерфейсы, то одному классу никак не запрещается реализовывать оба интерфейса `Singer` и `Songwriter`. В действительности мы можем определить третий интерфейс, который расширяет оба интерфейса `Singer` и `Songwriter` и добавляет новые методы, соответствующие сочетанию:

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

Такой уровень гибкости нужен не всегда. Если же он необходим, интерфейсы становятся спасительным средством. Альтернативой им является раздутая иерархия классов, которая содержит отдель-

ный класс для каждой поддерживаемой ею комбинации атрибутов. Если в системе имеется n атрибутов, то существует 2^n возможных сочетаний, которые, возможно, придется поддерживать. Это называется комбинаторным взрывом (*combinatorial explosion*). Раздутые иерархии классов может привести к созданию раздутых классов, содержащих массу методов, отличающихся друг от друга лишь типом аргументов, поскольку в такой иерархии классов не будет типов, отражающих общий функционал.

Интерфейсы позволяют безопасно и мощно наращивать функциональность, используя идиому класса-оболочки, описанной в статье 16. Если же для определения типов вы применяете абстрактный класс, то не оставляете программисту, желающему добавить новые функциональные возможности, иного выбора, кроме как использовать наследование. Получаемые в результате классы будут не такими мощными и не такими надежными, как классы-оболочки.

Хотя в интерфейсе нельзя хранить реализацию методов, определение типов с помощью интерфейсов не мешает оказывать программистам помощь в реализации класса. Вы можете объединить преимущества интерфейсов и абстрактных классов, сопроводив каждый предоставляемый вами нетривиальный интерфейс абстрактным классом с наброском (скелетом) реализации (*skeletal implementation class*). Интерфейс по-прежнему будет определять тип, а вся работа по его воплощению ляжет на скелетную реализацию.

По соглашению скелетные реализации носят названия вида *AbstractInterface*, где *Interface* — это имя реализуемого ими интерфейса. Например, в архитектуре *Collections Framework* представлены скелетные реализации для всех основных интерфейсов коллекций: *AbstractCollection*, *AbstractSet*, *AbstractList* и *AbstractMap*.

При правильном проектировании скелетная реализация позволяет программистам без труда создавать свои собственные реализации ваших интерфейсов. В качестве примера приведем статический метод генерации, содержащий завершенную, полнофункциональную реализацию интерфейса *List*:

```
// Конкретная реализация built поверх скелетной реализации
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null)
        throw new NullPointerException();
    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i]; // Autoboxing (Item 5)
        }
        @Override public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val; // Auto-unboxing
            return oldVal; // Autoboxing
        }
        public int size() {
            return a.length;
        }
    };
}
```

Если принять во внимание все, что делает реализация интерфейса `List`, то этот пример демонстрирует всю мощь скелетных реализаций. Кстати, пример является адаптером (*Adapter*) [Gamma95, с. 139], который позволяет представить массив `int` в виде списка экземпляров `Integer`. Из-за всех этих преобразований из значений `int` в экземпляры `Integer` и обратно производительность метода не очень высока. Отметим, что здесь приведен лишь статический метод генерации, сам же класс является недоступным *анонимным классом* (статья 22), спрятанным внутри статического метода генерации.

Достоинство скелетных реализаций заключается в том, что они оказывают помощь в реализации абстрактного класса, не налагая при этом строгих ограничений, как это имело бы место, если бы для определения типов использовались абстрактные классы. Для большинства программистов, реализующих интерфейс, расширение скелетной реализации — это очевидный, хотя и необязательный выбор. Если уже имеющийся класс нельзя заставить расширять скелетную реализацию, он всегда может реализовывать представленный интерфейс сам. Более того, скелетная реализация помогает в решении стоящей

перед разработчиком задачи. Класс, который реализует данный интерфейс, может переадресовывать вызов метода, указанного в интерфейсе, содержащемуся внутри его экземпляру закрытого класса, расширяющего скелетную реализацию. Такой прием, известный как искусственное множественное наследование (*simulated multiple inheritance*), тесно связан с идиомой класса-оболочки (статья 16). Он обладает большинством преимуществ множественного наследования и при этом избегает его подводных камней.

Написание скелетной реализации — занятие относительно простое, хотя иногда и скучное. Во-первых, вы должны изучить интерфейс и решить, какие из методов являются примитивами (*primitive*) в терминах, в которых можно было бы реализовать остальные методы интерфейса. Эти примитивы и будут абстрактными методами в вашей скелетной реализации. После этого вы должны предоставить конкретную реализацию всех остальных методов данного интерфейса. В качестве примера приведем скелетную реализацию интерфейса Map.Entry.

```
// Скелетная реализация
public abstract class AbstractMapEntry<K, V>
    implements Map.Entry<K, V> {
    // Примитивы
    public abstract K getKey();
    public abstract V getValue();
    // Элементы в изменяемых схемах должны переопределять этот метод
    public V setValue(V value) {
        throw new UnsupportedOperationException();
    }
    // Реализует основные соглашения для метода Map.Entry.equals
    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<?, ?> arg = (Map.Entry) o;
        return getKey().equals(arg.getKey()) &&
               getValue().equals(arg.getValue());
    }
}
```

```
        return equals(getKey(), arg.getKey()) &&
               equals(getValue(), arg.getValue());
    }

private static boolean equals(Object o1, Object o2) {
    return o1 == null ? o2 == null : o1.equals(o2);
}

// Реализует основные соглашения для метода Map Map.Entry.hashCode
@Override public int hashCode() {
    return hashCode(getKey()) ^ hashCode(getValue());
}

private static int hashCode(Object obj) {
    return obj == null ? 0 : obj.hashCode();
}

}
```

Поскольку скелетная реализация предназначена для наследования, вы должны выполнять все указания по разработке и документированию, представленные в статье 17. Для краткости в предыдущем примере опущены комментарии к документации, однако качественное документирование для скелетных реализаций абсолютно необходимо.

Уменьшенным вариантом скелетной реализации является простая реализация, показанная в `AbstractMap.SimpleEntry`. Простая реализация похожа на скелетную реализацию тем, что 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.

При определении типов, допускающих множественность реализаций, абстрактный класс имеет одно огромное преимущество перед интерфейсом: **абстрактный класс совершенствуется гораздо легче, чем интерфейс**. Если в очередной версии вы захотите добавить в абстрактный класс новый метод, вы всегда сможете представить законченный метод с правильной реализацией, предлагаемой по умолчанию. После этого новый метод появится у всех имеющихся реа-

лизаций данного абстрактного класса. Для интерфейсов этот прием не работает.

Вообще говоря, в открытый интерфейс невозможно добавить какой-либо метод, не разрушив все имеющиеся программы, которые используют этот интерфейс. В классе, ранее реализовавшем этот интерфейс, этот новый метод не будет представлен, и, как следствие, класс компилироваться не будет. Ущерб можно несколько уменьшить, если новый метод одновременно добавить и в скелетную реализацию, и в интерфейс, однако по-настоящему это не решит проблему. Любая реализация интерфейса, не наследующая скелетную реализацию, все равно работать не будет.

Следовательно, открытые интерфейсы необходимо проектировать аккуратно. **Как только интерфейс создан и повсюду реализован, поменять его почти невозможно.** В действительности его нужно правильно строить с первого же раза. Если в интерфейсе есть незначительный изъян, он уже всегда будет раздражать и вас, и пользователей. Если же интерфейс имеет серьезные дефекты, он способен погубить API. Самое лучшее, что можно предпринять при создании нового интерфейса, — заставить как можно больше программистов реализовать этот интерфейс самыми разнообразными способами, прежде чем он будет «заморожен». Это позволит вам найти все ошибки, пока у вас еще есть возможность их исправить.

Подведем итоги. Интерфейс обычно наилучший способ определения типа, который допускает несколько реализаций. Исключением из этого правила является случай, когда легкости совершенствования придается большее значение, чем гибкости и эффективности. В этом для определения типа вы должны использовать абстрактный класс, но только если вы осознаете и готовы принять все связанные с этим ограничения. Если вы предоставляете сложный интерфейс, вам следует хорошо подумать над созданием скелетной реализации, которая будет сопровождать его. Наконец, вы должны проектировать открытые интерфейсы с величайшей тщательностью, всесторонне проверяя их путем написания многочисленных реализаций.

Статья
79

Используйте интерфейсы только для определения типов

Если класс реализует интерфейс, то этот интерфейс может служить как некий *тип*, который можно использовать для ссылки на экземпляры этого класса. То, что класс реализует некий интерфейс, должно говорить нечто о том, что именно клиент может делать с экземплярами этого класса. Создавать интерфейс для каких-либо иных целей неправомерно.

Среди интерфейсов, которые не отвечают этому критерию, числится так называемый интерфейс констант (*constant interface*). Он не имеет методов и содержит исключительно поля static final, передающие константы. Классы, в которых эти константы используются, реализуют данный интерфейс для того, чтобы исключить необходимость в добавлении к названию констант название класса. Приведем пример:

```
// Шаблон интерфейса констант - не использовать!
public interface PhysicalConstants {
    // Число Авогадро (1/моль)
    static final double AVOGADROS_NUMBER = 6.02214199e23;
    // Постоянная Больцмана (Дж/К)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    // Масса электрона (кг)
    static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Шаблон интерфейса констант представляет собой неудачный вариант использования интерфейсов. Появление внутри класса каких-либо констант является деталью реализации. Реализация интерфейса констант приводит к утечке таких деталей во внешний API данного класса. То, что класс реализует интерфейс констант, для пользователей этого класса не представляет никакого интереса. На практике это может даже сбить их с толку. Хуже того, это является неким обязательством: если в будущих версиях класс поменяется так, что

ему уже не будет нужды использовать данные константы, он все равно должен будет реализовывать этот интерфейс для обеспечения совместимости на уровне двоичных кодов (*binary compatibility*). Если же интерфейс констант реализует неокончательный класс, константами из этого интерфейса будет засорено пространство имен всех его подклассов.

В библиотеках для платформы Java есть несколько интерфейсов с константами, например `java.io.ObjectStreamConstants`. Подобные интерфейсы следует воспринимать как отклонение от нормы, и подражать им не следует.

Для передачи констант существует несколько разумных способов. Если константы сильно связаны с имеющимся классом или интерфейсом, вы должны добавить их непосредственно в этот класс или интерфейс. Например, все классы-оболочки в библиотеках платформы Java, связанные с числами, такие как `Integer` или `Float`, предоставляют константы `MIN_VALUE` и `MAX_VALUE`. Если же константы лучше рассматривать как члены перечисления, то передавать их нужно с помощью класса *перечисления* (статья 30). В остальных случаях вы должны передавать константы с помощью вспомогательного класса (*utility class*), не имеющего экземпляров (статья 4). Представим вариант вспомогательного класса для предыдущего примера `PhysicalConstants`:

```
// Вспомогательный класс для констант
Package com.effectivejava.science;
public class PhysicalConstants {
    private PhysicalConstants() { } // Предотвращает появление
                                    // экземпляра
    public static final double AVOGADROS_NUMBER = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Обычно вспомогательному классу требуется, чтобы клиенты связывали названия констант с именем класса, например

PhysicalConstants.AVOGADROS_NUMBER. Если будет трудно использовать константы, передаваемые вспомогательным классом, то вы можете отказаться от необходимости связывания констант с названием класса, используя возможности *статического импорта*, представленные в версии 1.5.

```
// Используем статический импорт для избежания необходимости
// связывания констант
import static com.effectivejava.science.PhysicalConstants.*;
public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Другие варианты использования uses PhysicalConstants
    // для статического импорта.
}
```

Таким образом, интерфейсы нужно использовать только для определения типов. Их не надо использовать для передачи констант.

Статья
20

Объединение заменяйте иерархией классов

Иногда вам может встретиться класс, экземпляры которого бывают двух и более видов и могут содержать поле с признаком, который определяет вид экземпляра. Например, рассмотрим класс, который может представлять круг или прямоугольник:

```
// Tagged class - vastly inferior to a class hierarchy!
class Figure {
    enum Shape { RECTANGLE, CIRCLE };
    // Tag field - the shape of this figure
    final Shape shape;
    // These fields are used only if shape is RECTANGLE
```

```
double length;
double width;
// This field is used only if shape is CIRCLE
double radius;
// Constructor for circle
Figure(double radius) {
    shape = Shape.CIRCLE;
    this.radius = radius;
}
// Constructor for rectangle
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();
    }
}
```

У таких объединенных классов множество недостатков. Они загромождены шаблонами, включая в себя объявление перечислимых типов, меченых полей и выражений переключения. Читаемость кода еще больше страдает, потому что большое количество реализаций смещивается в один класс.

Растет расход памяти, потому что экземпляры нагружены ненужными полями другого вида. Поля не могут быть завершенными, если конструкторы не инициализируют ненужные поля, что приводит к созданию большего объема кода. Конструкторы должны за-

дать поля с отметками и инициализировать корректные поля данных без помощи компилятора: если будут инициализированы не те поля, то программа даст сбой при выполнении. Вы не можете добавить признак к объединенному классу, если вы не измените его исходный файл. Если вы действительно добавите признак, то должны не забыть добавить регистр к каждому выражению при переключении, в противном случае класс будет выполняться с ошибкой. Наконец, тип данных в экземпляре не дает понятия об имеющемся признаком. Короче говоря, **связанные классы слишком многословны, могут содержать ошибки и неэффективны.**

К счастью, у объектно ориентированных языков программирования есть намного лучший механизм определения типа данных, который можно использовать для представления объектов разных типов: создание подтипов. **Явное объединение в действительности является лишь бледным подобием иерархии классов.**

Чтобы преобразовать объединение в иерархию классов, определите абстрактный класс, содержащий метод для каждой операции, работа которой зависит от значения тега. В предыдущем примере единственной такой операцией является area. Полученный абстрактный класс будет корнем иерархии классов. Если есть какая-либо операция, функционирование которой не зависит от значения тега, представьте ее как неабстрактный метод корневого класса. Точно так же, если в явном объединении помимо tag и union есть какие-либо поля данных, то эти поля представляют данные единые для всех типов, а потому их нужно перенести в корневой класс. В приведенном примере подобных операций и полей данных, не зависящих от типа, нет.

Далее, для каждого типа, который может быть представлен объединением, определите неабстрактный подкласс корневого класса. В предыдущем примере такими типами являются круг и прямоугольник. В каждый подкласс поместите те поля данных, которые характерны для соответствующего типа. В нашем примере радиус является характеристикой круга, а длина и ширина характеризуют прямоугольник. Кроме того, в каждый подкласс поместите соответствующую реализацию для всех абстрактных методов в корневом классе.

Представим иерархию классов, которая соответствует нашему примеру явного объединения:

```
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}
class Circle extends Figure {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    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;
    }
    double area() { return length * width; }
}
```

Иерархия классов исправляет все недостатки связанных классов, отмеченных ранее. Код прост и ясен, не содержит шаблонов. Реализация каждого типа выделена в свой собственный класс, и ни один из этих классов не нагружен ненужными полями данных. Все поля окончательны. И компилятор проверяет, что каждый конструктор классов инициализирует свои поля данных и что у каждого класса есть реализация для каждого абстрактного метода, продекларированного в корневом классе. Это помогает избежать возможности ошибки при выполнении, ввиду отсутствия выражений переключения. Различные программисты могут расширить иерархию независимо друг от друга без необходимости доступа к исходному коду корневого класса. Для каждого типа имеется отдельный тип данных, позволяющий программистам определять тип переменных и входные параметры для конкретного типа.

Еще одно преимущество иерархии классов связано с ее способностью отражать естественные иерархические отношения между типами, что обеспечивает повышенную гибкость и улучшает проверку типов на этапе компиляции. Допустим, что явное объединение в исходном примере допускает также построение квадратов. В иерархии классов можно показать, что квадрат — это частный случай прямоугольника (при условии, что оба они неизменны):

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

Обратите внимание, что для классов в этой иерархии, за исключением класса `Square`, доступ предоставляется непосредственно к полям, а не через методы доступа. Делается это для краткости, и это было бы ошибкой, если бы классы были открытыми (статья 14).

Подводя итоги, можно сказать, что связанные классы редко приемлемы. Если у вас будет искушение написать такой класс, подумайте, можно ли его заменить иерархией. Если вам попадется уже существующий класс, подумайте над возможностью преобразования его в иерархию.

Статья
21

Используйте объект функции для выполнения сравнения

Некоторые языки поддерживают указатели на функции (*function pointer*), делегаты (*delegates*), выражения лямбда (*lambda expressions*) и другие возможности, что дает программе возможность хранить и передавать возможность вызова конкретной функции. Такие возможности обычно используются для того, чтобы позволить клиенту, вызвавшему функцию, уточнять схему ее работы, для этого он передает ей указатель на вторую функцию. Например, функция `qsort` из стандартной библиотеки Си получает указатель на функцию-компаратор (*comparator*), которую затем использует для сравнения

элементов, подлежащих сортировке. Функция-компаратор принимает два параметра, каждый из которых является указателем на некий элемент. Она возвращает отрицательное целое число, если элемент, на который указывает первый параметр, оказался меньше элемента, на который указывает второй параметр, нуль, если элементы равны между собой, и положительное целое число, если первый элемент больше второго. Передавая указатель на различные функции-компараторы, клиент может получать различный порядок сортировки. Как демонстрирует шаблон *Strategy* из [Gamma95, с. 315], функция-компаратор представляет алгоритм сортировки элементов.

В языке Java указатели отсутствуют, поскольку те же самые возможности можно получить с помощью ссылок на объекты. Вызывая в объекте некий метод, действие обычно производят над самим *этим объектом*. Между тем можно построить объект, чьи методы выполняют действия над *другими объектами*, непосредственно предоставленным этим методами. Экземпляр класса, который предоставляет клиенту ровно один такой метод, фактически является указателем на этот метод. Подобные экземпляры называются *объектами-функциями*. Например, рассмотрим следующий класс:

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
```

Этот класс передает единственный метод, который получает две строки и возвращает отрицательное число, если первая строка короче второй, нуль, если эти две строки имеют одинаковую длину, и положительное число, если первая строка длиннее второй. Данный метод — не что иное, как компаратор, который вместо более привычного лексикографического упорядочивания задает упорядочение строк по длине. Ссылка на объект *StringLengthComparator* служит для этого компаратора в качестве «указателя на функцию», что позволяет его использовать для любой пары строк. Иными словами, экземпляр класса *StringLengthComparator* — это *определенная методика (concrete strategy)* сравнения строк.

Как часто бывает с классами конкретных методик сравнения, класс `StringLengthComparator` не имеет состояния: у него нет полей, а потому все экземпляры этого класса функционально эквивалентны друг другу. Таким образом, чтобы избежать расходов на создание ненужных объектов, этот класс можно сделать синглтоном (статьи 3 и 5):

```
class StringLengthComparator {  
    private StringLengthComparator() {}  
    public static final StringLengthComparator  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

Чтобы передать методу экземпляр класса `StringLengthComparator`, нам необходим соответствующий тип параметра. Использовать непосредственно тип `StringLengthComparator` было бы нехорошо, поскольку это лишило бы клиентов возможности выбирать какие-либо другие алгоритмы сравнения. Вместо этого нам следует определить интерфейс `Comparator` и переделать класс `StringLengthComparator` таким образом, чтобы он реализовывал этот интерфейс. Иначе говоря, нам необходимо определить интерфейс методики сравнения (*strategy interface*), который должен соответствовать классу конкретной стратегии. Представим этот интерфейс:

```
// Интерфейс методики сравнения  
public interface Comparator<T> {  
    public int compare(T t1, T t2);  
}
```

Оказывается, что представленное определение интерфейса `Comparator` есть в пакете `java.util`, хотя никакого волшебства в этом нет и вы могли точно так же определить его сами. Интерфейс `Comparator` является одним из средств обобщенного программирования (статья 26), что применимо для компараторов объектов, не являю-

щихся строковыми. Его метод compare предполагает брать два параметра типа T (*его формальные параметры типа*) вместо String. Класс StringLengthComparator, показанный выше, можно заставить реализовывать Comparator<String> просто при его декларировании:

```
class StringLengthComparator implements Comparator<String> {  
    ... // class body is identical to the one shown above  
}
```

Классы конкретных методик сравнения часто создаются с помощью анонимных классов (статья 22).

```
Arrays.sort(stringArray, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

Обратите внимание на то, что использование анонимного класса приведет к созданию нового экземпляра каждый раз при выполнении вызова. Если выполнение должно происходить постоянно, рассмотрите возможность хранения объекта функции в закрытом статическом завершенном поле и повторного его использования. Другое преимущество этого заключается в том, что вы можете дать полю дескриптивное имя для объекта функции.

Поскольку интерфейс методики сравнения используется как тип для всех экземпляров конкретных методик сравнения, то для того, чтобы предоставить конкретную методику сравнения, нет необходимости делать соответствующий класс открытым. Вместо этого класс-хозяин (*host*) может передать открытое статическое поле (или статический метод генерации), тип которого соответствует интерфейсу методики сравнения, сам же класс методики сравнения может оставаться закрытым классом, вложенным в класс-хозяин. В следующем примере вместо анонимного класса используется статический класс-член, что позволяет реализовать в классе методики сравнения второй интерфейс — Serializable:

```
// Предоставление конкретной методики сравнения
class Host {
    private static class StrLenCmp
        implements Comparator<String>, Serializable {
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
    }
    // Возвращаемый компаратор является сериализуемым
    public static final Comparator<String>
    STRING_LENGTH_COMPARATOR = new StrLenCmp();
    ... // Основная часть класса опущена
}
```

Представленный шаблон используется в классе `String` для того, чтобы через его поле `CASE_INSENSITIVE_ORDER` передавать компаратор строк, не зависящий от регистра.

Подведем итоги. Первоначально указатели использовались для реализации шаблона `Strategy`. Для того чтобы реализовать этот шаблон в языке программирования Java, необходимо создать интерфейс, представляющий стратегии, а затем для каждой конкретной стратегии нужно построить класс, который этот интерфейс реализует. Если конкретная стратегия используется только один раз, ее класс обычно декларируется и реализуется с помощью анонимного класса. Если же конкретная стратегия передается для многократного использования, ее класс обычно становится закрытым статическим классом-членом и передается через поле `public static final`, чей тип соответствует интерфейсу стратегии.

Статья
22

Предпочитайте статические классы-члены нестатическим

Класс называется вложенным (*nested*), если он определен внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать окружающий его класс. Если вложенный

класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня. Существует четыре категории 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 категории классов называются внутренними (*inner class*). В этой статье рассказывается о том, когда и какую категорию вложенного класса нужно использовать и почему.

Статический класс-член — это простейшая категория вложенного класса. Лучше всего рассматривать его как обычный класс, который декларирован внутри другого класса и имеет доступ ко всем членам окружающего его класса, даже к закрытым. Статический класс-член является статическим членом своего внешнего класса и подчиняется тем же правилам доступа, что и остальные статические члены. Если он декларирован как закрытый, доступ к нему имеет лишь окружающий его класс, и т.д.

В одном из распространенных вариантов статический класс-член используется как открытый вспомогательный класс, который пригоден для применения, только когда есть внешний класс. Например, рассмотрим перечисление, описывающее операции, которые может выполнять калькулятор (статья 30). Класс `Operation` должен быть открытым статическим классом-членом класса `Calculator`. Клиенты класса `Calculator` могут ссылаться на эти операции, выполняемые калькулятором, используя такие имена, как `Calculator.Operation.PLUS` или `Calculator.Operation_MINUS`. Этот вариант приводится ниже. С точки зрения синтаксиса единственное различие между статическими и нестатическими классами-членами заключается в том, что в декларации статических классов-членов присутствует модификатор `static`. Несмотря на свою синтаксическую похожесть, эти две категории вложенных классов совершенно разные. Каждый экземпляр нестатического члена-класса неявным образом связан с содержащим его экземпляром класса-контейнера (*enclosing instance*). Из метода в экземпляре нестатического класса-члена можно вызывать методы содержащего его экземпляра, либо, используя специальную кон-

структурой `this` [JLS 15.8.4], можно получить ссылку на включающий экземпляр. Если экземпляр вложенного класса может существовать в отрыве от экземпляра внешнего класса, то вложенный класс не может быть нестатическим членом-классом: нельзя создать экземпляр нестатического класса-члена, не создав включающего его экземпляра.

Связь между экземпляром нестатического класса-члена и включающим его экземпляром устанавливается при создании первого, и после этого поменять ее нельзя. Обычно эта связь задается автоматически путем вызова конструктора нестатического класса-члена из экземпляра метода во внешнем классе. Иногда можно установить связь вручную, используя выражение `enclosingInstance.newMemberClass(args)`. Как можно предположить, эта связь занимает место в экземпляре нестатического класса-члена и увеличивает время его создания.

Нестатические классы-члены часто используются для определения адаптера (*Adapter*) [Gamma95, с. 139], при содействии которого экземпляр внешнего класса воспринимается своим внутренним классом как экземпляр некоторого класса, не имеющего к нему отношения. Например, в реализациях интерфейса `Map` нестатические классы-члены обычно применяются для создания представлений коллекций (*collection view*), возвращаемых методами `keySet`, `entrySet` и `values` интерфейса `Map`. Аналогично, в реализациях интерфейсов коллекций, таких как `Set` или `List`, нестатические классы-члены обычно используются для создания итераторов:

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

Если вы объявили класс-член, которому не нужен доступ к экземпляру содержащего его класса, не забудьте поместить в соответствующую декларацию модификатор static с тем, чтобы сделать этот класс-член статическим. Если вы не установите модификатор static, каждый экземпляр класса будет содержать ненужную ссылку на внешний объект. Поддержание этой связи требует и времени, и места, но не приносит никакой пользы. Если же вам когда-нибудь потребуется разместить в памяти экземпляр этого класса без окружающего его экземпляра, вы не можете это делать, так как нестатические классы-члены обязаны иметь окружающий их экземпляр.

Закрытые статические классы-члены обычно должны представлять составные части объекта, доступ к которым осуществляется через внешний класс. Например, рассмотрим экземпляр класса Map, который сопоставляет ключи и значения. Внутри экземпляра Map для каждой пары ключ/значение обычно создается объект Entry. Хотя каждая такая запись ассоциируется со схемой, клиенту не надо обращаться к собственным методам этой записи (getKey, getValue и setValue). Соответственно, использовать нестатические классы-члены для представления отдельных записей в схеме Map было бы расточительностью, самое лучшее решение — закрытый статический класс-член. Если в декларации этой записи вы случайно пропустите модификатор static, схема станет работать, но каждая запись будет содержать ненужную ссылку на общую схему, напрасно тратя время и место в памяти.

Вдвойне важно правильно сделать выбор между статическим и нестатическим классом-членом, когда этот класс является открытым или защищенным членом класса, передаваемого клиентам. В этом случае класс-член является частью внешнего API, и в последующих версиях уже нельзя будет сделать нестатический класс-член статическим, не потеряв совместимости на уровне двоичных кодов.

Анонимные классы в языке программирования Java не похожи ни на какие другие. Анонимный класс не имеет имени. Он не является членом содержащего его класса. Вместо того чтобы быть деклари-

рованным с остальными членами класса, он одновременно декларируется и порождает экземпляр в момент использования. Анонимный класс можно поместить в любом месте программы, где разрешается применять выражения. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический класс-член: в нестатическом контексте появляется окружающий его экземпляр.

Применение анонимных классов имеет несколько ограничений. Поскольку анонимный класс одновременно декларируется и порождает экземпляр, его можно использовать, только когда его экземпляр должен порождаться лишь в одном месте программы. Анонимный класс не имеет имени, поэтому может применяться только в том случае, если после порождения экземпляра не нужно на него ссылаться. Анонимный класс обычно реализует лишь методы своего интерфейса или суперкласса. Он не объявляет каких-либо новых методов, так как для доступа к ним нет поименованного типа. Поскольку анонимные классы стоят среди выражений, они должны быть очень короткими, возможно, строк двадцать или меньше. Использование более длинных анонимных классов может усложнить программу с точки зрения ее чтения.

Анонимный класс обычно используется для создания объекта функции (*function object*) (статья 21), такого как экземпляр класса Comparator. Например, при вызове метода sort отсортирует строки массива по их длине. Другой распространенный случай использования анонимного класса — создание объекта процесса (*process object*), такого как экземпляры классов Thread, Runnable или TimerTask. Третий вариант: в статическом методе генерации (см. метод intArrayAsList в статье 18).

Локальные классы, вероятно, относятся к наиболее редко используемой из этих четырех категорий вложенных классов. Локальный класс можно декларировать везде, где разрешается декларировать локальную переменную, и он подчиняется тем же самым правилам видимости. Локальный класс имеет несколько признаков, объединя-

ющих его с каждой из трех других категорий вложенных классов. Как и классы-члены, локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр тогда и только тогда, когда применяются в нестатическом контексте. Как и анонимные классы, они должны быть достаточно короткими, чтобы не мешать удобству чтения метода или инициализатора, в котором они содержатся.

Подведем итоги. Существует четыре категории вложенных классов, каждая из которых занимает свое место. 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, сделайте его нестатическим, в остальных случаях он должен быть статическим. Предположим, что класс находится внутри метода. Если вам нужно создавать экземпляры этого класса только в одном месте программы и уже есть тип, который этот класс характеризует, сделайте его анонимным классом. В противном случае это должен быть локальный класс.

5

Г л а в а

Средства обобщенного программирования (Generics)

В релизе 1.5 в языке Java появились средства обобщенного программирования (*Generics*). До их появления необходимо было передавать все объекты из коллекции. Если кто-то случайно вставит объект неправильного типа, передача завершится с ошибкой при выполнении. С помощью средств обобщенного программирования вы даете указания компилятору, какие типы объектов разрешены в каждой коллекции. Компилятор автоматизирует передачу объектов и на этапе компиляции говорит вам о том, что вы пытаетесь вставить объект неверного типа. Это приводит к тому, что программы становятся не только более безопасными, но и более «чистыми». Однако эти преимущества сопряжены с определенными сложностями. В этой главе мы расскажем о том, как свести к минимуму сложности, и максимально извлечь пользу из преимуществ. Для более детального изучения материала посмотрите учебник Лангера [Langer08] или книгу Нафталина и Уодлера [Naftalin07].

Не используйте необработанные типы в новом коде

Для начала несколько терминов. Класс или интерфейс, декларация которого содержит один или более *типовых параметров* (*type parameters*), является *обобщенным* (*generic*) классом или интерфейсом [JLS, 8.1.2, 9.1.2]. Например, в релизе 1.5 у интерфейса List есть единственный типовой параметр E, представляющий тип элемента списка. Технически наименование интерфейса теперь будет List<E> (читается «Список E»), но часто для краткости его называют List. Обобщенные классы и интерфейсы известны под общим наименованием *обобщенные типы* (*generic types*).

Каждый обобщенный тип определяет набор *типов с параметрами* (*parameterized types*), который состоит из наименования класса или интерфейса, после которого следует в угловых скобках список *актуальных типовых параметров* (*actual type parameters*), соответствующий формальным типовым параметрам, относящимся к обобщенному типу [JLS, 4.4, 4.5]. Например, List<String> (читается «список String») — это тип с параметрами, представляющий список, элементы которого принадлежат типу String. (String — это актуальный параметр типа, соответствующий формальному параметру типа E.)

Наконец, каждый обобщенный тип определяет *необработанный тип* (*raw type*), который является наименованием обобщенного типа без каких-либо актуальных типовых параметров [JLS, 4.8]. Например, необработанным типом, соответствующим List<E>, будет List. Необработанные типы ведут себя так, словно вся информация об обобщенном типе оказалась стерта при его декларировании. На практике необработанный тип List ведет себя аналогично тому, как вел себя интерфейс List до добавления к платформе обобщенных средств программирования .

До релиза 1.5 это было бы типичной декларацией коллекции:

```
// Необработанный тип коллекции – не стоит так делать!
/**
 * Моя коллекция марок (stamp). Содержит только экземпляры Stamp.
 */
private final Collection stamps = ... ;
```

Если вы случайно добавите в вашу коллекцию марок монету, ошибочное добавление будет откомпилировано и выполнено без ошибок:

```
// Ошибочная вставка монеты в коллекцию марок
stamps.add(new Coin( ... ));
```

Ошибки не будет до тех пор, пока вы не станете извлекать монету из коллекции марок:

```
// Необработанный тип итератора – не стоит так делать!
for (Iterator i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = (Stamp) i.next(); // Throws ClassCastException
    ... // Do something with the stamp
}
```

Как уже было упомянуто в этой книге, лучше всего находить ошибки сразу, как только они сделаны, в идеале на этапе компиляции. В нашем же случае вы не найдете ошибку до момента выполнения программы, а найдете ее намного позднее, чем она сделана, и совершенно не в том коде, в котором она была допущена. Как только вы увидите сообщение ClassCastException, вам придется искать по всему коду и смотреть, как запускаются методы, которые помешают монету в коллекцию марок. Компилятор тут вам не поможет, потому что он не понимает комментарий, в котором сказано «Contains only Stamp instances».

С использованием средств обобщенного программирования вы заменяете комментарий расширенной декларацией типов для коллекции, которая даст компилятору дополнительную информацию, недоступную ранее в комментарии:

```
// Коллекция типов с параметрами – безопасно
private final Collection<Stamp> stamps = ... ;
```

Благодаря такой декларации компилятор знает, что stamps должна содержать только экземпляры Stamp, и гарантирует, что это условие будет соблюдаться, при условии, что весь код компилируется компилятором из релиза 1.5 или более позднего и что весь код компилируется без каких-либо предупреждений (или сокрытия предупреждений, см. статью 24). Когда коллекция stamps декларируется с использованием типов с параметрами, ошибочное добавление приводит к сообщению об ошибке уже на этапе компиляции, которое точно сообщает, в чем ошибка:

```
Test.java:9: add(Stamp) in Collection<Stamp> cannot be applied
to (Coin)
stamps.add(new Coin());
^
```

Дополнительным преимуществом будет то, что вам больше не потребуется передавать объекты вручную при извлечении элементов из коллекции. Компилятор добавляет невидимые механизмы передачи для вас и гарантирует, что они обязательно сработают (опять-таки при условии, что весь ваш код откомпилирован компилятором, который знает о существовании обобщенных средств программирования и не скроет никаких предупреждений). Это утверждение верно вне зависимости от того, используете ли вы цикл for-each (статья 46):

```
// Цикл for-each поверх коллекции с параметрами – безопасно
for (Stamp s : stamps) { // No cast
    // Do something with the stamp
}
```

или обычный цикл:

```
// Цикл for, декларируемый с использованием итератора с параметрами, –
// безопасно
for (Iterator<Stamp> i = stamps.iterator(); i.hasNext(); ) {
    Stamp s = i.next(); // No cast necessary
    ... // Do something with the stamp
}
```

Хотя перспектива случайной вставки монеты в коллекцию марок может показаться маловероятной, тем не менее проблема вполне реальна. Например, легко представить, что кто-то вставит экземпляр `java.util.Date` в коллекцию, которая должна содержать только экземпляры `java.sql.Date`.

Как уже упоминалось выше, до сих пор разрешено использовать типы коллекций и другие обобщенные типы без параметров, но делать такого не рекомендуется. **Если вы используете необработанные типы, то вы теряете все преимущества использования средств обобщенного программирования.** В то же время, если вам не нужно использовать необработанные типы, почему тогда создатели языка навязывают вам это? Чтобы обеспечить совместимость. Платформа Java существовала два десятилетия до того момента появления средств обобщенного программирования, и теперь в мире существует огромное количество кода, который это использует. Это критически важно, чтобы код все же позволял многое и мог взаимодействовать с новым кодом, использующим средства обобщенного программирования. Необходимо было разрешить передавать экземпляры типов с параметрами методам, которые были созданы для использования только простых типов, и наоборот. Это требование, известное как *миграционная совместимость* (*migration compatibility*), привело к решению сохранить поддержку необработанных типов.

Когда вам не нужно использовать необработанные типы, такие как `List` в новом коде, совершенно нормально использовать типы с параметрами, чтобы разрешить добавление произвольного объекта, такого как `List<Object>`. В чем же разница между необработанным типом `List` и типом с параметрами `List<Object>?` Проще говоря, в первом нет опции проверки типа, в то время как последний точно говорит компилятору, что он может содержать объект любого типа. В то время как вы можете передать `List<String>` параметру с типом `List`, вы не можете передать его с типом `List<Object>`. Существуют правила образования подтипов для средств обобщенного программирования. `List<String>` является подтипов необработанного типа `List`, но не является таковым для типа с параметрами `List<Object>` (ста-

тья 25). Как следствие, вы теряете в безопасности типов в случае использования необработанного типа, такого как `List`, но сохраняете ее при использовании типа с параметрами `List<Object>`.

Для конкретного примера рассмотрим следующую программу:

```
// Использует необработанный тип (List) - происходит ошибка
// при выполнении!
public static void main(String[] args) {
    List<String> strings = new ArrayList<String>();
    unsafeAdd(strings, new Integer(42));
    String s = strings.get(0); // Compiler-generated cast
}
private static void unsafeAdd(List list, Object o) {
    list.add(o);
}
```

Эта программа компилируется, но из-за того, что используется необработанный тип `List`, выходит предупреждение:

```
Test.java:10: warning: unchecked call to add(E) in raw type List
list.add(o);
^
```

И действительно, если вы запускаете программу, то вы получаете сообщение `ClassCastException`, когда программа пытается передать результат запуска `strings.get(0)` в `String`. Эта передача уже сгенерирована компилятором, поэтому гарантировано, что она успешно завершится. В данном случае мы проигнорировали предупреждение компилятора и заплатили за это.

Если вы замените необработанный тип `List` типом с параметрами `List<Object>` в декларации `unsafeAdd` и попробуете снова скомпилировать программу, то обнаружите, что она более не компилируется. Появляется сообщение об ошибке:

```
Test.java:5: unsafeAdd(List<Object>,Object) cannot be applied
to (List<String>,Integer)
unsafeAdd(strings, new Integer(42));
^
```

Возможно, у вас появится искушение использовать необработанный тип для коллекции, типы которой неизвестны или не имеют значения. Например, предположим, что вы хотите написать метод, который берет два набора и возвращает количество элементов, которые имеют между собой что-то общее. Так будет выглядеть метод, если вы незнакомы со средствами обобщенного программирования:

```
// Использование необработанных типов для неизвестных типов
// элементов - так делать не стоит!
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 он использует необработанные типы, что опасно. Релиз Java 1.5 предоставляет безопасную альтернативу, известную под названием несвязанный тип подстановки (unbound wildcard type). Если вы хотите использовать обобщенные типы, но не знаете или вам не важны актуальные параметры типов, то вы можете использовать вместо этого знак вопроса. Например, несвязанным типом подстановки для обобщенного типа `Set<E>` будет `Set<?>` (читается как «набор какого-то типа»). Это наиболее общий тип `Set` с параметрами, способный содержать любой набор. Вот пример как метод `numElementsInCommon` выглядит с использованием несвязанного типа подстановки:

```
// Unbounded wildcard type - typesafe and flexible
static int numElementsInCommon(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

В чем разница между несвязанным типом подстановки `Set<?>` и необработанным типом `Set?` Дает ли нам что-либо знак вопроса(`?`)? Об этом можно не задумываться, здесь понятно, что тип подстановки безопасен, а необработанный тип нет. Вы можете добавить в коллекцию любой элемент, используя необработанный тип, с легкостью повредив инварианты типа коллекции (как показано на примере метода `unsafeAdd`), но **вы не можете добавить любой элемент (кроме нулевого) в коллекцию `Collection<?>`**. При попытке сделать это на этапе компиляции будет выведено сообщение об ошибке:

```
WildCard.java:13: cannot find symbol
symbol : method add(String)
location: interface Collection<capture#825 of ?>
c.add(«verboten»);
^
```

Данное сообщение недостаточно информативно, но компилятор выполнил свою задачу, защитив инварианты от повреждения. Вы не только не можете добавить любой элемент (кроме нулевого) в `Collection<?>`, но также не можете ничего предположить относительно типа объекта, который извлечете. Если эти ограничения неприемлемы, то можно использовать *обобщенные методы (generic methods)* (статья 27) или *связанные типы подстановки (bounded wildcard types)* (статья 28).

Есть два небольших исключения из правила: не использовать необработанные типы в новом коде, оба они основаны на факте, что информация об обобщенных типах стирается при запуске (статья 25). **Вы должны использовать необработанные типы в лiteralных константах класса или литералах.** Спецификация не разрешает использование типов с параметрами в данном случае (хотя разрешает типы массивов и примитивные типы) [JLS 15.8.2]. Другими словами, `List.class`, `String[].class` использовать разрешено, но `List<String>.class` и `List<?>.class` не разрешено.

Второе исключение из правила связано с оператором `instanceof`. Поскольку информация об обобщенном типе стирается при выпол-

нении, то не разрешается использовать оператор `instanceof` в типах с параметрами, за исключением несвязанных типов подстановки. Использование несвязанных типов подстановки вместо необработанных типов никогда не влияет на поведение оператора `instanceof`. В данном случае угловые скобки и знаки вопроса излишни. Вот как предпочтительнее использовать оператор `instanceof` с обобщенными типами:

```
// Разрешенное использование необработанных типов - оператор instanceof
if (o instanceof Set) { // Raw type
    Set<?> m = (Set<?>) o; // Wildcard type
    ...
}
```

Обратите внимание, что как только вы определите `o` как `Set`, то должны передать его типу подстановки `Set<?>`, а не необработанному типу `Set`. Это правильная передача параметра и не приведет к предупреждениям со стороны компилятора.

Подводя итоги, можно сказать, что использование необработанных типов может привести к ошибке при выполнении. Не используйте их в новом коде. Они существуют лишь для совместимости и взаимодействия с кодом, написанным до появления средств обобщенного программирования. В качестве небольшого резюме: `Set<Object>` — это тип с параметрами, который может содержать объекты любого типа, `Set<?>` — тип подстановки, который может содержать только объекты некоторого неизвестного типа, и `Set` — это необработанный тип, который лишает нас возможности использовать систему обобщенных типов. Первые два безопасны, а вот последний — нет.

Термины, представленные в этой статье (и некоторые представленные в других местах этой главы), сведены в следующую таблицу:

Термин	Пример	Статья
Тип с параметрами	<code>List<String></code>	Статья 23
Актуальный параметр типа	<code>String</code>	Статья 23
Обобщенный тип	<code>List<E></code>	Статьи 23, 26

Термин	Пример	Статья
Формальный параметр типа	E	Статья 23
Несвязанный тип подстановки	List<?>	Статья 23
Необработанный тип	List	Статья 23
Связанный параметр типа	<E extends Number>	Статья 26
Рекурсивная связка типа	<T extends Comparable <T>>	Статья 27
Связанный тип подстановки	List<? Extends Number>	Статья 28
Обобщенный метод	Static <E> List<E> asList(E[] a)	Статья 27
Метка типа	String.class	Статья 29

Статья 24

Избегайте предупреждений о непроверенном коде

Если вы программируете с использованием обобщенных средств, то столкнетесь со множеством предупреждений компилятора: предупреждения о непроверенной передаче, предупреждение о непроверенном запуске метода, предупреждение о непроверенном создании обобщенного массива и о непроверенном преобразовании. По мере приобретения опыта работы с обобщенными методами программирования таких предупреждений будет все меньше и меньше, но не стоит быть уверенными, что новый написанный код с использованием обобщенных средств будет абсолютно безошибочным.

Многих предупреждений о непроверенном коде можно легко избежать. Например, предположим, вы случайно написали такую декларацию:

```
Set<Lark> exaltation = new HashSet();
```

Компилятор аккуратно напомнит вам, что вы сделали не так:

```
Venery.java:4: warning: [unchecked] unchecked conversion  
found : HashSet, required: Set<Lark>  
Set<Lark> exaltation = new HashSet();  
^
```

После чего вы можете сделать нужные исправления, который приведут к тому, что ошибка исчезнет:

```
Set<Lark> exaltation = new HashSet<Lark>();
```

Некоторых предупреждений избежать будет *намного* сложнее. В этой главе представлены примеры таких предупреждений. Когда вы получите предупреждения, которые потребуют от вас размышлений, внимательно разбирайтесь, в чем дело. **Избежать надо всех предупреждений, насколько это возможно.** Если всех предупреждений удастся избежать, то вы можете быть уверены, что код безопасен, и это очень хорошо. Это означает, что вы не получите сообщения ClassCastException при выполнении, и увеличит вашу уверенность в том, что программа ведет себя так, как вы и планировали.

Если же вы не можете избежать предупреждений, но сами уверены, что код, о котором вы получили предупреждение, безопасен, то тогда (и только тогда) можно скрыть предупреждения с помощью аннотации @SupressWarnings(«unchecked»). Если вы скроете ошибки, не убедившись, что код безопасен, то вы тем самым лишь даете себе ложное ощущение безопасности. Код может откомпилироваться без предупреждений, но при выполнении все равно вывести ошибку ClassCastException. Если тем не менее вы просто проигнорируете предупреждения о непроверенном коде, зная при этом, что он безопасен (вместо того чтобы скрыть эти предупреждения), то тогда вы не заметите, когда появится новое предупреждение, представляющее реальную проблему. Новое предупреждение потерянется среди ложных тревог, которые вы не скрыли.

Аннотация SupressWarnings может использоваться с детализацией любого уровня, от декларирования локальных переменных

до целых классов. **Всегда используйте аннотацию SuppressWarnings на как можно меньшем диапазоне.** Обычно это будет декларирование переменной, очень короткий метод или конструктор. Никогда не используйте SuppressWarnings на целом классе. Это может спрятать действительно критические предупреждения.

Если вы все же используете аннотацию SuppressWarnings для метода или конструктора длиной более чем в одну строку, то, возможно, сможете свести его до применения на одном декларировании локальной переменной. Возможно, вам понадобится декларировать новую локальную переменную, но это стоит того. Например, рассмотрим это на методе toArray, который идет 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 вы скомпилируете ArrayList, метод выведет следующее предупреждение:

```
ArrayList.java:305: warning: [unchecked] unchecked cast
found : Object[], required: T[]
return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

Не разрешено применять аннотацию SuppressWarnings на выражении возврата, потому что это не декларирование [JLS, 9.7]. Вы можете захотеть поместить аннотацию на весь метод, но делать этого не надо. Вместо этого декларируйте локальную переменную для хранения возвращаемого значения и аннотацию поместите на декларирование, например:

```
// Adding local variable to reduce scope of @SuppressWarnings
public <T> T[] toArray(T[] a) {
    if (a.length < size) {
```

```
// This cast is correct because the array we're creating
// is of the same type as the one passed in, which is 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;
```

Этот метод компилируется чисто и на минимальном диапазоне, в котором скрываются предупреждения.

Каждый раз при использовании аннотации `@SuppressWarnings`(«unchecked») добавляйте комментарий, в котором объясняйте, почему так делать в данном случае безопасно. Это поможет другим понять код и, что более важно, уменьшит шанс того, что код будет изменен и вычисления станут небезопасными. Если вам сложно будет написать такой комментарий, все равно подумайте, как это сделать. В конце концов вы можете прийти к выводу, что данная операция не безопасна в принципе.

Подводя итоги, можно сказать, что предупреждения о непроверенном коде важны. Не стоит их игнорировать. Каждое такое предупреждение представляет собой потенциальную предпосылку возникновения ошибки `ClassCastException` при выполнении. Сделайте все возможное, чтобы избежать их. Если же избежать не удается и вы уверены, что код безопасен, скройте предупреждения аннотацией `@SuppressWarnings`(«unchecked») на минимальном диапазоне. Опишите в комментарии причину, почему вы приняли решение скрыть предупреждения.

Статья
25

Предпочитайте списки массивам

Массивы отличаются от средств обобщенного программирования в двух важных аспектах. Во-первых, массивы ковариантны. Это

жуткое слово значит просто, что если Sub является подтипов Super, тогда тип массива Sub[] является подтипов Super[]. Средства обобщенного программирования, напротив, инвариантны: для любых двух отдельных типов Type1 и Type2, List<Type1> не является ни подтипов, ни превосходящим типом для List<Type2> [JLS, 4.10; Naftalin07, 2.5]. Вы можете подумать, что это недостаток средств обобщенного программирования, но, напротив, недостатками обладают массивы.

Этот фрагмент кода разрешен:

```
// Выводит ошибку при запуске!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Выводит сообщение
ArrayStoreException
```

А этот — нет:

```
// Не компилировать!
List<Object> ol = new ArrayList<Long>(); // Несовместимые типы
ol.add("I don't fit in");
```

В любом случае вы не можете вставить **String** в контейнер **Long**, но, применив массив, увидите, что сделали ошибку только при выполнении, в то время как использование списка приведет к ошибке на этапе компиляции.

Вторым важным отличием массивов от средств обобщенного программирования является то, что массивы материальны [JLS, 4.7]. Это значит, что массивы знают и выполняют свои типы элементов при выполнении. Как выше было сказано, если вы попытаетесь сохранить **String** в массив **Long**, вы получите сообщение об ошибке **ArrayStoreException**. Средства обобщенного программирования, напротив, реализуются стиранием [JLS, 4.6]. Это значит, что они выполняют свои ограничения типов только на этапе компиляции и затем выбрасывают (или стирают) информацию о типах элементов при выполнении. Стирание позволяет обобщенным средствам легко взаимодействовать с разрешенным кодом, который не использует средства обобщенного программирования (статья 23).

По причине этих фундаментальных различий массивы и средства обобщенного программирования не могут применяться одновременно. Например, нельзя создавать массив обобщенных типов и типов с параметрами. Ни одно из этих выражений создания массивов не является разрешенным: `newList<E>[]`, `newList<String>[]`, `new E []`. Все выражения приведут к ошибкам *создания обобщенных массивов* на этапе компиляции.

Почему нельзя создавать обобщенные массивы? Потому что это небезопасно. Если бы это было разрешено, приведение типов, генерируемое компилятором в правильно написанной программе, вызывало бы ошибку при выполнении `ClassCastException`. Это бы нарушило фундаментальные гарантии, которые дает система обобщенных типов.

Чтобы конкретнее разъяснить это, рассмотрим следующий фрагмент кода:

```
// Почему создание обобщенных массивов не разрешено - не компилировать!
List<String>[] stringLists = new List<String>[1]; // (1)
List<Integer> intList = Arrays.asList(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. Стока 2 создает и инициализирует `List<Integer>`, содержащий единственный элемент. Стока 3 сохраняет массив `List<String>` в переменную `Object`, что разрешено, потому что массивы ковариантны. Стока 4 сохраняет `List<Integer>` в единственный элемент массива `Object`, что тоже закончится удачно, потому что обобщенные типы реализуются стиранием: выполняемым типом экземпляра `List<Integer>` является просто `List` и выполняемым типом экземпляра `List<String>[]` является `List[]`, такое назначение не приводит к ошибке `ArrayStoreException`. Теперь у нас проблема. Мы сохранили экземпляр `List<Integer>` в массив, который декларирован, чтобы хранить только экземпляры `List<String>`. В строке 5

мы выводим один элемент из списка в этом массиве. Компилятор автоматически передаст извлеченный элемент в `String`, но он на самом деле `Integer`, так что мы получим при выполнении ошибку `ClassCastException`. Чтобы такого не случилось, строка 1 (создающая обобщенный массив) выдает ошибку при компиляции.

Типы, такие как `E`, `List<E>` и `List<String>`, известны под техническим названием *нематериальных* типов [JLS, 4.7]. Говоря интуитивно, нематериальные типы — это такие типы, представление которых содержит меньше информации при выполнении, чем при компиляции. Единственными материальными типами с параметрами являются несвязанные типы подстановки, такие как `List<?>` и `Map<?, ?>` (статья 23). Разрешено, хотя и не очень полезно, создание массивов несвязанных типов подстановки.

Запрет на создание обобщенных массивов может вызвать недоумение. Это означает, что невозможно обобщенным типам вернуть массив этого типа элемента (но см. статью 29 для частичного решения). Это также означает, что вы можете получить запутанные предупреждения при использовании методов `varargs` (статья 42), как как каждый раз при запуске метода `varargs` создается массив для хранения параметров этого метода. Если тип элемента в этом массиве не является материальным, то вы получите предупреждение. Не многое можно сделать, чтобы избежать этих предупреждений, — остается только их скрыть (статья 24) и избегать одновременного использования средств обобщенного программирования и методов `varargs` в API.

Когда будет выведена ошибка при создании обобщенного массива, лучшим решением будет использование типа коллекции `List<E>` вместо типа массива `E[]`. Вы можете пожертвовать частично производительностью или краткостью, но взамен вы получите большую безопасность типов и взаимодействие.

Например, предположим, что у вас есть синхронизированный список (который выводится `Collection.synchronizedList`) и функция, которая берет два значения из типов, хранящихся в списке, и возвращает третье. Теперь предположим, что вы хотите написать

метод для «уменьшения» списка с помощью функции. Если список содержит целые числа и функция складывает значения двух чисел, то метод `reduce` вернет сумму всех значений в списке. Если функция умножает два целых числа, метод вернет произведение значений списка. Если список содержит строковые значения и функция объединяет две строки, то метод вернет строку, содержащую обе строки соединенные последовательно. Помимо списков и функций метод `reduce` принимает начальное значение уменьшения, которое возвращается, если список пуст. (Начальное значение обычно идентифицирует элемент для функции, где 0 обозначает сложение, 1 — умножение, и “” объединение строк.) Вот как может выглядеть код без использования средств обобщенного программирования:

```
// Применение метода reduce без использования средств обобщенного
// программирования и с недостатками параллелизма.
static Object reduce(List list, Function f, Object initVal) {
    synchronized(list) {
        Object result = initVal;
        for (Object o : list)
            result = f.apply(result, o);
        return result;
    }
}
interface Function {
    Object apply(Object arg1, Object arg2);
}
```

Предположим, вы прочитали статью 67, в которой сказано, что нельзя вызывать «внешний метод» из синхронизированной зоны. Так что вам нужно модифицировать метод `reduce` для копирования комментариев списка, удерживая замок, который позволит вам выполнять уменьшение на копии. До релиза 1.5 мы бы использовали метод `List`'s `toArray` (который запирает список изнутри):

```
// Применение метода reduce без средств обобщенного
// программирования и недостатков параллелизма.
static Object reduce(List list, Function f, Object initVal) {
```

```

Object[] snapshot = list.toArray(); // Locks list internally
Object result = initialValue;
for (Object o : snapshot)
    result = f.apply(result, o);
return result;
}

```

Если вы попытаетесь сделать это, используя средства обобщенного программирования, то вы столкнетесь с проблемой, которую мы обсудили выше. Вот как будет выглядеть версия интерфейса Function со средствами обобщенного программирования:

```

interface Function<T> {
    T apply(T arg1, T arg2);
}

```

А вот наивная попытка использовать средства обобщенного программирования в пересмотренной версии метода reduce. Это обобщенный метод (статья 27). Не обращайте внимания, если вам не понятна декларация. Для В данном случае надо изучить содержимое метода:

```

// Наивная версия с использованием обобщенного программирования
// для метода reduce - не компилировать!
static <E> E reduce(List<E> list, Function<E> f, E initialValue) {
    E[] snapshot = list.toArray(); // Locks list
    E result = initialValue;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}

```

Если вы попытаетесь откомпилировать этот метод, то получится следующая ошибка:

```

Reduce.java:12: incompatible types
found : Object[], required: E[]
E[] snapshot = list.toArray(); // Locks list
^

```

«Не беда, — подумаете вы, — я передам массив Object в массив E»:

```
E[] snapshot = (E[]) list.toArray();
```

Это избавляет от ошибки, но теперь вы получите предупреждение:

```
Reduce.java:12: warning: [unchecked] unchecked cast  
found : Object[], required: E[]  
E[] snapshot = (E[]) list.toArray(); // Запирает список  
^
```

Компилятор говорит, что он не может проверить безопасность передачи при запуске, так как не представляет, чем будет E при запуске. Помните, что информация о типах элементов стирается при запуске. Будет ли работать программа? Да, окажется, что она будет работать, но она не безопасна. Небольшое изменение — и будет выходить сообщение ClassCastException в строке, которая не содержит исключительной ситуации. Во время компиляции E[] соответствует типу, который может быть String[], Integer[] или любым другим типом массива. Исполняемым типом является Object[], и именно это опасно. Передача массивов нематериальных типов должна использоваться только при определенных обстоятельствах (статья 26).

Итак, что же вам следует делать? Использовать списки вместо массивов. Вот вариант метода reduce, который откомпилируется без ошибок и предупреждений.

```
// Метод reduce с использованием списка и средств обобщенного  
// программирования.  
static <E> E reduce(List<E> list, Function<E> f, E initVal) {  
    List<E> snapshot;  
    synchronized(list) {  
        snapshot = new ArrayList<E>(list);  
    }  
    E result = initVal;  
    for (E e : snapshot)  
        result = f.apply(result, e);  
    return result;  
}
```

Эта версия гораздо более «многословна», нежели версия с использованием массивов, но оно того стоит для успокоения. Теперь мы точно знаем, что у нас не будет ошибки ClassCastException при выполнении.

Подводя итоги, можно сказать, что у массивов и средств обобщенного программирования различные правила для типов. Массивы ковариантны и материальны, обобщенные средства — инвариантны и используют механизм «стирания». Следствием этого становится то, что массивы при запуске обеспечивают безопасность, но не безопасность типов при компиляции. С обобщенными средствами все совсем наоборот, т.е. массивы и обобщенные средства не стоит использовать вместе. Если вам все же придется сделать это и получить при этом ошибки или предупреждения при компиляции, то в первую очередь постарайтесь заменить массивы списками.

Статья 26

Поддерживайте обобщенные типы

Обычно не так сложно задать параметры в декларировании коллекции и использовать обобщенные типы и методы, поставляемые JDK. Написание же своего собственного обобщенного типа несколько сложнее. Однако это стоит того, и вам следует этому научиться.

Рассмотрим простую реализацию стека из статьи 6:

```
// Коллекция на основе объекта – типичный кандидат для обобщения
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);
    }
}
```

Данный класс — основной претендент для обобщения, другими словами, идеально подходит для применения преимуществ средств обобщенного программирования. В том виде, в каком оно сейчас есть, вы должны передавать объекты, которые выпадают из стека, и эти передачи будут выдавать ошибку при выполнении. Первый шаг — обобщения класса связан с тем, что можно было добавить один или несколько параметров при его декларировании. В нашем случае есть только один параметр, представляющий тип элемента стека и наименование этого параметра E (статья 56).

Следующий шаг — надо заменить все использования типа Object соответствующим параметром типа и выполнить компиляцию получившейся программы:

```
// Начальная попытка обобщить стек - не компилировать!
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; // Eliminate obsolete reference
    return result;
}
// no changes in isEmpty or ensureCapacity
}

```

Вы получите по крайней мере одну ошибку или предупреждение в этом классе, и этот класс не исключение. К счастью, этот класс выдаст только одну ошибку:

```

Stack.java:8: generic array creation
elements = new E[DEFAULT_INITIAL_CAPACITY];
^

```

Как объяснено в статье 25, вы не можете создавать массив из нематериальных типов, таких как E. Эта проблема появляется каждый раз, когда вы пишете обобщенный тип, поддерживаемый массивом. Есть два варианта решения проблемы. Первое решение 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 целом не является безопасным для типов:

```

Stack.java:8: warning: [unchecked] unchecked cast
found : Object[], required: E[]
elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
^

```

Возможно, компилятор не сможет доказать, что ваша программа безопасна, но это можете сделать вы. Вам необходимо убедить себя, что непроверенная передача не нарушит безопасность типов программы. Массив в запросе (`elements`) хранится в закрытом поле и никогда не возвращается клиенту и не передается методу `push` (которые имеет тип `E`), так что непроверенная передача не может причинить вреда.

После того как вы убедились, что непроверенная передача безопасна, скройте все предупреждения в насколько возможно узком диапазоне (статья 24). В этом случае конструктор содержит только непроверенное создание массива, так что вполне normally скрыть предупреждения во всем конструкторе. После добавления аннотации `Stack` отлично компилируется и вы можете использовать его, не боясь явных передач и исключений `ClassCastException`:

```
// Элементы массива будут содержать только экземпляры E из push(E).
// Достаточно удостовериться в безопасности типов, однако
// исполняемый тип массива будет не E[]; а Object[]!
@SuppressWarnings("unchecked")
public Stack() {
    elements = (E[]) new Object[DEFAULT_INITIAL_CAPACITY];
}
```

Второй способ избежать ошибок при создании обобщенных массивов — это изменить тип поля `elements` с `E[]` на `Object[]`. Если вы так сделаете, то получите другую ошибку:

```
Stack.java:19: incompatible types
found : Object, required: E E result = elements[-size];
^
```

Вы можете изменить эту ошибку на предупреждение, передавая извлекаемый элемент массива от `Object` в `E`:

```
Stack.java:19: warning: [unchecked] unchecked cast
found : Object, required: E E result = (E) elements[-size];
^
```

Поскольку `E` является нематериальным типом, то компилятор не может проверить его передачу при выполнении. Снова вы можете легко доказать себе, что непроверенная передача безопасна, поэтому нормально будет скрыть предупреждения. Следуя совету из 24-й статьи, мы скрываем предупреждения только для той части, которая содержит непроверенную передачу, а не на всем методе `pop`:

```
// Допустимое скрытие предупреждений о непроверенном коде
public E pop() {
    if (size == 0)
        throw new EmptyStackException();
    // push требует от элементов принадлежности к типу E,
    // поэтому передача верна
    @SuppressWarnings("unchecked") E result =
        (E) elements[-size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

Какую из двух технологий выбрать для создания обобщенных массивов — это вопрос вашего вкуса. В основном все одинаково. Более рискованно скрывать предупреждения о непроверенных передачах при использовании типов массивов, нежели при использовании скалярных типов, что предполагает использование второго решения. Но в более реалистичном классе, чем `Stack`, вы, вероятнее всего, будете производить чтение из массива во многих местах кода, так что выбор второго решения потребует несколько передач `E` вместо однократной передачи `E[]`, поэтому первое решение является более общепринятым [Naftalin07, 6.7].

Следующая программа демонстрирует использование обобщенного класса `Stack`. Программа печатает свои аргументы командной строки в обратном порядке и переведет их в верхний регистр. Не требуется явных передач для запуска метода `String's toUpperCase` на элементах, выпадающих из стека, и автоматически генерирует передачу для гарантии успешного выполнения:

```
// Маленькая программа, использующая на практике обобщенный класс Stack
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg : args)
        stack.push(arg);
    while (!stack.isEmpty())
        System.out.println(stack.pop().toUpperCase());
}
```

Может оказаться, что вышеупомянутый пример противоречит статье 25, которая рекомендует нам использовать списки вместо массивов. Не всегда возможно и желательно использовать списки внутри обобщенных типов. Java не поддерживает списки сами по себе, так что некоторые обобщенные типы, такие как `ArrayList`, должны реализовываться поверх массивов. Другие обобщенные типы, такие как `HashMap`, реализуются поверх массивов для улучшения производительности.

Подавляющее большинство обобщенных типов похожи на наш пример со `Stack` в том, что его параметры не имеют ограничений: вы можете создать `Stack<Object>`, `Stack<int[]>`, `Stack<List<String>>` или `Stack` от любого другого типа, ссылающегося на объект. Обратите внимание, что вы не можете создавать `Stack` примитивного типа: попытка создать `Stack<int>` или `Stack<double>` приведет к ошибке компиляции. Это фундаментальное ограничение, накладываемое системой обобщенного программирования в языке Java. Вы можете обойти ограничения, используя упаковываемые примитивные типы (статья 49).

Есть несколько обобщенных типов, которые ограничивают разрешенные значения своих параметров. Например, рассмотрим `java.util.concurrent.DelayQueue`, декларация которой 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 `java.util.concurrent.`

Delayed. Это позволит реализации DelayQueue и ее клиентам использовать преимущества методов Delayed на элементе DelayQueue без необходимости явной передачи или риска ошибки ClassCastException. Параметр типа E называется *связанный параметр*. Обратите внимание, что отношение подтипа определено таким образом, что каждый тип является подтипом самого себя [JLS, 4.10], так что разрешается создавать DelayQueue<Delayed>.

Подводя итоги, можно сказать, что обобщенные типы более безопасны и легки в применении, чем типы, требующие передачи в клиентском коде. Когда вы проектируете новые типы, убедитесь, что они могут быть использованы без подобного рода передач. Зачастую это будет означать их обобщение. Обобщайте существующие типы, насколько позволит время. Это облегчит жизнь новым пользователям этих типов, не нарушая существующего клиентского кода (статья 23).

Статья 27

Поддерживайте обобщенные методы

Методы, подобно классам, получают свои преимущества в виде обобщения. Статические служебные методы — хорошие претенденты для этого. Все алгоритмические методы в Collection (такие, как binarySearch и Sort) обобщены.

Написание обобщенных методов схоже с написанием обобщенных типов. Рассмотрим метод, который возвращает объединение двух множеств:

```
// Использует необработанные типы — недопустимо! (статья 23)
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(s1);
    result.addAll(s2);
    return result;
}
```

Этот метод можно откомпилировать, но 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);  
    ^
```

Для того чтобы исправить эти предупреждения и сделать метод безопасным для типов, измените декларирование метода так, чтобы он декларировал параметры, представляющие типы элемента для трех множеств (два аргумента и возвращаемое значение), и используйте параметры в методе. **Список параметров, который их декларирует, лежит между модификатором метода и его возвращаемым типом.** В этом примере списком параметров является `<E>` и возвращаемый тип `Set<E>`. Условия наименований для параметров для типа те же, что и для обобщенных методов и для обобщенных типов (статьи 26, 56):

```
// Обобщенный метод  
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {  
    Set<E> result = new HashSet<E>(s1);  
    result.addAll(s2);  
    return result;  
}
```

По крайней мере что касается простых обобщенных методов, то это все, что следует знать. Теперь метод компилируется без каких-либо предупреждений и обеспечивает безопасность типов и простоту использования. Это — простая программа для того, чтобы практиковаться с нашим методом. Программа не содержит передач и компилируется без ошибок и предупреждений:

```
// Simple program to exercise generic method
```

```

public static void main(String[] args) {
    Set<String> guys = new HashSet<String>(
        Arrays.asList("Tom", "Dick", "Harry"));
    Set<String> stooges = new HashSet<String>(
        Arrays.asList("Larry", "Moe", "Curly"));
    Set<String> aflcio = union(guys, stooges);
    System.out.println(aflcio);
}

```

При запуске программы она напишет [Moe, Harry, Tom, Curly, Larry, Dick]. Порядок элементов зависит от их реализации.

Ограничение метода `union` заключается в том, что типы всех трех наборов (как и входные параметры, так и возвращаемые значения) должны быть одинаковы. Вы можете сделать методы более гибкими, используя *связанные типы подстановки* (статья 28).

Еще одна особенность обобщенных методов, которую стоит отметить, заключается в том, что вам нет необходимости явно задавать значение параметра для типа, как это необходимо делать при запуске обобщенных конструкторов. Компилятор определяет значение параметров, рассматривая типы аргументов метода. В случае с вышеупомянутой программой компилятор видит, что оба аргумента для `union` принадлежат типу `Set<String>`, следовательно, он знает, что параметр типа Е должен быть `String`. Этот процесс называется *выводом типов*.

Как уже говорилось в статье 1, вы можете использовать вывод типов, предоставляемый запуском обобщенных методов, чтобы облегчить процесс создания экземпляров типов с параметрами. Для обновления памяти необходимость явно передавать значения параметров для типов при запуске обобщенных конструкторов может раздражать. Параметры оказываются избыточными слева и справа от декларирования переменных:

```

// Создание экземпляра типа с параметрами с помощью конструктора
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();

```

Чтобы избежать этой избыточности, напишите обобщенный метод статической генерации, соответствующий каждому конструктору, который вы хотите использовать. Например, вот метод статической генерации, соответствующий конструктору без параметров `HashMap`:

```
// Обобщенный метод статической генерации
public static <K, V> HashMap<K, V> newHashMap() {
    return new HashMap<K, V>();
}
```

С этим обобщенным методом статической генерации вы можете заменить вышеописанную повторяющуюся декларацию этой, более краткой:

```
// Создание экземпляра типа с параметрами с помощью метода
// статической генерации
Map<String, List<String>> anagrams = newHashMap();
```

Было бы замечательно, если бы язык выполнял вывод типа при запуске конструктора на обобщенных типах, как он делает это при запуске обобщенных методов. Когда-нибудь такое будет возможно, а сейчас, в релизе 1.6, этого нельзя сделать.

Схожий шаблон — *статический обобщенный синглтон*. Допустим, вам надо создать объект, который неизменяем, но должен применяться на многих различных типах. Поскольку обобщенные средства реализуются «стиранием» (статья 25), вы можете использовать единственный объект для всех требуемых типов параметризации, но вам нужно будет написать метод статической генерации, чтобы постоянно раздавать объекты для каждой запрошенной параметризации типов. Этот шаблон зачастую используется для объектов функций (статья 21), таких как `Collections.reverseOrder`, но он также используется и для коллекций, подобных `Collections.emptySet`.

Предположим, у вас есть интерфейс, описывающий функцию, которая принимает и возвращает значение некоторого типа `T`:

```
public interface UnaryFunction<T> {
    T apply(T arg);
}
```

Теперь предположим, что вы хотите сделать функцию идентификации. Будет напрасной тратой времени создавать новую функцию каждый раз, когда она потребуется. Если обобщенные средства были бы материальны, то вам бы понадобилась одна функция идентификации для каждого типа, но поскольку обобщенные средства «стираются», то вам потребуется синглтон. Это выглядит так:

```
// Шаблон обобщенного статического синглтона
private static UnaryFunction<Object> IDENTITY_FUNCTION =
new UnaryFunction<Object>() {
    public Object apply(Object arg) { return arg; }
};
// IDENTITY_FUNCTION является ненужной, и ее параметры не являются
// (звучит как-то странно) связанными, так что безопасно
// использовать общий экземпляр для всех типов
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

Передача IDENTITY_FUNCTION в (UnaryFunction<T>) выдает предупреждение о непроверенной передаче, поскольку UnaryFunction<Object> не является UnaryFunction<T> для любого значения T. Но функция идентификации особая: она возвращает свои аргументы неизменными, так что мы знаем, что безопасно использовать UnaryFunction<T> вне зависимости от значения T. Следовательно, мы можем с уверенностью скрыть предупреждение, сгенерированное передачей. После того как мы сделали это, код компилируется без ошибок или предупреждений.

Приведем пример программы, использующей наш обобщенный синглтон как UnaryFunction<String> и UnaryFunction<Number>. Как обычно, он не содержит передачи и компилируется без ошибок и предупреждений:

```
// Простая программа с использованием обобщенного синглетона
public static void main(String[] args) {
    String[] strings = { "jute", "hemp", "nylon" };
    UnaryFunction<String> sameString = identityFunction();
    for (String s : strings)
        System.out.println(sameString.apply(s));
    Number[] numbers = { 1, 2.0, 3L };
    UnaryFunction<Number> sameNumber = identityFunction();
    for (Number n : numbers)
        System.out.println(sameNumber.apply(n));
}
```

Допускается, хотя и относительно редко, чтобы параметр для типа был ограничен неким выражением, включающим в себя сам этот параметр. Это явление известно под названием рекурсивное ограничение типа. Наиболее часто она используется в связи с интерфейсом Comparable, который определяет естественный порядок типов:

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

Параметр типа T определяет тип, с которым можно сравнить реализацию элементов типа Comparable<T>. На практике почти все типы можно сравнить с элементами своего собственного типа. Так, например, String реализует Comparable<String>, Integer реализует Comparable<Integer>, и т.д.

Есть много методов работы со списками элементов, которые реализуют Comparable для сортировки списка, поиска внутри его, расчета минимума и максимума и подобных операций. Для того чтобы сделать любое из этих действий, необходимо, чтобы каждый элемент списка можно было сравнить с любым другим элементом в списке, другими словами, чтобы элементы списка были *взаимно сопоставимы*. Вот как можно выразить это ограничение:

```
// Используя рекурсивное ограничение типа для выражения взаимной
// сопоставимости.
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

Ограничение типа `<T extends Comparable<T>>` можно прочитать «для каждого типа T , который может быть сопоставлен с самим собой», что более или менее точно передает явление взаимной сопоставимости.

Приведем метод, который надо использовать с декларированием, упомянутым выше. Он рассчитывает максимальное значение списка согласно натуральному порядку его элементов и компилируется без ошибок и предупреждений:

```
// Возвращает максимальное значение из списка - использует
// рекурсивное ограничение типа
public static <T extends Comparable<T>> T max(List<T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

Рекурсивное ограничение типа может быть намного сложнее, чем представлено здесь, но, к счастью, оно не используется слишком часто. Если вы понимаете идиому и ее вариант в виде групповых символов (статья 28), то сможете справиться со многими рекурсивными ограничениями типов, которые встречаются на практике.

Подводя итоги, можно сказать, что обобщенные методы, как и обобщенные типы, более безопасны и их легче использовать, чем методы, требующие от клиентов передачи входных параметров и возвращаемых значений. Как и с типами, вы должны быть уверены, что ваши новые методы можно будет использовать без передач, что зачастую будет означать, что они обобщенные. Подобно работе с типами, вам нужно сделать обобщенными ваши существующие методы, чтобы облегчить жизнь новым пользователям, не изменяя код существующих клиентов (статья 23).

*Статья
28*

Используйте ограниченные групповые символы для увеличения гибкости API

Как было отмечено в статье 25, типы с параметрами являются **инвариантными**. Другими словами, для любых двух отдельных типов Type 1 и Type 2, `List<Type1>` не является ни подтипов, ни супертипов для `List<Type2>`. Хотя в том, что `List<String>` не является подтипов `List<Object>` и звучит парадоксально, в этом есть здравый смысл. Вы можете поместить любой объект в `List<Object>`, но вы можете поместить только строку в `List<String>`.

Иногда вам нужна большая гибкость, чем та, которую могут вам дать инвариантные типы. Рассмотрим класс `Stack` из статьи 26. Чтобы освежить вашу память, вот 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 берет последовательность элементов и вставляет их в стек. Сделаем первую попытку:

```
// Метод pushAll без использования групповых символов – имеет
// недостатки!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

Этот метод компилируется чисто, но он не является идеальным. Если тип элементов `Iterable src` идеально совпадает с аналогичным типом в стеке, то тогда он работает хорошо. Но предположим, что у вас есть `Stack<Number>` и вы запускаете `push(intVal)`, где `intVal`

принадлежит к типу `Integer`. Это работает, но только потому, что `Integer` является подтипов `Number`. Так что все выглядит довольно логично, и это тоже должно работать:

```
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

Если тем не менее вы попробуете выполнить это, то получите сообщение об ошибке, так как, было выше сказано, типы с параметрами являются инвариантными:

```
StackTest.java:7: pushAll(Iterable<Number>) in Stack<Number>
cannot be applied to (Iterable<Integer>)
numberStack.pushAll(integers);
^
```

К счастью, выход есть. Язык дает нам специальный вид типов с параметрами, называемый *ограниченным типом групповых символов (wildcard-type)*, для того чтобы справляться с подобными ситуациями. Тип входных параметров для `PushAll` должен быть не «`Iterable от E`», а «`Iterable некоего подтипа E`», и это есть тип группового символа, который в точности читается как: `Iterable<? Extends E>`. (Использование ключевого слова `extends` слегка запутывает: вспоминаем из статьи 26, что *подтип* определяется таким образом, что каждый тип является подтиром самого себя.). Давайте изменим `pushAll` для использования этого типа:

```
// Wildcard-тип для параметра, служащего производителем E public
void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

С этими изменениями не только `Stack` компилируется чисто, но также и любой клиентский код, который бы не компилировался с декларированием оригинального метода `pushAll`. Поскольку и `Stack` и его клиенты компилируются чисто, то вы знаете, что использование типов безопасно.

Теперь предположим, что вы хотите написать метод `popAll` для работы с `pushAll`. Метод `popAll` выталкивает из стека каждый элемент и добавляет элементы к данной коллекции. Вот как может выглядеть первая попытка написать такой метод `popAll`:

```
// Метод popAll без Wildcard-типа – имеет недостатки!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

Опять-таки, компилируется чисто и работает прекрасно, если элементы коллекции назначения точно совпадают с элементами в стеке. Но снова это не выглядит достаточно удовлетворительно. Предположим, у вас есть `Stack<Number>` и переменная типа `Object`. Если вы вытолкнете элемент из стека и сохраните его в переменной, то компилироваться и запускаться все будет без ошибок. Можно так поступить?

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

Если вы попробуете скомпилировать этот клиентский код вместе с вышеописанной версией метода `popAll`, то будет выведена ошибка, похожая на ту, которую мы видели в первом варианте метода `pushAll`: `Collection<Object>` не является подтипом `Collection<Number>`. И снова wildcard-тип дает нам выход. Тип входного параметра для `popAll` должен быть не «коллекцией `E`», а «коллекцией некоего подтипа `E`» (где супертип определен таким образом, что `E` является супертипов для самого себя [JLS, 4.10]). Здесь есть wildcard-тип, который точно следует читать как: `Collection<? super E>`. Давайте изменим метод `popAll`, чтобы использовать его:

```
// Wildcard-тип для параметра, являющегося потребителем E
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

С этими изменениями и Stack, и клиентский код компилируются без ошибок.

Урок понятен. Для максимальной гибкости нужно использовать wildcard-типы для входных параметров, представляющих производителей или потребителей. Если входные параметры являются одновременно и производителем и потребителем, тогда wildcard-типы не смогут вам помочь: вам нужно будет точное совпадение типов, что вы можете получить и без их использования.

Вот схема, которая поможет вам помнить, какой wildcard-тип использовать:

PECS значит производитель — extends,
потребитель — super (producer-extends, consumer-super).

Другими словами, если тип с параметрами представляет производителя T, используйте $\langle ? \text{ Extends } T \rangle$, а если он представляет потребителя T, используйте $\langle ? \text{ Super } T \rangle$. В нашем примере с классом Stack параметры src метода pushAll производят экземпляры E для использования стеком Stack. Подходящими типами для src будут Iterable $\langle ? \text{ Extends } E \rangle$, где параметры dst метода popAll потребляют экземпляры E из Stack и соответствующим типом для dst будет Collection $\langle ? \text{ Super } E \rangle$. Схема PECS демонстрирует фундаментальный принцип, который управляет использованием wildcard-типов. Naftalin и Wadler называют это *Get and Put Principle* [Naftalin07, 2.4].

Имея в голове эту схему, давайте посмотрим на декларацию некоторых методов из предыдущих статей. У метода reduce в статье 25 декларация:

```
static <E> E reduce(List<E> list, Function<E> f, E initVal)
```

Хотя списки могут быть как потребителями, так и производителями значений, метод reduce использует параметры list только как производитель E, так что его декларация должна использовать wildcard-тип, который расширяет (extends) E. Параметр f представляет функцию, которая как потребляет, так и производит экземпляры E,

так что ее wildcard тип не подойдет для этого. Приведем пример получившейся декларации метода:

```
// Wildcard-тип для параметра, служащего производителем E  
static <E> E reduce(List<? extends E> list, Function<E> f,  
E initVal)
```

Будет ли данное изменение на практике значить что-либо? Как оказывается, будет. Предположим, у нас есть `List<Integer>` и вы хотите уменьшить его с помощью `Function<Number>`. При оригинальной декларации это компилироваться не будет, однако если вы добавите ограниченный wildcard-тип, то все будет иначе.

Давайте рассмотрим метод `union` из статьи 27. Пример его декларации:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Оба параметра `s1` и `s2` являются производителями `E`, так что схема РЕС говорит нам, что декларация должна иметь вид:

```
public static <E> Set<E> union(Set<? extends E> s1,  
Set<? extends E> s2)
```

Обратите внимание, что возвращаемый тип все еще `Set<E>`. **Не используйте wildcard-тип в качестве возвращаемого типа.** Вместо того чтобы давать больше гибкости пользователям, он будет заставлять их использовать wildcard-тип в клиентском коде.

Wildcard-типы при правильном использовании почти невидимы для пользователей класса. Они заставляют методы принимать параметры, которые они должны принимать, и отвергать те, которые должны отвергать. **Если пользователь класса должен думать о наличии wildcard-типов, то, значит, что-то не так с API класса.**

К сожалению, правила вывода типов довольно сложны. Они занимают 16 страниц в спецификации языка [JLS, 15.12.2.7-8] и не всегда делают то, что вы от них хотите. Глядя на пересмотренную декларацию `union`, можно подумать, что сделаете так:

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);
```

Если вы попробуете, то получите сообщение об ошибке:

```
Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number & Comparable<?>>>
required: Set<Number>
Set<Number> numbers = union(integers, doubles);
```

К счастью, есть способ справиться с подобными ошибками. Если компилятор не выводит типы, которые вы хотите, тогда скажем ему, какие типы использовать с помощью *исключительного параметра*. Не так часто придется это делать, что хорошо, так как исключительные параметры типа не очень хороши сами по себе. С исключительным параметром программа компилируется без ошибок:

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Теперь обратим внимание на метод `max` из статьи 27. Вот его оригинальная декларация:

```
public static <T extends Comparable<T>> T max(List<T> list)
```

Вот переделанная декларация, использующая wildcard-типы:

```
public static <T extends Comparable<? super T>> T max(
    List<? extends T> list)
```

Для получения измененной декларации мы дважды применили схему PECS. Напрямую использовали ее для параметра `list`. Он произвел экземпляры `T`, поэтому мы заменили тип с `List<T>` на `List<? Extends T>`. Рассмотрим более сложное использование на параметре типа `T`. Рассмотрим пример использования группового символа на параметре типа. Изначально `T` должен был расширять `Comparable<T>`, но `Comparable<T>` потребляет экземпляры `T` (и производит целые числа, отражающие порядковые отношения). Следовательно, тип с параметрами `Comparable<T>` заменяется ограниченным wildcard-типом `Comparable<? super T>`. Все `Comparable` всегда являются потребителями, так что всегда следует использовать `Comparable<?`

super T> вместо Comparable<T>. То же самое относится и к Comparator, так что вы всегда должны использовать Comparator<? super T> вместо Comparator<T>.

Измененное декларирование max вероятнее является самым сложным декларированием метода во всей книге. Дает ли нам что-то эта увеличенная сложность? Да, дает. Вот простой пример списка, который будет исключен в оригинальной декларации, но допустим в измененной:

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

Причина, почему вы не можете применить оригинальное декларирование метода к данному списку, в том, что java.util.concurrent.ScheduledFuture не реализует Comparable<ScheduledFuture>. Вместо этого есть субинтерфейс от Delayed, который расширяет Comparable<Delayed>. Другими словами, экземпляр ScheduledFuture не просто сопоставим с другими экземплярами ScheduledFuture; он сопоставим с любым экземпляром Delayed, и этого недостаточно, чтобы декларация его отвергла.

Есть одна небольшая проблема с измененной декларацией max: она не дает методу компилироваться. Вот метод с измененным декларированием:

```
// Не будет компилироваться - групповым символам могут
// потребоваться изменения в теле метода!
public static <T extends Comparable<? super T>> T max( List<?
extends T> list) {
    Iterator<T> i = list.iterator();
    T result = i.next();
    while (i.hasNext()) {
        T t = i.next();
        if (t.compareTo(result) > 0)
            result = t;
    }
    return result;
}
```

Вот что произойдет при попытке компиляции:

```
Max.java:7: incompatible types
found : Iterator<capture#591 of ? extends T>
required: Iterator<T>
Iterator<T> i = list.iterator();
^
```

Что значит это сообщение об ошибке и как решить проблему? Это значит, что `list` не является `List<T>`, так что его метод `iterator` не возвращает `Iterator<T>`. Он возвращает `iterator` некоего подтипа `T`, так что мы заменяем декларацию `iterator` на эту, которая использует ограниченные wildcard-типы.

```
Iterator<? extends T> i = list.iterator();
```

Это единственное изменение, которое нужно сделать в теле метода. Элементы, возвращаемые методом `iterator next`, являются некоторыми подтипами `T`, так что они могут быть безопасно сохранены в переменной типа `T`.

Есть еще одна тема, связанная с wildcard-типами, которая стоит обсуждения. Есть двойственность между параметрами для типов и групповыми символами, так как многие методы могут декларироваться с использованием либо одного, либо другого. Например, вот две возможные декларации статического метода перестановки двух индексированных пунктов в списке. Первый использует неограниченный параметр wildcard-типа (статья 27), а второй использует неограниченный групповой символ:

```
// Две возможные декларации метода swap
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Какая из двух деклараций наиболее предпочтительна, и почему? В открытом API вторая лучше, потому что она проще. Вы передаете в список любой список, и метод переставляет индексированные элементы. Не надо волноваться о параметрах. Как правило, **если параметр для типа появляется только один раз в декларации метода**

да, замените его групповым символом. Если это не ограниченный параметр, замените его неограниченным групповым символом, если же это ограниченный параметр, замените его ограниченным групповым символом.

Есть одна проблема со второй декларацией swap, использующей групповой символ вместо параметра: непосредственная реализация его не будет компилироваться:

```
public static void swap(List<?> list, int i, int j) {  
    list.set(i, list.set(j, list.get(i)));  
}
```

Попытка компиляции приведет к малоинформативному сообщению об ошибке:

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?>  
cannot be applied to (int, Object)  
list.set(i, list.set(j, list.get(i)));  
^
```

Кажется, но это неверно, что мы не можем поместить элемент обратно в список, из которого мы его только что взяли. Проблема в том, что типом List является List<?> и вы не можете поместить любое значение, кроме нулевого в List<?>. К счастью, есть способ реализовать метод, не прибегая к небезопасной передаче или необработанным типам. Вспомогательный метод должен быть обобщенным методом для того, чтобы захватить тип. Вот как он должен выглядеть:

```
public static void swap(List<?> list, int i, int j) {  
    swapHelper(list, i, j);  
}  
  
// Закрытый вспомогательный метод для захвата групповых символов  
private static <E> void swapHelper(List<E> list, int i, int j) {  
    list.set(i, list.set(j, list.get(i)));  
}
```

Метод swapHelper знает list как list<E>. Следовательно, он знает, что любое значение, которое он получает из этого списка, является

типов \mathbb{E} и что безопасно поместить любое значение типа \mathbb{E} в список. Эта несколько сложная реализация swap компилируется без ошибок. Она позволяет экспортировать декларацию swap на основе групповых символов, одновременно извлекая пользу от более сложных обобщенных методов изнутри. Клиентам метода swap не придется сталкиваться с более сложной декларацией swapHelper, но они только выигрывают от этого.

Подведем итоги. Использование wildcard-типов в вашем API делает их более гибкими, хотя и более сложными. Если вы напишете библиотеку, которая будет широко использоваться, то правильное использование wildcard-типов должно быть в ней обязательным. Помните основное правило: схема РЕС и что все Comparable и Comparator являются потребителями.

*Статья
29*

Использование неоднородных контейнеров

Наиболее часто обобщенные средства программирования используются для коллекций, таких как Set и Map, или для одноэлементных контейнеров, таких как ThreadLocal и AtomicReference. Во всех этих случаях параметры присваиваются именно контейнерам. Это ограничивает вас до определенного количества параметров на один контейнер. Обычно это именно то, что нам нужно. У типа Set есть один параметр, представляющий его элементы; у типа Map есть два, представляющие его ключ и типы значений, и.т.д.

Иногда, тем не менее, нам требуется большая свобода действий. Например, строка базы данных может содержать большое количество столбцов, и было бы неплохо получить к ним безопасный доступ с точки зрения использования типов. К счастью, есть такой очень легкий способ. Идея в том, чтобы параметры присвоить *ключу*, а не контейнеру. Затем представить ключ с параметрами контейнеру, чтобы добавлять или извлекать значение. Система обобщенных типов используется для гарантии, что тип значений согласуется с его ключом.

В качестве простого примера данного подхода рассмотрим класс Favorites, который позволяет своим клиентам хранить и извлекать экземпляры «favorite» из произвольно большого количества классов. Объект Class будет играть роль ключа с параметрами. Причина, почему это работает, в том, что класс Class был обобщен в версии 1.5. Тип литерала класса больше не просто Class, а Class<T>. Например, String.class относится к типу Class<String> и класс Integer.class относится к классу Class<Integer>. Когда литерал класса передается между методами, чтобы обращаться к информации во время компиляции и во время выполнения, то это называется *метка типа* (*type token*) [Bracha04].

API для класса Favorites прост. Он выглядит как простая карта, кроме того, что ключ имеет параметры, а не карта. Клиент представляет объект Class при установке и получении избранных. Вот как выглядит API:

```
// Безопасный шаблон неоднородного контейнера – API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Приведем пример программы, которая практикуется на классе Favorites, сохраняя, извлекая и печатая экземпляры String, Integer и Class:

```
// Безопасный шаблон неоднородного 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 вы и ожидали, программа печатает Java cafebabe Favorites. Экземпляр Favorites является *безопасным с точки зрения типов*: он никогда не вернет Integer, если вы просите его вывести String. Он также является *неоднородным*: все ключи являются ключами разных типов, не как в обычной карте. Следовательно, мы можем назвать Favorites *безопасным неоднородным контейнером*.

Реализация Favorites на удивление крошечная. Вот она вся:

```

// Безопасный шаблон неоднородного контейнера - реализация
public class Favorites {
    private Map<Class<?>, Object> favorites =
        new HashMap<Class<?>, Object>();
    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }
    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}

```

Здесь происходит несколько тонких вещей. Каждый экземпляр Favprites поддерживается закрытым классом Map<Class<?>, Object>, который называется favorites. Вы можете подумать, что здесь невозможно поместить что-либо в Map из-за неограниченного wildcard-типа, но на самом деле все наоборот. Дело в том, что wildcard-тип является вложенным: он не является типом Map, который является wildcard-типом, а является типом его ключа. Это значит, что у каждого ключа могут быть разные типы с параметрами: один может быть Class<String>, другой Class<Integer> и т.д. Вот откуда берется неоднородность.

Также необходимо отметить, что типом значения favoritesMap является просто Object. Другими словами, Map не является гарантом

взаимоотношений типов между ключами и значениями, это означает, что каждое значение относится к типу, представленному его ключом. В действительности система типов Java не имеет достаточно возможностей, чтобы это выразить. Но мы знаем, что это правда, и извлечем из этого преимущество, когда придет время извлечь избранное.

Реализация метода `putFavorite` довольно тривиальна: она просто помещает в `favorites` схему связей из объекта `Class` в экземпляр `favorites`. Как было отмечено, это устраняет «связывание типов» между ключом и значением; утрачивается идея, что значение есть экземпляр ключа. Но это нормально, потому что метод `getFavorites` может и должен восстанавливать это связывание.

Реализация метода `getFavorite` сложнее, чем метода `putFavorites`. Во-первых, она берет из карты `favorites` значение, соответствующее данному объекту `Class`. Это верная ссылка на объект для возврата, но у нее неверный тип во время компиляции. Ее тип — просто `Object` (тип значения карты `favorites`), а нам нужно возвратить `T`. Таким образом, реализация метода `getFavorite` динамически передает ссылку на объект типу, представленному в качестве объекта `Class`, с использованием метода `cast` объекта `Class`.

Метод `Cast` — динамический аналог оператора Java `cast`. Он просто проверяет, что его аргумент является экземпляром типа, представленного объектом `Class`. Если так, он возвращает аргумент; в противном случае он выводит `ClassCastException`, предполагая, что клиентский код компилируется без ошибок. Нужно сказать, что мы знаем, что значения в карте `favorites` всегда совпадают с типами ключей.

Итак, что делает для нас метод `cast`, зная, что он просто возвращает свои аргументы? Подпись метода `cast` максимально использует преимущества того факта, что класс `Class` был обобщен. Его возвращаемый тип — это параметр типа объекта `Class`:

```
public class Class<T> {  
    T cast(Object obj);  
}
```

Это в точности, что нам нужно от метода getFavorites. Это то, что позволит нам сделать безопасным Favorites, не прибегая к непроверенным передачам T.

Имеются два ограничения для класса Favorites, которые стоит упомянуть. Во-первых, злонамеренные клиенты могут легко взломать безопасность типов экземпляра Favorites, просто используя объект Class в его необработанной форме. Но получившийся клиентский код выведет предупреждения о непроверенных передачах при компиляции. Это не отличается от обычной реализации коллекции, такой как HashSet и HashMap. Вы можете легко поместить String в HashSet<Integer>, используя необработанный тип HashSet (статья 23). Как говорилось, у вас может быть безопасность при выполнении, если вы хотите заплатить за нее. Единственный способ убедиться, что Favorites никогда не нарушит инварианты своих типов, — это заставить метод putFavorite проверить, что instance является на самом деле экземпляром типа, представленным объектом type . И мы уже знаем, как это сделать. Просто используя динамическую передачу:

```
//Безопасность типов обеспечивается динамической передачей
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

Имеются упаковщики коллекций в java.util.Collections, которые делают то же самое. Они называются checkedSet, checkedList, checkedMap и т.д. Их методы статической генерации берут объект Class (или два объекта) и дополняют ими коллекцию (или карту). Методы статической генерации являются обобщенными методами, гарантирующими, что типы объекта Class во время компиляции и коллекция согласуются. Упаковщики добавляют материализацию к коллекции, которую они упаковывают. Например, упаковщик выводит ClassException при выполнении, если кто-то пытается добавить Coin в вашу коллекцию Collection<Stamp>. Эти упаковщики полезны для отслеживания, кто добавляет некорректные элементы

в коллекцию в приложении, в котором смешаны обобщенный и уже существующий код.

Второе ограничение класса Favorites заключается в том, что он не может использоваться в нематериальных типах (статья 25). Другими словами, вы можете хранить избранное String или String[], но только не List<String>. Если вы попытаетесь сохранить List<String>, программа не будет компилироваться. Причина этого заключается в том, что вы не можете получить объект Class для List<String>: в List<String>.class, имеется синтаксическая ошибка, но, с другой стороны, это и хорошо. List<String> и List<Integer> используют один общий объект Class, которым является List.class. Он бы натворил немало бед свойствами объекта Class, если бы «литералы типа» List<String>.class и List<Integer>.class были допустимыми и возвращали одну и ту же ссылку на объект.

Нет достаточно удовлетворительного обхода второго ограничения. Есть технология, называемая метка супертипа, которая длинным путем обращается к этому ограничению, однако она сама имеет свои ограничения [Gafter07].

Метки типа, используемые Favorites, являются неограниченными: getFavorite и putFavorite принимают любой объект Class. Иногда вам может потребоваться ограничить типы, которые могут передаваться методу. Этого можно достичь ограниченной меткой типа, которая просто является меткой типа, устанавливающей ограничения на то, какие типы могут быть представлены, используя ограниченный параметр типа (статья 27) или ограниченный групповой символ (статья 28).

Аннотации API (статья 35) расширенно используют ограниченные метки типа. Например, вот метод для чтения аннотаций при выполнении. Этот метод происходит от интерфейса AnnotatedElement, который реализуется рефлексивным типом, представляющим классы, методы, поля и другие программные элементы:

```
public <T extends Annotation>
T getAnnotation(Class<T> annotationType);
```

Аргумент annotationType является ограниченной меткой типа, представляющей тип аннотации. Метод возвращает элементы аннотации данного типа, если таковые имеются, или null, если нет. В сущности, аннотированные элементы являются безопасными неоднородными контейнерами, ключи которых являются типами аннотации.

Предположим, что у вас есть объект типа Class<?> и вы хотите передать его методу, которому требуется ограниченная метка типа, та-кая как getAnnotation. Вы можете передать объект в Class<? Extends Annotation>, но эта передача непроверена, поэтому будет выведено предупреждение во время компиляции (статья 24). К счастью, класс Class дает нам метод экземпляра, который безопасно (и динамично) выполняет данную передачу. Метод называется asSubclass и передает объект Class, на котором он должен представлять подкласс класса, представленного его аргументами. Если передача удастся, то метод возвращает свои аргументы, если нет, то выводится ошибка ClassCastException.

Вот как используют метод asSubclass для чтения аннотаций, типы которых неизвестны во время компиляции. Этот метод компилируется без ошибок и предупреждений:

```
// Use of asSubclass to safely cast to a bounded type token
static Annotation getAnnotation(AnnotatedElement element,
String annotationTypeName) {
    Class<?> annotationType = null; // Unbounded type token
    try {
        annotationType = Class.forName(annotationTypeName);
    } catch (Exception ex) {
        throw new IllegalArgumentException(ex);
    }
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

Подведем итог. Нормальное использование средств обобщенного программирования, представленное коллекцией API, ограни-

чивает вас фиксированным количеством параметров на контейнер. Вы можете обойти это ограничение, поместив параметр в ключ вместо контейнера. Вы можете использовать объекты Class в качестве ключей для таких безопасных неоднородных контейнеров. Объект Class, используемый таким образом, называется меткой типа. Вы можете также использовать обычный тип ключа. Например, вы могли бы заставить тип DatabaseRow представлять строку базу данных (контейнер) и обобщенный тип Column<T> в качестве его ключа.

Г л а в а 6

Перечислимые типы и аннотации

В версии 1.5 были добавлены две группы типов ссылок к языку: новый вид класса, называемый перечислимым типом, и новый вид интерфейса, называемый аннотационным типом. В этой главе обсуждаются наилучшие способы использования этих двух групп.

Статья
30

Используйте перечислимые типы вместо констант int

Перечислимый тип — это тип, разрешенные значения которого состоят из фиксированного набора констант, таких как времена года, планеты Солнечной системы или наборы в колоде карт. До того как были добавлены перечислимые типы, они были представлены декларированием группы именных констант int, один член для каждого типа:

```
// Перечислимый шаблон int – с ужасными недостатками!
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;
```

Технология, известная как перечислимый шаблон `int`, обладает многими недостатками. Она не обеспечивает безопасность типов и не очень удобна. Компилятор не будет жаловаться, если вы передадите яблоко методу, который ожидает апельсин, сравните яблоки с апельсинами оператором `==` или хуже:

```
// Вкусный цитрусовый приправлен яблочным соусом!
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

Обратите внимание, что название каждой константы яблока содержит префикс `APPLE_` и что название каждой константы апельсина содержит `ORANGE_`. Это потому, что Java не предоставляет пространства имен для перечислимых групп `int`. Префиксы препятствуют совпадению имен, если две перечислимые группы `int` будут иметь константы с одинаковыми названиями.

Программы, использующие перечислимый шаблон `int`, довольно «хрупки». Поскольку перечислимые шаблоны `int` являются константами на время компиляции, они компилируются в клиенты, которые их используют. Если `int` связанный с перечислимой константой изменяется, то его клиенты должны быть заново скомпилированы. Если этого не сделать, то запускаться они будут, но их поведение будет непредсказуемо.

Нет легкого способа перевести перечислимые константы в печатаемые строки. Если вы напечатаете такую константу или отобразите ее в отладчике, все, что вы увидите, — это число, что не очень поможет. Нет надежного способа выполнить итерацию перечислимых констант в группе или даже получить размер перечислимой группы `int`.

Вы можете столкнуться с вариантом шаблона, в котором константы `String` используются вместо констант `int`. Этот вариант, известный как перечислимый шаблон `String`, желательно еще меньше

использовать. В то время как он дает печатаемые строки для своих констант, он может привести к проблемам с производительностью, потому что полагается на сравнение строк. Что еще хуже, он может привести неопытных пользователей к использованию строковых констант с тяжелым кодом в клиентском коде, вместо использования наименований полей. Если такие нагруженные кодом строковые константы содержат типографическую ошибку, то она будет пропущена при компиляции и в результате проявится при выполнении.

К счастью, в версии 1.5 язык дает нам альтернативу, которая помогает избежать недостатков перечислимых шаблонов `int` и `string` и предлагает ряд дополнительных преимуществ. Это *перечислимые типы* [JLS, 8.9]. Вот как они выглядят в простейшей форме:

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

При поверхностном взгляде может показаться, что перечислимые типы похожи на свои аналоги в других языках, таких как C, C++, и C#, но это сходство обманчиво. Перечислимые типы Java – это полноценные классы, обладающие большими возможностями, чем их аналоги в других языках, где перечислимые типы являются по сути значениями `int`.

Основная идея перечислимых типов в Java проста: они являются классами, которые экспортируют один экземпляр каждой перечислимой константы, используя открытое статическое завершенное поле. Перечислимые типы являются абсолютно завершенными благодаря тому, что у них нет конструкторов. Клиенты не могут ни создавать экземпляры перечислимых типов, ни расширять их, не создаются никакие другие экземпляры, кроме декларированных перечислимых констант. Другими словами, перечислимые типы подвергаются контролю создания экземпляров. Они являются обобщением синглтона (статья 3), являясь по сути одним перечислимым элементом. Для читателей, знакомых с первой редакцией этой книги, перечислимые типы представляют лингвистическую поддержку *безопасных перечислимых шаблонов* [Bloch01, статья 21].

Перечислимые типы обеспечивают безопасность типов при компиляции. Если вы объявили, что параметр должен относиться к типу Apple, то у вас есть гарантия, что любая ненулевая ссылка на объект, переданная параметру, является одним из трех разрешенных значений Apple. Попытка передать значение неверного типа приведет к ошибке при компиляции, так же как и попытки присвоить выражение одного перечислимого типа переменной другого или использовать оператор `==` для сравнения значений различных перечислимых типов.

Перечислимые типы с одинаковыми наименованиями констант существуют «мирно» потому, что у каждого типа имеется свое пространство имен. Вы можете добавлять или менять порядок констант в перечислимом типе без необходимости повторно компилировать клиентов, потому что поля, экспортирующие константы, предоставляют защитный слой между перечислимым типом и его клиентами: значения констант не компилируются в клиентов, как в случае с перечислимым шаблоном `int`. Наконец, вы можете перевести перечислимые типы в печатаемые строки вызовом метода `toString`.

В дополнение к тому, что перечислимые типы позволяют нам избавиться от недостатков перечислимых шаблонов `int`, они позволяют нам добавлять произвольные методы и поля, а также реализовывать произвольные интерфейсы. Они предлагают высококачественные реализации всех методов `Object` (глава 11), а их сериализованная форма сделана так, чтобы перенести большую часть изменений в перечислимых типах.

Так зачем нам добавлять методы или поля к перечислимым типам? Например, вы можете захотеть ассоциировать данные с их константами. Наши типы `Apple` или `Orange`, например, могут извлечь выгоду из метода, который вернет цвет фрукта или его изображение. Вы можете присвоить аргументы перечислимому типу с помощью любого подходящего метода. Перечислимый тип может начать жизнь, как простая коллекция перечислимых констант и развивается со временем в полноценную абстракцию.

В качестве примера перечислимого типа с широкими возможностями рассмотрим восемь планет нашей Солнечной системы. У каждой планеты есть масса и радиус, и из этих двух атрибутов вы можете рассчитать их гравитацию на поверхности. Это, в свою очередь, позволит вам рассчитать вес объекта на поверхности планеты, зная массу объекта. Вот как выглядит этот перечислимый тип. Числа в скобках после каждой перечислимой константы — параметры, которые передаются конструктору. В данном случае это масса и радиус планет:

```
// Перечислимый тип с данными и отношением
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; // In kilograms
    private final double radius; // In meters
    private final double surfaceGravity; // In m / s^2
    // Universal gravitational constant in m^3 / kg s^2
    private static final double G = 6.67300E-11;
    // Constructor
    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 перечислимый тип с большими возможностями, такой как Planet. **Для ассоциации данных с перечислимыми константами объявите поля экземпляра и напишите конструктор, который возьмет данные и сохранит их в поле.** Перечислимые типы по своей природе неизменяемы, так что все поля должны быть завершенными (статья 15). Они могут быть открытыми, но лучше сделать их закрытыми и снабдить открытыми средствами доступа (статья 14). В случае с Planet конструктор также рассчитывает и сохраняет значение гравитации у поверхности, но это лишь оптимизация. Гравитация может быть заново рассчитана на основе массы и радиуса каждый раз, когда используется метод surfaceWeight, который берет массу объекта и возвращает его вес на планете, представленной константой.

В то время как перечислимый тип Planet является простым, он обладает удивительными возможностями. Вот короткая программа, которая берет земной вес объекта (в любой единице) и печатает таблицу весов объектов на всех восьми планетах (в одних и 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("Weight on %s is %f%n",  
                p, p.surfaceWeight(mass));  
    }  
}
```

Обратите внимание, что у Planet, как и у всех перечислимых типов, есть статический метод values, который возвращает массив его значений в той последовательности, в какой они были объявлены. Также заметьте, что метод toString возвращает декларированные названия каждого значения перечислимого типа, давая возможность их легко распечатать методами println и printf. Если вас не устра-

ивает представление данной строки, вы можете изменить его методом `toString`. Вот результат запуска нашей небольшой программы `WeightTable` с аргументом командной строки 175:

```
Weight on MERCURY is 66.133672
Weight on VENUS is 158.383926
Weight on EARTH is 175.000000
Weight on MARS is 66.430699
Weight on JUPITER is 442.693902
Weight on SATURN is 186.464970
Weight on URANUS is 158.349709
Weight on NEPTUNE is 198.846116
```

Если вы впервые видите, как действует метод `printf` в Java, обратите внимание, что он отличается от аналога в С в том, что использует `%p` там, где в С используют `\n`.

Некоторые особенности, связанные с перечислимыми константами, возможно необходимо использовать только в рамках класса или пакета, в котором определен перечислимый тип. Такие особенности лучше всего реализованы как закрытые или открытые в рамках пакета методы. Каждая константа содержит скрытую коллекцию, которая позволяет классу или пакету, содержащему перечислимый тип, должным образом реагировать, когда появляется константа. Как и с другими классами, если, конечно, у вас нет причины открывать перечислимый метод, объягните его как закрытый или, если необходимо, открытый только в рамках пакета (статья 13).

Если перечислимый тип полезен, то он должен быть классом высшего уровня; если его использование привязано к классу высшего уровня, то он должен быть классом-членом класса высшего уровня (статья 22). Например, перечислимый тип `java.math.RoundingMode` представляет режим округления десятичной дроби. Эти режимы округления используются классом `BigDecimal` и дают полезную абстракцию, которая фундаментально не привязана к `BigDecimal`. Если сделать `RoundingMode` перечислимым типом верхнего уровня, разработчики библиотеки будут заставлять про-

граммистов, которым потребуется режим округления, повторно использовать данный перечислимый тип, что приведет к улучшению логичности во всех API.

Технология, показанная в примере с Planet, достаточна для применения в большем количестве перечислимых типов, но иногда вам требуется большего. С каждой константой Planet связаны различные данные, но иногда требуется связать фундаментально различные реакции с каждой константой. Например, предположим, вы пишете перечислимый тип, представляющий операции на основе калькулятора четырех функций, и вы хотите снабдить его методом для выполнения арифметических операций, представленным каждой коллекцией. Одним из способов достижения этого является переключение на значение перечислимого типа:

```
// Перечислимый тип, переключающийся на свое собственное
// значение, - спорно
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;
    // Do the arithmetic op represented by this constant
    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("Unknown op: " + this);
    }
}
```

Этот код работает, но он не очень хорош. Он не будет компилироваться без выражения throw, потому что конец метода технически может быть достигнут, хотя на самом деле он никогда не будет достигнут [JLS, 14.21]. Что еще хуже, код довольно «хрупкий». Если вы добавите новую перечислимую константу, но забудете добавить соответствующий регистр к switch, перечислимый тип все равно от-

компилируется, но даст ошибку при выполнении, когда вы попытаетесь выполнить новую операцию.

К счастью, есть более удачный способ ассоциации различных реакций с каждой перечислимой константой: объягите абстрактный метод `apply` в перечислимом типе и переопределите его конкретным методом для каждой константы в теле класса, *зависимого от константы*. Такие методы известны как *реализации, зависящие от константы*:

```
// Перечислимый тип с реализацией метода в зависимости от константы
public enum Operation {
    PLUS { double apply(double x, double y){return x + y; } },
    MINUS { double apply(double x, double y){return x - y; } },
    TIMES { double apply(double x, double y){return x * y; } },
    DIVIDE { double apply(double x, double y){return x / y; } };
    abstract double apply(double x, double y);
}
```

Если вы добавите новую константу ко второй версии `Operation`, то вряд ли вы забудете применить метод `apply`, так как метод немедленно следует за любой декларацией константы. В случае, если все же забудете, компилятор вам об этом напомнит, поскольку абстрактные методы в перечислимом типе должны быть переопределены в конкретные методы во всех их константах.

Реализация методов, привязанная к константам, может сочетаться с привязанным к константам данным. Например, вот версия `Operation`, которая переопределяет `toString` метод для возвращения символов, обычно связанных с операцией:

```
// Перечислимые типы с классами и данными, связанными
// с константами
public enum Operation {
    PLUS["+"] {
        double apply(double x, double y) { return x + y; }
    },
    MINUS["-"] {
```

```
        double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        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; }
    abstract double apply(double x, double y);
}
```

В некоторых случаях переопределение `toString` в перечислимом типе полезно. Например, реализация `toString`, описанная выше, облегчает задачу печати арифметического выражения, как показано в этой небольшой 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));
}
```

Запуск этой программы с аргументами командной строки 2 и 4 выводит на экран следующее:

```
2.000000 + 4.000000 = 6.000000
2.000000 - 4.000000 = -2.000000
2.000000 * 4.000000 = 8.000000
2.000000 / 4.000000 = 0.500000
```

У перечислимых типов есть автоматически генерируемые методы `valueOf(String)`, которые переводят названия констант в сами константы. Если вы переопределите метод `toString` в перечислимом

типе, рассмотрите возможность написать метод `fromString` для перевода обычных строковых представлений обратно в соответствующие перечислимые типы. Следующий код (с измененнымиенным образом именами типов) будет выполнять эту операцию для любого перечислимого типа до тех пор, пока у каждой константы есть уникальное строковое представление:

```
// Реализация метода fromString на перечислимом типе
private static final Map<String, Operation> stringToEnum
= new HashMap<String, Operation>();
static { // Initialize map from constant name to enum constant
    for (Operation op : values())
        stringToEnum.put(op.toString(), op);
}
// Returns Operation for string, or null if string is invalid
public static Operation fromString(String symbol) {
    return stringToEnum.get(symbol);
}
```

Обратите внимание, что константа `Operation` помещается в схему `stringToEnum` из статического блока, который запускается после создания константы. Если попытаться заставить каждую константу помещать себя в эту схему из своего собственного конструктора, то это вызовет ошибку компиляции. Это хорошо, поскольку если бы такое было разрешено, то приводило бы к ошибке при выполнении `NullPointerException`. Конструкторам перечислимых типов не разрешен доступ к статическим полям перечислимых типов, кроме полей констант на время компиляции. Это ограничение необходимо, потому что эти статические поля еще не инициализированы на момент запуска конструктора.

Недостаток реализации метода, зависимого от константы, заключается в том, что становится сложнее использовать общий код для разных перечислимых констант. Например, рассмотрим перечислимый тип, представляющий дни недели в пакете, рассчитывающем заработную плату. У этого перечислимого типа есть метод, рассчиты-

вающий оплату рабочего на данный день, зная ставку зарплаты рабочего (в час) и количество рабочих часов в тот день. При пятидневной неделе любое время сверх обычной смены должно генерировать повышенную оплату. С выражением `switch` легко выполнить расчет, применяя различные метки для различных случаев двух фрагментов кода. Для краткости код в данном примере использует `double`, но обратите внимание, что `double` не является подходящим типом данных для приложений расчета оплаты труда (статья 48):

```
// Перечислимый тип, переключающийся на свое значение, чтобы
// сделать код общим, - спорно
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
    private static final int HOURS_PER_SHIFT = 8;
    double pay(double hoursWorked, double payRate) {
        double basePay = hoursWorked * payRate;
        double overtimePay; // Рассчитывает переработку
        switch(this) {
            case SATURDAY: case SUNDAY:
                overtimePay = hoursWorked * payRate / 2;
                break;
            default: // Weekdays
                overtimePay = hoursWorked <= HOURS_PER_SHIFT ?
                    0 : (hoursWorked - HOURS_PER_SHIFT) * payRate / 2;
        }
        return basePay + overtimePay;
    }
}
```

Этот код, несомненно, краток, но его опасно использовать с точки зрения поддержки. Предположим, вы добавите элемент к перечислимому типу, возможно особое значение для представления дня отпуска, но забудете добавить соответствующий регистр к выражению `switch`. Программа будет все равно компилироваться, но метод `Pay` будет начислять рабочему оплату за день отпуска так же, как если бы он работал.

Чтобы безопасно выполнять расчеты оплаты с помощью реализации зависимого от констант метода, вам нужно дублировать расчет оплаты переработки для каждой константы или переместить расчет в два вспомогательных метода (один для рабочих дней, другой для выходных) и запустить подходящий вспомогательный метод для каждой константы. Любой подход приведет к большому объему кода, существенно уменьшая его читаемость и увеличивая возможность ошибки.

Код можно уменьшить, заменив абстрактный метод `overtimePay` на `PayrollDay` конкретным методом, который выполняет расчет переработки в выходные дни. Но это приведет к тому же недостатку, что и выражение `switch`: если вы добавите еще день без переопределения метода `overtimePay`, то расчет будет вестись неправильно, как оплата в рабочие дни.

Что вам действительно нужно — это необходимость выбирать стратегию оплаты переработки каждый раз при добавлении перечислимой константы. К счастью, есть способ, как этого достичь. Суть в том, чтобы переместить расчет оплаты переработки в закрытый вложенный перечислимый тип и передать экземпляр этого *стратегического перечислимого типа* в конструктор перечислимому типу `PayrollDay`. Перечислимый тип `PayrollDay` затем передаст расчет оплаты переработки стратегическому перечислимому типу, избегая необходимости для выражения `switch` или использования реализации зависимого от констант метода в `PayrollDay`. Хотя этот шаблон менее краток, чем выражение `switch`, он безопаснее и более гибок:

```
// Стратегический перечислимый шаблон
enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),
    WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),
    FRIDAY(PayType.WEEKDAY),
    SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);
    private final PayType payType;
    PayrollDay(PayType payType) { this.payType = payType; }
    double pay(double hoursWorked, double payRate) {
```

```
        return payType.pay(hoursWorked, payRate);
    }
    // Стратегический перечислимый тип
    private enum PayType {
        WEEKDAY {
            double overtimePay(double hours, double payRate) {
                return hours <= HOURS_PER_SHIFT ? 0 :
                    (hours - HOURS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            double overtimePay(double hours, double payRate) {
                return hours * payRate / 2;
            }
        };
        private static final int HOURS_PER_SHIFT = 8;
        abstract double overtimePay(double hrs, double payRate);
        double pay(double hoursWorked, double payRate) {
            double basePay = hoursWorked * payRate;
            return basePay + overtimePay(hoursWorked, payRate);
        }
    }
}
```

Если выражения `switch` перечислимых типов не являются хорошим выбором для реализации реакций в зависимости от констант, чем же тогда они хороши? **Переключение на перечислимых типах хорошо подходит для задания аргументов внешних перечислимых типов с помощью зависимых от констант реакций.** Например, предположим, что перечислимый тип `Operation` еще не находится под вашим контролем и вы хотели бы, чтобы у него был метод экземпляров для возврата противоположности каждой операции. Вы можете смоделировать данный эффект следующим статическим методом:

```
// Переключение на перечислимом типе для имитации недостающего 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("Unknown op: " + op);  
}  
}
```

В целом перечислимые типы можно сравнить с константами `int`. Небольшой недостаток в производительности перечислимых типов по сравнению с константами `int` заключается в том, что требуется пространство и время для загрузки и инициализации перечислимых типов. Кроме как на устройствах с ограниченной производительностью, таких как сотовый телефон или тостер, такой недостаток вряд ли будет заметен вообще.

Так когда же нам следует использовать перечислимые типы? В любом случае, когда вам требуется фиксированный набор констант. Конечно, это включает «натуральные перечислимые типы», такие как планеты, дни недели и шахматные фигуры. Но это также включает в себя другие наборы, для которых вы знаете все возможные значения во время компиляции, такие как выбор пунктов меню, кодов операций и метки командной строки. Нет необходимости, чтобы набор констант в перечислимом типе оставался неизменным все время. Специально была добавлена возможность, чтобы позволить развиваться перечислимым типам в рамках двоичной совместимости.

Подведем итоги. Преимущества перечислимых типов по сравнению с константами `int` сложно переоценить. Перечислимые типы хорошо читаемы, безопасны и имеют больше возможностей. Многим перечислимым типам требуются явные конструкторы и члены, но многие другие извлекают пользу от ассоциации данных с каждой из констант и предоставляя методы, на поведение которых влияют данные. Намного меньше пользы перечислимые типы извлекают от ассоциации нескольких реакций с одним мето-

дом. В этом относительно редком случае предпочтайте перечислимым типам, которые переключаются на собственные значения, методы, зависящие от констант. Рассмотрите стратегию перечислимых шаблонов, если у нескольких перечислимых констант есть общие реакции.

Статья 31

Используйте поля экземпляра вместо числовых значений

Многие перечислимые типы по природе своей ассоциируются с одним значением `int`. У всех типов есть метод `ordinal`, который возвращает расположение по порядку каждой перечислимой константы своего типа. Вы можете столкнуться с искушением вывести ассоциированное значение `int` из порядкового расположения:

```
// Неверное использование порядкового расположения для вывода
// ассоциированного 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 и работает, его поддержка может стать кошмаром. Если поменять порядок констант, метод `numberOfMusicians` будет сломан. Если вы захотите добавить вторую перечислимую константу, связанную со значением `int`, которую вы уже использовали, то вам снова не повезет. Например, может быть прекрасно, если вы захотите добавить константу, представляющую тройной квартер, состоящий из 12 музыкантов. Нет стандартного термина для ансамбля, состоящего из 11 музыкантов, так что вам придется добавить константу для неиспользованного значения `int` (11). В лучшем случае это ужасно выглядит. Если много значений `int` не использованы, то это непрактично.

К счастью, есть простое решение этих проблем. Никогда не выводите значение, связанное с перечислимыми типами, из их порядкового положения; вместо этого храните их 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 к Enum говорит об ordinal: «Большинство программистов вынуждены использовать этот метод. Он создан для использования только структурами данных, на основе перечислимых типов общего назначения, таких как EnumSet и EnumMap». Если только вы не пишете такую структуру данных, лучше всего избегать использования метода ordinal вообще.

*Статья
32*

Используйте EnumSet вместо битовых полей

Если элементы перечислимого типа используются преимущественно в наборах, то традиционно используется перечислимый шаблон enum (статья 30), дающий различные возможности 2 каждой константе:

```
// Перечислимые константы 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
    // Parameter is bitwise OR of zero or more STYLE_ constants
```

```
public void applyStyles(int styles) { ... }  
}
```

Это представление позволяет вам использовать побитовую операцию OR для объединения нескольких констант в набор, известный как *битовое поле*:

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

Представления битовых полей позволяют эффективно выполнять встроенные операции, такие как объединение и пересечение, используя побитовую арифметику. Но у битового поля есть все недостатки перечислимых констант int, и даже больше. Битовое поле труднее интерпретировать, чем простую перечислимую константу int, если она распечатана в виде числа. Также нет простого способа итерации для всех элементов, представленных битовым полем.

Некоторые программисты, предпочитающие использовать перечислимые типы вместо констант int, все еще используют битовые поля, если им необходимо обойти наборы констант. Нет нужды так делать — есть лучшая альтернатива. Пакет java.utils дает нам класс EnumSet для эффективного представления набора значений, извлеченных из одного перечислимого типа. Этот класс реализует интерфейс Set, давая вам все богатство, безопасность типов и возможность взаимодействия, которые вы можете получить при любой другой реализации Set. Но внутренне каждый EnumSet представлен как битовый вектор. Если у основного перечислимого типа 64 или менее элементов — у большинства так, — то весь EnumSet представлен одним long, так что его производительность сравнима с производительностью битового поля. Групповые операции, такие как RemoveAll и RetainAll, реализуются с использованием побитовой арифметики, так как если бы вы делали это вручную для битовых полей. Но вы теперь избавлены от сложности и ошибок ручной манипуляции с битами: EnumSet выполняет эту сложную работу для вас.

Вот как выглядит предыдущий пример, если его изменить для использования перечислимых типов вместо битовых полей. Он короче, яснее и безопаснее:

```
// EnumSet - современная замена битовым полям
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }
    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

Это клиентский код, который передает экземпляр EnumSet методу applyStyles. EnumSet дает богатый набор методов статической генерации для упрощения создания наборов, один из которых приведен в данном коде:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Обратите внимание, что метод applyStyles отдает предпочтение Set<Style>, а не EnumSet<Style>. Хотя кажется, что все клиенты должны передавать методу EnumSet, хорошей практикой является принятие типа интерфейса вместо типа реализации. Это дает возможность необычному клиенту передать в какой-либо другой реализации Set и не обладает какими-либо недостатками.

Подведем итоги. Поскольку перечислимые типы будут использоваться в наборах, нет необходимости представлять их битовыми полями. Класс EnumSet объединяет в себе краткость и производительность битовых полей со всеми преимуществами перечислимых типов, описанных в статье 30. Есть один недостаток EnumSet — невозможность создать неизменяемый EnumSet (по крайней мере, в версии 1.6), но скорее всего он будет исправлен в будущих версиях. А пока вы можете поместить EnumSet в оболочку, с помощью Collections.unmodifiableSet, но при этом и краткость и производительность пострадают.

*Статья
33*

Используйте EnumMap вместо порядкового индексирования

Вам может попасться код, использующий метод ordinal (статья 31) для индексирования внутри массива. Например, рассмо-

тим этот простой класс, который должен представлять кулинарную траву:

```
class Herb {  
    enum Type { ANNUAL, PERENNIAL, BIENNIAL }  
    final String name;  
    final Type type;  
    Herb(String name, Type type) {  
        this.name = name;  
        this.type = type;  
    }  
    @Override public String toString() {  
        return name;  
    }  
}
```

Теперь предположим, что у вас есть массив трав, представляющий растения в саду, и вы хотите составить список этих растений, организованный по типу (однолетние, двухлетние или многолетние). Для того чтобы это сделать, вы создаете три набора, по одному для каждого типа, и выполняете итерацию по саду, помещая каждую траву в соответствующий набор. Некоторые программисты сделают это помещая наборы в массив, индексированный по типу:

```
// Using Используется ordinal() для индексирования массива -  
// НЕ ДЕЛАЙТЕ ТАК!  
Herb[] garden = ... ;  
Set<Herb>[] herbsByType = // Indexed by Herb.Type.ordinal()  
(Set<Herb>[]) new Set[Herb.Type.values().length];  
for (int i = 0; i < herbsByType.length; i++)  
    herbsByType[i] = new HashSet<Herb>();  
for (Herb h : garden)  
    herbsByType[h.type.ordinal()].add(h);  
// Выводит результаты  
for (int i = 0; i < herbsByType.length; i++) {  
    System.out.printf("%s: %s%n",  
        Herb.Type.values()[i], herbsByType[i]);  
}
```

Данный прием работает, но в нем много проблем. Так как массив несовместим с средствами обобщенного программирования (статья 25), программе потребуется необработанная передача и она не будет чисто компилироваться. Поскольку массиву неизвестно, что представляет собой его индекс, то необходимо пометить результат вручную. Но самая серьезная проблема — когда вы получите доступ к массиву, индексированному порядковым перечислимым типом, то на вас ляжет ответственность использования правильного значения int; int не обеспечивает безопасность перечислимых типов. Если вы используете неверное значение, программа будет тихо делать не то, что нужно, или — если вам повезет — выдаст ошибку `ArrayIndexOutOfBoundsException`.

К счастью, есть намного более удачный способ добиться того же эффекта. Массив эффективно выполняет роль карты между перечислимым типом и значением, так что можно использовать Map. Заметим, что есть быстрая реализация Map, созданная для использования с перечислимыми ключами, известная как `java.util.EnumMap`. Вот как выглядит программа, переписанная для использования `EnumMap`:

```
// Использование EnumMap для связи данных с перечислимым типом
Map<Herb.Type, Set<Herb>> herbsByType =
    new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);
for (Herb.Type t : Herb.Type.values())
    herbsByType.put(t, new HashSet<Herb>());
for (Herb h : garden)
    herbsByType.get(h.type).add(h);
System.out.println(herbsByType);
```

Эта программа короче, чище, безопаснее и по скорости сравнима с оригинальной версией. Здесь нет небезопасных передач, нет необходимости вручную помечать результат, поскольку ключи карты — это перечислимые типы, которые знают, как перевести самих себя в печатаемые строки, и нет вероятности ошибок в расчете индексов массивов. Причина, почему `EnumMap` сравним по скорости с массивом,

индексированным по порядку, заключается в том, что EnumMap использует этот 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 берет объект Class ключевого типа: это маркер связанного типа, который обеспечивает информацию о выполнении обобщенных типов (статья 29).

Вы можете увидеть массив индексов массива (дважды) по порядку, используемый для представления карты из двух значений перечислимых типов. Например, эта программа использует такой массив для связывания двух фаз с переходом в фазу (жидкое состояние в твердое называется замерзанием, жидкое в газообразное — кипением и т.д.):

```
// Использует ordinal() для индексирования массива в массив -
// НЕ ДЕЛАЙТЕ ТАК!
public enum Phase { SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Строки индексируются с помощью src-ordinal, столбцы
        // с помощью dst-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null, MELT, SUBLIME },
            { FREEZE, null, BOIL },
            { DEPOSIT, CONDENSE, null }
        };
        // Возвращает переход из одного состояния в другое
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

Эта программа работает и может выглядеть элегантно, но это впечатление обманчиво. Как и в более простом примере с травами в саду, компилятор не знает отношений между порядковыми номе-

рами и индексами массива. Если вы сделаете ошибку в таблице перехода или забудете обновить ее при изменении перечислимых типов Phase или Phase.Transition, то ваша программа даст ошибку при выполнении. Ошибка может быть в виде ArrayOutOfBoundsException, NullPointerException или, что еще хуже, будет тихо работать некорректно. И размер таблицы будет квадратичен количеству фаз, даже если количество ненулевых входных значений будет меньше.

И опять же, вы можете сделать все гораздо лучше с помощью EnumMap. Поскольку каждый переход из одной фазы в другую индексируется парой перечислимых типов фаз, то лучше всего представить отношения в виде карты от одного перечислимого типа (исходная фаза) к карте другого (фаза, в которую осуществляется переход) и к результату (сам переход). Две фазы, связанные с переходом, лучше всего записывать, связывая данные с перечислимыми типами перехода из фазы в фазу, который затем используется для инициализации вложенного класса EnumMap:

```
// Использование вложенного класса EnumMap для ассоциации данных
// с парами перечислимых типов
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 src;
private final Phase dst;
Transition(Phase src, Phase dst) {
    this.src = src;
    this.dst = dst;
}
// Инициализирует карту перехода из фазы в фазу
private static final Map<Phase, Map<Phase, Transition>> m =
    new EnumMap<Phase, Map<Phase, Transition>>(Phase.class);
static {
```

```
        for (Phase p : Phase.values())
            m.put(p, new EnumMap<Phase, Transition>(Phase.class));
        for (Transition trans : Transition.values())
            m.get(trans.src).put(trans.dst, trans);
    }
    public static Transition from(Phase src, Phase dst) {
        return m.get(src).get(dst);
    }
}
```

Код для инициализации фазы перехода может выглядеть сложнее, но он не так уж плох. Типами карты являются `Map<Phase, Map<Phase, Transition>>`, что означает «карта от исходной фазы к фазе назначения и переходу». Первый цикл в блоке статической инициализации инициализирует наружную карту, содержащую три пустые внутренние карты. Второй цикл в блоке инициализирует внутренние карты, используя информацию об источнике и назначении, котороедается с помощью константы перехода состояния.

Теперь предположим, что вы хотите добавить новую фазу в систему: плазму или ионизированный газ. Есть только два перехода, связанные с этой фазой: ионизация, которая преобразует газ в плазму, и деионизация, которая обратно возвращает плазму в газ. Для обновления программы на основе массива вам придется добавить одну новую константу к `Phase` или две к `Phase.Transition` и заменить оригинальный массив из массива из девяти элементов на новую версию из шестнадцати элементов. Если вы добавите слишком много или слишком мало элементов в массив или поместите элементы не в том порядке, то вам не повезет: программа откомпилируется, но выведет ошибку при выполнении. Для обновления версии на основе `EnumMap` все, что вам надо сделать, — это добавить список фаз `PLASMA` а также `IONIZE (GAS, PLASMA)` и `DEIONIZE (PLASMA, GAS)` к списку переходов. Программа обо всем остальном сама позаботится, и у вас не будет возможности сделать ошибку. Внутренне карта карт реализуется как массив массивов, так что вы не много теряете в плане затрат и вре-

мени выполнения, взамен получая большую ясность, безопасность и простоту поддержки.

Подведем итоги. Редко бывает ситуация, когда для индексирования массивов используются порядковые номера: вместо них надо использовать EnumMap. Если взаимоотношения, которые вы хотите представить, находятся во многих измерениях, используйте EnumMap<..., EnumMap<...>>. Это особый случай общего принципа, в котором программисты приложений должны использовать Enum.ordinal (статья 31).

Статья
34

Имитируйте расширяемые перечислимые типы с помощью интерфейсов

Во всех отношениях перечислимые типы превосходят безопасные перечислимые шаблоны, описанные в первой редакции этой книги [Bloch01]. Однако при всем этом имеется одно исключение, оно касается расширяемости, которая была возможна в оригинальном шаблоне, но не поддерживается конструкцией языка. Другими словами, используя шаблон, можно было сделать так, чтобы один перечислимый тип расширял другой, используя те же средства языка. Это не случайно. По большей части расширение перечислимых типов оказывается плохой идеей. Становится запутанным то, что элементы расширяемого типа являются экземплярами основного типа, а не наоборот. Это не очень хорошая идея — перечислять все элементы основного типа и его расширения. Наконец, расширяемость усложнит многие аспекты дизайна и реализации.

Ранее отмечалось, что есть случай, когда следует использовать расширяемые перечислимые типы, называемые операционными кодами. Операционный код — это перечислимый тип, элементы которого представляют операции на некой «машине» типа Operation в статье 30, который представляет функции простого калькулятора. Иногда желательно позволить пользователям API предоставлять

свои собственные операции, эффективно расширяя набор операций, предоставляемых API.

К счастью, есть прекрасный способ достичь такого эффекта с использованием перечислимых типов. Основная идея — это извлечь преимущества из факта, что перечислимые типы могут реализовывать произвольные интерфейсы, определяя интерфейс для типа операционных кодов и перечислимого типа, являющегося стандартной реализацией интерфейса. Например, вот пример расширяемой версии типа Operation статьи 30:

```
// Имитированный расширяемый перечислимый тип 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 перечислимый тип (`BasicOperation`) и не является расширяемым, тип интерфейса (`Operation`) расширяем, и он является типом интерфейса, который используется для представления операций в API. Вы можете определить другой перечислимый тип, который реализует этот интерфейс, и использовать экземпляры этого нового типа вместо основного типа. Например, предположим, вы хотите определить расширение для операционного типа, о котором шла речь выше, состоящим из возведения в степень и остальных операций. Все, что вам нужно сделать, — это написать перечислимый тип для реализации интерфейса `Operation`:

```
// Имитированный перечислимый тип расширения extension
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;
    }
}
```

Вы можете использовать новые операции там же, где вы могли бы использовать основные операции, при условии, что API написаны так, чтобы они могли принимать тип интерфейса (`Operation`), а не реализацию (`BasicOperation`). Обратите внимание, что вам не нужно

объявлять абстрактный метод `apply` в перечислимом типе, как вы бы делали это в нерасширяемом перечислимом типе с реализацией метода, специфичного для экземпляра. Это из-за того, что абстрактный метод (`apply`) является членом интерфейса (`Operation`).

Есть возможность не только передать один экземпляр расширяемого перечислимого типа везде, где ожидается основной перечислимый тип, — можно передать все расширение перечислимого типа и использовать его элементы в дополнение или вместо элементов основного типа. Например, вот версия программы, которая выполняет все операции расширения, определенные выше:

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation>
void test(Class<T> opSet, double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
            x, op, y, op.apply(x, y));
}
```

Обратите внимание, что класс литерал для расширяемого операционного типа (`ExtendedOperation.class`) передается от `main` в `test` для описания набора расширяемых операций. Класс литерал служит в качестве маркера связанных типов (статья 29). Довольно сложная декларация параметра `opSet` (`<T extends Enum<T> & Operation>Class<T>`) гарантирует, что объект `Class` представляет и перечислимый тип и подтип, принадлежащий к `Operation`, который является в точности тем, что требуется для итерации элементов и выполнения операций, связанных с каждым из них.

Второй альтернативой будет использование `Collection<? extends Operation>`, который является связанным типом группового символа (статья 28), в качестве типа для параметра `opSet`:

```

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}
private static void test(Collection<? extends Operation> opSet,
double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
x, op, y, op.apply(x, y));
}

```

Получившийся код менее сложен, и метод `test` более гибкий: он позволяет вызывающему объединить операции из нескольких типов реализации. С другой стороны, вы откажетесь от использования `EnumSet` (статья 32) и `EnumMap` (статья 33) для определенной операции, так что вам лучше воспользоваться маркером связанного типа, если только вам не требуется гибкость для объединения операций в нескольких типах реализаций.

Обе вышеупомянутые программы произведут этот результат, если их запускать с аргументами командной строки 4 и 2:

```

4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000

```

Основной недостаток использования интерфейсов для имитации расширяемых перечислимых типов заключается в том, что реализации не могут быть унаследованы от одного перечислимого типа другим. В случае с примером `outOperation` логика сохранения и извлечения символов, связанных с операцией, дублируется в `BasicOperation` и `ExtendedOperation`. В этом случае это важно, так как очень небольшое количество кода дублируется. Если бы был большой объем общего функционала, вы могли бы инкапсулировать его во вспомогательный класс или статический вспомогательный метод для избежания дублирования кода.

Подведем итог. **Хотя вы не можете написать расширяемый перечислимый тип, вы можете имитировать его, написав интерфейс с основным перечислимым типом, который реализует интерфейс.** Это позволяет клиентам писать свои собственные перечислимые типы, реализующие интерфейс. Эти перечислимые типы могут потом использоваться везде, где основной перечислимый тип может быть использован при условии, что API написаны на условиях интерфейса.

Страница
35

Предпочитайте аннотации шаблонам присвоения имен

До версии 1.5 было общепринятым использование именования шаблонов (*naming pattern*) для определения, что некоторые элементы программы требуют специального подхода инструментом или структурой. Например, среда тестирования JUnit изначально требовала от пользователей назначать методы тестирования, начиная их наименования со слова `test` [Beck04]. Этот прием работает, но у него есть несколько недостатков. Во-первых, типографические ошибки могут привести к невидимым ошибкам. Например, предположим, что вы случайно назвали метод тестирования `tsetSafetyOverride` вместо `testSafetyOverride`. JUnit не сообщит вам об ошибке, но и не будет выполнять тестирование, что приведет к ложному чувству безопасности.

Второй недостаток именования шаблонов заключается в том, что нет способа гарантировать, что они используются только на тех элементах программы, на которых нужно. Например, предположим, вы вызываете класс `testSafetyMechanisms`, надеясь, что JUnit автоматически проверит все свои методы, вне зависимости от их названий. И снова JUnit не сообщит об ошибке, но и не выполнит тест.

Их третий недостаток заключается в том, что они не дают хорошего способа ассоциировать значения параметров с элементами программы. Например, предположим, вы хотите выполнить такой вид

теста, который завершится успешно только тогда, когда он выведет определенное сообщение об исключении. Тип исключения является по сути параметром теста. Вы можете закодировать название исключения типа в названии метода тестирования, используя готовые шаблоны, но это будет ненадежно и выглядеть ужасно (статья 50). Компилятор не будет знать, что ему надо проверить, что строка, которая должна присвоить название исключению действительно это сделала. Если именованный класс не существует или не был исключением, то вы никогда этого не узнаете, пока не выполните тест.

Аннотации [JLS 9.7] решают все эти проблемы. Предположим, вы хотите определить тип аннотации для назначения простых тестов, которые запускаются автоматически и завершаются с ошибкой, если появляется исключение. Вот как такой аннотационный тип с называнием `Test` выглядит:

```
// Marker annotation type declaration
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }
```

Декларация для типа аннотации `Test` сама по себе является аннотированной аннотациями `Retention` и `Target`. Такие аннотации на декларации типа аннотации называются мета-аннотации. Мета-аннотация `@Retention(RetentionPolicy.RUNTIME)` означает, что аннотация `Test` будет задержана при выполнении. Без этого аннотация `Test` была бы невидима инструменту тестирования. Мета-аннотация `@Target(ElementType.METHOD)` обозначает, что аннотация `Test` разрешена только на декларациях методов: она не может применяться при объявлении классов, полей и других элементов.

Обратите внимание на комментарий при декларации аннотации `Test`, который говорит «Используйте только статические методы без параметров». Было бы хорошо, если бы компилятор мог наложить это ограничение, но он не может. Есть определенные ограничения на количество проверок на ошибки, которые компилятор может выполнить для вас даже с аннотациями. Если вы поместите аннотацию `Test` на декларацию метода экземпляра или метода с одним или более параметров, программа тестирования все равно откомпилируется, предоставив инструменту тестирования разбираться с проблемами при выполнении.

Вот как аннотация `Test` будет выглядеть на практике. Она называется аннотацией с маркерами, потому что у нее нет параметров, а только маркеры аннотированных элементов. Если программист сделает ошибку при написании названия `Test` или применит аннотацию `Test` на программном элементе, отличном от декларации метода, то программа компилироваться не будет:

```
// Программа, содержащая аннотации с маркерами
public class Sample {
    @Test public static void m1() { } // Test should pass
    public static void m2() { }
    @Test public static void m3() { // Test should fail
        throw new RuntimeException("Boom");
    }
    public static void m4() { }
    @Test public void m5() { } // INVALID USE: nonstatic method
    public static void m6() { }
    @Test public static void m7() { // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

У класса `Sample` имеется семь статических методов, четыре из которых аннотируются как наборы. Два из них, `m3` и `m7`, выводят исключения, и два, `m1` и `m5`, не выводят. Но один из аннотируемых

методов, который не выводит исключения, т.е., является экземпляром метода, поэтому нельзя использовать аннотацию в данном случае. В итоге Sample содержит четыре теста: один будет успешным, другой нет, и один не является разрешенным. Эти четыре метода, которые не аннотируются аннотацией Test, будут проигнорированы инструментом тестирования.

Аннотации Test не имеют прямого воздействия на семантику класса Sample. Они служат только для предоставления информации для использования заинтересованных программ. Говоря более общо, аннотации никогда не меняют семантики на аннотируемом коде, дают возможность для специального отношения к нему со стороны инструментов, таких как эта простая программа, запускающая тест:

```
// Программа для обработки аннотаций с маркерами
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("Passed: %d, Failed: %d%n",
    }
```

```
    passed, tests - passed);
}
}
```

Этот инструмент берет полноценное имя класса в командную строку и запускает все аннотируемые методом `Test` методы класса рефлексивно путем вызова `Method.invoke`. Метод `isAnnotationPresent` говорит инструменту, какой именно метод запускать. Если метод тестирования выводит исключение, то функция отражения заворачивает его в оболочку `InvocationTargetException`. Инструмент фиксирует это исключение и печатает отчет об ошибке, содержащий оригинальное исключение, выведенное методом тестирования, которое извлекается из `InvocationTargetException` с помощью метода `getCause`.

Если попытка запуска метода тестирования отражением выводит какое-либо исключение, кроме `InvocationTargetException`, это означает неправильное использование аннотации `Test`, которое не было зафиксировано при компиляции. Такое использование включает в себя метод экземпляра метода с одним или более параметров или недоступного метода. Следующий блок в программе запуске теста фиксирует ошибки использования `Test` и выводит соответствующее сообщение об ошибке. Вот результат, который выводится, если на `Sample` запускается `RunTests`:

```
public static void Sample.m3() failed: RuntimeException: Boom
INVALID @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

Теперь добавим поддержку тестов, которые завершатся успешно, если она выведет определенное исключение. Нам понадобится новый тип аннотации для этого:

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
```

```
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

Тип параметра для этой аннотации — `Class<? Extends Exception>`. Этот тип символа подстановки довольно громоздкий. Он означает «объект `Class` некоего класса, расширяющий `Exception`» и позволяет пользователю аннотации задать любой тип исключения. Такое использование является примером маркера связанного типа (статья 29). Вот как это выглядит на практике. Обратите внимание, что литералы класса используются здесь как значения параметров аннотации:

```
// Программа, содержащая аннотацию с параметром
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() { // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() { // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { } // Should fail (no exception)
}
```

Теперь давайте изменим инструмент, запускающий тест, чтобы обработать новую аннотацию. Это действие состоит из добавления следующего кода к методу `main`:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
```

```
try {
    m.invoke(null);
    System.out.printf("Test %s failed: no exception%n", m);
} catch (InvocationTargetException wrappedEx) {
    Throwable exc = wrappedEx.getCause();
    Class<? extends Exception> excType =
        m.getAnnotation(ExceptionTest.class).value();
    if (excType.isInstance(exc)) {
        passed++;
    } else {
        System.out.printf(
            "Test %s failed: expected %s, got %s%n",
            m, excType.getName(), exc);
    }
} catch (Exception exc) {
    System.out.println("INVALID @Test: " + m);
}
```

Этот код похож на тот, который мы использовали при обработке аннотации `Test` с только одним исключением: код извлекает значение параметра аннотации и использует его для проверки правильности типа выводимого исключения. Явной передачи нет, поэтому исключение `ClassCastException` не представляет опасности. Тот факт, что программа тест скомпилировалась гарантирует, что параметры аннотации представляют верные типы исключений, с одним лишь предостережением: вполне возможно, что параметры аннотации были верными на момент компиляции, но файла класса, представляющего пределенный тип исключения, больше нет при выполнении. В этом,udem надеяясь, редком случае программа, запускающая тест, выведет исключение `TypeNotPresentException`.

Продвинем пример с проверкой исключений на один шаг вперед, вполне возможно представить тест, который проходит успешно, если выводится одно из нескольких определенных исключений. Механизм аннотации имеет функционал, который облегчает поддержку его ис-

пользования. Предположим, что мы меняем тип параметра аннотации `ExceptionTest` так, чтобы он был массивом объекта `Class`:

```
// Тип аннотации с параметрами массива
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

Синтаксис для массива параметров в аннотации довольно гибок. Он оптимизирован для одноэлементных массивов. Все предыдущие аннотации `ExceptionTest` все еще действительны в новой версии `ExceptionTest` с параметрами массива, и в результате получается одиночный элементный массив. Для определения многоэлементного массива необходимо окружить элементы фигурными скобками и отделить их запятыми:

```
// Код, содержащий аннотацию с параметрами массива
@ExceptionTest({ IndexOutOfBoundsException.class,
    NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();
    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

Разумно изменить программу, запускающую тест для обработки новой версии `ExceptionTest`. Этот код заменяет оригинальную версию:

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
```

```
Throwable exc = wrappedExc.getCause();
Class<? extends Exception>[] excTypes =
m.getAnnotation(ExceptionTest.class).value();
int oldPassed = passed;
for (Class<? extends Exception> excType : excTypes) {
    if (excType.isInstance(exc)) {
        passed++;
        break;
    }
}
if (passed == oldPassed)
System.out.printf("Test %s failed: %s %n", m, exc);
}
```

Структура тестирования, разработанная в этой статье, всего лишь игрушечная, но она ясно демонстрирует преимущества аннотаций над шаблонами именования. И она всего лишь поверхностно затрагивает то, что вы можете делать с аннотациями. Если вы пишете инструмент, который требует от программистов добавление информации из исходных файлов, определите подходящий набор типов аннотации. Просто не существует причины для использования шаблонов наименования теперь, когда есть аннотации.

Как уже говорилось, за исключением системных программистов большей части программистов не потребуется определять типы аннотаций. Все программисты должны, тем не менее, использовать предопределенные типы аннотаций, которыми их обеспечивает платформа Java (статьи 36 и 24). Также рассмотрите использование любых аннотаций, предоставляемых IDE или инструментами статического анализа. Такие аннотации могут улучшить качество диагностической информации, предоставляемой этими инструментами. Обратите внимание, чтобы эти аннотации были стандартизированы, так что вам придется проделать некоторую работу, если вы захотите сменить инструменты или если стандарт, наконец, появится.

Используйте аннотацию `Override` последовательно

Когда аннотации добавились в версии 1.5, несколько типов аннотаций добавилось к библиотекам [JLS, 9.6.1]. Для обычного программиста самым важным из них является `Override`. Эта аннотация может использоваться только при декларировании методов, и она может определить, что декларация аннотируемого метода переопределяет декларацию в супертипе. Если вы последовательно будете использовать данную аннотацию, то она защитит вас от большого количества ужасных ошибок. Рассмотрим эту программу, в которой класс `Bigram` представляет биграмму, или упорядоченную пару символов:

```
// Можете ли определить, где ошибка?
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<Bigram>();
        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 постоянно добавляет к набору 26 биграмм, каждая из которых состоит из двух идентичных символов нижнего регистра. Затем она выводит размер набора. Вы ожидаете, что программа напечатает 26, так как набор не может дублироваться. Если вы попытаетесь запустить программу, то увидите, что она распечата-ет не 26, а 260. Что не так?

Вполне понятно, что автор класса `Bigram` намеревался переопре-делить метод `equals` (статья 8) и даже не забыл переопределить метод `hashCode` (статья 9). К сожалению, наш неудачливый программист не пе-реопределил `equals`, тем самым перегрузив его (статья 41). Для переопре-деления `Object.equals` вы должны определить метод `equals`, параметры которого принадлежат типу `Object`, но параметры метода `equals` класса `Bigram` не принадлежат типу `Object`, следовательно, `Bigram` наследует метод `equals` из `Object`. Этот метод проверяет идентичность объекта, как и оператор `==`. Каждая из десяти копий каждой биграммы отличается от девяти других, так что они не будут равны согласно `Object.equals`, который объясняет, почему программа пишет 260.

К счастью, компилятор может помочь обнаружить эту ошибку, но только если вы поможете ему, сказав, что вы собираетесь перео-пределить `Object.equals`. Чтобы это сделать, необходимо аннотиро-вать `Bigram.equals` с помощью `@Override`, как показано ниже:

```
@Override public boolean equals(Bigram b) {  
    return b.first == first && b.second == second;  
}
```

Если вы вставите эту аннотацию и попытаетесь заново скомпи-лировать программу, компилятор выдаст сообщение об ошибке, вро-де этого:

```
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 сделали неправильно, стукните себя по лбу и замените испорченную реализацию `equals` правильной (статья 8):

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof Bigram))  
        return false;  
    Bigram b = (Bigram) o;  
    return b.first == first && b.second == second;  
}
```

Следовательно, нужно **использовать аннотацию Override для каждой декларации метода, которой вы хотите переопределить декларацию суперкласса**. Есть одно небольшое исключение из этого правила. Если вы пишете класс, который не помечен как абстрактный, и вы уверены, что он переопределяет абстрактный метод, вам не надо беспокоиться о том, чтобы поместить аннотацию Override на этот метод. В классе, который не объявлен как абстрактный, компилятор выдаст сообщение об ошибке, если вам не удастся переопределить абстрактный метод суперкласса. Тем не менее вы можете обратить внимание на все методы в вашем классе, которые переопределяют методы суперкласса, в любом случае вы можете и этим методам также дать аннотацию.

Современные IDE дают нам еще одну причину, почему необходимо последовательно использовать аннотации Override. У таких IDE есть автоматические проверки, известные как *инспекции кода*. Если вы примените соответствующую инспекцию кода, то IDE выдаст предупреждение, если у вас есть метод, который не содержит аннотации Override, но на самом деле переопределяет метод суперкласса. Если вы используете аннотацию Override последовательно, то эти сообщения предупредят вас о ненамеренном переопределении. Эти сообщения дополняют сообщения об ошибках от компилятора, который предупредит о том, что переопределение не удалось. Между IDE и компилятором вы можете быть уверены, что вы переопределяете методы там, где вы хотите, и нигде более.

Если вы используете версию 1.6 или более позднюю, аннотация Override даст вам еще больше помощи в поиске ошибок. В версии 1.6 стало разрешено использование аннотации Override на декларациях методов, которые переопределяют декларации из интерфейсов

и классов. В конкретном классе, который объявлен для реализации интерфейса, вам нет необходимости аннотировать методы, которые, по вашему мнению, должны переопределять методы интерфейсов, потому что компилятор выдаст сообщение об ошибке, если ваш класс не сможет реализовать каждый метод интерфейса. И снова вы можете, если хотите, добавить эти аннотации просто для привлечения внимания на методы интерфейса, но для этого нет строгой необходимости.

В абстрактном классе или интерфейсе, тем не менее, лучше аннотировать все методы, которые должны переопределять методы суперкласса или методы суперинтерфейса, вне зависимости от того, конкретные они или абстрактные. Например, интерфейс Set не добавляет новые методы к интерфейсу Collection, поэтому он должен содержать аннотацию Override на всех своих декларациях методов, чтобы гарантировать, что он случайно не добавит новые методы к интерфейсу Collection.

Подведем итоги. Компилятор может защитить вас от большого количества ошибок, если вы используете аннотацию Override для каждой декларации методов, которые должны переопределять декларацию супертипов, за одним исключением. В конкретных классах вам не нужно аннотировать методы, которые должны переопределять декларации абстрактных методов (хотя делать так вредно).

Статья
37

Используйте маркерные интерфейсы для определения типов

Маркерный интерфейс — это такой интерфейс, который не содержит деклараций методов, но просто определяет (или «маркирует») класс, который реализует интерфейс как имеющий определенное свойство. Например, рассмотрим интерфейс Serializable (глава 11). Реализуя этот интерфейс, класс сообщает, что его экземпляры могут быть приписаны к ObjectOutputStream (или «сериализованы»).

Возможно, вы слышали, что маркерные аннотации (статья 35) делают маркерные интерфейсы устаревшими. Это неверное утверждение. Маркерные интерфейсы обладают двумя преимуществами над маркерными аннотациями. Первое и основное: **маркерные интерфейсы определяют тип, который реализуется экземплярами маркированного класса, а маркерные аннотации – нет.** Существование этого типа позволяет вам фиксировать ошибки во время компиляции, которые вы не можете заметить до выполнения в случае использования маркерных аннотаций.

В случае с маркерным интерфейсом Serializable метод ObjectOutputStream.write(Object) не будет работать, если его аргумент не реализует интерфейс. Неизвестно, почему автор API ObjectOutputStream не использовал преимущества интерфейса Serializable при объявлении метода write. Тип аргумента метода должен был быть Serializable, а не Object. Попытка вызова ObjectOutputStream.write на объекте, который не реализует Serializable, будет неудачной только при выполнении, но так не должно быть.

Другим преимуществом маркерных интерфейсов над маркерными аннотациями является то, что на них можно более точно нацелиться. Если тип аннотации объявлен с указанием цели ElementType.TYPE, то он может применяться к любому классу или интерфейсу. Предположим, у вас есть маркер, который применим только к реализациям определенного интерфейса. Если вы определите его как маркерный интерфейс, то вы сможете расширить единственный интерфейс, к которому он применим, гарантируя, что все маркованные типы являются также подтипами единственного интерфейса, к которому он применим.

Есть спорное утверждение, что интерфейс Set всего лишь ограниченный маркерный интерфейс. Это применимо только к подтипам Collection, но это не добавляет никаких методов, кроме определенных в Collection. Он обычно не рассматривается как маркерный интерфейс, потому что он уточняет соглашения нескольких методов Collection, которые включают add, equals и hashCode. Но легко пред-

ставить маркерный интерфейс, который применим только к подтипам нескольких определенных интерфейсов и не уточняет соглашений каких-либо методов интерфейса. Такой маркерный интерфейс может описать некоторые инварианты всего объекта или отразить, что экземпляры подходят для обработки методом какого-либо другого класса (таким же образом, как и интерфейс `Serializable` отражает, что экземпляры подходят для обработки `ObjectOutputStream`).

Главным преимуществом маркерных аннотаций над маркерными интерфейсами является возможность добавить больше информации к типу аннотации после того, как она уже используется путем добавления одного или более элементов типа аннотации к уже имеющимся по умолчанию [JLS, 9.6], что дает начало простым типам маркерных аннотаций и может развиться в большой тип аннотации со временем. Такая эволюция невозможна с маркерными интерфейсами, поскольку в принципе невозможно добавить метод к интерфейсу после того, как он уже реализован (статья 18).

Другим преимуществом маркерных аннотаций является то, что они, по сути, часть большого аннотационного функционала. Следовательно, маркерные аннотации обеспечивают последовательность в структурах, разрешающих применение аннотаций большого количества программных элементов.

И так, когда же мы должны использовать маркерную аннотацию, а когда — маркерный интерфейс? Определенно, вам нужно использовать аннотацию, если маркер относится к любому программному элементу, кроме класса или интерфейса, поскольку только классы и интерфейсы могут реализовывать или расширять интерфейс. Если маркер относится только к классам и интерфейсам, задайте себе вопрос, могу ли я захотеть написать один или более методов, которые принимали бы только объекты, имеющие такую маркировку? Если это так, то вам предпочтительнее использовать маркерный интерфейс вместо аннотации. Это даст вам возможность использовать интерфейс в качестве типа параметра для методов в вопросе, который приведет к реальной выгоде при проверке типов на этапе компиляции.

Если вы ответили на первый вопрос, задайте себе еще один: хочу ли я ограничить использование этого маркера элементами определенного интерфейса навсегда? Если так, то разумно определить маркер в качестве субинтерфейса для данного интерфейса. Если вы ответили «нет» на оба вопроса, то вам вероятнее всего стоит использовать маркерную аннотацию.

Подведем итоги. И маркерный интерфейс, и маркерная аннотация имеют свои области применения. Если вы хотите определить тип, который не имеет каких-либо новых методов, связанных с ним, то лучший способ — использовать маркерный интерфейс. Если вы хотите маркировать элементы программы, кроме классов и интерфейсов, чтобы дать возможность добавления информации к маркеру в будущем или чтобы уместить маркер в структуру, которая уже использует типы аннотаций, тогда верным выбором будет использование маркерной аннотации. **Если вы пишете тип маркерной аннотации, цель которого elementType.TYPE, потратьте время, чтобы определить, действительно ли это должен быть тип аннотации, или же маркерный интерфейс более подходит.**

В некотором роде, эта статья — противоположность статьи 19, в которой говорится: «Если вы не хотите определять тип, не используйте интерфейс». При первом приближении эта статья говорит: «Если вы действительно хотите определить тип, используйте интерфейс».

7

Г л а в а

Методы

В данной главе рассматривается несколько аспектов проектирования методов: как обрабатывать параметры и возвращаемые методом значения, как проектировать сигнатуры методов и как документировать методы. Значительная часть материала относится как к методам, так и к конструкторам. Как и в главе 5, особое внимание здесь уделяется удобству, устойчивости и гибкости программ.

Статья
38

Проверяйте достоверность параметров

Большинство методов и конструкторов имеют ограничения на то, какие значения могут быть переданы с параметрами. Например, нередко указывается, что значения индекса должны быть неотрицательными, а ссылки на объекты отличны от null. Все эти ограничения вы обязаны четко документировать и начать метод с их проверки. Это частный случай более общего принципа: вы должны стараться выявлять ошибки как можно скорее после того, как они произойдут. В противном случае обнаружение ошибки станет менее вероятным, а определение источника ошибки — более трудоемким.

Если методу передано неверное значение параметра, но перед началом обработки он проверяет полученные параметры, то вызов этого метода быстро и аккуратно завершится с инициированием соответствующего исключения. Если же метод не проверяет своих параметров, может произойти несколько событий. Метод может завершиться посередине обработки, инициировав непонятное исключение. Хуже, если метод завершится normally, без возражений вычислив неверный результат. Но самое худшее, если метод завершится normally, но оставит некий объект в опасном состоянии, что впоследствии в непредсказуемый момент времени вызовет появление ошибки в какой-либо другой части программы, никак не связанной с этим методом.

В открытых методах для описания исключений, которые будут инициироваться, когда значения параметров нарушают ограничения, используйте тег `@throws` генератора документации Javadoc (статья 62). Как правило, это будет исключение `IllegalArgumentException`, `IndexOutOfBoundsException` или `NullPointerException` (статья 60). После того как вы документировали ограничения для параметров метода и исключения, которые будут инициироваться в случае нарушения ограничений, установить эти ограничения для метода не составит труда. Приведем типичный пример:

```
/*
 * Возвращает объект BigInteger, значением которого является модуль данного
 * числа по основанию m. Этот метод отличается от метода remainder тем,
 * что всегда возвращает неотрицательное значение BigInteger.
 *
 * @param      m - модуль, должен быть положительным числом
 * @return     this mod m
 * @throws    ArithmeticException, if m <= 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus not positive");
    ... // Вычисления
}
```

Если метод не предоставляется в распоряжение пользователей, то вы, как автор пакета, контролируете все условия, при которых этот метод вызывается, а потому можете и обязаны убедиться в том, что ему будут передаваться только правильные значения параметра. Методы, не являющиеся открытыми, должны проверять свои параметры, используя утверждения (*assertion*), как показано ниже:

```
// 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;
    ... // Do the computation
}
```

Данные утверждения заявляют, что условия утверждения будут положительные, вне зависимости от того, какие пакеты используются его клиентом. В отличие от нормальной проверки корректности утверждения выводят ошибку `AssertionError` в случае неудачного запуска не вызывают никакого эффекта и ничего не стоят, пока они не задействованы. Это можно сделать, передав метку — `ea` (или `enableassertions`) в интерпретатор Java. Для более подробной информации об утверждениях см. учебник Sun [`Asserts`].

Особенно важно проверять правильность параметров, которые не используются методом, а откладываются для обработки в дальнейшем. Например, рассмотрим статический метод генерации из статьи 16, который получает массив целых чисел и возвращает представление этого массива в виде экземпляра `List`. Если клиент метода передаст значение `null`, метод инициирует исключение `NullPointerException`, поскольку содержит явную проверку. Если бы проверка отсутствовала, метод возвращал бы ссылку на вновь сформированный экземпляр `List`, который будет инициировать исключение `NullPointerException`, как только клиент попытается им воспользоваться. К сожалению, к тому моменту определить происхождение экземпляра `List` будет уже трудно, что может значительно усложнить задачу отладки.

Частным случаем принципа, требующего проверки параметров, которые должны быть сохранены для использования в дальнейшем, являются конструкторы. Для конструкторов очень важно проверять правильность параметров, чтобы не допустить создания объектов, которые нарушают инварианты соответствующего класса.

Существуют исключения из правила, обязывающего перед выполнением вычислений проверять параметры метода. Важное значение имеет ситуация, когда явная проверка правильности является дорогостоящей или невыполнимой операцией, но вместе с тем параметры все же неявно проверяются непосредственно в процессе их обработки. Например, рассмотрим метод `Collections.sort(List)`, сортирующий список объектов. Все объекты в представленном списке должны быть взаимно сравнимы. В ходе его сортировки каждый объект в нем будет сравниваться с каким-либо другим объектом из того же списка. Если объекты не будут взаимно сравнимы, в результате одного из таких сравнений будет инициировано исключение `ClassCastException`, а это именно то, что должен делать в таком случае метод `sort`. Таким образом, нет смысла выполнять упреждающую проверку взаимной сравнимости элементов в списке. Заметим, однако, что неразборчивое использование такого подхода может привести к потере такого качества, как атомарность сбоя (статья 64).

Иногда в ходе обработки неявно осуществляется требуемая проверка некоторых параметров, однако, когда проверка фиксирует ошибку, инициируется совсем не то исключение. Другими словами, исключение, которое инициируется в ходе обработки и связано с обнаружением неверного значения параметра, не соответствует тому исключению, которое, согласно вашему описанию, должно инициироваться этим методом. В таких случаях для преобразования внутреннего исключения в требуемое вы должны использовать идиому трансляции исключений, описанную в статье 61.

Не следует из этой статьи делать вывод, что произвольное ограничение параметров является хорошим решением. Наоборот, вы должны создавать методы как можно более общими. Чем меньше

ограничений вы накладываете на параметры, тем лучше, при условии, что метод для каждого полученного значения параметра может сделать что-то разумное. Часто, однако, некоторые ограничения обусловлены реализацией абстракций.

Подведем итог. Каждый раз, когда вы пишете метод или конструктор, вы должны подумать над тем, какие существуют ограничения для его параметров. Эти ограничения необходимо отразить в документации и реализовать в самом начале метода в виде явной проверки. Важно привыкнуть к такому порядку. Та скромная работа, которая с ним связана, будет с лихвой вознаграждена при первом же обнаружении неправильного параметра.

Статья
39

При необходимости создавайте резервные копии

Одной из особенностей, благодаря которой работа с языком программирования Java доставляет такое удовольствие, является его **безопасность**. Это означает, что в отсутствие машино-зависимых методов (*native method*) он неуязвим по отношению к переполнению буферов и массивов, к неконтролируемым указателям, а также другим ошибкам, связанным с разрушением памяти, которые мешают при работе с такими небезопасными языками, как Си и С++. При использовании безопасного языка можно писать класс и не сомневаться, что его инварианты будут оставаться правильными, что бы ни произошло с остальными частями системы. В языках, где память трактуется как один гигантский массив, такое невозможно.

Но даже в безопасном языке вы не изолированы от других классов, если не приложите со своей стороны некоторые усилия. **Вы должны писать программы с защитой, исходя из предположения, что клиенты вашего класса будут предпринимать все возможное для того, чтобы разрушить его инварианты.** Это действительно так, когда кто-то пытается взломать систему безопасности. Однако, скорее всего, вашему классу придется иметь дело с непредвиденным по-

ведением других классов, которое обусловлено простыми ошибками программиста, пользующегося вашим API. В любом случае имеет смысл потратить время и написать классы, которые будут устойчивы при неправильном поведении клиентов.

Хотя другой класс не сможет поменять внутреннее состояние объекта без какой-либо поддержки со стороны последнего, оказать такое содействие, не желая того, на удивление просто. Например, рассмотрим класс, задачей которого является представление неизменного периода времени:

```
// Неправильный класс «неизменяемого» периода времени
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start – начало периода.
     * @param end – конец периода; не должен предшествовать началу.
     * @throws IllegalArgumentException, если start позже, чем end.
     * @throws NullPointerException, если start или end равны null.
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " after "
                + end);
        this.start = start;
        this.end = end;
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }

    ... // Остальное опущено
}
```

На первый взгляд может показаться, что это неизменяемый класс, который успешно выполняет условие, заключающееся в том, что началу периода не предшествует его же конец. Однако, воспользовавшись изменяемостью объекта Date, можно с легкостью нарушить этот инвариант:

```
// Атака на содержимое экземпляра Period  
Date start = new Date();  
Date end = new Date();  
Period p = new Period(start, end);  
end.setYear(78); // Изменяет содержимое объекта p!
```

Чтобы защитить содержимое экземпляра Period от нападений такого типа, **для каждого изменяемого параметра конструктор должен создавать резервную копию (*defensive copy*) и использовать именно эти копии**, а не оригинал, как составные части экземпляра Period:

```
// Исправленный конструктор: для представленных параметров создает  
// резервные копии  
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 + " позже " + end);  
}
```

С новым конструктором описанная ранее атака уже не может воздействовать на экземпляр Period. Заметим, что **резервные копии создаются до проверки правильности параметров (статья 38)**, так что сама проверка выполняется уже не для оригинала, а для его копии. Такой порядок может показаться искусственным, но он необходим, поскольку защищает класс от подмены параметров, которая выполняется из параллельного потока в пределах «окна уязвимости» (*window of vulnerability*): с момента, когда параметры проверены, и до того момента, когда для них созданы копии. (Среди специалистов по компьютерной

безопасности эта атака называется атакой во время проверки / во время использования

(а *time-of-check/time-of-use*) или иначе атакой *TОСТОУ*[Viega01].)

Заметим также, что для создания резервных копий мы не пользовались методом `clone` из класса `Date`. Поскольку `Date` не является окончательным классом, нет гарантии, что метод `clone` возвратит объект именно класса `java.util.Date`, — он может вернуть экземпляр ненадежного подкласса, созданного специально для нанесения ущерба. Например, такой подкласс может записывать в закрытый статический список ссылку на экземпляр в момент создания последнего, а затем предоставить злоумышленнику доступ к этому списку. В результате злоумышленник получит полный контроль над всеми этими экземплярами. Чтобы предотвратить атаки такого рода, **не используйте метод `clone` для создания резервной копии параметра, который имеет тип, позволяющий ненадежным партнерам создавать подклассы.**

Обновленный конструктор успешно защищает от вышеописанной атаки, однако все равно остается возможность модификации экземпляра `Period`, поскольку его методы предоставляют доступ к его внутренним частям, которые можно поменять:

```
// Вторая атака на содержимое экземпляра Period
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Изменяет внутренние данные p!
```

Чтобы защититься от второй атаки, просто модифицируйте методы доступа таким образом, чтобы **возвращать резервные копии изменяемых внутренних полей**.

```
// Исправленные методы доступа: создаются резервные копии
// внутренних полей
public Date start() {
    return (Date) start.clone();
```

```
}

public Date end() {
    return (Date) end.clone();
}
```

Получив новый конструктор и новые методы доступа, класс `Period` действительно стал неизменяемым. Теперь некомпетентному программисту или злоумышленнику не удастся нарушить инвариант, гласящий, что начало периода предшествует его концу. Это так, поскольку, за исключением самого класса `Period`, никакой другой класс не имеет возможности получить доступ хоть к какому-нибудь изменяемому полю экземпляра `Period`. Указанные поля действительно инкапсулированы в этом объекте.

Заметим, что новые методы доступа, в отличие от нового конструктора, для создания резервных копий используют метод `clone`. Такое решение приемлемо (хотя и не обязательно), поскольку мы точно знаем, что внутренние объекты `Date` в классе `Period` относятся к классу `java.util.Date`, а не какому-то потенциально ненадежному подклассу.

Резервное копирование параметров производится не только для неизменяемых классов. Всякий раз, когда вы пишете метод или конструктор, который помещает во внутреннюю структуру объект, созданный клиентом, задумайтесь, не является ли этот объект потенциально изменяемым. Если да, проанализируйте, будет ли ваш класс устойчив к изменениям в объекте, когда он уже получил доступ к этой структуре данных. Если ответ отрицательный, то вы должны создать резервную копию этого объекта и поместить ее в структуру данных вместо оригинала. Например, если ссылку на объект, предоставленный клиентом, вы предполагаете использовать как элемент во внутреннем экземпляре `Set` или как ключ во внутреннем экземпляре `Map`, следует учитывать, что инварианты этого набора или схемы могут быть нарушены, если после добавления в них объект вдруг поменяется.

То же самое справедливо и в отношении резервного копирования внутренних компонентов перед возвращением их клиенту. Вы должны дважды подумать, является ли ваш класс изменяемым или нет,

прежде чем передавать клиенту ссылку на внутреннюю компоненту, которую можно изменить. Возможно, вам все же следует возвращать резервную копию. Крайне важно также помнить о том, что массивы ненулевой длины всегда являются изменяемыми. Поэтому для внутреннего массива вы всегда должны делать резервную копию, прежде чем возвращать его клиенту. Как альтернатива, вы можете возвращать пользователю неизменяемое представление этого массива. Оба этих приема показаны в статье 13.

Таким образом, урок, который можно извлечь из всего сказанного, заключается в том, что в качестве составных частей объектов вы должны по возможности использовать неизменяемые объекты, чтобы не пришлось беспокоиться о резервном копировании (статья 15). В случае же с нашим примером `Period` стоит отметить, что опытные программисты для внутреннего представления времени часто используют не ссылку на объект `Date`, а простой тип `long`, возвращаемый методом `Date.getTime()`. И поступают они так в первую очередь потому, что `Date` является изменяемым.

Не всегда можно создать резервную копию изменяемого параметра перед его включением в объект. Если класс доверяет тому, кто его вызывает, и не будет изменять внутреннее содержимое, то, возможно, класс и его клиенты являются частями одного и того же пакета, и тогда можно обойтись без резервного копирования. При подобных обстоятельствах в документации к классу должно быть четко объяснено, что вызывающий не должен менять задействованные параметры или возвращаемые значения.

Даже за рамками одного пакета не всегда стоит использовать резервное копирование изменяемых параметров перед интегрированием их в объект. Существуют такие методы и конструкторы, чей вызов означает явную передачу объекта, на который указывает параметр-ссылка. Вызвав такой метод, клиент дает обещание, что он не будет напрямую менять этот объект. Для метода или конструктора, предполагающего, что ему будет полностью передано управление изменяемым объектом, предоставленным клиентом, это обстоятельство должно быть четко оговорено в документации.

Классы, где содержатся методы или конструкторы, вызов которых означает передачу управления объектом, не способны защитить себя от злоумышленника. Такие классы можно использовать только тогда, когда есть взаимное доверие между классом и его клиентами или же когда нарушение инвариантов класса не способно нанести ущерба, кроме самого клиента. Последнюю ситуацию иллюстрирует шаблон класса-оболочки (статья 16). При определенном характере класса-оболочки клиент может разрушить инварианты этого класса, используя прямой доступ к объекту уже после того, как он попал в оболочку, однако обычно это не наносит вреда никому, кроме самого этого клиента.

Подведем итоги. Если у класса есть изменяемые компоненты, которые он получает от клиента или возвращает ему, то необходимо резервное копирование всех компонентов класса. Если затраты на копирование слишком высоки и класс доверяет своим клиентам, зная, что они не изменят неподходящим образом компоненты, тогда резервное копирование можно заменить документированием, отражающим, что клиенты не должны менять задействованные компоненты.

Статья
ЧО

Тщательно проектируйте сигнатуру метода

В этой статье приводятся советы по проектированию API, не удостоившиеся собственной статьи. Собранные вместе, они помогут сделать ваш API не столь подверженным ошибкам, более удобным и простым в изучении.

Тщательно выбирайте названия методов. Названия всегда должны соответствовать стандартным соглашениям по именованию (статья 56). Вашей главной целью должен быть выбор таких имен, которые будут понятны и согласуются с остальными названиями в том же пакете. Второй целью должен быть выбор имен, отвеча-

ющих общим соглашениям, если таковые имеются. В случае сомнений смотрите руководство по API библиотек языка Java. Несмотря на массу противоречий, которые неизбежны, если учитывать размер и возможности библиотек, здесь также присутствует консенсус.

Не заходите слишком далеко в погоне за удобством своих методов. Каждый метод должен выполнить собственную часть работы. Избыток методов делает класс слишком сложным для изучения, использования, описания, тестирования и сопровождения. В отношении интерфейсов это верно вдвое: большое количество методов усложняет жизнь и разработчикам, и пользователям. Для каждого действия, поддерживаемого вашим типом, создайте полнофункциональный метод. Сокращенный вариант операции рассматривайте лишь в том случае, если она будет использоваться часто. **Если есть сомнения, забудьте об этом варианте.**

Избегайте длинного перечня параметров. Правило таково, что на практике четыре параметра нужно рассматривать как максимум, а чем параметров меньше, тем лучше. Большинство программистов не способны помнить более длинные списки параметров. Если метод превышает этот предел, вашим API невозможно будет пользоваться, не обращаясь беспрестанно к его описанию. **Особенно вредны длинные последовательности параметров одного и того же типа.** И это не только потому, что ваш пользователь не сможет запомнить порядок их следования. Если он по ошибке поменяет их местами, его программа все равно будет компилироваться и работать. Только вот делать она будет совсем не то, что хотел ее автор.

Для сокращения слишком длинных списков параметров можно использовать три приема. Первый заключается в разбиении метода на несколько методов, каждому из которых нужно лишь какое-то подмножество его параметров. Если делать это неаккуратно, может получиться слишком много методов, однако этот же прием помогает сократить количество методов путем увеличения их ортогональности. Например, рассмотрим интерфейс `java.util.List`. У него нет методов для поиска индекса первого и последнего элемента в подсписке,

каждому из них потребовалось бы по три параметра. Вместо этого он предоставляет метод `subList`, который принимает два параметра и возвращает представление подсписка. Для получения желаемого результата метод `subList` можно объединить с методами `indexOf` или `lastIndexOf`, принимающими по одному параметру. Более того, метод `subList` можно сочетать с любыми другими методами экземпляра `List`, чтобы выполнять самые разные операции для подсписков. Полученный API имеет очень высокое соотношение мощности и размера.

Второй прием сокращения чрезмерно длинных перечней параметров заключается в создании вспомогательных классов, обеспечивающих агрегирование параметров. Обычно эти вспомогательные классы являются статическими классами-членами (статья 22). Этот прием рекомендуется использовать, когда становится понятно, что часто возникающая последовательность параметров на самом деле представляет некую отдельную сущность. Предположим, что вы пишете класс, реализующий карточную игру, и выясняется, что постоянно передается последовательность из двух параметров: достоинство карты и ее масть. И ваш API, и содержимое вашего класса, вероятно, выиграют, если для представления карты вы создадите вспомогательный класс и каждую такую последовательность параметров замените одним параметром, соответствующим этому вспомогательному классу.

Третий прием, сочетающий в себе аспекты первых двух, должен адаптировать шаблон «построитель» (статья 2) от конструктора объекта до запуска метода. Если у вас есть метод с многими параметрами, особенно если некоторые из них необязательны, то будет предпочтительнее определить объект, который будет представлять все параметры и позволить клиенту выполнять многократный вызов «сеттеров» на данном объекте, каждый из которых будет устанавливать один параметр на маленькой, связанной группе. Как только нужный параметр задан, клиент запускает «исполняемый» метод на объекте, который выполняет окончательные проверки параметров и производит уже актуальный расчет.

Выбирайте тип параметра, отдавайте предпочтение интерфейсу, а не классу (статья 52). Если для декларации параметра имеется подходящий интерфейс, всегда используйте его, а не класс, который

реализует этот интерфейс. Например, нет причин писать метод, принимающий параметр типа `Hashtable`, лучше использовать `Map`. Это позволит вам передавать этому методу `Hashtable`, `HashMap`, `TreeMap`, подмножество `TreeMap`, и вообще любую, пока еще не написанную реализацию интерфейса `Map`. Применяя же вместо интерфейса класс, вы навязываете вашему клиенту конкретную реализацию и вынуждаете использовать ненужное и потенциально трудоемкое копирование в том случае, если входные данные будут представлены в какой-либо иной форме.

Предпочитайте использовать двухэлементные перечисляемые типы вместо параметров `boolean`. Код легче читать и писать, особенно если вы используете IDE, который поддерживает автозаполнение. Также легче добавлять больше опций. Например, у вас есть тип `Thermometer` с методом статической генерации, который берет значение этого перечислимого типа:

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Не только использование `Thermometer.newInstance(TemperatureScale, CELCIUS)` намного понятнее, чем использование `Thermometer.newInstance(true)`, но вы можете в будущей версии добавить `KELVIN` к `TemperatureScale` без необходимости добавлять новый метод статической генерации к `Thermometer`. Также вы можете переделать зависимости температурной шкалы в методе для перечислимых констант (статья 30). Например, у каждой константы шкалы может быть метод, который берет значение `double` и нормализует его к шкале Цельсия.

Статья
Ч 1

Перезагружая методы, соблюдайте осторожность

Приведем пример попытки классифицировать коллекции по признаку — набор, список и другой вид коллекций, — предпринятой из лучших побуждений:

```
// Ошибка! – 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));
    }
}
```

Возможно, вы ожидаете, что эта программа напечатает сначала «Set», затем «List» и, наконец, «Unknown Collection». Ничего подобного! Программа напечатает «Unknown Collection» три раза. Почему это происходит? Потому что метод `classify` перегружается (*overloading*) и **выбор варианта перегрузки осуществляется на стадии компиляции**. Для всех трех проходов цикла параметр на стадии компиляции имеет один и тот же тип `Collection<?>`. И хотя во время выполнения программы при каждом проходе используется другой тип, это уже не влияет на выбор варианта перегрузки. Поскольку во время компиляции параметр имел тип `Collection<?>`, может применяться только третий вариант перегрузки: `classify(Collection<?>)`. И именно этот перегруженный метод вызывается при каждом проходе цикла.

Поведение этой программы такое странное потому, что **выбор перегруженных методов является статическим, тогда как выбор**

переопределенных методов — динамическим. Правильный вариант переопределенного метода выбирается при выполнении программы, исходя из того, какой тип в этот момент имел объект, для которого этот метод был вызван. Напомним, что переопределение (*overriding*) метода осуществляется тогда, когда подкласс имеет декларацию метода с точно такой же сигнатурой, что и у декларации метода предка. Если в подклассе метод был переопределен и затем данный метод был вызван для экземпляра этого подкласса, то выполняться будет уже переопределенный метод независимо от того, какой тип экземпляр подкласса имел на стадии компиляции. Для пояснения рассмотрим маленькую 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) {  
        Wine[] wines = {  
            new Wine(), new SparklingWine(), new Champagne()  
        };  
        for (Wine wine : wines)  
            System.out.println(wine.name());  
    }  
}
```

Метод `name` декларируется в классе `Wine` и переопределяется в классах `SparklingWine` и `Champagne`. Как и ожидалось, эта программа печатает `Wine`, `SparklingWine` и `Champagne`, хотя на стадии компиляции при каждом проходе в цикле экземпляр имеет тип `Wine`. Тип объекта на стадии компиляции не влияет на то, какой из методов будет

исполняться, когда поступит запрос на вызов переопределенного метода: всегда выполняется «самый точный» переопределяющий метод. Сравните это с перезагрузкой, когда тип объекта на стадии выполнения уже не влияет на то, какой вариант перезагрузки будет использоваться: выбор осуществляется на стадии компиляции и всецело основывается на том, какой тип имеют параметры на стадии компиляции.

В примере с `CollectionClassifier` программа должна была определять тип параметра, автоматически переключаясь на соответствующий перезагруженный метод на основании того, какой тип имеет параметр на стадии выполнения. Именно это делает метод `name` в примере с `Wine`. Перезагрузка метода не имеет такой возможности. Предполагая, что тут требуется статический метод, исправить программу можно, заменив все три варианта перезагрузки метода `classify` единым методом, который выполняет явную проверку `instanceOf`:

```
public static String classify(Collection<?> c) {  
    return c instanceof Set ? "Set" :  
        c instanceof List ? "List" : "Unknown Collection";  
}
```

Поскольку переопределение является нормой, а перезагрузка — исключением, именно переопределение задает, что люди ожидают увидеть при вызове метода. Как показал пример `CollectionClassifier`, перезагрузка может не оправдать эти ожидания. Не следует писать код, поведение которого неочевидно для среднего программиста. Особенno это касается интерфейсов API. Если рядовой пользователь API не знает, какой из перезагруженных методов будет вызван для указанного набора параметров, то работа с таким API, вероятно, будет сопровождаться ошибками. Причем ошибки эти проявятся, скорее всего, только на этапе выполнения в виде некорректного поведения программы, и многие программисты не смогут их диагностировать. Поэтому необходимо **избегать запутанных вариантов перезагрузки**.

Стоит обсудить, что же именно сбивает людей с толку при использовании перезагрузки. **Безопасная, умеренная политика предписывает никогда не предоставлять два варианта перезагрузки с одним и тем же числом параметров.** Если метод использует varargs, то лучше вообще его не перегружать, кроме случаев, описанных в статье 42. Если вы придерживаетесь этого ограничения, у программистов никогда не возникнет сомнений по поводу того, какой именно вариант перезагрузки соответствует тому или иному набору параметров. Это ограничение не слишком обременительно, поскольку, вместо того чтобы использовать перезагрузку, вы всегда можете дать методам различные названия.

Например, рассмотрим класс `ObjectOutputStream`. Он содержит варианты методов `write` для каждого простого типа и нескольких ссылочных типов. Вместо того чтобы перезагружать метод `write`, они применяют такие сигнатуры, как `writeBoolean(boolean)`, `writeInt(int)` и `writeLong(long)`. Дополнительное преимущество такой схемы именования по сравнению с перезагрузкой заключается в том, что можно создать методы `read` с соответствующими названиями, например `readBoolean()`, `readInt()` и `readLong()`. И действительно, в классе `ObjectInputStream` есть методы чтения с такими названиями.

В случае с конструкторами у вас нет возможности использовать различные названия, несколько конструкторов в классе всегда подлежат перезагрузке. Правда, в отдельных ситуациях вы можете вместо конструктора предоставлять статический метод генерации (статья 1), но это не всегда возможно. Однако, с другой стороны, при применении конструкторов вам не нужно беспокоиться о взаимосвязи между перезагрузкой и переопределением, так как конструкторы нельзя переопределять. Поскольку вам, вероятно, придется предоставлять несколько конструкторов с одним и тем же количеством параметров, полезно знать, в каких случаях это безопасно.

Предоставление нескольких перезагруженных методов с одним и тем же количеством параметров вряд ли запутает программистов, если всегда понятно, какой вариант перезагрузки соответствует за-

данному набору реальных параметров. Это как раз тот случай, когда у каждой пары вариантов перезагрузки есть хотя бы один формальный параметр с «совершенно непохожим» типом. Два типа считаются совершенно непохожими, если экземпляр одного из этих типов невозможно привести к другому типу. В этих условиях выбор варианта перезагрузки для данного набора реальных параметров полностью диктуется тем, какой тип имеют параметры в момент выполнения программы, и никак не связан с их типом на стадии компиляции. Следовательно, исчезает главный источник путаницы.

Например, класс `ArrayList` имеет конструктор, принимающий параметр `int`, и конструктор, принимающий параметр типа `Collection`. Трудно представить себе условия, когда возникнет путаница с вызовом двух этих конструкторов,

До версии 1.5 все примитивные типы радикально отличались от всех типов ссылок, но это утверждение более не верно с появлением автоупаковщиков и привело к реальным проблемам. Рассмотрим следующую программу:

```
public class SetList {  
    public static void main(String[] args) {  
        Set<Integer> set = new TreeSet<Integer>();  
        List<Integer> list = new ArrayList<Integer>();  
        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);  
    }  
}
```

Данная программа добавляет целые числа от -3 до 2 к отсортированному набору и к списку, затем делает три идентичных обращения

к remove и в наборе, и в списке. Как и любой нормальный человек, вы ожидаете, что программа удалит неотрицательные значения (0, 1 и 2) из набора и списка и напечатает [-3, -2, -1] [-3, -2, -1]. На самом деле программа удаляет неотрицательные значения из набора и нечетные значения из списка и выводит [-3, -2, -1] [-0, 0, 2]. Будет преуменьшением назвать такое поведение программы путанным.

Итак, вот что происходит: Обращение к `set.remove()` выбирает перегруженный метод `remove(E)`, где E элемент типа из набора (`Integer`), и преобразует i с `int` в `integer`. Мы ожидаем такое поведение программы, следовательно, программа удаляет положительные значения из набора. Обращение к `list.remove()`, с другой стороны, выбирает перезагруженный `remove(int i)`, который удаляет элементы из определенных позиций в списке. Если вы начнете со списка [-3, -2, -1, 0, 1, 2] и удалите все нулевые элементы, затем первый и затем второй, у вас останется [-2, 0, -2] и загадка будет разгадана. Для решения проблемы передайте аргумент `list.remove` в `Integer`, заставляя выбрать верный тип перезагрузки. Вы также можете запустить `Integer.valueOf` на i и передать результат в `list.remove`. В любом случае 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))  
}
```

Путаное поведение, продемонстрированное в предыдущем примере, произошло, потому что у интерфейса `List<E>` две перегрузки метода `remove`: `remove(E)` и `remove(int)`. До версии 1.5, когда это все было обобщено, у интерфейса `List` был метод `remove(Object)` вместо `remove(E)` и соответствующие типы параметров `Object` и `int` сильно отличались. Но с появлением средств обобщенного программирования и преобразований эти два типа параметров больше так не отличаются. Другими словами, появление средств обобщенного программирования и преобразований в языке нанесло ущерб интерфейсу `List`. К счастью, лишь незначительное количество API в библиотеках Java

пострадало, но эта история разъясняет нам (что более важно), что перегрузку надо использовать осторожно, так как преобразования и средства обобщенного программирования теперь тоже часть языка.

Типы массивов и классы, кроме `Object`, сильно отличаются. Так же типы массивов и интерфейсы, кроме `Serializable` и `Cloneable`, сильно отличаются. Два отдельных класса считаются несвязанными, если ни один из классов не является потомком от другого [JLS, 5.5]. Например, `String` и `Throwable` несвязанные классы. Ни один объект не может быть экземпляром двух несвязанных классов, потому что несвязанные классы сильно отличаются.

Есть и другие пары типов, которые не могут быть преобразованы ни в одну сторону [JLS 5.1.12], но если вы выходите за рамки простых вышеописанных случаев, то для многих программистов будет трудно понять, какая перегрузка относится к набору актуальных параметров. Правила, определяющие, какая перегрузка выбирается, крайне сложны. Они занимают 33 страницы спецификации языка [JLS, 15.12.1-3], и не многие программисты понимают все их тонкости.

Иногда вы будете хотеть нарушить указания данной статьи, особенно для развития существующих классов. Например, начиная с версии 1.4 у класса `String` был метод `contentEquals(StringBuffer)`. В версии 1.5 был добавлен новый интерфейс `CharSequence` для обеспечения общим интерфейсом `StringBuffer`, `StringBuilder`, `String`, `CharBuffer` и других похожих типов, которые настроены для реализации того интерфейса. В то же время, как `CharSequence` добавился к платформе, `String` получил перегрузку метода `contentEquals`, который берет `CharSequence`.

Получившаяся перегрузка очевидно нарушает инструкции данной статьи, но она не причиняет вреда, до тех пор пока оба перегруженных метода делают одно и то же при запуске на одной и той же ссылке на объект. Программисты могут не знать, какая именно перезагрузка запускается, но никаких последствий это не вызывает, пока они ведут себя одинаково. Стандартный способ убедиться, что поведение будет именно таким, — это перейти от более специфичной перегрузки к более общей:

```
public boolean contentEquals(StringBuffer sb) {  
    return contentEquals((CharSequence) sb);  
}
```

Хотя библиотеки для платформы Java в основном следуют приведенным здесь советам, все же можно найти несколько мест, где они нарушаются. Например, класс `String` передает два перезагруженных статических метода генерации `valueOf(char[])` и `valueOf(Object)`, которые, получив ссылку на один и тот же объект, выполняют совершенно разную работу. Этому нет четкого объяснения, и относиться к данным методам следует как к аномалии, способной вызвать настоящую неразбериху.

Подведем итоги. То, что вы можете осуществлять перезагрузку методов, еще не означает, что вы должны это делать. Обычно лучше воздерживаться от перезагрузки методов, которые имеют несколько сигнатур с одинаковым количеством параметров. Но иногда, особенно при наличии вызова конструкторов, невозможно следовать этому совету. Тогда постарайтесь избежать ситуации, при которой благодаря приведению типов один и тот же набор параметров может использоваться разными вариантами перезагрузки. Если такой ситуации избежать нельзя, например, из-за того, что вы переделываете уже имеющийся класс под реализацию нового интерфейса, удостоверьтесь в том, что все варианты перезагрузки, получая одни и те же параметры, будут вести себя одинаковым образом. Если же вы этого не сделаете, программисты не смогут эффективно использовать перезагруженный метод или конструктор и не смогут понять, почему он не работает.

Статья
Ч2

Используйте varargs с осторожностью

В версии 1.5 были добавлены методы `vararg`, формально известные под названием методы нагрузки переменных аргументами

(*variable arity methods*) [JLS, 8.4.1]. Методы varargs принимают ноль или более аргументов определенного типа. Vararg сначала создает массив, размер которого равен числу аргументов, переданных в месте вызова, затем помещает значения аргументов в массив и, наконец, передает массив методу.

Например, вот метод varargs, который берет последовательность аргументов int и возвращает их сумму. Как вы можете ожидать, значение sum(1, 2, 3) равно 6 и значение sum() равно 0:

```
// Простое использование varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

Иногда можно написать метод, которому требуется один или более аргументов некоего типа больше нуля. Например, предположим, вы хотите рассчитать минимальное количество аргументов int. Эта функция не очень хорошо определена, если клиент не передает никаких аргументов. Вам необходимо проверить длину массива при запуске:

```
// Неверное использование varargs для передачи одного или более
// аргументов!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

У этого решения несколько проблем. Самая серьезная заключается в том, что если клиент запустит этот метод без аргументов,

то произойдет ошибка при выполнении. Другая проблема заключается в том, что оно ужасно выглядит. Вам необходимо включить явную проверку корректности для args, и вы не можете использовать цикл for-each, если только не инициализируете min в Integer.MAX_VALUE, что также ужасно выглядит.

Но, к счастью, есть лучший способ достигнуть желаемого эффекта. Объявите метод таким образом, чтобы он принимал два параметра, один нормальный параметр определенного типа и один параметр varargs этого же типа. Это решение исправит все недостатки предыдущего:

```
// Правильный способ использовать varargs для передачи одного
// или более аргументов
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

Как видно из этого примера, использование varargs эффективно там, где вам действительно требуется метод с переменным числом аргументов. Методы varargs были созданы для метода printf, который появился в платформе версии 1.5, и для средств отражения (статья 53), которые были модифицированы для использования преимуществ varargs в этой версии. И printf и отражение очень много выиграли от появления varargs.

Вы можете изменить существующий метод, берущий массив как окончательный параметр, с тем чтобы он принимал параметры varargs вместо него, не затрагивая существующих клиентов. Однако то, что вы это можете, вовсе не значит, что это следует делать. Рассмотрим случай с Array.asList. Этот метод никогда не разрабатывался для того, чтобы собирать несколько аргументов в список, но переработать его, чтобы он мог это делать после появления

varargs, выглядит довольно неплохой идеей. В результате стало возможно сделать это:

```
List<String> homophones = Arrays.asList("to", "too", "two");
```

Данное решение работает, но применять его будет большой ошибкой. До появления релиза 1.5 была общая идиома для распечатывания содержимого массива:

```
// Устаревшая идиома для печати массива!
System.out.println(Arrays.asList(myArray));
```

Она была необходима, потому что массивы наследовали свою реализацию `toString` от `Object`, таким образом, вызов `toString` непосредственно на массиве выдает бесполезную строку, таюю как `[Ljava.lang.Integer;@3e25a5]`. Идиома работала только на массивах типов объектных ссылок, но, если вы случайно попробуете ее на массивах примитивов, программа не будет компилироваться. Например, эта программа:

```
public static void main(String[] args) {
    int[] digits = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 };
    System.out.println(Arrays.asList(digits));
}
```

сгенерировала бы ошибку в версии 1.4:

```
Va.java:6: asList(Object[]) in Arrays can't be applied to (int[])
System.out.println(Arrays.asList(digits));
^
```

Из-за неудачного решения переработать `Arrays.asList` как метод `varargs` в версии 1.5 эта программа теперь компилируется без ошибок или предупреждений. Запуск программы тем не менее приводит к результату, который не является ожидаемым и понятным: `[[I@3e25a5]`. Метод `Arrays.asList`, теперь «улучшенный», чтобы он мог использовать `varargs`, собирает объектные ссылки на массив целых чисел `digits` в одноэлементный массив массивов и сворачивает его в экземпляр `List<int[]>`. Распечатка этого списка приводит

к тому, что запускается `toString` на своем единственном элементе, массиве `int`, с неудачным результатом, описанным выше.

С другой стороны, идиома `Arrays.asList` теперь устарела для перевода массивов в строки, а современная идиома имеет куда больше возможностей. Также в версии 1.5 классу `arrays` дается полное дополнение методов `Arrays.toString` (не методов `varargs`!), созданных специально для перевода массивов любого типа в строки. Если вы используете `Arrays.toString` вместе `Arrays.asList`, программа даст нужный результат:

```
// Правильный способ напечатать массив
System.out.println(Arrays.toString(myArray));
```

Вместо модификации `Array.asList` было бы лучше добавить метод к `Collections` специально с целью сбора аргументов в список:

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

Такой метод дает возможность сбора не причиняя вреда проверке типов в уже существующем методе `Array.asList`.

Урок понятен. Не стоит модифицировать каждый метод, у которого есть конечный параметр массива; используйте `varargs`, только когда вызов определяет действительно с последовательностью значений с переменным числом аргументов.

Особенно подозрительны две сигнатуры методов:

```
ReturnType1 suspect1(Object... args) { }
<T> ReturnType2 suspect2(T... args) { }
```

Методы с любой из этих сигнатур примут любой список параметров. Любая проверка типов на этапе компиляции, которую вы делали до модификации, будет утрачена, как было показано на примере, что происходит с `Arrays.asList`.

Нужно быть осторожными при использовании возможностей `varargs` в ситуациях, где критична производительность. Каждый запуск метода `varargs` приводит к размещению и инициализации массива. Если вы понимаете, что не можете позволить себе такие рас-

ходы, но вам нужна гибкость методов varargs, есть шаблон, который позволит решить вам вашу задачу. Предположим, вы определили, что 95% обращений к методу содержит 3 и менее параметров. Тогда объявим 5 перегрузок метода, каждый с количеством обычных параметров от нуля до трех, и один метод varargs для использования, когда количество аргументов превысит 3:

```
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 затраты на создание массивов составят лишь 5% от всех вызовов в случаях, где количество параметров будет больше трех. Как и в большинстве случаев оптимизации производительности, этот прием обычно неприемлем, но там, где он нужен, становится спасательным кругом.

Класс EnumSet использует данный прием для своих методов статической генерации для уменьшения затрат на создание наборов перечислимых типов до абсолютного минимума. Было нормально использовать данный прием, потому что наборы перечислимых типов предоставляют конкурентоспособную замену битовым полям (статья 32).

Подведем итоги. Методы varargs удобны в плане определения методов, которые требуют переменного количества аргументов, но не стоит переусердствовать в их использовании. Они могут выдавать запутанные результаты в случаях неправильного использования.

Статья
Ч3

Возвращайте массив нулевой длины, а не null

Нередко встречаются методы, имеющие следующий вид:

```
private List cheesesInStock = ...;
/**
```

```

* @return массив, содержащий все сыры, имеющиеся в магазине,
* или null, если сыров для продажи нет.
*/
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
        return null;
}

}

```

Нет причин рассматривать как особый случай ситуацию, когда в продаже нет сыра. Это требует от клиента написания дополнительного кода для обработки возвращаемого методом значения null, например:

```

Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");

```

вместо простого:

```

if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");

```

Такого рода многоречивость необходима почти при каждом вызове метода, который вместо массива нулевой длины возвращает null. Это чревато ошибками, так как разработчик клиента мог и не написать специальный код для обработки результата null. Ошибка может оставаться незамеченной годами, поскольку подобные методы, как правило, возвращают один или несколько объектов. Следует еще упомянуть о том, что возврат null вместо массива приводит к усложнению самого метода, возвращающего массив.

Иногда можно услышать возражения, что возврат значения null предпочтительнее возврата массива нулевой длины потому, что это позволяет избежать расходов на размещение массива в памяти. 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 яв-

ляется основной причиной падения производительности (статья 55). Во-вторых, при каждом вызове метода, который не возвращает записей, клиенту можно передавать один и тот же массив нулевой длины, поскольку любой массив нулевой длины неизменяем, а неизменяемые объекты доступны для совместного использования (статья 15). На самом деле именно это и происходит, когда вы применяете стандартную идиому для выгрузки элементов из коллекции в массив с контролем типа:

```
// Правильный способ вывести массив из коллекции
private final List cheesesInStock = ...;
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
/**
 * @return массив, содержащий все сыры, имеющиеся в магазине
 */
public Cheese[] getCheeses() {
    return (Cheese[]) cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

В этой идиоме константа в виде массива нулевой длины передается методу `toArray` для того, чтобы показать, какой тип он должен возвратить. Обычно метод `toArray` выделяет место в памяти для возвращаемого массива, однако, если коллекция пуста, она размещается во входном массиве, а спецификация `Collection.toArray(Object[])` дает гарантию, что, если входной массив будет достаточно вместителен, чтобы содержать коллекцию, возвращен будет именно он. Поэтому представленная идиома никогда не будет сама размещать в памяти массив нулевой длины.

Аналогичным образом можно заставить метод с коллекцией значений возвращать ту же самую пустую неизменяемую коллекцию каждый раз, когда это требуется. Методы `Collections.emptySet`, `emptyList` и `emptyMap` дают в точности то, что нам нужно, как показано ниже:

```
// Правильный способ возврата копии коллекции.
public List<Cheese> getCheeseList() {
```

```
if (cheesesInStock.isEmpty())
    return Collections.emptyList(); // Always returns same list
else
    return new ArrayList<Cheese>(cheesesInStock);
}
```

Подведем итоги. **Нет никаких причин для того, чтобы работающий с массивами метод возвращал значение null, а не массив нулевой длины.** Такая идиома, по-видимому, проистекает из языка программирования C, где длина массива возвращается отдельно от самого массива. В языке C бесполезно выделять память под массив нулевой длины.

Статья ЧЧ

Для всех открытых элементов API пишите doc-комментарии

Если API будет использоваться, его нужно описывать. Обычно документация к API пишется вручную, и поддержание соответствия между документацией и программным кодом — весьма неприятная работа. Среда программирования Java облегчает эту задачу с помощью утилиты, называемой *Javadoc*. Она автоматически генерирует документацию к API, отталкиваясь от исходного текста программы, дополненного специальным образом оформленными комментариями к документации (*documentation comment*), которые чаще называют *doc-комментариями* (*doc comment*). Утилита Javadoc предлагает простой, эффективный способ документирования API и используется повсеместно.

Если вы еще не знакомы с соглашениями для doc-комментариев, то обязаны их изучить. Эти соглашения не являются частью языка программирования Java, но де-факто они образуют свой API, который обязан знать каждый программист. Соглашения описаны на веб странице Sun *How to Write Doc Comments* [учебник Javadoc]. Хотя эта страница и не обновлялась с момента выхода версии 1.4, она все

еще остается бесценным ресурсом. Два важных тега были добавлены в Javadoc с версией 1.5, {@literal} и {@code}[Javadoc-5.0]. Эти теги и обсуждаются в данной статье.

Чтобы должным образом документировать API, следует предварять doc-комментарием каждую предоставляемую пользователям декларацию класса, интерфейса, конструктора, метода и поля. Если класс является сериализуемым, нужно документировать также его сериализуемую форму (статья 75).

Единственное исключение обсуждается в конце статьи. Если doc-комментарий отсутствует, самое лучшее, что может сделать Javadoc, — это воспроизвести декларацию элемента API как единственную возможную для него документацию. Работа с API, у которого нет комментариев к документации, чревата ошибками. Чтобы создать программный код, приемлемый для сопровождения, вы должны написать doc-комментарии даже для тех классов, интерфейсов, конструкторов, методов и полей, которые не предоставляются пользователям.

Doc-комментарий для метода должен лаконично описывать соглашения между этим методом и его клиентами. Соглашение должно оговаривать, что делает данный метод, а не как он это делает. Исключение составляют лишь методы в классах, предназначенных для наследования (статья 17). В doc-комментарии необходимо перечислить все *предусловия* (*precondition*), т.е. утверждения, которые должны быть истинными 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 тегами @throws для необработанных исключений. Каждое необработанное исключение соответствует нарушению некоего предусловия. Предусловия также могут быть указаны вместе с параметрами, которых они касаются, в соответствующих тегах @param.

Помимо пред- и постусловий для методов должны быть также документированы любые побочные эффекты. Побочный эффект

(side effect) — это поддающееся наблюдению изменение состояния системы, которое является неявным условием для достижения постусловия. Например, если метод запускает фоновый поток, это должно быть отражено в документации. Наконец, комментарии к документации должны описывать *безопасность класса при работе с потоками (thread safety)*, которая обсуждается в статье 70.

В целях полного описания соглашений doc-комментарий для метода должен включать в себя: тег @param для каждого параметра, тег @return, если только метод не возвращает тип void, и тег @throws для каждого исключения, инициируемого этим методом, как обработанного, так и необработанного (статья 62). По соглашению, текст, который следует за тегом @param или @return, представляет собой именную конструкцию (noun phrase — термин грамматики английского языка), описывающую значение данного параметра или возвращаемое значение. Текст, следующий за тегом @throws, должен состоять из слова if и именной конструкции, описывающей условия, при которых инициируется данное исключение. Иногда вместо именных конструкций используются арифметические выражения. Все эти соглашения иллюстрирует следующий краткий doc-комментарий из интерфейса List:

```
/**
 * Возвращает элемент, который занимает заданную позицию в данном списке.
 *
 * <p>Этот метод <i>не</i> дает гарантии, что будет выполняться в постоянное время.
 * В некоторых реализациях он будет выполнять во время, пропорциональное положению его элементов.
 *
 * @param index индекс элемента, который нужно возвратить;
 * индекс должен быть меньше размера списка и неотрицательным.
 * @return элемент, занимающий в списке указанную позицию.
 * @throws IndexOutOfBoundsException if индекс лежит вне диапазона
 * ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

Заметим, что в этом doc-комментарии используются метасимволы и теги языка HTML. Утилита Javadoc преобразует doc-комментарии в код HTML, и любые содержащиеся в doc-комментариях элементы HTML оказываются в полученном HTML-документе. Иногда программисты заходят настолько далеко, что встраивают в свои doc-комментарии таблицы HTML, хотя это не является общепринятым.

Также отметим использование Javadoc тега {@code} вокруг фрагмента кода в выражении @throw. Это необходимо для выполнения двух задач: приводит к тому, что фрагмент кода отображается шрифтом кода (code font) и он скрывает обработку HTML разметки вложенных Javadoc тегов в фрагменте кода. Последнее свойство — это то, что нам дает возможность использования знака меньше чем (<) в фрагменте кода, даже если это метасимвол HTML. До версии 1.5 фрагменты кода включались в doc-комментарии, используя HTML. Больше нет необходимости использовать HTML теги `<code>` или `<tt>` в doc-комментариях: Javadoc тег {@code} использовать предпочтительнее, так как он избавляет нас от необходимости избегать использование метасимволов HTML. Для добавления многострочного примера кода в doc-комментарий используйте тег Javadoc {@code} внутри HTML тега `<pre>`. Другими словами, пример многострочного кода должен начинаться с `<pre> { @code` и заканчиваться символами } `</pre>`.

Наконец, отметим появление в doc-комментарии слова «this». По соглашению, слово «this» всегда ссылается на тот объект, которому принадлежит вызываемый метод, соответствующий данному doc-комментарию.

Не забудьте, что вам нужно предпринять специальные действия, чтобы сгенерировать документацию, содержащую метасимволы HTML, такие как знаки <, >, &. Лучший способ вставить эти символы в документацию — это заключить их в тег {@literal}, который скрывает обработку HTML разметки во вложенном Javadoc теге. Это как с тегом {@code}, за исключением того, что он не переводит текст в шрифт кода. Например, фрагмент Javadoc:

* The triangle inequality is {@literal |x + y| < |x| + |y|}.

Генерирует следующую документацию: «Неравенство треугольника $|x + y| < |x| + |y|$.» Тег {@literal} мог быть помещен только вокруг знака меньше чем, вместо того чтобы все неравенство, причем документация получилась бы такой же, но тогда doc-комментарий было бы сложнее читать в исходном коде. Этот пример иллюстрирует общий принцип, что doc-комментарии должны быть читаемы не только в сгенерированной документации, но и в исходном коде. Если и того и другого достичь не удается, то предпочтительнее, конечно, читаемость сгенерированной документации.

Первым предложением любого doc-комментария является общее описание (*summary description*) того элемента, к которому этот комментарий относится. Общее описание должно позиционироваться как описывающее функции соответствующей сущности. Во избежание путаницы никакие два члена или конструктора в одном классе или интерфейсе не должны иметь одинакового общего описания. Особое внимание обращайте на перезагруженные методы, описание которых часто хочется начать с одного и того же предложения.

Внимательно следите за тем, чтобы в первом предложении doc-комментария не было точки. В противном случае общее описание будет завершено прежде времени. Например, комментарий к документации, который начинается с фразы «A college degree, such as B.S., M.S., or Ph.D.», приведет к появлению в общем описании фразы «A college degree, such as B.» Во избежание подобных проблем лучше не использовать в общем описании сокращений и десятичных дробей. Для получения символа точки нужно заменить его числовым представлением (*numeric encoding*) «'.'». Хотя это работает, исходный текст программы приобретает не слишком красивый вид:

```
/**  
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.  
 * College is a fountain of knowledge where many go to drink.*/  
public class Degree { }
```

Тезис, что первым предложением в doc-комментарии является общее описание, отчасти вводит в заблуждение. По соглашению, оно

редко бывает законченным предложением. Общее описание методов и конструкторов должно представлять собой глагольную конструкцию (*verb phrase* — термин грамматики английского языка), которая описывает операцию, осуществляемую этим методом. Например:

- `ArrayList(int initialCapacity)` — создает пустой список с заданной начальной емкостью.
- `Collection.size()` — возвращает количество элементов в указанной коллекции.

Общее описание классов, интерфейсов и полей должно быть именной конструкцией, описывающей сущность, которую представляет экземпляр этого класса, интерфейса или само поле. Например:

- `TimerTask` — задача, которая может быть спланирована классом `Timer` для однократного или повторяющегося исполнения.
- `Math.PI` — значение типа `double`, наиболее близкое к числу «пи» (отношение длины окружности к ее диаметру).

Три возможности, добавленные к языку в версии 1.5, требуют особого отношения в doc-комментариях: средства обобщенного программирования, перечислимые типы и аннотации. **Когда вы пишете документацию по обобщенным типам или методам, обязательно документируйте все параметры типа:**

```
/**  
 * An object that maps keys to values. A map cannot contain  
 * duplicate keys; each key can map to at most one value.  
 *  
 * (Remainder omitted)  
 *  
 * @param <K> the type of keys maintained by this map  
 * @param <V> the type of mapped values  
 */  
public interface Map<K, V> {  
    ... // Remainder omitted  
}
```

При документировании перечислимого типа убедитесь, что документированы константы, тип и любой открытый метод. Обратите внимание, что вы можете поместить весь doc-комментарий в одну строку, если он краткий:

```
/**  
 * An instrument section of a symphony orchestra.  
 */  
public enum OrchestraSection {  
    /** Woodwinds, such as flute, clarinet, and oboe. */  
    WOODWIND,  
    /** Brass instruments, such as french horn and trumpet. */  
    BRASS,  
    /** Percussion instruments, such as timpani and cymbals */  
    PERCUSSION,  
    /** Stringed instruments, such as violin and cello. */  
    STRING;  
}
```

При документировании аннотационных типов убедитесь, что документирован любой член, так же как и сам тип. Документируйте члены фразой с существительным, как будто они являются полями. Для общего описания типа используйте фразу с глаголом, которая говорит, что это значит, когда элемент программы содержит аннотацию такого типа:

```
/**  
 * Indicates that the annotated method is a test method that  
 * must throw the designated exception to succeed.  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface ExceptionTest {  
    /**  
     * The exception that the annotated test method must throw  
     * in order to pass. (The test is permitted to throw any  
     * subtype of the type described by this class object.)  
    */  
}
```

```
 */  
Class<? extends Throwable> value();  
}
```

В версии 1.5 doc-комментарии пакетного уровня следует помещать в файл, называемый `package-info.java`, вместо `package.html`. В дополнение к doc-комментарию пакетного уровня `package-info.java` должен содержать декларацию пакета и может содержать аннотацию пакета для данной декларации.

Двумя аспектами экспортируемого API класса, которые зачастую игнорируются, являются потоковая безопасность и возможность сериализации. Является или нет класс безопасным с точки зрения потока, должно быть документировано на уровне потока, как описано в статье 70. Если класс может быть сериализуем, нужно документировать его сериализованную форму, как описано в статье 75.

Утилита Javadoc получила возможность «наследовать» комментарии к методам. Если метод не имеет doc-комментария, Javadoc находит среди приемлемых наиболее близкий doc-комментарий, отдавая при этом предпочтение интерфейсам, а не суперклассам. Подробности алгоритма поиска комментариев можно найти в *The Javadoc Reference Guide* [Javadoc-ref]. Вы можете наследовать части doc-комментариев их супертипов, используя тег `{@inheritDoc}`.

Это означает, что классы теперь могут заимствовать doc-комментарии из реализуемых ими интерфейсов вместо того, чтобы копировать комментарии у себя. Такая возможность способна сократить или вовсе снять бремя поддержки многочисленных наборов почти идентичных doc-комментариев, но у нее есть несколько ограничений. Их детали не вошли в книгу.

Простейший способ уменьшить вероятность появления ошибок в комментариях к документации — это пропустить HTML-файлы, сгенерированные утилитой Javadoc, через программу проверки кода HTML (*HTML validity checker*). При этом будет обнаружено множество случаев неправильного использования тегов и метасимволов HTML, которые нужно исправить. Некоторые из программ провер-

ки кода HTML доступны в Интернете для загрузки, или вы можете проверить HTML код онлайн [[W3C-validator](#)].

Следует добавить еще одно предостережение, связанное с комментариями к документации. Комментарии должны сопровождать все элементы внешнего API, но этого не всегда достаточно. Для сложных API, состоящих из множества взаимосвязанных классов, комментарии к документации часто требуется дополнять внешним документом, описывающим общую архитектуру данного API. Если такой документ существует, то комментарии к документации в соответствующем классе или пакете должны на него ссылаться.

Соглашения, описанные в данной статье, составляют основы. Более детальное руководство, как писать doc-комментарии, содержится в [How To Write Doc Comments \[Javadoc-guide\]](#), выпущенном компанией Sun. Имеются IDE плагины, которые проверяют следование многим из этих правил [[Burn01](#)].

Подведем итоги. Комментарии к документации — самый лучший, самый эффективный способ документирования API. Написание комментариев нужно считать обязательным для всех элементов внешнего API. Выберите стиль, который не противоречит стандартным соглашениям. Помните, что в комментариях к документации можно использовать любой код HTML, причем метасимволы HTML необходимо маскировать escape-последовательностями.

Г л а в а 8

Общие вопросы программирования

Данная глава посвящена обсуждению основных элементов языка Java. В ней рассматриваются интерпретация локальных переменных, использование библиотек и различных типов данных, а также две выходящие за рамки языка возможности: *отражение (reflection)* и *машинозависимые методы (native method)*. Наконец, обсуждаются оптимизация и соглашения по именованию.

Статья
Ч5

**Сводите к минимуму
область видимости
локальных переменных**

Эта статья по своей сути схожа со статьей 13 «Сводите к минимуму доступность классов и членов». Сужая область видимости локальных переменных, вы повышаете удобство чтения и сопровождения вашего кода, сокращаете вероятность возникновения ошибок.

Старые языки программирования, такие как C, указывают, что локальные переменные должны декларироваться в начале блока. И программисты продолжают придерживаться этого порядка, хотя

от него уже нужно отказываться. Напомним, что язык программирования Java позволяет объявлять переменную в любом месте, где может стоять оператор.

Самый сильный прием сужения области видимости локальной переменной заключается в декларировании ее в том месте, где она впервые используется. Декларация переменной до ее использования только засоряет программу: появляется еще одна строка, отвлекающая читателя, который пытается разобраться в том, что делает программа. К тому моменту, когда переменная применяется, читатель может уже не помнить ни ее тип, ни начальное значение. Если программа совершенствуется и переменная больше не нужна, легко забыть убрать ее декларацию, если та находится далеко от места первого использования переменной.

К расширению области видимости локальной переменной приводит не только слишком раннее, но и слишком позднее ее декларирование. Область видимости локальной переменной начинается в том месте, где она декларируется, и заканчивается с завершением блока, содержащего эту декларацию. Если переменная декларирована за пределами блока, где она используется, то она остается видимой и после того, как программа выйдет из этого блока. Если переменная случайно была использована до или после области, в которой она должна была применяться, последствия могут быть катастрофическими.

Почти каждая декларация локальной переменной должна содержать инициализатор. Если у вас недостаточно информации для правильной инициализации переменной, вы должны отложить декларацию до той поры, пока она не появится. Исключение из этого правила связано с использованием операторов `try/catch`. Если для инициализации переменной применяется метод, инициирующий появление обрабатываемого исключения, то инициализация переменной должна осуществляться внутри блока `try`. Если переменная должна использоваться за пределами блока `try`, декларировать ее следует перед блоком `try`, там, где она еще не может быть «правильно инициализирована» (статья 53).

Цикл предоставляет уникальную возможность для сужения области видимости переменных. Цикл `for` позволяет объявлять переменные цикла (*loop variable*), ограничивая их видимость ровно той областью, где они нужны. (Эта область состоит из собственно тела цикла, а также из предшествующих ему полей инициализации, проверки и обновления.) Следовательно, если после завершения цикла значения его переменных не нужны, предпочтение следует отдавать циклам `for`, а не `while`.

Представим, например, предпочтительную идиому для организации цикла по некой коллекции (статья 46):

```
// Не используются циклы for-each или средства обобщенного
// программирования до версии 1.51
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next());
}
```

Для пояснения, почему данный цикл `for` предпочтительнее более очевидного цикла `while`, рассмотрим следующий фрагмент кода, в котором содержатся два цикла `while` и одна ошибка:

```
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 ошибку копирования фрагмента программы: инициализируется новая переменная цикла `i2`, но используется старая `i`, которая, к сожалению, остается в поле видимости. Полученный код компилируется без замечаний и выполняется без инициализации исключительных ситуаций, только вот делает не то, что нужно. Вместо того чтобы организовывать итерацию по `c2`, второй

цикл завершается немедленно, создавая ложное впечатление, что коллекция `c2` пуста. И поскольку программа ничего об этой ошибке не сообщает, та может оставаться незамеченной долгое время.

Если бы аналогичная ошибка копирования была допущена при применении цикла `for`, полученный код не был бы даже скомпилирован. Для той области, где располагается второй цикл, переменная первого цикла уже была бы за пределами видимости:

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}

...
// Ошибка компиляции - символ i не может быть идентифицирован
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
    doSomething(i2.next());
}
```

Более того, если вы пользуетесь идиомой цикла `for`, уменьшается вероятность того, что вы допустите ошибку копирования, поскольку нет причин использовать в двух этих циклах различные названия переменных. Эти циклы абсолютно независимы, а потому нет никакого вреда от повторного применения названия для переменной цикла. На самом деле это даже стильно.

Идиома цикла `for` имеет еще одно преимущество перед идиомой цикла `while`, хотя и не столь существенное. Идиома цикла `for` короче на одну строку, что помогает при редактировании уместить содержащий ее метод в окне фиксированного размера и повышает удобство чтения.

Приведем еще одну идиому цикла для просмотра списка, которая минимизирует область видимости локальных переменных:

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    doSomething(i);
}
```

По поводу этой идиомы важно заметить, что у нее есть две переменные цикла: `i` и `n`, и обе имеют абсолютно правильную область

видимости. Вторая переменная `p` используется для хранения ограничений первой, тем самым избегая дополнительных затрат на расчеты по каждой итерации. Как правило, вам следует использовать эту идиому, если проверка цикла включает в себя запуск метода, гарантирующий возврат одного и того же результата по каждой итерации.

Последний прием, позволяющий уменьшить область видимости локальных переменных, заключается в **создании небольших, четко позиционированных методов**. Если в пределах одного и того же метода вы сочетаете две операции, то локальные переменные, относящиеся к одной из них, могут попасть в область видимости другой. Во избежание этого разделите метод на два, по одному методу для каждой операции.

Статья
Ч6

Предпочитайте использование цикла `for-each`

До версии 1.5 предпочтительнее было использовать следующую идиому для итерации коллекции:

```
// Более не является предпочтительной идиомой для итерации коллекции!
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next()); // (No generics before 1.5)
}
```

Так выглядела наиболее предпочтительная идиома для итерации массива:

```
// Более не является предпочтительной идиомой для итерации массива!
for (int i = 0; i < a.length; i++) {
    doSomething(a[i]);
}
```

Эти идиомы лучше, чем циклы `while` (статья 45), но они несовершенны. Итератор и переменная индекса путаются друг с другом.

В дальнейшем они предоставлят возможность для ошибки. Итератор встречается трижды в каждом цикле и переменная индекса четыре раза, что дает большую возможность использовать неверную переменную. Если вы так сделаете, нет никакой гарантии, что компилятор сможет заметить проблему.

Цикл `for-each`, представленный в версии 1.5, избавляет нас от этой путаницы и возможности для ошибки, полностью скрывая итератор или переменную индекса. В результате получается идиома, подходящая и для коллекций, и для массивов:

```
// Предпочтительная идиома для итерации коллекций или массивов
for (Element e : elements) {
    doSomething(e);
}
```

Двоеточие следует читать как «в». Следовательно, цикл будет читаться как «для каждого элемента `e` в элементах». Обратите внимание, что использование цикла `for-each` не наносит никакого ущерба производительности, даже для массивов. На самом деле данный цикл может предложить небольшое преимущество в производительности над простым циклом `for` в некоторых обстоятельствах, так как он подсчитывает ограничения индекса массива только один раз. В то время как вы можете сделать это вручную (статья 45), программисты так не делают.

Преимущество цикла `for-each` над циклом `for` даже больше, когда речь заходит о вложенных итерациях нескольких коллекций. Вот какую общую ошибку делают, когда пытаются выполнить вложенную итерацию двух коллекций:

```
// Можете 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 }

...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());
```

```
List<Card> deck = new ArrayList<Card>();
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. Многие эксперты в программировании делают эту ошибку. Проблема в том, что метод `next` вызывается слишком много раз на итераторе для внешней коллекции (`suits`). Он должен вызываться из внешнего цикла, так чтобы на каждый элемент приходился один вызов, но вместо этого он вызывается из внутреннего цикла, потому вызывается по одному разу на каждую карточку. После того как у вас закончатся костюмы, цикл выведет ошибку `NoSuchElementException`.

Если вам на самом деле не повезет и размер внешней коллекции есть кратное число размера внутренней коллекции — возможно, потому что эта одна и та же коллекция, — то цикл завершится нормально, не он не будет делать то, что вы хотите. Например, рассмотрим неудачную попытку напечатать все возможные комбинации пары костей:

```
// Та же ошибка, но с другими симптомами!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }

...
Collection<Face> faces = Arrays.asList(Face.values());
for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

Данная программа не выводит ошибку, но она печатает только шесть двойных значений (от «1:1» до «6:6»), вместо ожидаемых 36 комбинаций.

Для решения данной проблемы вам надо добавить переменную в диапазоне внешнего цикла, чтобы хранить внешние элементы:

```
// Решено, но выглядит ужасно — 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 вложенный цикл `for-each`, то проблема просто исчезает. Получившийся в результате код настолько краток, насколько вы хотите:

```
// Предпочитательная идиома для вложенной итерации коллекций и массивов
for (Suit suit : suits)
for (Rank rank : ranks)
deck.add(new Card(suit, rank));
```

Цикл `for-each` не только дает вам возможность итерации коллекций и массивов, но и позволяет выполнять итерации любых объектов, реализующих интерфейс `Iterable`. Этот простой интерфейс, содержащий лишь один метод, был добавлен к платформе вместе с циклом `for-each`. Вот как он выглядит:

```
public interface Iterable<E> {
// Returns an iterator over the elements in this iterable
Iterator<E> iterator();
}
```

Не так сложно реализовать интерфейс `Iterable`. Если вы пишете тип, представляющий группу элементов, заставьте его реализовать `Iterable`, даже если вы решите не реализовывать `Collection`. Это позволит вашим пользователям применить итерацию на типах с использованием цикла `for-each`, за что они будут вам благодарны.

Подведем итоги. Цикл `for-each` обладает сильнейшими преимуществами по сравнению с традиционным циклом `for` по части ясности и профилактике ошибок. К сожалению, есть три ситуации, в которых вы *не можете использовать цикл `for-each`*:

1. **Фильтрация** — если вам требуется пройти через коллекцию и удалить выбранные элементы, тогда вам нужно использовать явный итератор, чтобы вы могли вызвать его метод `remove`.
2. **Преобразование** — если вам требуется пройти через список или массив и заменить некоторые или все значения его эле-

ментов, тогда вам нужен итератор списка или индекс массива, чтобы задать значение элемента.

3. Параллельная итерация — если вам необходимо пройти через несколько коллекций параллельно, тогда вам нужен явный контроль над итератором или переменной индекса, чтобы все итераторы или переменные индекса могли быть расширены в рамках строгой системы (как было непреднамеренно показано выше в примерах с картами и костями).

Если вы окажетесь в любой из этих ситуаций, используйте обычный цикл `for`, избегайте ловушек, описанных в данной статье, и будьте уверены, что вы делаете все, что можете.



Изучите библиотеки и пользуйтесь ими

Предположим, что нужно генерировать случайные целые числа в диапазоне от нуля до некоторой верхней границы. Столкнувшись с такой распространенной задачей, многие программисты написали бы небольшой метод примерно следующего содержания:

```
private static final Random rnd = new Random();
// Неправильно, хотя встречается часто
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

Неплохой метод, но он несовершенен: у него есть три недостатка. Первый состоит в том, что если n — это небольшая степень числа два, то последовательность генерируемых случайных чисел через очень короткий период начнет повторяться. Второй заключается в том, что если n не является степенью числа два, то в среднем некоторые числа будут получаться гораздо чаще других. Если n большое, указанный недостаток может проявляться довольно четко. Графически это де-

монстрируется следующей программой, которая генерирует миллион случайных чисел в тщательно подобранном диапазоне и затем печатает, сколько всего чисел попало 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 печатала бы число близкое к полутора миллионам, однако, запустив эту программу, вы обнаружите, что она печатает число близкое к 666 666. Две трети чисел, сгенерированных методом `random`, попадают в нижнюю половину диапазона!

Третий недостаток представленного метода `random` заключается в том, что он может, хотя и редко, потерпеть полное фиаско, выдавая результат, выходящий за пределы указанного диапазона. Это происходит потому, что метод пытается преобразовать значение, возвращенное методом `rnd.nextInt()`, в неотрицательное целое число, используя метод `Math.abs`. Если `nextInt()` вернул `Integer.MIN_VALUE`, то `Math.abs` также возвратит `Integer.MIN_VALUE`. Затем, если `n` не является степенью числа два, оператор остатка (`%`) вернет отрицательное число. Это почти наверняка вызовет сбой в вашей программе, и воспроизвести обстоятельства этого сбоя будет трудно.

Чтобы написать такой вариант метода `random`, в котором были бы исправлены все эти три недостатка, необходимо изучить генераторы линейных конгруэнтных псевдослучайных чисел, теорию чисел и арифметику дополнения до двух. К счастью, делать это вам не нужно, все это уже сделано для вас. Необходимый метод называется `Random.nextInt(int)`, он был добавлен в пакет `java.util` стандартной библиотеки в версии 1.2.

Нет нужды вдаваться в подробности, каким образом метод `nextInt(int)` выполняет свою работу (хотя любопытные личности могут изучить документацию или исходный текст метода). Старший инженер с подготовкой в области алгоритмов провел много времени за разработкой, реализацией и тестированием этого метода, а затем показал метод экспертам в данной области с тем, чтобы убедиться в его правильности. После этого библиотека прошла стадию предварительного тестирования и была опубликована, тысячи программистов широко пользуются ею в течение нескольких лет. До сих пор ошибок в указанном методе найдено не было. Но если какой-либо дефект обнаружится, он будет исправлен в следующей же версии. **Обращаясь к стандартной библиотеке, вы используете знания написавших ее экспертов, а также опыт тех, кто работал с ней до вас.**

Второе преимущество от применения библиотек заключается в том, что вам не нужно терять время на решение специальных задач, имеющих лишь косвенное отношение к вашей работе. Как и большинство программистов, вы должны тратить время на разработку своего приложения, а не на подготовку его фундамента.

Третье преимущество от использования стандартных библиотек заключается в том, что их производительность имеет тенденцию повышаться со временем, причем без каких-либо усилий с вашей стороны. Множество людей пользуется библиотеками, они применяются в стандартных промышленных тестах, поэтому организация, которая осуществляет поддержку этих библиотек, заинтересована в том, чтобы заставить их работать быстрее. Например, стандартная библиотека арифметических операций с многократно увеличенной точностью `java.math` была переписана в версии 1.3, что привело к впечатляющему росту ее производительности.

Со временем библиотеки приобретают новые функциональные возможности. Если в каком-либо классе библиотеки не хватает важной функции, сообщество разработчиков даст знать об этом недостатке. Платформа Java всегда развивалась при серьезной поддержке со стороны сообщества разработчиков.

Последнее преимущество от применения стандартных библиотек заключается в том, что ваш код соответствует господствующим в данный момент тенденциям. Такой код намного легче читать и сопровождать, его могут использовать множество разработчиков.

Учитывая все эти преимущества, логичным казалось бы применение библиотек, а не частных разработок, однако многие программисты этого не делают. Но почему? Может быть, потому, что они не знают о возможностях имеющихся библиотек. **С каждой следующей версией в библиотеки включается множество новых функций, и стоит быть в курсе этих новшеств.** Вы можете внимательно изучать соответствующую документацию в режиме online либо прочесть о новых библиотеках в самых разных книгах [J2SE-APIs, Chan00, Flanagan99, Chan98]. Библиотеки слишком объемны, чтобы просматривать всю документацию, однако **каждый программист должен хорошо знать** java.lang, java.util и в меньшей степени java.io. Остальные библиотеки изучаются по мере необходимости.

Обзор всех возможностей библиотек выходит за рамки данной статьи, однако некоторые из них заслуживают особого упоминания. В версии 1.2 в пакет java.util была добавлена архитектура *Collections Framework*. Она должна входить в основной набор инструментов каждого программиста. Collections Framework — унифицированная архитектура, предназначенная для представления и управления коллекциями и позволяющая манипулировать коллекциями независимо от деталей представления. Она сокращает объемы работ по программированию и в то же время повышает производительность. Эта архитектура позволяет достичь унифицированности несвязанных API, упрощает проектирование и освоение новых API, способствует повторному использованию программного обеспечения. Если вы хотите больше узнать, см. документацию на веб-сайте Sun или прочтайте учебник [Bloch06].

В версии 1.5 появился набор взаимозаменяемых утилит в пакете java.util.concurrent. Этот пакет содержит как высокоуровневые ути-

литы совместимости для упрощения многопоточного программирования, так и низкоуровневые взаимозаменяемые примитивы, что позволяет экспертам писать свои собственные высокоуровневые взаимозаменяемые абстракции. Высокоуровневая часть `java.util.concurrent` также должна быть частью набора каждого программиста (статьи 68, 69).

Иногда функция, заложенная в библиотеке, не отвечает вашим потребностям. И чем специфичнее ваши запросы, тем это вероятнее. Если вы изучили возможности, предлагаемые библиотеками в некоей области, и обнаружили, что они не соответствуют вашим потребностям, используйте альтернативную реализацию. В функциональности, предоставляемой любым конечным набором библиотек, всегда найдутся пробелы. И если необходимая вам функция отсутствует, у вас нет иного выбора, как реализовать ее самостоятельно.

Подведем итоги. Не изобретайте колесо. Если вам нужно сделать нечто, что кажется вполне обычным, в библиотеках уже может быть класс, который делает это. Вообще говоря, 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 тех же самых функций.

Статья
Ч8

Если требуются точные ответы, избегайте использования типов `float` и `double`

Типы `float` и `double` в первую очередь предназначены для научных и инженерных расчетов. Они реализуют *бинарную арифметику с плавающей точкой* (*binary floating-point arithmetic*), которая была тщательно выстроена с тем, чтобы быстро получать правильное приближение для широкого диапазона значений. Однако эти типы не дают точного результата, и в ряде случаев их нельзя использовать.

вать. **Типы float и double не подходят для денежных расчетов**, поскольку с их помощью невозможно представить число 0.1 (или любую другую отрицательную степень числа десять).

Например, у вас в кармане лежит \$1,03, и вы тратите 42 цента. Сколько денег у вас осталось? Приведем фрагмент наивной программы, которая пытается ответить на этот вопрос:

```
System.out.println(1.03 - .42);
```

Как ни печально, программа выводит 0.6100000000000001. И это не единственный случай. Предположим, что у вас в кармане есть доллар и вы покупаете 9 прокладок для крана по 10 центов за каждую. Какую сдачу вы получите?

```
System.out.println(1.00 - 9*.10);
```

Если верить этому фрагменту программы, то вы получите \$0.0999999999999995. Может быть, проблему можно решить, округлив результаты перед печатью? К сожалению, это срабатывает не всегда. Например, у вас в кармане есть доллар, и вы видите полку, где выстроены в ряд вкусные конфеты за 10, 20, 30 центов и т.д. вплоть до доллара. Вы покупаете по одной конфете каждого вида, начиная с той, что стоит 10 центов, и т.д., пока у вас еще есть возможность взять следующую конфету. Сколько конфет вы купите и сколько получите сдачи? Решим эту задачу следующим образом:

```
// Ошибка: использование плавающей точки для денежных // расчетов!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = .10; funds >= price; price += .10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

Запустив программу, вы выясните, что можете позволить себе три конфеты и у вас останется еще \$0.3999999999999999. Но это неправильный ответ! Правильный путь решения задачи заключается в **применении для денежных расчетов типов BigDecimal, int или long**. Представим простое преобразование предыдущей программы, которое позволяет использовать тип BigDecimal вместо double:

```
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)) {  
        itemsBought++;  
        funds = funds.subtract(price);  
    }  
    System.out.println(itemsBought + " items bought.");  
    System.out.println("Money left over: $" + funds);  
}
```

Запустив исправленную программу, вы обнаружите, что можете позволить себе четыре конфеты и у вас останется \$0,00. Это верный ответ. Однако тип BigDecimal имеет два недостатка: он не столь удобен и медленнее, чем простой арифметический тип. Последнее можно считать несущественным, если вы решаете единственную маленькую задачу, а вот неудобство может раздражать.

Вместо BigDecimal можно использовать int или long (в зависимости от обрабатываемых величин) и самостоятельно отслеживать положение десятичной точки. В нашем примере расчеты лучше производить не в долларах, а в центах. Продемонстрируем этот подход:

```
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(itemsBought + " items bought.");
System.out.println("Money left over: $" + funds);
}

```

Подведем итоги. Для вычислений, требующих точного результата, не используйте типы float и double. Если вы хотите, чтобы система сама отслеживала положение десятичной точки, и вас не пугают неудобства, связанные с отказом от простого типа, используйте BigDecimal. Применение этого класса имеет еще то преимущество, что он дает вам полный контроль над округлением: для любой операции, завершающейся округлением, предоставляется на выбор восемь режимов округления. Это пригодится, если вы будете выполнять экономические расчеты с жестко заданным алгоритмом округления. Если для вас важна производительность, 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, можете применить тип int. Если в значении не больше восемнадцати десятичных цифр, используйте тип long. Если же в значении более восемнадцати цифр, вам придется работать с BigDecimal.

*Статья
49*

Отдавайте предпочтение использованию обычных примитивных типов, а не упакованных примитивных типов

У языка Java есть система, состоящая из двух частей, — примитивные типы, такие как int, double и Boolean, и типы ссылок, такие как String и List. Каждый примитивный тип обладает соответствующим типом ссылок, называемых упакованными примитивными ти-

пами. Упакованные примитивные типы, соответствующие простым примитивным типам `int`, `double` и `Boolean`, являются `Integer`, `Double` и `Boolean`.

В версии 1.5 к языку добавилось преобразование между скалярными и упакованными типами. Как сказано в статье 5, эти возможности размывают, но не стирают границу между обычными примитивными типами и упакованными примитивными типами. Между ними есть реальные отличия, и, что важно, вам нужно всегда знать, какой из них вы сейчас используете, и нужно осторожно делать выбор, какой из них использовать.

Между обычными и упакованными примитивными типами существует три главных отличия. Во-первых, у обычных примитивных типов есть только их значения, в то время как у упакованных примитивных типов есть еще и идентичность, отличающаяся от их значения. Другими словами, экземпляры двух упакованных примитивных типов могут иметь одно и то же значение, но разные идентичности. Во-вторых, у обычных примитивных типов есть только функциональные значения, в то время как упакованные примитивные типы имеют только одно нефункциональное значение, `null`, в дополнение ко всем функциональным значениям соответствующих им обычных примитивных типов. И последнее — обычные примитивные типы более эффективны с точки зрения и времени и пространства, чем упакованные примитивные типы. Все эти три отличия могут создать реальные проблемы, если их использовать неосторожно.

Рассмотрим следующий компаратор, созданный для представления в возрастающем порядке значения `Integer`. (Вспоминаем, что метод компаратора `Compare` возвращает число — отрицательное, положительное или ноль, в зависимости от того, является ли первый аргумент меньше, равен или больше второго.). На практике вам не нужно писать компаратор, так как он реализует натуральный порядок в `Integer`, который можно получить и без компаратора, но пример получается интересный:

```
// Нерабочий компаратор - можете ли найти, где ошибка?
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
    public int compare(Integer first, Integer second) {
        return first < second ? -1 : (first == second ? 0 : 1);
    }
};
```

Этот компаратор кажется нормальным и пройдет много тестов. Например, его можно использовать с `Collections.sort`, чтобы правильно отсортировать список из миллионов элементов, вне зависимости от того, содержатся ли в списке дублирующие элементы. Но у этого компаратора существенные недостатки. Чтобы убедиться в этом, просто напечатайте значения `Natural.Order.compare(new Integer(42), newInteger(42))`. Оба экземпляра `Integer` представляют одно и то же значение **(42)**, так что значение этого выражения должно быть **0**, но оно на самом деле **1**, что отражает, что первое значение `Integer` больше, чем второе.

Итак, в чем же проблема? Первая проверка в `naturalOrder` срабатывает normally. Оценка выражения `first < second` приводит к тому, что экземпляр `Integer`, на который ссылаются `first` и `second`, становится автоматически распакован; т.е. он извлекает свои примитивные значения. Проверка продолжает проверять, является ли первый результат значения `int` меньше второго. Но предположим, что нет. Тогда следующая проверка проверяет выражение `first == second`, который выполняет сравнение идентичностей двух ссылок на объект. Если `first` и `second` ссылаются на различные экземпляры `Integer`, представляющие различные значения `int`, то это сравнение вернет значение `false` и компаратор ошибочно выведет значение **1**, которое означает, что первое значение `Integer` больше второго. **Использование оператора == на упакованных примитивных типах почти всегда неверно.**

Самый понятный способ решения проблемы — это добавить две локальные переменные для хранения примитивных значений `int`, соответствующих `first` и `second`, и выполнять все сравнения на этих

переменных. Это поможет избежать ошибочного сравнения их идентичностей:

```
Comparator<Integer> naturalOrder = new Comparator<Integer>() {  
    public int compare(Integer first, Integer second) {  
        int f = first; // Auto-unboxing  
        int s = second; // Auto-unboxing  
        return f < s ? -1 : (f == s ? 0 : 1); // No unboxing  
    }  
};
```

Теперь рассмотрим эту небольшую программу:

```
public class Unbelievable {  
    static Integer i;  
    public static void main(String[] args) {  
        if (i == 42)  
            System.out.println("Unbelievable");  
    }  
}
```

Нет, она не печатает `Unbelievable` — но что она делает, почти также странно. Она выдает ошибку `NullPointerException` при сравнении выражения (`i == 42`). Проблема в том, что `i` — это `Integer`, а не `int` и все поля ссылки на объекты, его начальное значение `null`. Когда программа проверяет выражение (`i == 42`), она сравнивает `Integer` и `int`. Почти в любом случае, когда вы смешиваете обычные и упакованные примитивные типы одной операцией, упакованный примитивный тип автоматически распаковывается, и это не вызывает ошибки. Если нулевая ссылка на объект распаковывается, то вы получите `NullPointerException`. То, что демонстрирует данная программа, может случиться где угодно. Исправить программу также просто, как объявить, что `i` является `int`, а не `Integer`.

Наконец, рассмотрим программу (статья 5):

```
// Ужасно медленная программа! Можете 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 должны быть, потому что она случайно декларирует, что локальная переменная (`sum`) является упакованным примитивным типом `Long`, а не обычным примитивным типом `long`. Программа компилируется без ошибок или предупреждений, и переменная постоянно упаковывается и распаковывается, снижая производительность.

У всех трех программ, которые мы обсудили, одна и та же проблема: программист проигнорировал различия между обычными примитивными типами и упакованными примитивными типами, что привело к неприятным последствиям. В первых двух случаях это привело к ошибкам при запуске, в третьем случае — серьезному снижению производительности

Итак, когда же надо использовать упакованные примитивные типы? У них есть несколько разрешенных видов использования. 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 типах с параметрами (глава 5), потому что язык не позволяет вам использовать обычные примитивные типы. Например, вы не можете объявить переменную, чтобы она имела тип `ThreadLocal<int>`, вместо этого вам нужно использовать `ThreadLocal<Integer>`. Наконец, вы должны использовать упакованные примитивные типы при запуске рефлексивных методов (статья 53).

Подведем итоги. Используйте обычные примитивные типы вместо упакованных примитивных типов, когда у вас есть выбор. Обычные примитивные типы проще и быстрее. Если вам приходится

использовать упакованные примитивные типы, будьте осторожны! **Автоупаковка уменьшает многословность, но не опасность использования упакованных примитивных типов.** Когда ваша программа сравнивает два упакованных примитивных типа оператором `==`, то происходит сравнение идентичностей, что, скорее всего, совсем не то, что вам нужно. Когда программа выполняет расчеты и с обычными и с упакованными примитивными типами, она выполняет распаковку, а когда ваша программа выполняет распаковку, она может вывести ошибку `NullPointerException`. Наконец, когда программа упаковывает примитивные значения, это может привести к затратному и ненужному созданию объектов.

Статья
50

Не используйте строку там, где более уместен иной тип

Тип `String` создавался для того, чтобы представлять текст, и делает он это прекрасно. Поскольку строки широко распространены и имеют хорошую поддержку в языке Java, возникает естественное желание использовать строки для решения тех 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 часто имеют вид строки. Естественным является стремление оставить их в том же виде, однако это оправдано лишь тогда, когда данные по своей сути являются текстом. Если получены числовые данные, они должны быть приведены к соответствующему числовому типу, такому как `int`, `float` или `BigInteger`. Ответ на вопрос «да/нет» следует преобразовать в `boolean`. В общем случае, если есть соответствующий тип значения (примитивный либо ссылка на объект), вы обязаны им воспользоваться. Если такого нет, вы должны его написать. Этот совет кажется очевидным, но его часто нарушают.

Строки — плохая замена перечислениям. Как говорилось в статье 30, и перечисления типов, и перечисления целых чисел лучше представлять в виде констант перечисления, а не в виде строк.

Строки — плохая замена составным типам (aggregate type). Если некая сущность имеет несколько составных частей, то попытка представить ее одной строкой — обычно неподходящее решение. Для примера приведем строку кода из реальной системы (идентификатор имени изменен, чтобы не выдавать виновника):

```
// Неправомерное использование строки в качестве составного типа
String compoundKey = className + "#" + i.next();
```

Такой подход имеет множество недостатков. Если в одном из полей встретится символ, используемый для разделения, может возникнуть беспорядок. Для получения доступа к отдельным полям вы должны выполнить разбор строки, а это медленная, трудоемкая и подверженная ошибкам операция. У вас нет возможности создать методы equals, toString и compareTo, и вы вынуждены принять решение, предлагаемое классом String. Куда лучше написать класс, представляющий составной тип, часто это бывает закрытый статический класс-член (статья 22).

Строки — плохая замена мандатам. Иногда строки используются для обеспечения доступа к некоторым функциональным возможностям. Например, рассмотрим механизм создания переменных, привязанных к определенному потоку. Этот механизм позволяет создавать переменные, в которых каждый поток может хранить собственное значение. У библиотек Java с версии 1.2 есть возможность использования локально-потоковых переменных, но до этого программистам приходилось выкручиваться самим. Несколько лет назад, столкнувшись с необходимостью реализации такого функционала, несколько человек независимо друг от друга пришли к одному и тому же решению, при котором доступ к подобной переменной осуществляется через строку-ключ, предоставляемую клиентом:

```
// Ошибка: неправомерное использование класса String в качестве манадата
public class ThreadLocal {
```

```
private ThreadLocal() { } // Не порождает экземпляров
// Заносит в именованную переменную значение,
// соответствующее текущему потоку
public static void set(String key, Object value);
// Извлекает из именованной переменной значение,
// соответствующее текущему потоку
public static Object get(String key);
}
```

Проблема здесь заключается в том, что эти ключи относятся к общему пространству имен. И если два независимых клиента, работающие с этим пакетом, решат применить для своих переменных одно и то же название, то, сами того не подозревая, они будут совместно использовать одну и ту же переменную. Обычно это приводит к сбою у обоих клиентов. Кроме того, нарушается безопасность: чтобы получить незаконный доступ к данным другого клиента, злумышленник может намеренно воспользоваться тем же ключом, что и клиент.

API можно исправить, если эту строку заменить неподделываемым ключом (его иногда называют *мандатом* (*capability*)):

```
public class ThreadLocal {
    private ThreadLocal() { } // Не порождает экземпляров
    public static class Key {
        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);
}
```

Это решает обе проблемы с API, использующим строки, но 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: это переменная, имеющая привязку к потоку. Теперь не порождающий экземпляров класс верхнего уровня больше ничего для вас не делает, так что вы можете от него освободиться, а вложенный класс переименовать в ThreadLocal:

```
public final class ThreadLocal {  
    private ThreadLocal();  
    public void set(Object value);  
    public Object get();  
}
```

Этот API не является безопасным, потому что необходимо передавать значение от Object его актуальному типу, когда вы извлекаете его из привязанной к потоку локальной переменной. Невозможно сделать безопасным оригинальный API на основе строки, и сложно сделать безопасным API на основе ключа, но очень просто сделать API безопасным, обобщив класс ThreadLocal (статья 26):

```
public final class ThreadLocal<T> {  
    public ThreadLocal();  
    public void set(T value);  
    public T get();  
}
```

Такой API реализует java.util.ThreadLocal. Помимо того, что этот интерфейс разрешает проблемы API, применяющего строки, он быстрее и элегантнее любого API, использующего ключи.

Подведем итоги. Не поддавайтесь естественному желанию представить объект в виде строки, если для него имеется или может быть написан более приемлемый 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 простых и составных типов, перечислений.

*Статья
51*

При конкатенации строк опасайтесь потери производительности

Оператор конкатенации строк (+) – удобный способ объединения нескольких строк в одну. Он превосходно справляется с генерацией отдельной строки для вывода и с созданием строкового представления для небольшого объекта с фиксированным размером, но не допускает масштабирования. **Время, которое необходимо оператору конкатенации для последовательного объединения n строк, пропорционально квадрату числа n .** К сожалению, это следствие того факта, что строки являются неизменяемыми (статья 15). При объединении двух строк копируется содержимое обеих строк.

Например, рассмотрим метод, который создает строковое представление для выписываемого счета, последовательно объединяя строки для каждого пункта в счете:

```
// Неуместное объединение строк -- плохая производительность!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i); // Объединение строк
    return result;
}
```

Если количество пунктов велико, этот метод работает очень медленно. Чтобы добиться приемлемой производительности, используйте `StringBuilder` вместо `String` для хранения незавершенного выражения. (Класс `StringBuilder`, появившийся в версии 1.5, является несинхронизированной заменой `StringBuffer`, который теперь устарел.)

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}
```

Изменение производительности впечатляет. Если число пунктов (`numItems`) равно 100, а длина строки (`lineForItem`) постоянна и равна 80, то на моей машине второй метод работает в 90 раз быстрее первого. Поскольку первый метод демонстрирует квадратичную зависимость от количества пунктов, а второй — линейную, разница в производительности при большем количестве пунктов становится еще более разительной. Заметим, что второй метод начинается с предварительного размещения в памяти объекта `StringBuilder`, достаточно крупного, чтобы в нем поместился результат вычислений. Даже если отказаться от этого и создать `StringBuilder`, имеющий размер по умолчанию, он будет работать в 50 раз быстрее, чем первый метод.

Мораль проста: не пользуйтесь оператором конкатенации для объединения большого числа строк, если производительность имеет важное значение. Лучше применять метод `append` из класса `StringBuilder`. В качестве альтернативы можно использовать массив символов или обрабатывать строки по одной, не объединяя их.

*Статья
52*

Для ссылки на объект используйте его интерфейс

В статье 40 дается совет: в качестве типа параметра указывать интерфейс, а не класс. В более общей формулировке: ссылаясь на объект, вы должны отдавать предпочтение не классу, а интерфейсу. **Если есть подходящие типы интерфейсов, то параметры, возвращаемые значения, переменные и поля следует декларировать, указывая интерфейс.** Единственный случай, когда вам нужно ссылаться на класс объекта, — при его создании. Для пояснения рассмотрим случай с классом `Vector`, который является реализацией интерфейса `List`. Возьмите за правило писать так:

```
// Хорошо: указывается тип интерфейса.
List<Subscriber> subscribers = new Vector<Subscriber>();
```

а не так:

```
// Плохо: в качестве типа указан класс!
Vector<Subscriber> subscribers = new Vector<Subscriber>();
```

Если вы выработаете привычку указывать в качестве типа интерфейс, ваша программа будет более гибкой. Когда вы решите поменять реализацию, все, что для этого потребуется, — изменить название класса в конструкторе (или использовать другой статический метод генерации). Например, первую из представленных деклараций можно переписать следующим образом:

```
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

И весь окружающий код сможет продолжить работу. Код, окружающий эту декларацию, ничего не знал о прежнем типе, который реализовывал интерфейс. Поэтому он не должен заметить подмену декларации.

Однако нельзя упускать из виду следующее: если первоначальная реализация интерфейса выполняла некие особые функции, не предусмотренные общими соглашениями для этого интерфейса, и программный код зависел от этих функций, крайне важно, чтобы новая реализация интерфейса обеспечивала те же функции. Например, если программный код, окружавший первую декларацию, зависел от того обстоятельства, что `Vector` синхронизирован по отношению к потокам, то замена класса `Vector` на `ArrayList` будет некорректной. Если вы зависите от любых свойств реализации, документируйте эти требования при декларировании переменной.

Тогда зачем менять реализацию? Затем, что новая реализация предлагает повышенную производительность, либо затем, что она обеспечивает необходимую дополнительную функциональность. В качестве примера из реальной жизни возьмем класс `ThreadLocal`. Для связывания экземпляров `ThreadLocal` и потоков в нем используется поле `Map` из класса `Thread`, которое доступно только в пределах пакета. В версии 1.3 это поле инициализировалось экземпляром `HashMap`. В версии 1.4 в платформу Java была добавлена специализированная реализация `Map`, получившая название `IdentityHashMap`. Благодаря изменению в программном коде одной единственной

строки, обеспечившей инициализацию указанного поля экземпляром IdentityHashMap вместо прежнего HashMap, механизм ThreadLock стал работать быстрее.

Будь это поле декларировано как HashMap, а не Map, нельзя было бы гарантировать, что изменения одной строки будет достаточно. Если бы клиент использовал функции класса HashMap, выходящие за рамки интерфейса Map, или передавал экземпляр схемы методу, который требует HashMap, то в результате перехода на IdentityHashMap программа перестала бы компилироваться. Декларация поля с типом интерфейса вынуждает вас «быть проще».

К объекту можно обращаться как к экземпляру класса, а не интерфейса, если нет подходящего интерфейса. Например, рассмотрим *классы значений (value class)*, такие как String и BigInteger. Классы значений редко пишутся в расчете на несколько реализаций. Как правило, это окончательные (*final*) классы, у них редко бывают соответствующие интерфейсы. Лучше использовать класс значения в качестве параметра, переменной, поля или возвращаемого значения. Вообще говоря, если конкретный класс не имеет соответствующего интерфейса, у вас нет иного выбора, кроме как ссылаться на него как на экземпляр класса, независимо от того, представляет этот класс значение или нет. К этой категории относится класс Random.

Второй случай, связанный с отсутствием соответствующего типа интерфейса, относится к объектам, которые принадлежат к структуре, где основными типами являются классы, а не интерфейсы. Если объект принадлежит к *структуре, построенной на классах (class-based framework)*, то для ссылки на него лучше использовать не сам класс реализации, а соответствующий базовый класс (*base class*), который обычно является абстрактным. К этой категории относится класс java.util.TimerTask.

Последний случай относится к классам, которые хотя и реализуют интерфейс, но предоставляют пользователю дополнительные методы, отсутствовавшие в этом интерфейсе. Примером может служить LinkedList. Если программа использует дополнительные мето-

ды, такой класс следует применять только для ссылки на его экземпляры, но никогда как тип параметра (статья 25).

Приведенные варианты не исчерпывают всех случаев, а лишь дают представление о множестве ситуаций, когда для ссылки на объект лучше использовать его класс. На практике должно быть вполне очевидно, есть ли у данного объекта соответствующий интерфейс. Если есть, ваша программа станет более гибкой оттого, что вы будете применять его для ссылки на этот объект. Если же такого интерфейса нет, используйте самый старший класс в иерархии, который обеспечивает необходимую функциональность.

Статья
53

Предпочитайте интерфейс отражению класса

Механизм отражения (reflection, пакет `java.lang.reflect`) предоставляет программный доступ к информации о загруженных классах. Имея экземпляр `Class`, вы можете получить экземпляры `Constructor`, `Method` и `Field`, соответствующие конструкторам, методам и полям данного экземпляра. Эти объекты обеспечивают программный доступ к названиям методов в классе, к типам полей, сигнатурам методов и т.д.

Более того, экземпляры `Constructor`, `Method` и `Field` позволяют вам опосредованно манипулировать стоящими за ними *сущностями*: вы можете создавать экземпляры, вызывать методы и получать доступ к полям соответствующего класса, обращаясь к методам экземпляров `Constructor`, `Field` и `Method`. Например, `Method.invoke` дает возможность вызвать любой метод из любого объекта любого класса (при соблюдении обычных ограничений, связанных с безопасностью). Механизм отражения разрешает одному классу использовать другой, даже если последний во время компиляции первого еще не существовал. Однако такая мощь обходится недешево:

- **Вы лишаетесь всех преимуществ проверки типов на этапе компиляции**, в том числе проверки исключений. Если про-

грамма попытается опосредованно вызвать несуществующий или недоступный метод, то в случае отсутствия специальных мер предосторожности произойдет сбой программы во время выполнения.

- **Программный код, необходимый для отражения классов, неуклюж и многословен.** Его тяжело писать и трудно читать.
- **Страдает производительность.** Опосредованный вызов метода выполняется намного медленнее обычного вызова. Насколько медленнее — сложно точно сказать, потому что на его работу влияет много факторов. На моей машине разница в скорости может быть от 2 до 50 раз.

Механизм отражения классов первоначально создавался для средств разработки, обеспечивающих построение приложения из компонентов. Как правило, такие инструменты осуществляют загрузку классов по требованию и используют механизм отражения для выяснения того, какие методы и конструкторы поддерживаются этими классами. Подобные инструменты позволяют пользователю в интерактивном режиме создавать приложения, имеющие доступ к классам, однако полученные приложения используют уже обычный, а не опосредованный доступ. Отражение применяется только на этапе проектирования приложения. **Как правило, обычные приложения на стадии выполнения не должны пользоваться опосредованным доступом к объектам.**

Есть лишь несколько сложных приложений, которым необходим механизм отражения. В их число входят визуализаторы классов, инспекторы объектов, анализаторы программного кода и интерпретирующие встроенные системы. Отражение можно также использовать в системах вызова удаленных процедур (*RPC system*) с целью снижения потребности в компиляторах запросов. Если у вас есть сомнения, подпадает ли ваше приложение в одну из этих категорий, вероятнее всего, оно к ним не относится.

Вы можете без больших затрат использовать многие преимущества механизма отражения, если будете применять его

в усеченном виде. Во многих программах, которым нужен класс, отсутствовавший на момент компиляции, для ссылки на него можно использовать соответствующий интерфейс или суперкласс (статья 52). Если это то, что вам нужно, вы можете сначала опосредованно **создать экземпляры, а затем обращаться к ним обычным образом, используя их интерфейс или суперкласс.** Если соответствующий конструктор, как часто бывает, не имеет параметров, вам даже не нужно обращаться к пакету `java.lang.reflect` — требуемые функции предоставит метод `Class.newInstance`.

В качестве примера приведем программу, которая создает экземпляр интерфейса `Set`, чей класс задан первым аргументом командной строки. Остальные аргументы командной строки программа вставляет в полученный набор и затем распечатывает его. При выводе аргументов программа уничтожает дубликаты (первый аргумент игнорируется). Порядок печати аргументов зависит от того, какой класс указан в первом аргументе. Если вы указываете «`java.util.HashSet`», аргументы выводятся в произвольном порядке, если — «`java.util.TreeSet`», они печатаются в алфавитном порядке, поскольку элементы в наборе `TreeSet` сортируются.

```
// Опосредованное создание экземпляра с доступом
// через интерфейс
public static void main(String[] args) {
    // Преобразует имя класса в экземпляр класса
    Class<?> cl = null;
    try {
        cl = Class.forName(args[0]);
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found.");
        System.exit(1);
    }
    // Создает экземпляр класса
    Set<String> s = null;
    try {
        s = (Set<String>) cl.newInstance();
    }
```

```
    } catch(IllegalAccessException e) {
        System.err.println("Class not accessible.");
        System.exit(1);
    } catch(InstantiationException e) {
        System.err.println("Class not instantiate.");
        System.exit(1);
    }
    // Проверяет набор
    s.addAll(Arrays.asList(args).subList(1, args.length-1));
    System.out.println(s);
}
```

Хотя эта программа является всего лишь игрушкой, она показывает мощный прием. Этот код легко превратить в универсальную программу проверки наборов, которая проверяет правильность указанной реализации Set, активно воздействуя на один или несколько экземпляров и выясняя, выполняют ли они соглашения для интерфейса Set. Точно так же его можно превратить в универсальный инструмент для анализа производительности. Методика, которую демонстрирует эта программа, в действительности достаточна для создания полноценной инфраструктуры с предоставлением услуг (*service provider framework*) (статья 1). В большинстве случаев описанный прием — это все, что нужно знать об отражении классов.

Этот пример иллюстрирует два недостатка системы отражения. Во-первых, во время его выполнения могут возникнуть три ошибки, которые, если бы не использовался механизм отражения, были бы обнаружены еще на стадии компиляции. Во-вторых, для генерации экземпляра класса по его имени потребовалось двадцать строк кода, тогда как вызов конструктора уложился бы ровно в одну строку. Эти недостатки, однако, касаются лишь той части программы, которая создает объект как экземпляр класса. Как только экземпляр создан, он неотличим от любого другого экземпляра Set. Благодаря этому значительная часть кода в реальной программе не поменяется от локального применения механизма отражения.

При попытке компиляции программы вы получите следующее сообщение об ошибке:

Note: MakeSet.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

Это предупреждение касается использования программой обобщенных типов, но не отображает реальной проблемы. Как скрыть предупреждения, описано в статье 24.

Еще один отклоняющийся от темы вопрос, который заслуживает упоминания, — это использование программой `System.exit`. Очень редко бывает ситуация, когда стоит использовать данный метод, который закрывает полностью виртуальную машину. Тем не менее он подходит при ненормальном завершении утилиты командной строки.

Приемлемым, хотя и редким вариантом использования отражения является разрушение зависимости класса от других классов, методов и полей, которые в момент выполнения могут отсутствовать. Это может пригодиться при написании пакета, который должен работать с различными версиями какого-либо другого пакета. Прием заключается в том, чтобы компилировать ваш пакет для работы в минимальном окружении (обычно это поддержка самой старой версии), а доступ ко всем новым классам и методам осуществлять через механизм отражения. Чтобы это работало, необходимо предпринимать правильные действия, когда в ходе выполнения программы обнаружится, что тот или иной новый класс или метод, к которому вы пытаетесь получить доступ, в данный момент отсутствует. Эти действия могут заключаться в применении каких-либо альтернативных средств, позволяющих достичь той же цели, или же в использовании усеченного функционала.

Подведем итоги. Отражение — это мощный инструмент, который необходим для решения определенных сложных задач системного программирования. Однако у него есть много недостатков. Если вы пишете программу, которая должна работать с классами, неизвестными на момент компиляции, то вы должны по возможности ис-

пользовать механизм отражения только для создания экземпляров отсутствовавших классов, а для доступа к полученным объектам следует применять некий интерфейс или суперкласс, который известен уже на стадии компиляции.

Статья
54

Соблюдайте осторожность при использовании машинозависимых методов

Интерфейс Java Native Interface (JNI) дает возможность приложениям на языке Java делать вызов *машинозависимых методов* (*native method*), т.е. специальных методов, написанных на *машинозависимом языке программирования*, таком как С или С++. Перед тем как вернуть управление языку Java, машинозависимые методы могут выполнить любые вычисления, используя машинозависимый язык.

Исторически машинозависимые методы имели три основные области применения. Они предоставляли доступ к механизмам, соответствующим конкретной платформе, таким как реестр и блокировка файла. Они обеспечивали доступ к унаследованным библиотекам кода, которые, в свою очередь, могли дать доступ к унаследованным данным. Наконец, машинозависимые методы использовались для того, чтобы писать на машинозависимом языке те части приложений, которые критичны для быстродействия, тем самым повышая общую производительность.

Применение машинозависимых методов для доступа к механизмам, специфичным для данной платформы, абсолютно оправданно. Однако по мере своего развития платформа Java предоставляет все больше и больше возможностей, которые прежде можно было найти лишь на главных платформах. Например, появившийся в версии 1.4 пакет `java.util.prefs` выполняет функции реестра. Оправданно также использование машинозависимых методов для доступа к унаследованному коду, однако есть более хорошие способы доступа к унасле-

ледованному коду. Например, интерфейс JDBC (Java DataBase Connectivity) обеспечивает доступ к унаследованным базам данных.

В версии 1.3 использование машинозависимых методов для повышения производительности редко оправдывает себя. В предыдущих версиях это часто было необходимо, однако сейчас созданы более быстрые реализации JVM. И теперь для большинства задач можно получить сравнимую производительность, не прибегая к машинозависимым методам. Например, когда в версию 1.1 был включен пакет `java.math.BigInteger`, он был реализован поверх быстрой библиотеки арифметических операций с многократно увеличенной точностью, написанной на языке С. В то время это было необходимо для получения приемлемой производительности. В версии 1.3 класс `BigInteger` полностью переписан на языке Java и тщательно отрегулирован. Для большинства операций и размеров operandов новая версия оказывается быстрее первоначальной во всех реализациях JVM 1.3 компании Sun.

Применение машинозависимых методов имеет серьезные недостатки. Поскольку машинозависимые методы *небезопасны* (статья 39), использующие их приложения теряют устойчивость к ошибкам, связанным с памятью. Для каждой новой платформы машинозависимый программный код необходимо компилировать заново, может потребоваться даже его изменение. С переходом на машинозависимый код и с возвратом в Java связаны высокие накладные расходы, а потому, если машинозависимые методы выполняют лишь небольшую работу, их применение может *снизить* производительность приложения. Наконец, машинозависимые методы сложно писать и трудно читать.

Подведем итоги. Хорошо подумайте, прежде чем использовать машинозависимые методы. Если их и можно применять для повышения производительности, то крайне редко. Если вам необходимо использовать машинозависимые методы для доступа к низкоуровневым ресурсам или унаследованным библиотекам, пишите как можно меньше машинозависимого кода и тщательно его тестируйте. Единственная ошибка в машинозависимом коде может полностью разрушить все ваше приложение.

Соблюдайте осторожность при оптимизации

Есть три афоризма, посвященные оптимизации, которые обязан знать каждый. Возможно, они пострадали от слишком частого цитирования, однако приведем их на тот случай, если вы с ними не знакомы:

Во имя эффективности (без обязательности ее достижения) делается больше вычислительных ошибок, чем по каким-либо иным причинам, включая непроходимую тупость.

— Уильям Вульф (*William A. Wulf*) [*Wulf72*]

Мы обязаны забывать о мелких усовершенствованиях, скажем, на 97% рабочего времени: опрометчивая оптимизация — корень всех зол.

— Дональд Кнут (*Donald E. Knuth*) [*Knuth74*]

Что касается оптимизации, то мы следуем двум правилам:

Правило 1. Не делайте этого.

Правило 2 (только для экспертов). Пока не делайте этого — т.е. пока у вас нет абсолютно четкого, но неоптимизированного решения.

— М. А. Джексон (*M. A. Jackson*) [*Jackson75*]

Эти афоризмы на два десятилетия опередили язык программирования Java. В них отражена сущая правда об оптимизации: легче причинить вред, чем благо, особенно если вы взялись за оптимизацию преждевременно. В процессе оптимизации вы можете получить программный код, который не будет ни быстрым, ни правильным, и его уже так легко не исправить.

Не жертвуйте здравыми архитектурными принципами во имя производительности. Страйтесь писать хорошие программы, а не быстрые. Если хорошая программа работает недостаточно быстро, ее архитектура позволит осуществить оптимизацию. Хорошие про-

граммы воплощают принцип *сокрытия информации (information hiding)*: по возможности они локализуют конструкторские решения в отдельных модулях, а потому отдельные решения можно менять, не затрагивая остальные части системы (статья 13).

Это не означает, что вы должны игнорировать вопрос 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 систему. Изменение задним числом фундаментальных положений вашего проекта может породить систему с уродливой структурой, которую сложно сопровождать и совершенствовать. Поэтому вы должны думать о производительности уже в процессе разработки приложения.

Старайтесь избегать конструкторских решений, ограничивающих производительность. Труднее всего менять те компоненты, которые определяют взаимодействие модулей с окружающим миром. Главными среди таких компонентов являются API, протоколы физического уровня и форматы записываемых данных. Мало того, что эти компоненты впоследствии сложно или невозможно менять, любой из них способен существенно ограничить производительность, которую можно получить от системы.

Изучите влияние на производительность тех проектных решений, которые заложены в ваш API. Создание изменяемого открытого типа может потребовать создания множества ненужных резервных копий (статья 39). Точно так же использование наследования в открытом классе, для которого уместнее была бы композиция, на всегда привязывает класс к его суперклассу, а это может искусственно ограничивать производительность данного подкласса (статья 16). И последний пример: указав в API не тип интерфейса, а тип реализующего его класса, вы оказываетесь привязаны к определенной реализации этого интерфейса, даже несмотря на то, что в будущем, возможно, будут написаны еще более быстрые его реализации (статья 52).

Влияние архитектуры API на производительность велико. Рассмотрим метод getSize из класса java.awt.Component. То, что этот критичный для производительности метод возвращает экземпляр Dimension, а также то, что экземпляры Dimension являются изменяемыми, приводит к тому, что любая реализация этого метода при каждом вызове создает новый экземпляр Dimension. И хотя, начиная с версии 1.3, создание небольших объектов обходится относительно дешево, бесполезное создание миллионов объектов может нанести производительности приложения реальный ущерб.

В данном случае имеется несколько альтернатив. В идеале класс Dimension должен стать неизменяемым (статья 15). Либо метод getSize можно заменить двумя методами, возвращающими отдельные простые компоненты объекта Dimension. И действительно, с целью повышения производительности в версии 1.2 два таких метода были добавлены в интерфейс класса Component. Однако уже существовавший к тому времени клиентский код продолжает пользоваться методом getSize, и его производительность по-прежнему страдает от первоначально принятых проектных решений для API.

Хорошая схема API, как правило, сочетается с хорошей производительностью. **Не стоит искажать API ради улучшения производительности.** Проблемы с производительностью, которые заставили вас переделать API, могут исчезнуть с появлением новой платформы или других базовых программ, а вот искаженный API и связанная с ним головная боль останутся с вами навсегда.

После того как вы тщательно спроектировали программу и выстроили четкую, краткую и хорошо структурированную ее реализацию, можно подумать об оптимизации, если, конечно, вы еще не удовлетворены производительностью программы. Напомним два правила Джексона: «не делайте этого» и «не делайте этого пока (для экспертов)». Он мог бы добавить еще одно: **измеряйте производительность до и после попытки ее оптимизации.**

Возможно, вы будете удивлены, но нередко попытки оптимизации не оказывают поддающегося измерению влияния на производи-

тельность, иногда они даже ухудшают ее. Основная причина заключается в том, что сложно догадаться, где программа теряет время. Та часть программы, которую вы считаете медленной, может оказаться ни при чем, и вы зря потратите время, пытаясь ее оптимизировать. Общее правило гласит, что 80% времени программы теряют на 20% своего кода.

Средства профилирования помогут вам определить, где именно следует сосредоточить усилия по оптимизации. Подобные инструменты предоставляют вам информацию о ходе выполнения программы, например: сколько примерно времени требуется каждому методу, сколько раз он был вызван. Это укажет вам объект для настройки, а также может предупредить вас о необходимости замены самого алгоритма. Если в вашей программе скрыт алгоритм с квадратичной (или еще худшей) зависимостью, никакие настройки эту проблему не решат. Следовательно, вам придется заменить алгоритм более эффективным. Чем больше в системе программного кода, тем большее значение имеет работа с профилировщиком. Это все равно что искать иголку в стоге сена: чем больше стог, тем больше пользы от металлоискателя. JDK поставляется с простым профилировщиком, несколько инструментов посложнее можно купить отдельно.

Задача определения эффекта оптимизации для платформы Java стоит острее, чем для традиционных платформ, по той причине, что язык программирования Java не имеет четкой модели производительности (*performance model*). Нет четкого определения относительной стоимости различных базовых операций. «Семантический разрыв» между тем, что пишет программист, и тем, что выполняется центральным процессором, здесь гораздо значительнее, чем у традиционных компилируемых языков, и это сильно усложняет надежное предсказание того, как будет влиять на производительность какая-либо оптимизация. Существует множество мифов о производительности, которые на поверку оказываются полуправдой, а то и совершенной ложью.

Помимо того, что модель производительности плохо определена, она меняется от одной реализации JVM к другой и даже от версии

к версии. Если вы будете запускать свою программу в разных реализациях JVM, проследите эффект от вашей оптимизации для каждой из этих реализаций. Иногда в отношении производительности вам придется идти на компромисс между различными реализациями JVM.

Подведем итоги. Не старайтесь писать быстрые программы — лучше пишите хорошие, тогда у вас появится и скорость. Проектируя системы, обязательно думайте о производительности, особенно если вы работаете над API, протоколами нижних уровней и форматами записываемых данных. Закончив построение системы, измерьте ее производительность. Если скорость приемлема, ваша работа завершена. Если нет, локализуйте источник проблем с помощью профилировщика и оптимизируйте соответствующие части системы. Первым шагом должно быть исследование выбранных алгоритмов: никакая низкоуровневая оптимизация не компенсирует плохой выбор алгоритма. При необходимости повторите эту процедуру, измеряя производительность после каждого изменения, пока не будет получен приемлемый результат.

Статья
56

При выборе имен придерживайтесь общепринятых соглашений

Платформа Java обладает хорошо устоявшимся набором соглашений, касающихся выбора имен (*naming convention*). Многие из них приведены в «*The Java Language Specification*» [JLS, 6.8]. Соглашения об именовании делятся на две категории: типографские и грамматические.

Типографских соглашений, касающихся выбора имен для пакетов, классов, интерфейсов и полей, очень мало. Никогда не нарушайте их, не имея на то веской причины. API, не соблюдающий эти соглашения, будет трудно использовать. Если соглашения нарушены в реализации, ее будет сложно сопровождать. В обоих случаях нарушение соглашений может запутывать и раздражать других програм-

мистов, работающих с этим кодом, а также способствовать появлению ложных допущений, приводящих к ошибкам.

Названия пакетов должны представлять собой иерархию, отдельные части которой отделены друг от друга 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 в Интернете, которому предшествуют домены верхнего уровня, например edu.сми, com.sun, gov.nsa. Исключение из этого правила составляют стандартные библиотеки, а также необязательные пакеты, чьи названия начинаются со слов java и javax. Пользователи не должны создавать пакетов с именами, начинающимися с java или javax. Детальное описание правил, касающихся преобразования названий доменов Интернета в префиксы названий пакетов, можно найти в «*The Java Language Specification*» [JLS, 7.7].

Вторая половина в названии пакета должна состоять из одной или нескольких частей, описывающих этот пакет. Части должны быть короткими, обычно не длиннее восьми символов. Поощряются выразительные сокращения, например util вместо utilities. Допустимы акронимы, например awt. Такие части, как правило, должны состоять из одного-единственного слова или сокращения.

Многие пакеты имеют имена, в которых, помимо названия домена в Интернете, присутствует только одно слово. Большее количество частей в имени пакета нужно лишь для больших систем, чей размер настолько велик, что требует создания неформальной иерархии. Например, в пакете javax.swing представлена сложная иерархия пакетов с такими названиями, как javax.swing.plaf.metal. Подобные пакеты часто называют подпакетами, однако это относится исключительно к области соглашений, поскольку для иерархии пакетов нет лингвистической поддержки.

Названия классов и интерфейсов состоят из одного или нескольких слов, причем в каждом слове первая буква должна быть заглавной, например Timer или TimerTask. Необходимо избегать аббревиатур, за исключением акронимов и нескольких общепринятых

сокращений, таких как `max` и `min`. Нет полного единодушия по поводу того, должны ли акронимы полностью писаться прописными буквами или же заглавной у них должна быть только первая буква. Хотя чаще в верхнем регистре пишется все название, есть один сильный аргумент в пользу того, чтобы заглавной была только первая буква. В последнем случае всегда ясно, где кончается одно слово и начинается другое, даже если рядом стоят несколько акронимов. Какое название класса вы предпочли бы увидеть: `HTTPURL` или `HttpUrl`?

Названия методов и полей подчиняются тем же самым типографским соглашениям, за исключением того, что первый символ в названии всегда должен быть строчным, например `remove`, `ensureCapacity`. Если первым словом в названии метода или поля оказывается акроним, он весь пишется строчными буквами.

Единственное исключение из предыдущего правила касается полей-констант (*constant field*), названия которых должны состоять из одного или нескольких слов, написанных заглавными буквами и отделенных друг от друга символом подчеркивания, например `VALUES` или `NEGATIVE_INFINITY`. Поле-константа — это поле `static final`, значение которого не меняется. Если поле `static final` имеет простой тип или неизменяемый ссылочный тип (статья 15), то это поле-константа. Например, перечислимые константы являются полями-константами. Если у статического завершенного поля есть изменяемый тип ссылки, оно все еще может быть полем-константой, если объект, на который оно ссылается, является неизменяемым. Заметим, что поля-константы — это единственное место, где допустимо использование символа подчеркивания. Заметим, что поля-константы — единственное место, где допустимо использование символа подчеркивания.

Названия локальных переменных подчиняются тем же типографским соглашениям, что и названия членов классов, за исключением того, что в них можно использовать аббревиатуры, отдельные символы, а также короткие последовательности символов, смысл которых зависит от того контекста, где эти локальные переменные находятся. Например: `i`, `xref`, `houseNumber`. Наименование типа параметров обычно состоит из одной буквы.

Наиболее часто это одно из этих пяти: T для произвольных типов, E для типов элементов в коллекции, K и V для ключей и типов значений схемы, и X для исключений. Последовательность произвольных типов может быть T, U, V или T1, T2, T3.

Примеры типографских соглашений приведены в таблице 7.1.

Таблица 7.1.

Примеры типографских соглашений

Тип идентификатора	Примеры
Пакет	com.google.inject, org.jode.time.format
Класс или интерфейс	Timer, FutureTask, LinkedHashMap, HttpServlet
Метод или поле	remove, ensureCapacity, getCrc
Поле-константа	MIN_VALUES, NEGATIVE_INFINITY
Локальная переменная	i, xref, houseNumber
Тип параметра	T, E, K, V, X, T1, T2

По сравнению с типографскими грамматические соглашения, касающиеся именования, более гибкие и спорные. Для пакетов практически нет грамматических соглашений. Для именования классов обычно используются существительные или именные конструкции, например Timer и BufferedWriter или ChessPiece. Интерфейсы имеютсь так же, как классы, например Collection и Comparator, либо применяются названия с окончаниями, образованными от прилагательных «-able» и «-ible», например Runnable и Accessible. Поскольку у типов аннотаций есть много вариантов использования, то тут нет преобладания каких-либо частей речи. Существительные, глаголы, предлоги и прилагательные используются в равной степени, например BindingAnnotation, Inject, ImplementedBy или Singleton.

Для методов, выполняющих какое-либо действие, в качестве названия используются глаголы или глагольные конструкции, например append и drawImage. Для методов, возвращающих булево значение,

обычно применяются названия, в которых сначала идет слово «*is*», а потом существительное, именная конструкция или любое слово (фраза), играющее роль прилагательного, например *isDigit*, *isProbablePrime*, *isEmpty*, *isEnabled*, *isRunning*.

Для именования методов, не связанных с булевыми операциями, а также методов, возвращающих атрибут объекта, для которого они были вызваны, обычно используется существительное, именная конструкция либо глагольная конструкция, начинающаяся с глагола «*get*», например *size*, *hashCode*, *getTime*. Отдельные пользователи требуют, чтобы применялась лишь третья группа (начинающаяся с «*get*»), но для подобных претензий нет никаких оснований. Первые две формы обычно делают текст программы более удобным для чтения, например:

```
if (car.speed() > 2* SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

Форма, начинающаяся с «*get*», обязательна, если метод принадлежит к классу Bean [JavaBeans]. Ее можно также рекомендовать, если в будущем вы собираетесь превратить свой класс в Bean. Наконец, серьезные основания для использования данной формы имеются в том случае, если в классе уже есть метод, присваивающий этому же атрибуту новое значение. При этом указанные методы следует назвать *getAttribute* и *setAttribute*.

Несколько названий методов заслуживают особого упоминания. Методы, которые преобразуют тип объекта и возвращают независимый объект другого типа, часто называются *toType*, например *toString*, *toArray*. Методы, которые возвращают *представление* (статья 4), имеющее иной тип, чем сам объект, обычно называются *asType*, например *asList*. Методы, возвращающие простой тип с тем же значением, что и у объекта, в котором они были вызваны, называются *typeValue*, например *intValue*. Для статических методов генерации широко используются названия *valueOf* и *getInstance* (статья 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 какое-либо поле, то немного. Поля типа `boolean` обычно именуются так же, как логические методы доступа, но префикс «`is`» у них опускается, например `initialized`, `composite`. Поля других типов, как правило, именуются с помощью существительного или именной конструкции, например `height`, `digits`, `bodyStyle`. Грамматические соглашения для локальных переменных аналогичны соглашениям для полей, только их соблюдение еще менее обязательно.

Подведем итоги. Изучите стандартные соглашения поименованию и доведите их использование до автоматизма. Типографские соглашения просты и практически однозначны; грамматические соглашения более сложные и свободные. Как сказано в «*The Java Language Specification*» [JLS, 6.8], не нужно рабски следовать этим соглашениям, если длительная практика их применения диктует иное решение. Пользуйтесь здравым смыслом.

9

Г л а в а

Исключения

Если исключения (*exception*) используются наилучшим образом, они способствуют написанию понятных, надежных и легко сопровождаемых программ. При неправильном применении результат может быть прямо противоположным. В этой главе даются рекомендации по эффективному использованию исключений.

Статья
57

Используйте исключения лишь в исключительных ситуациях

Однажды, если вам не повезет, вы сделаете ошибку в программе, например такую:

```
// Неправильное использование исключений. Никогда так
// не делайте!
try {
    int i = 0;
    while(true)
        range[i++].climb();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

Что делает этот код? Изучение кода не вносит полной ясности, и это достаточная причина, чтобы им не пользоваться. Здесь приведена плохо продуманная идиома для циклического перебора элементов в массиве. Когда производится попытка обращения к первому элементу за пределами массива, бесконечный цикл завершается иницированием исключительной ситуации `ArrayIndexOutOfBoundsException`, ее перехватом и последующим игнорированием. Предполагается, что это эквивалентно стандартной идиоме цикла по массиву, которую узнает любой программист Java:

```
for (Mountain m : range)  
    m.climb();
```

Но почему же кто-то выбрал идиому, использующую исключения, вместо другой, испытанной и правильной? Это вводящая в заблуждение попытка улучшить производительность, которая исходит из ложного умозаключения, что, поскольку виртуальная машина проверяет границы при всех обращениях к массиву, обычная проверка на завершение цикла (`i < a.length`) избыточна и ее следует устранить. В этом рассуждении неверны три момента:

- Так как исключения создавались для применения в исключительных условиях, лишь очень немногие реализации JVM пытаются их оптимизировать (если таковые есть вообще). Обычно создание, иницирование и перехват исключения дорого обходится системе.
- Размещение кода внутри блока `try-catch` **препятствует выполнению определенных процедур оптимизации, которые в противном случае могли бы быть исполнены в современных реализациях JVM.**
- **Стандартная идиома цикла по массиву вовсе не обязательно приводит к выполнению избыточных проверок, в процессе оптимизации некоторые современные реализации JVM отбрасывают их.**

Практически во всех современных реализациях JVM идиома, использующая исключения, работает гораздо медленнее стандартной идиомы. На моей машине идиома, использующая исключения, выполняет массив из 100 элементов более чем в два раза медленнее стандартной.

Идиома цикла, использующая исключения, снижает производительность и делает непонятным программный код. Кроме того, нет гарантий, что она будет работать. При появлении не предусмотренной разработчиком ошибки указанная идиома может без предупреждений завершить работу, маскировав ошибку, и тем самым значительно усложнить процесс отладки. Предположим, вычисления в теле цикла содержат ошибку, которая приводит к выходу за границы при доступе к какому-то совсем другому массиву. Если бы применялась правильная идиома цикла, эта ошибка породила бы необработанное исключение, которое вызвало бы немедленное завершение потока с соответствующим сообщением об ошибке. В случае же порочной, использующей исключения идиомы цикла исключение, вызванное ошибкой, будет перехвачено и неправильно интерпретировано как обычное завершение цикла.

Мораль проста: исключения, как и подразумевает их название, должны применяться лишь для исключительных ситуаций, при обычной обработке использовать их не следует никогда. Вообще говоря, вы всегда должны предпочитать стандартные, легко распознаваемые идиомы идиомам с ухищрениями, предлагающим лучшую производительность. Даже если имеет место реальный выигрыш в производительности, он может быть поглощен неуклонным совершенствованием реализаций JVM. А вот коварные ошибки и сложность поддержки, вызываемые чрезсчур хитроумными идиомами, наверняка останутся.

Этот принцип относится также к проектированию API. Хорошо спроектированный API не должен заставлять своих клиентов использовать исключения для обычного управления потоком вычислений. Если в классе есть метод, зависящий от состояния (*state-dependent*), который может быть вызван лишь при выполнении определенных не-предсказуемых условий, то в этом же классе, как правило, должен

присутствовать отдельный метод, проверяющий состояние (*state-testing*), который показывает, можно ли вызывать первый метод. Например, класс `Iterator` имеет зависящий от состояния метод `next`, который возвращает элемент для следующего прохода цикла, а также соответствующий метод проверки состояния `hasNext`. Это позволяет применять для просмотра коллекции в цикле следующую стандартную идиому:

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {  
    Foo foo = i.next();  
  
}
```

Если бы в классе `Iterator` не было метода `hasNext`, клиент был бы вынужден использовать следующую конструкцию:

```
// Не пользуйтесь этой отвратительной идиомой  
// для просмотра коллекции в цикле!  
try {  
    Iterator<Foo> i = collection.iterator();  
    while(true) {  
        Foo foo = i.next();  
  
    }  
} catch (NoSuchElementException e) {  
}
```

Этот пример так же плох, как и пример с просмотром массива в цикле, приведенный в начале статьи. Идиома, использующая исключения, отличается от стандартной идиомы не только многословностью и запутанностью, но также худшей производительностью и способностью скрывать ошибки, возникающие в других, не связанных с ней частях системы.

В качестве альтернативы отдельному методу проверки состояния можно использовать особый зависящий от состояния метод: он будет возвращать особое значение, например `null`, при вызове для объекта, имеющего неподходящее состояние. Для класса `Iterator` этот прием

не годится, поскольку `null` является допустимым значением для метода `next`.

Приведем некоторые рекомендации, которые помогут вам сделать выбор между методом проверки состояния и особым возвращаемым значением. Если к объекту возможен одновременный доступ без внешней синхронизации или если смена его состояний инициируется извне, может потребоваться прием с особым возвращаемым значением, поскольку состояние объекта может поменяться в период между вызовом метода, который проверяет состояние, и вызовом соответствующего метода, который зависит от состояния объекта. Особое возвращаемое значение может потребоваться для повышения производительности, когда метод проверки состояния может при необходимости дублировать работу метода, зависящего от состояния объекта. Однако при прочих равных условиях метод проверки состояния предпочтительнее особого возвращаемого значения. При его использовании легче читать текст программы, а также проще обнаруживать и исправлять неправильное построение программы.

Подведем итоги. Исключения созданы для использования в исключительных условиях. Не используйте их для обычного контроля и не пишите такие API, которые будут заставлять других это делать.

Статья
58

Применяйте обрабатываемые исключения для восстановления, для программных ошибок используйте исключения времени выполнения

В языке программирования Java предусмотрены три типа объектов `Throwable`: обрабатываемые исключения (*checked exception*), исключения времени выполнения (*run-time exception*) и ошибки (*error*). Программисты обычно путают, при каких условиях 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 обрабатываемым и необрабатываемым исключениями гласит: используйте обрабатываемые ис-

ключения для тех условий, когда есть основания полагать, что инициатор вызова способен их обработать. Генерируя обрабатываемое исключение, вы принуждаете инициатора вызова обрабатывать его в операторе `catch` или передавать дальше. Каждое обрабатываемое исключение, которое, согласно декларации, инициирует некий метод, является, таким образом, серьезным предупреждением для пользователя API о том, что при вызове данного метода могут возникнуть соответствующие условия.

Предоставляя пользователю API обрабатываемое исключение, разработчик API передает ему право осуществлять обработку соответствующего условия. Пользователь может пренебречь этим правом, перехватив исключение и проигнорировав его. Однако, как правило, это оказывается плохим решением (статья 65).

Есть два типа необрабатываемых объектов `Throwable`: исключения времени выполнения и ошибки. Поведение у них одинаковое: ни тот ни другой не нужно и, вообще говоря, нельзя перехватывать. Если программа инициирует необрабатываемое исключение или ошибку, то, как правило, это означает, что восстановление невозможно и дальнейшее выполнение программы принесет больше вреда, чем пользы. Если программа не перехватывает такой объект, его появление вызовет остановку текущего потока команд с соответствующим сообщением об ошибке.

Используйте исключения времени выполнения для индикации программных ошибок. Подавляющее большинство исключений времени выполнения сообщает о нарушении предусловий (*precondition violation*). Нарушение предусловия означает лишь то, что клиент API не смог выполнить соглашения, заявленные в спецификации к этому API. Например, в соглашениях для доступа к массиву оговаривается, что индекс массива должен попадать в интервал от нуля до «длина массива минус один». Исключение `ArrayIndexOutOfBoundsException` указывает, что это предусловие было нарушено.

Хотя в спецификации языка Java это не оговорено, существует строго соблюданное соглашение о том, что ошибки зарезервированы

в JVM для того, чтобы фиксировать дефицит ресурсов, нарушение инвариантов и другие условия, делающие невозможным дальнейшее выполнение программы [Chan98, Horstman00]. Поскольку эти соглашения признаны практически повсеместно, лучше для Error вообще не создавать новых подклассов. Все реализуемые вами необрабатываемые исключения должны прямо или косвенно наследовать класс RuntimeException.

Для исключительной ситуации можно определить класс, который не наследует классов Exception, RuntimeException и Error. В спецификации языка Java такие классы напрямую не оговариваются, однако неявно подразумевается, что они будут вести себя так же, как обычные обрабатываемые исключения (которые являются подклассами класса Exception, но не RuntimeException). Когда же вы должны использовать этот класс? Если одним словом, то никогда. Не имея никаких преимуществ перед обычным обрабатываемым исключением, он будет запутывать пользователей вашего API.

Подведем итоги. Для ситуаций, когда можно обработать ошибку и продолжить исполнение, используйте обрабатываемые исключения, для программных ошибок применяйте исключения времени выполнения. Разумеется, ситуация не всегда однозначна, как белое и черное. Рассмотрим случай с исчерпанием ресурсов, которое может быть вызвано программной ошибкой, например размещением в памяти неоправданно большого массива, или настоящим дефицитом ресурсов. Если исчерпание ресурсов вызвано временным дефицитом или времененным увеличением спроса, эти условия вполне могут быть изменены. Именно разработчик API принимает решение, возможно ли восстановление работоспособности программы в конкретном случае исчерпания ресурсов. Если вы считаете, что работоспособность можно восстановить, используйте обрабатываемое исключение. В противном случае применяйте исключение времени выполнения. Если неясно, возможно ли восстановление, то по причинам, описанным в статье 59, лучше остановиться на необрабатываемом исключении.

Разработчики API часто забывают, что исключения — это вполне законченные объекты, для которых можно определять любые методы. Основное назначение таких методов — создание кода, который увязывал бы исключение с дополнительной информацией об условии, вызвавшем появление данной исключительной ситуации. Если таких методов нет, программистам придется разбираться со строковым представлением этого исключения, выуживая из него дополнительную информацию. Эта крайне плохая практика. Классы редко указывают какие-либо детали в своем строковом представлении, 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 обрабатываемые исключения обычно указывают на ситуации, когда возможно продолжение выполнения, для такого типа исключений важно создать методы, которые предоставляли бы клиенту информацию, помогающую возобновить работу. Предположим, что обрабатываемое исключение инициируется при неудачной попытке позвонить с платного телефона из-за того, что клиент не предоставил достаточной суммы денег. Для этого исключения должен быть реализован метод доступа, который запрашивает недостающую сумму с тем, чтобы можно было сообщить о ней пользователю телефонного аппарата.

Статья
59

Избегайте ненужных обрабатываемых исключений

Обрабатываемые исключения — замечательная особенность языка программирования Java. В отличие от возвращаемых кодов они заставляют программиста отслеживать условия возникновения исключений, что значительно повышает надежность приложения. Это означает, что злоупотребление обрабатываемыми исключениями может сделать API менее удобным для использования. Если ме-

тод инициирует одно или несколько обрабатываемых исключений, то в программном коде, из которого этот метод был вызван, должна присутствовать обработка этих исключений в виде одного или нескольких блоков catch либо должно быть декларировано, что этот код сам инициирует исключения и передает их дальше. В любом случае перед программистом стоит нелегкая задача.

Такое решение оправданно, если даже при надлежащем применении интерфейса API невозможно предотвратить возникновение условий для исключительной ситуации, однако программист, пользуясь данным API, столкнувшись с этим исключением, мог бы предпринять какие-либо полезные действия. Если не выполняются оба этих условия, лучше пользоваться необрабатываемым исключением. Роль лакмусовой бумажки в данном случае играет вопрос: как программист будет обрабатывать исключение? Является ли это решение лучшим:

```
 } catch(TheCheckedException e) {  
     throw new Error("Assertion error");  
     // Условие не выполнено. Этого не должно быть никогда!  
 }
```

А что скажете об этом:

```
 } catch(TheCheckedException e) {  
     e.printStackTrace(); // Ладно, закончили работу.  
     System.exit(1);  
 }
```

Если программист, применяющий API, не может сделать ничего лучшего, то больше подходит необрабатываемое исключение. Примером исключения, не выдерживающего подобной проверки, является CloneNotSupportedException. Оно инициируется методом Object.clone, который должен использоваться лишь для объектов, реализующих интерфейс Cloneable (статья 11). Блок catch практически всегда соответствует невыполнению утверждения. Так что обрабатываемое исключение не дает программисту преимуществ, но требует от последнего дополнительных усилий и усложняет программу.

Дополнительные действия со стороны программиста, связанные с обработкой обрабатываемого исключения, значительно увеличиваются, если это единственное исключение, инициируемое данным методом. Если есть другие исключения, метод будет стоять в блоке `try`, так что для этого исключения понадобится всего лишь еще один блок `catch`. Если же метод инициирует только одно обрабатываемое исключение, оно будет требовать, чтобы вызов соответствующего метода был помещен в блок `try`. В таких условиях имеет смысл подумать: не существует ли какого-либо способа избежать обрабатываемого исключения.

Один из приемов, позволяющих превратить обрабатываемое исключение в необрабатываемое, состоит в разбиении метода, инициирующего исключение, на два метода, первый из которых будет возвращать булево значение, указывающее, будет ли инициироваться исключение. Таким образом, в результате преобразования API последовательность вызова

```
// Вызов с обрабатываемым исключением
try {
    obj.action(args);
} catch(TheCheckedException e) {
    // Обработать исключительную ситуацию
}
```

принимает следующий вид:

```
// Вызов с использованием метода проверки состояния
// и необрабатываемого исключения
if (obj.actionPermitted(args)) {
    obj.action(args);
} else{
    // Обработать исключительную ситуацию
}
```

Такое преобразование можно использовать не всегда. Если же оно допустимо, это может сделать работу с API более удобной.

Хотя второй вариант последовательности вызова выглядит не лучше первого, полученный API имеет большую гибкость. В ситуации, когда программист знает, что вызов будет успешным, или согласен на завершение потока в случае неудачного вызова, преобразованный API позволяет использовать следующую упрощенную последовательность вызова:

```
obj.action(args);
```

Если вы предполагаете, что применение упрощенной последовательности вызова будет нормой, то описанное преобразование API приемлемо. API, полученный в результате этого преобразования, в сущности, тот же самый, что и API с методом «проверки состояния» (статья 57). Следовательно, к нему относятся те же самые предупреждения: если к объекту одновременно и без внешней синхронизации могут иметь доступ сразу несколько потоков или этот объект может менять свое состояние по команде извне, указанное преобразование использовать не рекомендуется. Это связано с тем, что в промежутке между вызовом `actionPermitted` и вызовом `action` состояние объекта может успеть поменяться. Если метод `actionPermitted` при необходимости мог бы дублировать работу метода `action`, то от преобразования, вероятно, стоит отказаться по соображениям производительности.

Статья
60

Предпочитайте стандартные исключения

Одной из сильных сторон экспертов, отличающих их от менее опытных программистов, является то, что эксперты борются за высокую степень повторного использования программного кода и обычно этого добиваются. Общее правило, гласящее, что повторно используемый код — это хорошо, относится и к технологии исключений. В библиотеках для платформы Java реализован основной набор необрабатываемых исключений, перекрывающий большую часть

потребностей в исключениях для API. В этой статье обсуждаются наиболее часто применяемые исключения.

Повторное использование уже имеющихся исключений имеет несколько преимуществ. Главное то, что они упрощают освоение и применение вашего API, поскольку соответствуют установленным соглашениям, с которыми программисты уже знакомы. С этим же связано второе преимущество, которое заключается в том, что программы, использующие ваш API, легче читать, поскольку там 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`. Обычно оно инициируется, когда вызываемому методу передается аргумент с неправильным значением. Например, `IllegalArgumentException` может инициироваться в случае, если для параметра, указывающего количество повторов для некоей процедуры, передано отрицательное значение.

Другое часто используемое исключение — `IllegalStateException`. Оно обычно инициируется, если в соответствии с состоянием объекта вызов метода является неправомерным. Например, это исключение может инициироваться, когда делается попытка использовать некий объект до его инициализации надлежащим образом.

Вряд ли можно утверждать, что все неправильные вызовы методов сводятся к неправильным аргументам или неправильному состоянию, поскольку для определенных типов неправильных аргументов и состояний стандартно используются совсем другие исключения. Если при вызове какому-либо параметру было передано `null`, тогда как значения `null` для него запрещены, то в этом случае в соответствии с соглашениями должно инициироваться исключение `NullPointerException`, а не `IllegalArgumentException`. Точно так же, если параметру, который соответствует индексу некоей последовательности, при вызове было передано значение, выходящее за границы допустимого диапазона, инициироваться должно исключение `IndexOutOfBoundsException`, а не `IllegalArgumentException`.

Еще одно универсальное исключение, о котором необходимо знать: `ConcurrentModificationException`. Оно должно инициироваться, когда объект, предназначавшийся для работы в одном потоке или с внешней синхронизацией, обнаруживает, что его изменяют (или изменили) из параллельного потока.

Последнее универсальное исключение, заслуживающее упоминания, — `UnsupportedOperationException`. Оно инициируется, если объект не имеет поддержки производимой операции. По сравнению с другими исключениями, обсуждавшимися в этой статье, `UnsupportedOperationException` применяется довольно редко, поскольку большинство объектов обеспечивает поддержку всех реализуемых ими методов. Это исключение используется при такой реализации интерфейса, когда отсутствует поддержка одной или нескольких заявленных в нем дополнительных функций. Например, реализация интерфейса `List`, имеющая только функцию добавления элементов, будет инициировать это исключение, если кто-то попытается удалить элемент.

В таблице 8.1 собраны самые распространенные из повторно используемых исключений.

Таблица 8.1

Часто используемые исключения	
Исключение	Повод для использования
<code>IllegalArgumentException</code>	Неправильное значение параметра
<code>IllegalStateException</code>	Состояние объекта неприемлемо для вызова метода
<code>NullPointerException</code>	Значение параметра равно <code>null</code> , а это запрещено
<code>IndexOutOfBoundsException</code>	Значение параметра, задающего индекс, выходит за пределы диапазона
<code>ConcurrentModificationException</code>	Обнаружена параллельная модификация объекта из разных потоков, а это запрещено
<code>UnsupportedOperationException</code>	Объект не имеет поддержки указанного метода

Помимо перечисленных исключений, при определенных обстоятельствах могут применяться и другие исключения. Например, при реализации таких арифметических объектов, как комплексные числа и матрицы, уместно пользоваться исключениями `ArithmetricException` и `NumberFormatException`. Если исключение отвечает вашим потребностям, пользуйтесь им, но только чтобы условия, при которых вы будете его инициировать, не вступали в противоречие с документацией к этому исключению. Выбирая исключение, следует исходить из его семантики, а не только из названия. Кроме того, если вы хотите дополнить имеющееся исключение информацией об отказе (статья 63), не стесняйтесь создавать для него подклассы.

И наконец, учитывайте, что выбор исключения — не всегда точная наука, поскольку «поводы для использования», приведенные в таблице 8.1, не являются взаимоисключающими. Рассмотрим, например, объект, соответствующий колоде карт. Предположим, что для него есть метод, осуществляющий выдачу карт из колоды, причем в качестве аргумента ему передается количество требуемых карт. Допустим, что при вызове с этим параметром было передано значение, превышающее количество карт, оставшихся в колоде. Эту ситуацию можно толковать как `IllegalArgumentException` (значение параметра «размер сдачи» слишком велико) либо как `IllegalStateException` (объект «колода» содержит слишком мало карт для обработки запроса). В данном случае, по-видимому, следует использовать `IllegalArgumentException`, но непреложных правил здесь не существует.

Статья 61

Инициируйте исключения, соответствующие абстракции

Если метод инициирует исключение, не имеющее видимой связи с решаемой задачей, это сбивает с толку. Часто это происходит, когда метод передает исключение, инициированное абстракцией нижнего уровня. Это не только приводит в замешательство, но и засоряет интерфейс верхнего уровня деталями реализации. Если в следующей

версии реализация верхнего уровня поменяется, то также 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, инициировать исключения, которые можно объяснить в терминах абстракции верхнего уровня. Описываемая идиома, которую мы называем трансляцией исключений (*exception translation*), выглядит следующим образом:

```
// Трансляция исключения
try {
    // Использование абстракции нижнего уровня
    // для выполнения наших указаний
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Приведем конкретный пример трансляции исключения, взятый из класса `AbstractSequentialList`, который представляет собой скелетную реализацию (статья 18) интерфейса `List`. В этом примере трансляция исключения продиктована спецификацией метода `get` в интерфейсе `List`:

```
/**
 * Возвращает элемент, находящийся в указанной позиции
 * в заданном списке.
 * @throws IndexOutOfBoundsException, если индекс находится
 * за пределами диапазона (index < 0 || index >= size()).
 */
public Object get(int index) {
    ListIterator 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 при анализе ситуации, вызвавшей исключение, лучше использовать особый вид трансляции исключений, называемый сцеплением исключений (*exception chaining*). При этом исключение нижнего уровня передается с исключением верхнего уровня; в последнем создается открытый метод доступа, позволяющий извлечь исключение нижнего уровня:

```
// Сцепление исключений
try {
    // Использование абстракции нижнего уровня
    // для выполнения наших указаний
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(e);
}
```

Конструктор исключений верхнего уровня передает `cause` конструктору суперклассов, учитывающему сцепление, так что он окончательно передается конструктору `Throwable`, учитывающему сцепление:

```
// Исключения с помощью конструктора, учитывающего сцепление
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
    }
}
```

У большинства стандартных исключений имеются учитывающие сцепление конструкторы. Для исключений, у которых нет конструкторов, вы можете воспользоваться методом `initCause` объекта `Throwable`. Не только сцепление исключений дает вам программный доступ к `cause` (с помощью `getCause`), но еще он интегрирует его стек в исключение верхнего уровня.

Хотя трансляция исключений лучше, чем бессмысленная передача исключений с нижних уровней, злоупотреблять ею не следует.

Самый хороший способ обработки исключений нижнего уровня — полностью исключить их возможность. Для этого перед вызовом метода нижнего уровня необходимо убедиться в том, что он будет выполнен успешно. Иногда добиться этого можно путем явной проверки 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 какой-либо механизм регистрации, например `java.util.logging`, появившийся в версии 1.4. Это дает возможность администратору исследовать возникшую проблему и в то же время изолирует от нее программный код клиента и конечного пользователя.

В ситуациях, когда невозможно предотвратить возникновение исключений на нижних уровнях или изолировать от них верхние уровни, как правило, должен применяться механизм трансляции исключений. Непосредственную передачу исключений с нижележащего уровня на верхний следует разрешать только тогда, когда, исходя из описания метода на нижнем уровне, можно дать гарантию, что все инициируемые им исключения будут приемлемы для абстракции верхнего уровня.

Статья
62

Для каждого метода документируйте все инициируемые исключения

Описание инициируемых методом исключений составляет важную часть документации, которая необходима для правильного применения метода. Поэтому крайне важно, чтобы вы уделили время тщательному описанию всех исключений, инициируемых каждым методом.

Обрабатываемые исключения всегда декларируйте по отдельности с помощью тега `@throws` (Javadoc), четко описывайте условия, при которых каждое из них инициируется. Не пытайтесь сократить описание, объявляя о том, что метод инициирует некий суперкласс исключений, вместо того чтобы декларировать несколько классов возможных исключений. Например, никогда не объявляйте, что метод инициирует исключение `Exception` или, что еще хуже, исключение `Throwable`. Помимо того, что такая формулировка не дает программисту никакой информации о том, какие исключения могут быть инициированы данным методом, она значительно затрудняет работу с методом, поскольку надежно перекрывает любое другое исключение, которое может быть инициировано в этом же месте.

Хотя язык Java не требует, чтобы программисты декларировали необрабатываемые исключения, которые могут быть инициированы данным методом, имеет смысл документировать их столь же тщательно, как и обрабатываемые исключения. Необрабатываемые исключения обычно представляют ошибки программирования (статья 40), ознакомление программиста со всеми этими ошибками может помочь ему избежать их. Хорошо составленный перечень необрабатываемых исключений, которые может инициировать метод, фактически описывает предусловия для его успешного выполнения. Важно, чтобы в документации к каждому методу были описаны его предусловия, а описание необрабатываемых исключений как раз и является наилучшим способом выполнения этого требования.

Особенно важно, чтобы для методов интерфейса были описаны необрабатываемые исключения, которые могут быть ими инициированы. Такая документация является частью основных соглашений для интерфейса и обеспечивает единообразное поведение различных его реализаций.

Для описания каждого необрабатываемого исключения, которое может быть инициировано методом, используйте тег `@throws` (Javadoc), однако не нужно с помощью ключевого слова `throws` включать необрабатываемые исключения в декларацию метода. Програм-

мист, пользующийся вашим API, должен знать, какие из исключений обрабатываются, а какие — нет, поскольку в первом и втором случаях на него возлагается различная ответственность. Наличие описания, соответствующего тегу `@throws`, и отсутствие заголовка к методу, соответствующему декларации `throws`, создает мощный визуальный сигнал, помогающий программисту отличить обрабатываемые исключения от необрабатываемых.

Следует отметить, что документирование всех необрабатываемых исключений, которые могут быть инициированы каждым методом, — это идеал, который не всегда достижим в реальности. Когда производится пересмотр класса и предоставляемый пользователю метод меняется так, что начинает инициировать новые необрабатываемые исключения, это не является нарушением совместимости ни на уровне исходных текстов, ни на уровне байт-кода. Предположим, некий класс вызывает метод из другого класса, написанного независимо. Авторы первого класса могут тщательно документировать все необрабатываемые исключения, инициируемые каждым методом. Однако, если второй класс был изменен так, что теперь он инициирует дополнительные необрабатываемые исключения, первый класс (не претерпевший изменений) тоже будет передавать эти новые необрабатываемые исключения, хотя он их и не декларировал.

Если одно и то же исключение по одной и то же причине инициируется несколькими методами, его описание можно поместить в общий комментарий к документации для всего класса, а не описывать его отдельно для каждого метода. Примером такого рода является исключение `NullPointerException`. Прекрасно было бы в комментарии к классу сказать «все методы этого класса инициируют исключение `NullPointerException`, если с каким-либо параметром была передана нулевая ссылка на объект» или другие слова с тем же смыслом.

Подведем итоги. Необходимо документировать все исключения, которые могут быть выведены каждым написанным вами методом. Это относится как к обрабатываемым, так и к необрабатываемым исключениям, как к абстрактным, так и к конкретным методам. Для каждого

обрабатываемого исключения необходимо свое собственное выражение `throw`, не надо использовать это выражение для необрабатываемых исключений. Если вы не будете документировать исключения, которые выводят ваши методы, то будет сложно и даже невозможно другим людям использовать написанные вами классы и интерфейсы.

Статья
63

В описание исключения добавляйте информацию о сбое

Если выполнение программы завершается аварийно из-за необработанного исключения, система автоматически распечатывает трассировку стека 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 (*detail message*). Часто это единственная информация, с которой приходится иметь дело программистам или специалистам по наладке, исследующим сбой программы. И если воспроизвести этот сбой нелегко, то получить какую-либо еще информацию будет трудно или даже вообще невозможно. Поэтому крайне важно, чтобы метод `toString` в классе исключения возвращал как можно больше информации о причинах отказа. Иными словами, строковое представление исключения должно зафиксировать отказ для последующего анализа.

Для фиксации сбоя строковое представление исключения должно содержать значения всех параметров и полей, «способствовавших появлению этого исключения». Например, описание исключения `IndexOutOfBoundsException` должно содержать нижнюю границу, верхнюю границу и действительный индекс, который не уложился в эти границы. Такая информация говорит об отказе очень многое. Любое из трех значений или все они вместе могут быть неправильными. Представленный индекс может оказаться на единицу меньше нижней границы или быть равен верхней границе («ошибка границы» — *fencepost error*) либо мо-

жет иметь несуразное значение, как слишком маленькое, так и слишком большое. Нижняя граница может быть больше верхней (серьезная ошибка нарушения внутреннего инварианта). Каждая из этих ситуаций указывает на свою проблему, и, если программист знает, какого рода ошибку следует искать, это в огромной степени облегчает диагностику.

Хотя добавление в строковое представление исключения всех относящихся к делу «достоверных данных» является критическим, обычно нет надобности в том, чтобы оно было пространным. Трасировка стека, которая должна анализироваться вместе с исходными файлами приложения, как правило, содержит название файла и номер строки, где это исключение возникло, а также файлы и номера строк из стека, соответствующие всем остальным вызовам. Многословные пространные описания сбоя, как правило, излишни — необходимую информацию можно собрать, читая 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 описание исключения нужно главным образом программистам и специалистам по наладке для анализа причин сбоя. Поэтому содержащаяся в строковом представлении информация гораздо важнее его вразумительности.

Один из приемов, гарантирующих, что строковое представление исключения будет содержать информацию, достаточную для описания сбоя, состоит в том, чтобы эта информация запрашивалась в конструкторах исключения, а не в строке описания. Само же описание исключения можно затем генерировать автоматически для представления этой информации. Например, вместо конструктора `String` исключение `IndexOutOfBoundsException` могло бы иметь следующий конструктор:

```
/**  
 * Конструируем IndexOutOfBoundsException  
 *  
 * @param lowerBound – самое меньшее из разрешенных значений  
 * индекса
```

```
* @param upperBound – самое большое из разрешенных значений
* индекса
*           плюс один
* @param index – действительное значение индекса
*/
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                 int index) {
    // Генерируем описание исключения,
    // фиксирующее обстоятельства отказа
    super("Lower bound: " + lowerBound +
          ", Upper bound: " + upperBound +
          ", Index: " + index);
    // Сохраняем информацию об ошибке для программного доступа
    this.lowerBound = lowerBound;
    this.upperBound = upperBound;
    this.index = index;
}
```

К сожалению, хотя ее очень рекомендуют, эта идиома в библиотеках для платформы Java используется не слишком интенсивно. С ее помощью программист, инициирующий исключение, может с легкостью зафиксировать обстоятельства сбоя. Вместо того чтобы заставлять каждого пользующегося классом генерировать свое строковое представление, в этой идиоме собран фактически весь код, необходимый для того, чтобы качественное строковое представление генерировал сам класс исключения.

Как отмечалось в статье 58, 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 обрабатываемых исключений еще важнее, чем для необрабатываемых, поскольку информация об обстоятельствах сбоя может быть полезна для восстановления работоспособности программы. Программный доступ к деталям необрабатываемого исключения редко интересует программистов (хотя это и не исключено). Однако, согласно общему

принципу (статья 10), такие методы доступа имеет смысл создавать даже для необрабатываемых исключений.

Статья 64

Добавайтесь атомарности методов по отношению к сбоям

После того как объект инициирует исключение, обычно необходимо, чтобы он оставался во вполне определенном, пригодном для дальнейшей обработки состоянии, даже несмотря на то, что сбой произошел непосредственно в процессе выполнения операции. Особенно это касается обрабатываемых исключений, когда предполагается, что клиент будет восстанавливать работоспособность программы. Вообще говоря, вызов метода, завершившийся сбоем, должен оставлять обрабатываемый объект в том же состоянии, в каком тот был перед вызовом. Метод, обладающий таким свойством, называют *атомарным по отношению к сбою* (*failure atomic*).

Добиться такого эффекта можно несколькими способами. Простейший способ заключается в создании неизменяемых объектов (статья 15). 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 неопределенном состоянии, поскольку состояние каждого неизменяемого объекта согласуется 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 (статья 38). Благодаря этому любое исключение будет инициироваться до того, как начнется модификация объекта. В качестве примера рассмотрим метод Stack.pop из статьи 6:

```
public Object pop() {  
    if (size == 0)
```

```
        throw new EmptyStackException();
Object result = elements[-size];
elements[size] = null; // Убираем устаревшую ссылку
return result;
}
```

Если убрать начальную проверку размера, метод все равно будет инициировать исключение при попытке получить элемент из пустого стека. Однако при этом он будет оставлять поле `size` в неопределенном (отрицательном) состоянии. А это приведет к тому, что сбоем будет завершаться вызов любого метода в этом объекте. Кроме того, само исключение, инициируемое методом `pop`, не будет соответствовать текущему уровню абстракции (статья 61).

Другой прием, который тесно связан с предыдущим и позволяет добиться атомарности по отношению к сбоям, заключается в упорядочении вычислений таким образом, чтобы все фрагменты кода, способные повлечь сбой, предшествовали первому фрагменту, который модифицирует объект. Такой прием является естественным расширением предыдущего в случаях, когда невозможно произвести проверку аргументов, не выполнив хотя бы части вычислений. Например, рассмотрим случай с классом `TreeMap`, элементы которого сортируются по некоему 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 сбоем (и исключением `ClassCastException`), который произойдет в процессе поиска этого элемента в дереве, но до того, как в этом дереве что-либо будет изменено.

Третий, редко встречающийся прием заключается в написании специального кода восстановления (*recovery code*), который перехватывает сбой, возникающий в ходе выполнения операции, и заставляет объект вернуться в то состояние, в котором он находился в момент, предшествующий началу операции. Этот прием используется главным образом для структур, записываемых в базу данных.

Наконец, последний прием, позволяющий добиться атомарности метода, заключается в том, чтобы выполнять операцию на временной копии объекта и, как только операция будет завершена, заменять содержимое объекта содержимым его временной копии. Такой прием подходит для случая, когда вычисления могут быть выполнены намного быстрее, если поместить данные во временную структуру. Например, метод `Collections.sort` перед выполнением сортировки загружает полученный список в некий массив с тем, чтобы облегчить доступ к элементам во время внутреннего цикла сортировки. Это сделано для повышения производительности, однако имеет и другое дополнительное преимущество — гарантию того, что предоставленный методу список останется нетронутым, если процедура сортировки завершится сбоем.

К сожалению, не всегда можно достичь атомарности по отношению к отказам. Например, если два потока одновременно, без должной синхронизации пытаются модифицировать некий объект, последний может остаться в неопределенном состоянии. А потому после перехвата исключения `ConcurrentModificationException` нельзя полагаться на то, что объект все еще пригоден к использованию. Ошибки (в отличие от исключений), как правило, невосстановимы, и потому методам не нужно даже пытаться сохранять атомарность в случае появления ошибки.

Даже там, где можно получить атомарность по отношению к сбоям, она не всегда желательна. Для некоторых операций она существенно увеличивает затраты ресурсов и сложность вычислений. Вместе с тем очень часто это свойство достигается без особого труда, если хорошо разобраться с проблемой. Как правило, любое исключение, добавленное в спецификацию метода, должно оставлять объект в том состоянии, в котором он находился до вызова метода. В случае нарушения этого правила в документации API должно быть четко указано, в каком состоянии будет оставлен объект. К сожалению, множество имеющейся документации к API не стремится достичь этого идеала.

Статья
65

Не игнорируйте исключений

Этот совет кажется очевидным, но он нарушается настолько часто, что заслуживает повторения. Когда разработчики API декларируют, что некий метод инициирует исключение, этим они пытаются что-то вам сказать. Не игнорируйте это! Игнорировать исключения легко: необходимо всего лишь окружить вызов метода оператором `try` с пустым блоком `catch`:

```
// Пустой блок catch игнорирует исключение – крайне
// подозрительный код!
try {
    ...
} catch (SomeException e) {
```

Пустой блок `catch` лишает исключение смысла, который состоит в том, чтобы вы обрабатывали исключительную ситуацию. Игнорировать исключение — это все равно что игнорировать пожарную тревогу: выключить сирену, чтобы больше ни у кого не было возможности узнать, есть ли здесь настоящий пожар. Либо вам удастся всех обмануть, либо результаты окажутся катастрофическими. Когда бы вы ни увидели пустой блок `catch`, в вашей голове должна включаться сирена. Блок `catch` обязан содержать, по крайней мере, комментарий, объясняющий, почему данное исключение следует игнорировать.

Ситуацию, когда игнорирование исключений может оказаться целесообразным, иллюстрирует такой пример, как закрытие `FileInputStream`. Вы не изменили состояние файла, так что нет необходимости предпринимать действий к восстановлению, и вы уже прочитали информацию из файла, так что нет причины отменять текущую операцию. Даже в этом случае лучше записать исключение, чтобы вы могли позднее изучить данный вопрос, если такие исключения будут часто происходить.

Представленная в этой статье рекомендация в равной степени относится как к обрабатываемым, так и к необрабатываемым 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, будет работать дальше, никак на нее не реагируя. Затем в любой произвольный момент времени программа может завершиться с ошибкой, и программный код, где это произойдет, не будет иметь никакого отношения к действительному источнику проблемы. Должным образом обработав исключение, вы можете избежать отказа. Даже простая передача необрабатываемого исключения вовсе вызовет, по крайней мере, быстрый останов программы, при котором будет сохранена информация, полезная при устранении сбоя.

Г л а в а 10

Потоки

Потоки позволяют выполнять одновременно несколько операций в пределах одной программы. Многопоточное программирование сложнее однопоточного, потому что многое может пойти не так, а сбои воспроизвести довольно сложно. Но избежать его невозможно. Во многих случаях это необходимо, кроме того для получения хорошего результата с точки зрения производительности требуется многопроцессорная система, что сейчас довольно распространено. В этой главе содержатся советы, которые помогут вам создавать понятные, правильные и хорошо документированные программы для работы с потоками.

Статья
66

Синхронизируйте доступ потоков к совместно используемым изменяемым данным

Использование ключевого слова `synchronized` дает гарантию, что в данный момент времени некий оператор или блок будет выполняться только в одном потоке. Многие программисты рассматривают синхронизацию лишь как средство блокировки потоков, которое не позволяет одному потоку наблюдать объект в промежуточном

состоянии, пока тот модифицируется другим потоком. С этой точки зрения, объект создается с согласованным состоянием (статья 13), а затем блокируется методами, имеющими к нему доступ. Эти методы следят за состоянием объекта и (дополнительно) могут вызывать для него *переход состояния* (*state transition*), переводя объект из одного согласованного состояния в другое. Правильное выполнение синхронизации гарантирует, что ни один метод никогда не сможет наблюдать этот объект в промежуточном состоянии.

Такая точка зрения верна, но не отражает всей картины.

Без синхронизации изменения в одном потоке не видны другому. Синхронизация не только дает потоку возможность наблюдать объект в промежуточном состоянии, но также дает гарантию, что объект будет переходить из одного согласованного состояния в другое в результате четкого выполнения последовательности шагов. Каждый поток, попадая в синхронизированный метод или блок, видит результаты выполнения всех предыдущих переходов под управлением того же самого кода блокировки.

Спецификация языка Java дает гарантию, что чтение и запись отдельной переменной, если это не переменная типа long или double, являются атомарными операциями [JLS, 17.4.7]. Иными словами, гарантировано, что при чтении переменной (кроме long и double) будет возвращаться значение, которое было записано в эту переменную одним из потоков, даже если новые значения в эту переменную без какой-либо синхронизации одновременно записывают несколько потоков.

Возможно, вы слышали, что для повышения производительности при чтении и записи атомарных данных нужно избегать синхронизации. Это неправильный совет с опасными последствиями. Хотя свойство атомарности гарантирует, что при чтении атомарных данных поток не увидит случайного значения, нет гарантии, что значение, записанное одним потоком, будет увидено другим: синхронизация необходима как для блокирования потоков, так и для надежного взаимодействия между ними. Это является следствием сугубо технического аспекта языка программирования Java, который называется моделью памяти (*memory model*) [JLS, 17].

Отсутствие синхронизации для доступа к совместно используемой переменной может иметь серьезные последствия, даже если переменная имеет свойство атомарности как при чтении, так и при записи.

Рассмотрим задачу остановки одного потока из другого. У библиотек есть метод Thread.stop, но он давно устарел и является *небезопасным*: работа с ним может привести к разрушению данных. **Не используйте Thread.stop.** Для остановки одного потока из другого рекомендуется использовать прием, который заключается в том, чтобы в одном потоке создать некое опрашиваемое поле, значение которого по умолчанию false, но может быть установлено true вторым потоком, для указания, что первый поток должен остановить сам себя. Обычно такое поле имеет тип boolean. Поскольку чтение и запись такого поля атомарны, у некоторых программистов появляется соблазн предоставить ему доступ без синхронизации:

```
// Ошибка! - Как вы думаете, как долго будет выполняться эта программа?  
public class StopThread {  
    private static boolean stopRequested;  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread backgroundThread = new Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stopRequested)  
                    i++;  
            }  
        });  
        backgroundThread.start();  
        TimeUnit.SECONDS.sleep(1);  
        stopRequested = true;  
    }  
}
```

Вы, возможно, думаете, что такая программа выполнится за секунду, после чего поток поменяет значение stopRequested на true, что приведет к тому, что цикл фонового потока завершится. Тем не менее

на моей машине программа никогда не завершается: цикл фонового потока выполняется вечно!

Проблема представленного кода заключается в том, что в отсутствие синхронизации нет гарантии (если ее вообще можно дать), что поток, подлежащий остановке, «увидит», что основной поток поменял значение stopRequested. В результате метод requestStop может оказаться абсолютно неэффективным. При отсутствии синхронизации для виртуальной машины приемлемо преобразовать данный код:

```
while (!stopRequested)
    i++;
```

в такой:

```
if (!stopRequested)
    while (true)
        i++;
```

Эта оптимизация называется *поднятием*, и это именно то, что делает сервер виртуальной машины HotSpot. Результат — падение живучести: программе не удается успешно завершиться. Один из способов разрешить эту проблему — непосредственно синхронизировать доступ к полю stopRequested:

```
// Правильно синхронизированное совместное завершение потока
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(new Runnable() {
            public void run() {
                int i = 0;
```

```
        while (!stopRequested())
            i++;
    }
});

backgroundThread.start();
TimeUnit.SECONDS.sleep(1);
requestStop();
```

Заметим, что и метод записи (`requestStop`), и метод чтения (`stopRequested`) синхронизированы. Синхронизировать только метод записи недостаточно. На самом деле **от синхронизации нет пользы, если и операции чтения и записи не синхронизированы**.

Действия синхронизированных методов `StopThread` были бы автоматными 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 альтернатива, которая не будет настолько нагружена текстом и производительность которой будет лучше. Блокировку во второй версии `StopThread` можно, если `stopRequested` будет объявлен непостоянным (`volatile`). Хотя модификатор `volatile` не выполняет взаимного исключения, он гарантирует, что любой поток, который прочитает поле, увидит недавно записанные значения:

```
// Совместное завершение потоков с помощью поля volatile
public class StopThread {
    private static volatile boolean stopRequested;
    public static void main(String[] args)
        throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                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:

```

// Ошибка – требуется синхронизация!
private static volatile int nextSerialNumber = 0;
public static int generateSerialNumber() {
    return nextSerialNumber++;
}

```

Этот метод должен гарантировать, что при каждом вызове будет возвращаться другой серийный номер до тех пор, пока не будет возвращено иное значение (пока не будет произведено 2³² вызова). Состояние генератора включает лишь одно атомарно записываемое поле (`nextSerialNumber`), для которого допустимы любые возможные значения. Для защиты инвариантов данного генератора серийных номеров синхронизация не нужна. Тем не менее без синхронизации этот метод не работает.

Проблема в том, что оператор приращения `(++)` атомарным не является. Он выполняет две операции в поле `nextSerialNumber`: сначала он читает его значение, потом записывает новое значение, равное старому плюс один. Если второй поток читает значение между периодами чтения первым потоком старых и записи новых значений, второй поток будет видеть те же значения, что и первый, и возвращать тот же серийный номер. Чтение и запись — это независимые операции, которые выполняются последовательно. Соответственно, несколько параллельных потоков могут наблюдать в поле `nextSerialNumber` одно и то же значение и возвращать один и тот же серийный номер. Это ошибка безопасности: программа считает неверные результаты.

Одним из способов исправления метода `generateSerialNumber` сводится к простому добавлению в его декларацию слова `synchron-`

nized. Тем самым гарантируется, что различные вызовы не будет смешиваться и каждый новый вызов будет видеть результат обработки всех предыдущих обращений. После того как вы сделаете это, вам необходимо удалить модификатор volatile из nextSerialNumber. Чтобы сделать этот метод «железобетонным», возможно, имеет смысл заменить int на long или инициализировать какое-либо исключение если nextSerialNumber будет близко к переполнению.

Все же лучше последовать совету из статьи 47 и использовать класс AtomicLong, являющийся частью java.concurrent.atomic. Он делает как раз то, что вы хотите, и, скорее всего, его производительность будет лучше, чем синхронизированная версия generalSerialNumber:

```
private static final AtomicLong nextSerialNum = new AtomicLong();
public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

Лучший способ избежать проблем, описанных в этой статье, — не делать общими изменяемые данные. Либо делать общими неизменяемые данные (статья 15), либо вообще общими ничего не делать. Другими словами, изменяемые данные держите в одном потоке. Если вы примените данную политику, необходимо это документировать, чтобы при развитии программы это можно было сохранить. Также необходимо глубокое понимание структур и библиотек, которыми вы пользуетесь, так как они могут запускать потоки, о которых вы можете не знать.

Вполне допустимо, если один поток изменит объект данных на некоторое время, а затем сделает его общим для других потоков, синхронизируя только действие обобщения ссылки на объект. Другие потоки тогда смогут читать объект без дальнейшей синхронизации до тех пор, пока он снова не изменится. Такие объекты называются эффективно неизменяемыми [Goetz06, 3.5.411]. Преобразование такой ссылки на объект от одного потока на другой называется безопасной публикацией [Goetz06, 3.5.3]. Существует много путей

для безопасной публикации ссылки на объект: вы можете хранить ее в статическом поле как часть инициализации класса; вы также можете хранить ее в непостоянном поле, в завершенном поле или в поле, к которому имеется доступ с помощью нормальной блокировки; или же вы можете поместить его в параллельную коллекцию (статья 69).

Подведем итоги. **Когда несколько потоков совместно работают с изменяемыми данными, каждый поток, который читает или записывает эти данные, должен пользоваться блокировкой.** Без синхронизации невозможно дать какую-либо гарантию, что изменения в объекте, сделанные одним потоком, были увидены другим. Несинхронизированный доступ к данным может привести к отказам, затрагивающим живучесть и безопасность системы. Воспроизвести такие отказы бывает крайне сложно. Они могут зависеть от времени и быть чрезвычайно чувствительны к деталям реализации JVM и особенностям компьютера. Если вам требуется только связь между потоками, модификатор `volatile` является подходящей формой синхронизации, но может быть довольно запутанным, если использовать его некорректно.

Статья
67

Избегайте избыточной синхронизации

Статья 66 предупреждает об опасностях недостаточной синхронизации. Данная статья посвящена обратной проблеме. В зависимости от ситуации избыточная синхронизация может приводить к снижению производительности приложения, взаимной блокировке потоков или даже к непредсказуемому поведению программы.

Для исключения возможности взаимной блокировки (*deadlock*) никогда не передавайте управление клиенту, если находитесь в синхронизированном методе или блоке. Иными словами, из области синхронизации не следует вызывать открытые или

защищенные методы, которые предназначены для переопределения, или метод, предоставленный клиентом в форме объекта функции (статья 21). С точки зрения класса, содержащего синхронизированную область, такой метод является чужим. У класса нет сведений о том, что этот метод делает, нет над ним контроля. В зависимости от того, что делает чужой метод, вызов его из синхронизированной области может привести к исключениям, блокировке или повреждению данных.

Для пояснения рассмотрим класс, который реализует упаковщик набора *Observable*. Он позволяет клиентам подписываться на уведомления каждый раз, когда к набору добавляется элемент. Это шаблон *Observable* [Gamma95, стр. 293]. Для краткости, класс не предоставляет уведомлений, когда элементы удаляются из набора, но настроить такие уведомления не составит труда. Этот класс реализуется поверх *ForwardingSet* из статьи 16, который можно реализовывать повторно:

```
// Ошибка - запускает чужой метод из синхронизированного блока!
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }
    private final List<SetObserver<E>> observers =
        new ArrayList<SetObserver<E>>();
    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); // calls notifyElementAdded
        return result;
    }
}

```

Наблюдатели подписываются на уведомления, запуская метод `addObserver`, и отписываются запуском метода `removeObserver`. В обоих случаях экземпляр интерфейса обратного вызова (*callback*) передается методу:

```

public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
}

```

При кратком взгляде `ObserverSet` работает. Например, следующая программа печатает числа от 0 до 99:

```

public static void main(String[] args) {
    ObservableSet<Integer> set =
        new ObservableSet<Integer>(new HashSet<Integer>());
    set.addObserver(new SetObserver<Integer>() {
        public void added(ObservableSet<Integer> s, Integer e) {
            System.out.println(e);
        }
    });
    for (int i = 0; i < 100; i++)
        set.add(i);
}

```

Теперь попробуем нечто более необычное. Предположим, мы заменим вызов `addObserver` другим методом, который передает наблюдателю, который печатает значение `Integer`, которое было добавлено к набору и удаляет его, если значение равно 23:

```
set.addObserver(new SetObserver<Integer>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) s.removeObserver(this);
    }
});
```

Вы можете подумать, что программа напечатает числа от 0 до 23, после чего наблюдатель отписывается и программа тихо завершает работу. Но на самом деле она выводит значения от 0 до 23, после чего выводит ошибку `currentModificationException`. Проблема в том, что `notifyElementAdded` находится в процессе итерации поверх списка `observers`, когда запускается метод для наблюдателя `added`. Метод `added` вызывает метод `removeObserver` набора `Observable`, который в свою очередь вызывает `observer.remove`. Теперь у нас проблема. Мы пытаемся удалить элемент из списка во время итерации над ним, что недопустимо. Итерация метода `notifyElementAdded` находится в синхронизированном блоке, чтобы воспрепятствовать параллельному изменению, но он не препятствует потоку, в котором происходит итерация, делать обратный вызов к набору `observable` и изменению его списка `observers`.

Попробуем теперь нечто странное: напишем попытку наблюдателя отписаться, но вместо непосредственного вызова `removeObserver` мы для этого воспользуемся услугами другого потока. Наблюдатель использует службу `executor` (статья 68).

```
// Наблюдатель, использующий без надобности фоновый поток
set.addObserver(new SetObserver<Integer>() {
    public void added(final ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
```

```

ExecutorService executor =
Executors.newSingleThreadExecutor();
final Set<Observer<Integer>> observer = this;
try {
    executor.submit(new Runnable() {
        public void run() {
            s.removeObserver(observer);
        }
    }).get();
} catch (ExecutionException ex) {
    throw new AssertionError(ex.getCause());
} catch (InterruptedException ex) {
    throw new AssertionError(ex);
} finally {
    executor.shutdown();
}
}
});

```

На этот раз ошибка исключения не выводится — мы получаем блокировку. Фоновый поток вызывает `s.removeObserver`, который пытается заблокировать `observers`, но он не может применить блокировку, потому что основной поток ее уже применил. В любом случае основной поток ждет, когда фоновый поток завершит удаление наблюдателя, что объясняет блокировку.

Этот пример неестественен, так как нет причин для наблюдателя использовать фоновый поток, но проблема реальна. Запуск чужого метода из синхронизированной области вызвал много блокировок в реальной системе, такой как набор GUI.

В обоих предыдущих примерах (исключение и блокировка) нам повезло. Ресурсы, которые защищены синхронизированной областью (`observers`), находились в стабильном состоянии на момент запуска чужого метода (`added`). Предположим, вам пришлось запустить чужой метод из синхронизированного потока в тот момент,

когда инвариант, защищенный синхронизированной областью, был временно недействителен. Поскольку блокировки в языке Java являются возвращающимися, такие вызовы не приведут к блокировке. Как и в первом примере, который завершился исключением, вызываемый поток уже содержит блокировку, так что поток завершится успешно, если он попытается повторно применить блокировку, даже если другая, концептуально несвязанная операция будет выполняться с данными, защищенными блокировкой. Последствия такого сбоя могут быть катастрофическими. По сути, блокировка нам удалась. Возвращающиеся блокировки позволяют упростить создание многопоточных объектно ориентированных программ, но они могут сбоя живучести превратить в сбои безопасности.

К счастью, обычно не слишком сложно решить такую проблему путем перемещения запуска чужого метода за пределы синхронизированных блоков. Для метода `notifyElementAdded` это подразумевает копию состояния списка `observers`, через который затем можно безопасно пройти без блокировки. При выполнении этих изменений оба предыдущих примера будут выполняться без ошибок и блокировок:

```
// Чужой метод перемещен за пределы синхронизированного блока -
// открытые вызовы
private void notifyElementAdded(E element) {
    List<Set0bserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<Set0bserver<E>>(observers);
    }
    for (Set0bserver<E> observer : snapshot)
        observer.added(this, element);
}
```

На самом деле есть лучший способ переместить запуск чужого метода за рамки синхронизированного блока. В версии 1.5 библиотеки Java дают нам параллельную коллекцию (статья 69), известную как `CopyOnWriteArrayList`, который специально сделан для этой цели. Это вариант `ArrayList`, в котором все операции записи реализуются

тем, что делаются свежие копии всего базового массива. Поскольку внутренний массив никогда не изменяется, итерации не требуется блокировка и выполняется она очень быстро. В большинстве случаев применения производительность `CopyOnWriteArrayList`, будет ужасной, но для списка наблюдателей он подойдет идеально, так как он редко меняется и очень часто через него можно пройти.

Методы `add` и `addAll` набора `ObservableSet` нет необходимости менять, если список изменится при использовании `CopyOnWriteArrayList`. Вот как выглядит оставшаяся часть класса. Обратите внимание, что здесь нет явной синхронизации:

```
// Безопасный набор observable с CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
    new CopyOnWriteArrayList<SetObserver<E>>();
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, запущенный за пределами синхронизированной области, известен как *открытый вызов* [Lea00 2.4.1.3]. Кроме того, что открытые вызовы предотвращают сбои, они могут существенно увеличить параллельность. Чужой метод может выполняться в течение произвольно длительного периода. Если чужой метод был запущен из синхронизируемой области, другие потоки не получат доступ к защищенным ресурсам без надобности.

Как правило, вам нужно выполнять как можно меньше действий внутри синхронизируемой области. Получить блокировку, просмотреть общие данные, преобразовать их как надо и затем убрать блокировку. Если вам требуется выполнение длительных действий,

найдите способ, чтобы переместить эти действия за пределы синхронизируемой области, не нарушая инструкций (статья 66).

Первая часть этой статьи была посвящена правильности. Теперь вкратце рассмотрим вопрос производительности. Хотя затраты на синхронизацию уменьшились еще на заре появления языка Java, тем не менее важно не переусердствовать с синхронизацией. В мультипроцессорной среде реальные затраты на излишнюю синхронизацию — это не время, затрачиваемое процессором на получение блокировок, а утерянные возможности параллельного запуска и задержки, вызванные необходимостью убедиться в том, что каждое ядро видит всю память постоянно. Другие скрытые затраты от излишней синхронизации — это то, что она может ограничить возможность виртуальной машины оптимизировать код при выполнении.

Вам необходимо сделать неизменяемый класс потокобезопасным (статья 70), если предполагается его параллельное использование и вы можете достичь большей параллельности выполняя синхронизацию изнутри, чем путем блокирования всего объекта извне. В противном случае не синхронизируйте изнутри. Пусть клиенты синхронизируются извне, где это приемлемо. Вначале, при появлении платформы Java многие классы нарушали данные инструкции. Например, экземпляры `StringBuffer` почти всегда используются одним потоком, но все-таки выполняют синхронизацию изнутри. Именно поэтому в версии 1.5 он был заменен на `StringBuilder`, который представляет собой несинхронизируемую версию `StringBuffer`. Если вы сомневаетесь, то не синхронизируйте ваш класс, но укажите в документации, что он не потокобезопасен (статья 70).

Если вы синхронизируете класс изнутри, вы можете пользоваться различными приемами для достижения параллельности, такими как разделение блокировки, распределение блокировки и контроль параллельности без блокировки. Эти приемы выходят за рамки этой книги, но описываются другими [Goetz06, Lee00].

Если класс или статический метод связан с изменяемым статическим полем, он должен иметь внутреннюю синхронизацию,

даже если обычно применяется только с одним потоком. В противоположность совместно используемому экземпляру здесь клиент не имеет возможности произвести внешнюю синхронизацию, поскольку нет никакой гарантии, что другие клиенты будут делать то же самое. Эту ситуацию иллюстрирует статический метод generateSerialNumber (статья 66).

Подведем итоги. Во избежание взаимной блокировки потоков и разрушения данных никогда не вызывайте чужие методы из синхронизированной области. Постарайтесь ограничить объем работы, выполняемой вами в синхронизированных областях. Проектируя изменяемый класс, подумайте о том, не должен ли он иметь свою собственную синхронизацию. Выигрыш, который вы рассчитываете получить, отказываясь от синхронизации, теперь уже не такой громадный, а вполне умеренный. Ваше решение должно исходить из того, будет ли ваша абстракция использоваться для работы с несколькими потоками. Четко документируйте свое решение.

В современном многопроцессорном мире как никогда важно, чтобы не злоупотреблять синхронизацией. Синхронизируйте класс изнутри, только если для этого есть хорошая причина, и четко документируйте ваше решение (статья 70).

Статья
68

Предпочитайте использование эзекуторов и заданий вместо потоков

В первой редакции книги содержался код для простой рабочей очереди [Bloch01, статья 49]. Этот класс позволял клиентам ставить в очередь рабочие задачи для асинхронной обработки фоновым потоком. Когда рабочая очередь становится более не нужна, клиенты могут запустить метод, чтобы попросить фоновый поток завершить самого себя после завершения работы, уже стоящей в очереди. Данная реализация была не более чем игрушкой, но для ее реализации тре-

бовалась целая страница очень тонкого кода, который был подвержен сбоям безопасности и живучести. К счастью, более нет причин писать такой код.

В версии 1.5 платформы появился `java.util.concurrent`. Этот пакет содержит структуру экзекуторов, являющуюся довольно гибкой, с основанной на интерфейсах возможностью выполнения задач. Создание рабочих очередей, которое сейчас реализовано лучше описанной в первой редакции книги, требует всего лишь одной строки кода:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Вот как можно запустить выполнение:

```
executor.execute(runnable);
```

А вот как корректно его завершить (если у вас не получится, то, скорее всего, ваша виртуальная машина не закроется):

```
executor.shutdown();
```

С помощью службы экзекуторов вы можете сделать больше вещей. Например, вы можете подождать, пока завершится определенная задача (как в «фоновом потоке SetObserver», описанном в статье 67), или подождать, пока одна или все задачи из коллекции задач завершатся (используя методы `invokeAny` или `invokeAll`). Можно подождать, пока служба экзекуторов корректно завершится (используя метод `awaitTermination`), вы сможете вывести результаты задач один за другим по мере их завершения (используя `ExecutorCompletionService`) и.т.д.

Если вы хотите, чтобы запросы из очереди обрабатывались более чем одним потоком, просто вызовите другой метод статической генерации, который создает другой вид службы экзекуторов, называемый *пул потоков*. Вы можете создавать пул потоков с фиксированным количеством переменных потоков. Класс `java.util.concurrent.Executors` содержит методы статической генерации, предоставляющие большинство экзекуторов, которые вам когда-либо понадобятся. Если вы захотите что-то необычное, вы можете использовать не-

посредственно класс ThreadPoolExecutor. Этот класс позволяет вам контролировать почти каждый аспект работы пула потоков.

Выбор службы экзекуторов для определенного приложения может быть довольно запутанным. Если вы пишете небольшую программу или легко нагруженный сервер, использование Executor.newCachedThreadPool обычно является хорошим выбором, поскольку он не требует конфигурации и «все делает правильно». Но выбор кэшированного пула потоков для тяжело нагруженных серверов будет неудачным. В кэшированном пуле потоков поставленные задачи не ставятся в очередь, а отправляются сразу на выполнение. Если нет доступных потоков, то просто создается новый. Если сервер нагружен настолько, что использует процессор на полную мощность, то создание новых потоков по мере поступления задач только ухудшит ситуацию. Следовательно, при тяжело нагруженном сервере вам лучше использовать Executor.newFixedThreadPool, который дает нам фиксированное число потоков, или использовать непосредственно класс ThreadPoolExecutor для максимального контроля.

Вам не только следует воздержаться от написания собственных рабочих очередей, но также и от работы непосредственно с потоками. Ключевым понятием больше не является Thread, который служил ранее в качестве и рабочей единицы и механизма его выполнения. Теперь рабочая единица и механизм разделены. Ключевым понятием теперь является рабочая единица, которая называется задачей. Есть два вида задач: Runnable и близкий ему Callable (который похож на Runnable, за исключением того, что он возвращает значение). Общий механизм выполнения задач называется службой экзекуторов. Если вы мыслите задачами и позволяете экзекуторам выполнять их для вас, вы получите огромную выгоду в гибкости в плане выбора подходящей политики выполнения. По сути, структура экзекуторов делает для выполнения то, что структура коллекций делает для накопления.

У структуры экзекуторов есть замена для java.util.Timer, которой является ScheduledThreadPoolExecutor. Хотя использовать

Timer легче, экзекутор пула потоков с расписанием является более гибким. Timer использует только один поток для выполнения задач. Если единственный поток Timer приведет к исключению, Timer прекратит работу. А экзекутор пула потоков с расписанием поддерживает многопоточность и корректно восстанавливается после необработанного исключения.

Полное описание структуры потоков выходит за рамки данной книги, но заинтересованный читатель может найти его в *Java Concurrency in Practice* [Goetz06].

Статья
69

Предпочитайте использовать утилиты параллельности, нежели `wait` и `notify`

В первой редакции этой книги была статья, посвященная корректному использованию `wait` и `notify` (Blocj01, статья 50). Содержащиеся в ней советы все еще актуальны и подытожены в конце этой статьи, но эти советы теперь имеют намного меньшее значение. Это произошло потому, что сейчас намного меньше причин для использования `wait` и `notify`. В версию 1.5 платформы Java включены утилиты параллельности верхнего уровня, которые делают те вещи, которые вы раньше писали вручную поверх `wait` и `notify`. **Зная о трудности использования `wait` и `notify`, вам вместо этого следует использовать высокоуровневые утилиты параллельности.**

Эти утилиты, содержащиеся в `java.util.concurrent`, делятся на три категории: структура экзекуторов, которая была кратко описана в статье 68, параллельные коллекции и синхронизаторы. Параллельные коллекции и синхронизаторы кратко описаны в этой статье.

Параллельные коллекции дают нам высокопроизводительные параллельные реализации стандартных интерфейсов коллекций, таких как `List`, `Queue` и `Map`. Для обеспечения высокого уровня параллельности эти реализации сами управляют своей синхронизацией изнутри (статья 67). Следовательно, **невозможно исключить параллельную**

деятельность из параллельной коллекции, ее блокировка не будет иметь действие, а только затормозит выполнение программы.

Это значит, что клиенты не могут атомарно создавать запуски методов на параллельных коллекциях. Некоторые из этих интерфейсов коллекций расширены *операциями изменения в зависимости от состояния* (*state-dependent modify operations*), которые объединяют в себе несколько примитивных операций в одну атомарную операцию. Например, ConcurrentMap расширяет Map и добавляет несколько методов, включая putIfAbsent(key, value), которые вставляют схему в ключ, если таковой не было, и возвращает предыдущее значение, ассоциированное с ключом или null, если такого нет. Это облегчает реализацию традиционных потокобезопасных схем. Например, этот метод имитирует поведение String.intern:

```
// Параллельная традиционная схема поверх ConcurrentHashMap -  
// не оптимальна  
private static final ConcurrentHashMap<String, String> map =  
    new ConcurrentHashMap<String, String>();  
public static String intern(String s) {  
    String previousValue = map.putIfAbsent(s, s);
```

В действительности можно сделать лучше. ConcurrentHashMap оптимизирован специально для операций извлечения, таких как get. Следовательно, имеет смысл запускать get вначале и вызывать putIfAbsent, если только get покажет, что это необходимо:

```
// Параллельная традиционная схема поверх ConcurrentHashMap -  
// работает быстрее!  
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 отличной параллельности, ConcurrentHashMap еще и работает быстрее. На моей машине оптимизированный метод intern работает в шесть раз быстрее, чем String.intern (но имейте в виду, что String.intern должен использовать слабую ссылку, чтобы избежать утечки памяти с течением времени). Если у вас нет серьезной причины сделать по-другому, **используйте ConcurrentHashMap вместо Collections.synchronizedMap или HashTable**. Простая замена старой синхронизированной схемы параллельными схемами может серьезно увеличить производительность параллельных приложений. Говоря более общо, предпочитайте использование параллельных коллекций вместо синхронизируемых извне коллекций.

Некоторые интерфейсы коллекций были расширены блокирующими операциями, которые ждут (или блокируют) до тех пор, пока они не смогут успешно выполниться. Например, BlockingQueue расширяет Queue и добавляет несколько методов, в том числе take, которые удаляют и возвращают головной элемент из очереди, ожидая, если очередь пуста. Это позволяет использовать блокирующие очереди для рабочих очередей (также известные как очереди производитель-потребитель), в которые один или несколько производящих потоков ставят рабочие задания и из которых один или более потребляющих потоков убирают и обрабатывают задания, как только они становятся доступными. Как вы можете ожидать, большинство реализаций ExecutorService, в том числе ThreadPoolExecutor, используют BlockingQueue ([статья 68](#)).

Синхронизаторы — объекты, которые дают возможность потокам ждать друг друга, позволяя им координировать их деятельность. Наиболее часто используемые синхронизаторы CountDownLatch и Semaphore. Наименее используемые CyclicBarrier и Exchanger.

Синхронизаторы CountDownLatch — это одноразовые барьеры, которые позволяют одному или более потокам ждать, когда один или более поток что-то сделает. Единственный конструктор для CountDownLatch берет int, который является количеством запуска метода CountDown, и помещает его на замок до тех пор, пока всем ожидающим потокам будет разрешено продолжить.

Удивительно легко создавать полезные вещи поверх простых примитивных. Например, предположим, что вы хотите создать простую структуру для подсчета времени параллельного выполнения действия. Эта структура состоит из одного метода, который берет экзекутор для выполнения действия, уровень параллельности, представляющий собой количество действий, которые должны параллельно выполняться, и `Runnable`, представляющий собой действие. Все рабочие потоки готовят сами себя для выполнения действия до того, как поток таймера запустит часы (это необходимо для получения точного результата). Когда последний рабочий поток будет готов к выполнению действия, `time` «нажимает на спусковой крючок», позволяя рабочим потокам выполнять действие. 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 будет удивительно проста:

```
// Простая структура для подсчета времени параллельного выполнения
public static long time(Executor executor, int concurrency,
final Runnable action) throws InterruptedException {
    final CountDownLatch ready = new CountDownLatch(concurrency);
    final CountDownLatch start = new CountDownLatch(1);
    final CountDownLatch done = new CountDownLatch(concurrency);
    for (int i = 0; i < concurrency; i++) {
        executor.execute(new Runnable() {
            public void run() {
                ready.countDown(); // Tell timer we're ready
                try {
                    start.await();
                    // Wait till peers are ready
                    action.run();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    done.countDown();
                }
            }
        });
    }
    start.countDown();
    try {
        done.await();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return done.getCount();
}
```

```
        // Tell timer we're done
    }
}

});

ready.await(); // Wait for all workers to be ready
long startNanos = System.nanoTime();
start.countDown(); // And they're off!
done.await(); // Wait for all workers to finish
return System.nanoTime() - startNanos;
}
```

Обратите внимание, что метод использует три защелки для подсчета. Первая, `ready`, используется рабочими потоками, чтобы сказать потоку таймера, когда он готов. Рабочие потоки затем ждут второй защелки, которой является `start`. Последний рабочий поток запускает `ready.countDown`, позволяя всем рабочим потокам продолжать. Поток таймера ждет третьей защелки, `done`, пока последние рабочие потоки закончат выполнение действия и вызовут `done.countDown`. Как только это произойдет, поток таймера пробуждается и фиксирует время окончания. Стоит упомянуть еще несколько деталей. Экзекутор, который передается методу `time`, должен позволить создание по крайней мере такого количества потоков, которое определено уровнем параллельности, иначе тест никогда не завершится. Это называется *блокировка потока от зависания* [Goetz06, 8.1.1]. Если рабочий поток натолкнется на `InterruptedException`, он выполняет прерывание, используя идиому `Thread.currentThread().interrupt()`, и возвращает его метод `run`. Это позволяет экзекутору поступать с прерыванием так, как он считает нужным, что и должно быть. Наконец, обратите внимание, что `System.nanoTime` используется для записи времени деятельности чаще, чем `System.currentTimeMillis`. Для записи времени по интервалам всегда лучше использовать `System.nanoTime`, чем `System.currentTimeMillis`. `System.nanoTime` более аккуратен и более точен, а также на него не влияют настройки системного времени.

Эта статья лишь поверхностно освещает утилиты параллельности. Например, три защелки счетчика из предыдущего примера могут заменяться одним циклическим барьером. Получающийся код даже более краткий, но его труднее понять. Для более детальной информации см. *Java Concurrency in Practice* [Goetz06]. Вам всегда следует предпочитать использовать утилиты параллельности вместо `wait` и `notify`. Метод `wait` используется, чтобы заставить поток ждать наступления определенных условий. Он должен запускаться в синхронизированной области, которая запирает объект, на котором он запускается. Вот стандартная идиома для использования метода `wait`:

```
// Стандартная идиома, использующая метод wait
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait(); // (Releases lock, and reacquires on wakeup)
    ... // Perform action appropriate to condition
}
```

Всегда используйте идиому цикла `wait` для запуска метода `wait`; никогда не запускайте ее вне цикла. Цикл служит для проверки условий до и после ожидания.

Проверка условия перед ожиданием и пропуск ожидания, если условие условия уже зафиксировано, необходимы для гарантии живучести. Если условие уже зафиксировано и метод `notify` (или `notifyAll`) уже запущен перед тем, как поток переходит к ожиданию, нет гарантии, что поток когда-либо проснется от ожидания.

Проверка условия после ожидания и ожидание, когда условие не будет выполняться, необходимы для обеспечения безопасности. Если поток продолжает работать, когда условию он уже не отвечает, он может уничтожить инвариант, охраняемый блокировкой. Есть несколько причин, почему поток должен проснуться, даже если условие не соблюдается:

Другой поток получил блокировку и изменил защищенное состояние в период между временем, когда поток запустил `notify`, и временем пробуждения ожидающего потока.

Другой поток мог запустить `notify` случайно или злумышленно, когда условие не соблюдалось. Классы подвержены такому недостатку, если находятся в режиме ожидания на объекте в открытом доступе. Любой метод `wait`, содержащийся в синхронизированном методе открытого объекта, подвержен этой проблеме.

Уведомляющий поток может быть чрезчур щедрым на пробуждение ожидающих потоков. Например, поток уведомления может запустить `notifyAll`, даже если только некоторые из ожидающих потоков удовлетворяют условиям.

Ожидающий поток может (правда, редко) проснуться при отсутствии уведомления. Это известно под названием ложное пробуждение [Posix, 11.4.3.6.1: JavaSE6].

Связанный вопрос, нужно ли использовать `notify` или `notifyAll` для пробуждения ожидающих потоков. (Вспомните, что `notify` пробуждает один ожидающий поток, принимая во внимание, что поток существует и `notifyAll` будит все ожидающие потоки.) Часто советуют использовать `notifyAll`. Это разумный консервативный совет. Он всегда выдаст корректный результат, потому что гарантировано, что вы разбудите потоки, которые должны быть разбужены. Вы можете также разбудить некоторые другие потоки, но это не повлияет на правильность выполнения программы. Эти потоки проверят условия, для которых они проснулись, и, обнаружив, что они не соответствуют им, останутся в режиме ожидания.

В качестве оптимизации вы можете выбрать запуск `notify` вместо `notifyAll`, если все потоки, которые могут быть в наборе ожидания, ждут одного и того же условия и только один поток в одно время может извлечь выгоду от того, что условие будет истиной.

Даже если эти условия окажутся истиной, могут быть другие причины для использования `notifyAll` вместо `notify`. Так же как помещение запуска `wait` в цикл защищает от случайных и злонамеренных уведомлений на общедоступном объекте, использование `notifyAll` вместо `notify` защищает от случайных или злонамеренных ожиданий несвязанного потока. Такие ожидания могут проглотить

критические уведомления, оставив их потенциальных получателей в неопределенном ожидании.

Подведем итоги. Использование методов `wait` и `notify` непосредственно — это как программирование на языке ассемблер по сравнению с высокоуровневым программированием, предоставленным `java.util.concurrent`. **Редко, если вообще когда-либо, бывает причина использовать `wait` и `notify` в новом коде.** Если вы поддерживаете код, содержащий `wait` и `notify`, убедитесь, что `wait` всегда запускается из цикла `while`, используя стандартную идиому. Метод `notifyAll` должен в общем и целом использоваться предпочтительнее, чем `notify`. Если используется `notify`, необходимо серьезно позаботиться о его живучести.

*Статья
70*

При работе с потоками документируйте уровень безопасности

То, как класс работает, когда его экземпляры и статические методы одновременно используются в нескольких потоках, является важной частью соглашений, устанавливаемых классом для своих клиентов. Если вы не отразите эту сторону поведения класса в документации, использующие его программисты будут вынуждены делать допущения. И если эти допущения окажутся неверными, полученная программа может иметь либо недостаточную (статья 66), либо избыточную (статья 67) синхронизацию. В любом случае это способно привести к серьезным ошибкам.

Иногда говорится, что пользователи могут сами определить, безопасен ли метод при работе с несколькими потоками, если проверят, присутствует ли модификатор `synchronized` в документации, генерируемой утилитой Javadoc. Это неверно по некоторым причинам. Хотя в ранних версиях утилита Javadoc действительно указывала в создаваемом документе модификатор `synchronized`, это было ошибкой, и в версии 1.2 она была устранена. **Наличие в декларации метода модификатора `synchronized` — это деталь реализации, а не**

часть внешнего API. Присутствие модификатора не является надежной гарантией того, что метод безопасен при работе с несколькими потоками. От версии к версии ситуация может меняться.

Более того, само утверждение о том, что наличия ключевого слова `synchronized` достаточно для того, чтобы говорить о безопасности при работе с несколькими потоками, содержит в себе распространенную микроконцепцию о категоричности этого свойства. **На самом деле класс может иметь несколько уровней безопасности. Чтобы класс можно было безопасно использовать в среде со многими потоками, в документации к нему должно быть четко указано, какой уровень безопасности он поддерживает.**

В следующем списке приводятся уровни безопасности, которых может придерживаться класс при работе с несколькими потоками. Этот список не претендует на полноту, однако в нем представлены самые распространенные случаи. Используемые здесь названия не являются стандартными, поскольку в этой области нет общепринятых соглашений.

- **Неизменяемый (*immutable*).** Экземпляры такого класса выглядят для своих клиентов как константы. Никакой внешней синхронизации не требуется. Примерами являются `String`, `Integer` и `BigInteger` (статья 13).
- **С поддержкой многопоточности (*thread-safe*).** Экземпляры такого класса могут изменяться, однако все методы имеют довольно надежную внутреннюю синхронизацию, чтобы эти экземпляры могли параллельно использовать несколько потоков безо всякой внешней синхронизации. Параллельные вызовы будут обрабатываться последовательно в некотором глобально согласованном порядке. Примеры: `Random` и `java.util.Timer`.
- **С условной поддержкой многопоточности (*conditionally thread-safe*).** То же, что и с поддержкой многопоточности, за исключением того, что класс (или ассоциированный

класс) содержит методы, которые должны вызываться один за другим без взаимного влияния со стороны других потоков. Для исключения возможности такого влияния клиент должен установить соответствующую блокировку на время выполнения этой последовательности. Примеры: `Hashtable` и `Vector`, чьи итераторы требуют внешней синхронизации.

- **Не поддерживающий многопоточность (not thread-safe)** – экземпляры такого класса изменяемы, и их можно безопасно использовать при работе с параллельными потоками, если каждый вызов метода (а в некоторых случаях каждую последовательность вызовов) окружить внешней синхронизацией. Среди примеров реализации коллекций общего назначения, такие как `ArrayList` и `HashMap`.
- **Несовместимый с многопоточностью (thread-hostile).** Этот класс небезопасен при параллельной работе с несколькими потоками, даже если вызовы всех методов окружены внешней синхронизацией. Обычно несовместимость связана с тем обстоятельством, что эти методы меняют некие статические данные, которые оказывают влияние на другие потоки. К счастью, в библиотеках платформы Java лишь очень немногие классы и методы несовместимы с многопоточностью. Так, метод `System.runFinalizersOnExit` несовместим с многопоточностью и признан устаревшим.

Эти категории (кроме несовместимого с многопоточностью) соответствуют примерно *аннотациям по безопасности потоков (thread safety annotation)*, приведенными в книге *Java Concurrency in Practice*, которыми являются `Immutable`, `ThreadSafe` и `NotThreadSafe` [Goetz06, Appendix A]. Поддерживающие и условно поддерживающие многопоточность категории описываются в аннотации `ThreadSafe`.

Документированию класса с условной поддержкой многопоточности нужно уделять особое внимание. Вы должны указать, какие последовательности вызовов требуют внешней синхронизации

и какую блокировку (в редких случаях — блокировки) необходимо поставить, чтобы исключить одновременный доступ. Обычно это блокировка самого экземпляра, но не всегда. Если объект является альтернативным представлением какого-либо другого объекта, клиент должен получить блокировку для основного объекта с тем, чтобы воспрепятствовать его непосредственной модификации со стороны других потоков. Например, в документации к методу `Collection.synchronizedMap` говорится следующее:

Обязательно следует использовать ручную синхронизацию на возвращаемой схеме при итерации любого из представлений коллекции:

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<K, V>());
...
Set<K> s = m.keySet(); // Needn't be in synchronized block

synchronized(m) { // Synchronizing on m, not s!
    for (K key : s)
        key.f();
}
```

Если не последовать этому совету, то может наблюдаться недeterminистическое поведение.

Описание безопасности потока класса обычно содержится в комментариях к документации на него, но методы с особыми настройками безопасности потока должны описывать эти свойства в своих собственных комментариях к документации. Нет необходимости документировать неизменяемость перечислимых типов, если только не понятно по возвращаемым типам. Методы статической генерации должны документировать безопасность потоков на возвращаемом объекте, как продемонстрировано на примере выше с `Collections.synchronizedMap`.

Если класс начнет использовать политику блокировки открытого доступа, он дает клиентам возможность выполнять последовательный запуск методов атомарно, но это стоит гибкости. Это недопустимо при высокопроизводительном внутреннем контроле параллельности такого вида, как `ConcurrentHashMap` и `ConcurrentLinkedQueue`. Клиент

также может вызвать DOS-атаку, удерживая блокировку открытого доступа длительное время. Это может произойти случайно либо намеренно.

Для предотвращения такой DOS-атаки вы можете использовать *закрытую блокировку объекта* (*private lock object*) вместо использования синхронизированных методов (которые используют блок с открытым доступом):

```
// Идиома закрытой блокировки объекта – препятствует DOS-атаке
private final Object lock = new Object();
public void foo() {
    synchronized(lock) {
    }
}
```

Поскольку закрытая блокировка объекта недоступна клиентам класса, то они не могут вмешиваться в процесс синхронизации объекта. Для эффекта мы применим совет из статьи 13, инкапсулируя блокированный объект в рамках объекта, который он синхронизирует.

Обратите внимание, что поле `lock` объявляется как `final`. Это не позволяет вам по неосторожности изменить его содержимое, что может привести к катастрофическим последствиям, в частности к несинхронизированному доступу к содержащемуся объекту (статья 66). Мы применим совет из статьи 15, сводя к минимуму изменяемость поля `lock`.

Для повторной итерации идиома закрытого объекта блокировки может использоваться только в классах с безусловной поддержкой многопоточности. Классы с условной поддержкой многопоточности не могут использовать идиому, потому что им надо сначала документировать, какую именно блокировку должны получить их клиенты при выполнении определенной последовательности запуска методов.

Использование внутренних объектов для блокировки особенно подходит классам, которые предназначены для наследования (статья 17). Если бы такой класс должен был использовать экземпля-

ры для блокировки, то подкласс мог бы легко и непреднамеренно вмешаться в операции основного класса и наоборот. Используя одну и ту же блокировку для разных целей, суперкласс и подкласс стали бы в конце концов «наступать друг другу на пятки». Это не теоретическая проблема. Например, это происходит с классом Thread [Bloch05, задача 77].

Подведем итоги. Для каждого класса необходимо четко документировать возможность работы с несколькими потоками. Единственная возможность сделать это — представить аккуратно составленный текст описания. К описанию того, как класс поддерживает многопоточность, наличие модификатора synchronized отношения не имеет. Однако для классов с условной поддержкой многопоточности в документации важно указывать, какой объект следует заблокировать, чтобы последовательность обращений к методам стала неделимой. Если вы пишете класс с безусловной поддержкой потоков, то рассмотрите использование закрытой блокировки вместо синхронизированных методов. Это защищает вас от вмешательства в синхронизацию клиентов, подклассов и дает вам гибкость в применении более сложного подхода к контролю параллельности в последующих версиях.



С осторожностью используйте отложенную инициализацию

Отложенная инициализация — это задержка инициализации поля до тех пор, пока значение не потребуется. Если значение не потребуется вовсе, поле никогда не инициализируется. Этот прием применяется как на статических полях, так и на полях экземпляров. Хотя в основном отложенная инициализация является оптимизацией, она также может разрушить вредоносную зацикленность при инициализации класса или экземпляра [Bloch05, задача 51].

Как и в случае с большинством оптимизаций, лучшим советом для отложенной инициализации будет «не используйте ее до тех пор,

пока она вам действительно не понадобится» (статья 55). Отложенная инициализация — это палка о двух концах. Она уменьшает затраты на инициализацию класса или создание экземпляра за счет увеличения затрат на доступ к полю, инициализация которого отложена. В зависимости от того, какая часть инициализируемого поля в конце концов требует инициализации, насколько затратной будет их инициализация и как часто будет производиться доступ к каждому полю, отложенная инициализация может (как и любая оптимизация) повредить производительности.

Как уже говорилось, отложенная инициализация имеет свою область применения. Если к полю получает доступ только часть экземпляра класса *and* и инициализация поля требует затрат, тогда, возможно, стоит использовать отложенную инициализацию. Единственный способ узнать наверняка — измерить производительность класса с или без отложенной инициализации.

При наличии нескольких потоков отложенная инициализация может быть довольно сложна. Если два или более потока используют общее инициализируемое поле, то важно, чтобы применялась какая -либо форма синхронизации, иначе могут быть ужаснейшие ошибки (статья 66). Все приемы инициализации, обсуждаемые в этой статье, поддерживают многопоточность.

В большинстве случаев нормальная инициализация более предпочтительна, чем отложенная. Вот типичный пример декларирования нормально инициализируемого поля экземпляра. Обратите внимание на использование модификатора `final` (статья 15):

```
// Нормальная инициализация экземпляра поля  
private final FieldType field = computeFieldValue();
```

Если вы используете отложенную инициализацию для прерывания 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;  
synchronized FieldType getField() {  
    if (field == null)  
        field = computeFieldValue();  
    return field;  
}
```

Обе эти идиомы (нормальная инициализация и отложенная инициализация с синхронизированным методом доступа) являются неизменными, когда используются на статическом поле, кроме того, что вы добавляете модификатор static к полю и декларацию метода доступа.

Если вам требуется отложенная инициализация статического поля для производительности, используйте идиому, содержащую отложенную инициализацию класса. Эта идиома (также известная как идиома, содержащая инициализацию класса по запросу) использует гарантию того, что класс не будет инициализирован до тех пор, пока не будет использован [JLS 12.4.1]. Вот как она выглядит.

```
// Идиома отложенной инициализации класса для статического поля  
private static class FieldHolder {  
    static final FieldType field = computeFieldValue();  
}  
static FieldType getField() { return FieldHolder.field; }
```

Когда впервые запускается метод getField, он впервые 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 затратам на доступ. Современная виртуальная машина синхронизирует доступ к полям только при инициализации класса. Как только класс инициализирован, виртуальная машина изменяет код, чтобы последующий доступ к полю не содержал в себе никакой проверки или синхронизации.

Если вам требуется отложенная инициализация для повышения производительности поля экземпляра, используйте иди-

ому двойной проверки. Эта идиома избавляет от затрат на блокировку при доступе к полю после того, как оно инициализировано (статья 67). Идея, заложенная в данной идиоме, заключается в том, чтобы дважды проверить значение поля (отсюда и название): один раз без блокировки, в случае если поле окажется неинициализированным, и второй раз с блокировкой. Только если вторая проверка покажет, что поле не инициализировано, выполняется его инициализация. Поскольку нет блокировки, если поле уже инициализировано, то важно, чтобы поле было объявлено volatile (статья 66). Вот как выглядит идиома:

```
// Идиома двойной проверки для отложенной инициализации поля
// экземпляра
private volatile FieldType field;
FieldType getField() {
    FieldType result = field;
    if (result == null) { // First check (no locking)
        synchronized(this) {
            result = field;
            if (result == null) // Second check (with locking)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```

Этот код может показаться довольно запутанным. В частности, может быть непонятной необходимость локальной переменной result. Эта переменная убеждается в том, что field читается только один раз в общем случае, где она уже инициализирована. Хотя в ней и нет острой необходимости, это может улучшить производительность и более элегантно с точки зрения стандартов низкоуровневого потокового программирования. На моей машине метод работал на 25% быстрее, чем обычная версия без локальной переменной.

До релиза 1.5 идиома двойной проверки не работала надежно, потому что семантика модификатора volatile не была достаточно

сильно определена для обеспечения надежности [Pugh01]. Модель памяти, представленная в версии 1.5, решила проблему [JLS, 17; Goetz06, 16]. Сегодня идиома двойной проверки является хорошим приемом для отложенной инициализации поля. Хотя можно выполнять двойную проверку на статическом поле, нет причин, чтобы так делать: идиома отложенной инициализации класса больше подходит для этого.

Два варианта идиомы двойной проверки заслуживают внимания. В некоторых случаях может понадобиться отложенная инициализация поля экземпляра, которая допускает повторную инициализацию. Если вы окажетесь в такой ситуации, то можно использовать вариант, где применяется идиома двойной проверки, которая обходится без второй проверки. Она известна под названием *идиома однократной проверки*. Вот как она выглядит. Обратите внимание, что поле `field` снова объявляется как `volatile`:

```
// Идиома однократной проверки – может вызвать повторную
// инициализацию!
private volatile FieldType field;
private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
```

Все приемы инициализации, обсужденные в этой статье, относятся к примитивным полям, а также к полям ссылок на объекты. Когда идиома двойной или однократной проверки применяется на примитивном числовом поле, значение поля сравнивается с 0 (значение по умолчанию для примитивных числовых переменных), а не с `null`.

Если для вас не важно, чтобы каждый поток пересчитывал значение поля, и если тип поля является примитивным, но не `long` или `double`, то можно убрать модификатор `volatile` из декларации поля в идиоме однократной проверки. Этот вариант известен как *специфичная идиома однократной проверки*. Она ускоряет доступ к полю в некоторых архитектурах за счет дополнительных инициализаций

(до одной на поток, получающей доступ к полю). Это определенно экзотический прием, не для каждого использования. Но, тем не менее, он используется экземпляром `String` для кэширования хэш-кодов.

Подведем итоги. Вам необходимо большинство полей инициализировать с помощью нормальной инициализации, а не отложенной. Если требуется отложенная инициализация поля с целью повышения производительности или для прерывания вредоносного зацикливания, тогда используйте подходящий прием отложенной инициализации. Для полей экземпляров это идиома двойной проверки; для статических полей — идиома инициализации содержащего класса. Для полей экземпляров, которые разрешают повторную инициализацию, вы можете рассматривать вариант с идиомой однократной проверки.

Статья
72

Не попадайте в зависимость от планировщика потоков

Когда в системе выполняется несколько потоков, соответствующий планировщик определяет, какие из них будут выполняться и в течение какого времени. Каждая правильная реализация JVM при этом будет пытаться добиться какой-то справедливости, однако конкретная стратегия диспетчериизации в различных реализациях отличается очень сильно. Соответственно, хорошо написанные многопоточные приложения не должны зависеть от особенностей этой стратегии. **Любая программа, чья корректность или производительность зависит от планировщика потоков, скорее всего, переносимой не будет.**

Лучший способ написать устойчивую, гибкую и переносимую многопоточную программу — обеспечить условия, когда в любой момент времени может выполняться несколько потоков. В этом случае планировщику потоков остается совсем небольшой выбор: он просто передает управление выполняемым потокам, пока те еще могут

выполняться. Как следствие, поведение программы не будет сильно меняться даже при выборе совершенно других алгоритмов диспетчирования потоков.

Основной прием, позволяющий сократить количество запущенных потоков, заключается в том, чтобы каждый поток выполнял небольшую порцию работы, а затем ждал следующей. С точки зрения Executor Framework (статья 68) это означает правильное определение размера пула потоков [Goetz06. 8.2], и надо делать так, чтобы задачи были небольшими и не зависели друг от друга. Задачи не должны быть слишком маленькими, иначе затраты на диспетчирование негативно отразятся на производительности.

Потоки не должны находиться в состоянии активного ожидания (*busy-wait*), регулярно проверяя структуру данных и ожидая, пока что-то с теми произойдет. Помимо того, что программа при этом становится чувствительной к причудам планировщика, активное ожидание может значительно повысить загрузку на процессор, соответственно уменьшая количество полезной работы, которую на той же машине могли бы выполнить остальные процессы. В качестве примера, как не надо поступать, рассмотрим неправильную повторную реализацию CountDownLatch:

```
// Ужасная реализация CountDownLatch – состояние активного
// ожидания не прерывается!
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 выполняется в 2000 раз медленнее, чем CountDownLatch, когда в ожидании находятся 1000 потоков. Хотя данный пример может казаться и нереалистичным, не так редко можно встретить системы, в которых без надобности запускаются один или более потоков. Результат может быть не настолько критичным, как при SlowCountDownLatch, но производительность и переносимость, скорее всего, пострадают.

Столкнувшись с программой, которая едва работает из-за того, что некоторым потокам недостаточно времени ЦП по сравнению с другими потоками, **не поддавайтесь искушению починить программу добавлением вызова Thread.yield**. Вам, может, удастся заставить работать программу, но она не будет переносимой. Один и тот же запуск yield может увеличить производительность на одной реализации виртуальной машины, но на второй увеличение будет хуже, а на третьей вообще не будет иметь никакого эффекта. **Проверить семантику Thread.yield невозможно**. Лучшим подходом будет изменить структуру приложения для снижения количества параллельно выполняемых потоков.

Аналогичный прием, к которому также относится похожий недостаток, — это настройка приоритетности потоков. **Приоритетность потоков — одна из наименее переносимых особенностей на платформе Java**. Разумным будет настроить отклик приложения, применив несколько свойств приоритетности потоков, но это редко бывает необходимо и делает приложения непереносимыми. Совсем неразумно для решения серьезных проблем с живучестью использовать приоритетность потоков. Проблема, скорее всего, будет актуальна до тех пор, пока вы не найдете и не устраниете основное препятствие.

В первой редакции этой книги говорилось, что единственное применение `Thread.yield`, используемое большинством программистов, — это искусственное увеличение параллельности для тестирования. Идея была в том, чтобы убрать ошибки путем исследования большой части пространства состояния программы. Этот прием когда-то был довольно эффективен, но никогда не была гарантирована его работа. В спецификации к `Thread.yield` говорилось, что он вообще ничего не делает, просто возвращает контроль над ним вызывающему. Некоторые виртуальные машины действительно так и делают. Следовательно, вам необходимо использовать `Thread.sleep(1)` вместо `Thread.yield` для проверки параллельности. Не используйте `Thread.sleep(0)`, который может немедленно возвратиться.

Подведем итоги. Правильность вашего приложения не должна зависеть от планировщика потоков. Иначе полученное приложение не будет устойчивым и переносимым. Как следствие, не надо связываться с методом `Thread.yield` и приоритетами. Эти функции предназначены только для планировщика. Их можно дозированно использовать для улучшения качества сервиса в уже работающей реализации, но ими никогда нельзя пользоваться для «исправления» программы, которая едва работает.

Статья
73

Избегайте группировки потоков

Помимо потоков, блокировок и мониторов система многопоточной обработки предлагает еще одну базовую абстракцию: группа потоков (*thread group*). Первоначально группировка потоков рассматривалась как механизм изоляции апплетов в целях безопасности. В действительности своих обязательств они так и не выполнили, а их роль в системе безопасности упала до такой степени, что в работе, где выстраивается модель безо-

пасности для платформы Java 2 [Gong03], они даже не упоминаются.

Но если группировка потоков не несет никакой функциональной нагрузки в системе безопасности, то какие же функции она выполняет?

Не много. Это дает вам возможность применять примитивы класса Thread сразу к целой группе потоков. Некоторые из этих примитивов уже устарели, остальные используются нечасто.

По иронии судьбы, API ThreadGroup слаб с точки зрения поддержки многопоточности. Чтобы для некоей группы получить перечень активных потоков, вы должны вызвать метод enumerate. В качестве параметра ему передается массив, достаточно большой, чтобы в него можно было записать все активные потоки. Метод activeCount возвращает количество активных потоков в группе, однако нет никакой гарантии, что это количество не изменится за то время, пока вы создаете массив и передаете его методу enumerate. Если указанный массив окажется слишком мал, метод enumerate без каких-либо предупреждений игнорирует потоки, не поместившиеся в массив.

Точно так же API некорректен, когда ему передается список подгрупп, входящих в группу потоков. И хотя указанные проблемы можно было решить, добавив в класс ThreadGroup новые методы, этого не было сделано из-за отсутствия реальной потребности. **Группировка потоков сильно устарела.**

До появления версии 1.5 существовал небольшой функционал, который был доступен только вместе с API ThreadGroup: метод ThreadGroup. uncaughtException был единственным способом получения контроля, когда поток выводил такое исключение. Этот функционал полезен, например, для непосредственного отслеживания пути стека к журналу определенного приложения. Тем не менее в релизе 1.5 тот же самый функционал доступен с методом setUncaughtExceptionHandler, относящимся к Thread.

Подведем итоги. Группировка потоков практически не имеет сколь-нибудь полезной функциональности, и большинство предо-

ставляемых ею возможностей имеет дефекты. Группировку потоков следует рассматривать как неудачный эксперимент, а существование групп можно игнорировать. Если вы проектируете класс, который работает с логическими группами потоков, вам нужно, вероятнее всего, использовать экзекуторы пула потоков (статья 68).

11

Г л а в а

Сериализация

В этой главе описывается API сериализации объекта (*object serialization*), который формирует среду для представления объекта в виде потока байтов и, наоборот, для восстановления объекта из соответствующего потока байтов. Процедура представления объекта в виде потока байтов называется сериализацией объекта (*serializing*), обратный процесс называется его десериализацией (*deserializing*). Как только объект был сериализован, его представление можно передавать с одной работающей виртуальной машины Java на другую или сохранять на диске для последующей десериализации. Сериализация обеспечивает стандартное представление объектов на базовом уровне, которое используется для взаимодействия с удаленными машинами, а также как стандартный формат для сохранения данных при работе с компонентами JavaBeans. Замечательной особенностью данной главы является шаблон *serialization proxy* (статья 78), которая поможет вам избежать многие ловушки, связанные с сериализацией объекта.

Статья
74

Соблюдайте осторожность при реализации интерфейса `Serializable`

Чтобы сделать экземпляры класса сериализуемыми, достаточно добавить в его декларацию слова «`implements Serializable`». По-

скольку это так легко, широкое распространение получило неправильное представление, что сериализация требует от программиста совсем небольших усилий. На самом деле все гораздо сложнее.

Значительная доля затрат на реализацию интерфейса Serializable связана с тем, что это уменьшается возможность изменения реализации класса в последующих версиях. Когда класс реализует интерфейс Serializable, соответствующий ему поток байтов (*сериализованная форма, serialized form*) становится частью его внешнего API. И как только ваш класс получит широкое распространение, вам придется поддерживать соответствующую сериализованную форму точно так же, как вы обязаны поддерживать все остальные части интерфейса, предоставляемого клиентам. Если вы не приложите усилий к построению *специальной сериализованной формы (custom serialized form)*, а примете форму, предлагаемую по умолчанию, эта форма окажется навсегда связанной с первоначальным внутренним представлением класса. Иначе говоря, если вы принимаете сериализованную форму, которая предлагается по умолчанию, те экземпляры полей, которые были закрыты или доступны только в пакете, станут частью его внешнего API и практика минимальной доступности полей (статья 13) потеряет свою эффективность как средство скрытия информации.

Если вы принимаете сериализованную форму, предлагаемую по умолчанию, а затем поменяете внутреннее представление класса, это может привести к таким изменениям в форме, что она станет несовместима с предыдущими версиями. Клиенты, которые пытаются сериализовать объект с помощью старой версии класса и десериализовать его уже с помощью новой версии, получат сбой программы. Можно поменять внутреннее представление класса, оставив первоначальную сериализованную форму (с помощью методов ObjectOutputStream.putFields и ObjectInputStream.readFields), но этот механизм довольно сложен и оставляет в исходном коде программы видимые изъяны. Поэтому вам следует тщательно выстраивать очень качественную сериализованную форму, с которой вы сможете

отправиться в долгий путь (статьи 75, 78). Эта работа усложняет создание приложения, но дело того стоит. Даже хорошо спроектированная сериализованная форма ограничивает дальнейшее развитие класса, плохо же спроектированная форма может его искалечить.

Простым примером того, какие ограничения на изменение класса накладывает сериализация, могут служить *уникальные идентификаторы потока* (*stream unique identifier*), более известные как *serial version UID*. С каждым сериализуемым классом связан уникальный идентификационный номер. Если вы не указываете этот идентификатор явно, декларируя поле `private static final long` с названием `serialVersionUID`, система генерирует его автоматически, используя для класса сложную схему расчетов. При этом на автоматически генерируемое значение оказывают влияние название класса, названия реализуемых им интерфейсов, а также все открытые и защищенные члены. Если вы каким-то образом поменяете что-либо в этом наборе, например, добавите простой и удобный метод, изменится и автоматически генерируемый *serial version UID*. Следовательно, если вы не будете явным образом декларировать этот идентификатор, совместимость с предыдущими версиями будет потеряна.

Второе неудобство от реализации интерфейса `Serializable` заключается в том, что повышается вероятность появления ошибок и дыр в защите. Объекты обычно создаются с помощью конструкторов, сериализация же представляет собой механизм создания объектов, который *выходит за рамки языка Java*. Принимаете ли вы схему, которая предлагается по умолчанию, или переопределяете ее, десериализация — это «скрытый конструктор», имеющий все те же проблемы, что и остальные конструкторы. Поскольку явного конструктора здесь нет, легко упустить из виду то, что при десериализации вы должны гарантировать сохранение всех инвариантов, устанавливаемых настоящими конструкторами, и исключить возможность получения злоумышленником доступа к внутреннему содержимому создаваемого объекта. Понадеявшись на механизм десериализации, предоставляемый по умолчанию, вы можете получить

объекты, которые не препятствуют несанкционированному доступу к внутренним частям и разрушению инвариантов (статья 76).

Третье неудобство реализации интерфейса Serializable связано с тем, что выпуск новой версии класса сопряжен с большой работой по тестированию. При пересмотре сериализуемого класса важно проверить возможность сериализации объекта в новой версии и последующей его десериализации в старой и наоборот. Таким образом, объем необходимого тестирования прямо пропорционален произведению числа сериализуемых классов и числа имеющихся версий, что может быть большой величиной. К подготовке таких тестов нельзя подходить формально, поскольку, помимо совместимости на бинарном уровне, вы должны проверять совместимость на уровне семантики. Иными словами, необходимо гарантировать не только успешность процесса сериализации-десериализации, но и то, что он будет создавать точную копию первоначального объекта. И чем больше изменяется сериализуемый класс, тем сильнее потребность в тестировании. Если при написании класса специальная сериализованная форма была спроектирована тщательно (статья 75, 78), потребность в проверке уменьшается, но полностью не исчезает.

Реализация интерфейса Serializable должна быть хорошо продумана. У этого интерфейса есть реальные преимущества: его реализация играет важную роль, если класс должен участвовать в какой-либо схеме, которая для передачи или обеспечения живучести объекта использует сериализацию. Более того, это упрощает применение класса как составной части другого класса, который должен реализовать интерфейс Serializable. Однако с реализацией интерфейса Serializable связано и множество неудобств. Реализуя класс, соотносите неудобства с преимуществами. Практическое правило таково: классы значений, такие как Date и BigInteger, и большинство классов коллекций обязаны реализовывать этот интерфейс. Классы, представляющие активные сущности, например пул потоков, должны реализовывать интерфейс Serializable крайне редко. Так, в версии 1.4 появился механизм сохранения компонентов JavaBean, кото-

рый использует стандарт XML, а потому этим компонентам больше не нужно реализовывать интерфейс Serializable.

Классы, предназначенные для наследования (статья 17), 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. В ряде случаев это правило можно нарушать. Например, если класс или интерфейс создан в первую очередь для использования в некоторой системе, требующей, чтобы все ее участники реализовывали интерфейс Serializable, то лучше всего, чтобы этот класс (интерфейс) реализовывал (расширял) Serializable.

Классы, предназначенные для наследования, которые реализуют интерфейс Serializable, включают в себя Throwable, Component и HttpServlet. Throwable реализует Serializable для того, чтобы исключения из удаленного запуска методов (remote method invocation, RMI) могли передаваться от сервера к клиенту. Component реализует Serializable, чтобы можно было отправить, сохранить и восстановить GUI. HttpServlet реализует Serializable для кэширования состояния сессии.

Если вы реализуете класс с полем экземпляра, который сериализуем и расширяем, то вы должны знать об одной опасности. Если у класса есть инварианты, которые могут быть нарушены, если поля его экземпляров были инициализированы на значение по умолчанию (ноль для цельных типов, false для типов Boolean, null для типов ссылок на объект), то вы должны добавить к классу этот метод readObjectNoData:

```
// readObjectNoData для расширяемых сериализуемых классов
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

Если вам интересно, метод readObjectNoData был добавлен в версии 1.4 на случай добавления сериализуемого суперкласса к существующему сериализуемому классу. Детали можно найти в спецификации по сериализации [Serialization, 3.5].

Нужно сделать одно *предупреждение* относительно реализации интерфейса `Serializable`. Если класс предназначен для наследования и не является сериализуемым, может оказаться, что для него невозможно написать сериализуемый подкласс. В частности, этого нельзя сделать, если у суперкласса нет доступного конструктора без параметров. Следовательно, **для несериализуемого класса, который предназначен для наследования, вы должны рассмотреть возможность создания конструктора без параметров.** Часто это не требует особых усилий, поскольку многие классы, предназначенные для наследования, не имеют состояния. Но так бывает не всегда.

Самое лучшее — это создавать объекты, у которых все инварианты уже установлены (статья 15). Если для установки инвариантов необходима информация от клиента, это будет препятствовать использованию конструктора без параметров. Бесхитростное добавление конструктора без параметров и метода инициализации в класс, остальные конструкторы которого устанавливают инварианты, усложняет пространство состояний этого класса и увеличивает вероятность появления ошибки.

Приведем вариант добавления конструктора без параметров в несериализуемый расширяемый класс, свободный от этих пороков. Предположим, что в этом классе есть один конструктор:

```
public AbstractFoo(int x, int y) ...
```

Следующее преобразование добавляет *зашитенный* конструктор без параметров и отдельный метод инициализации. Причем метод инициализации имеет те же параметры и устанавливает те же инварианты, что и обычный конструктор. Обратите внимание, что переменные, сохраняющие состояние объекта (`x` и `y`), не могут быть завершенными, так как они установлены методом `initialize`:

```
// Несериализуемый, имеющий состояние класс, для которого можно
// создать подклассы
public abstract class AbstractFoo {
    private int x, y; // Our state
    protected void initialize() {
        x = 0;
        y = 0;
    }
}
```

```

// This enum and field are used to track initialization
private enum State { NEW, INITIALIZING, INITIALIZED };
private final AtomicReference<State> init =
    new AtomicReference<State>(State.NEW);
public AbstractFoo(int x, int y) { initialize(x, y); }
// This constructor and the following method allow
// subclass's readObject method to initialize our state.
protected AbstractFoo() { }
protected final void initialize(int x, int y) {
    if (!init.compareAndSet(State.NEW, State.INITIALIZING))
        throw new IllegalStateException(
            "Already initialized");
    this.x = x;
    this.y = y;
    ... // Do anything else the original constructor did
    init.set(State.INITIALIZED);
}
// These methods provide access to internal state so it can
// be manually serialized by subclass's writeObject method.
protected final int getX() { checkInit(); return x; }
protected final int getY() { checkInit(); return y; }
// Должен вызываться из всех открытых и защищенных методов
// экземпляра
private void checkInit() {
    if (init.get() != State.INITIALIZED)
        throw new IllegalStateException("Uninitialized");
}
... // Остальное опущено
}

```

Все методы в экземпляре `AbstractFoo`, прежде чем выполнять свою работу, должны вызывать `checkInit`. Тем самым гарантируется быстрое и четкое аварийное завершение этих методов в случае, если неудачно написанный подкласс не инициализировал соответствующий экземпляр. Имея этот механизм взамен прежнего, имеет смысл перейти к реализации сериализуемого подкласса:

```
// Сериализуемый подкласс несериализуемого класса, имеющего состояние
public class Foo extends AbstractFoo implements Serializable {
    private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        // Ручная десериализация и инициализация состояния
        // суперкласса
        int x = s.readInt();
        int y = s.readInt();
        initialize(x, y);
    }
    private void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
        // Ручная сериализация состояния суперкласса
        s.writeInt(getX());
        s.writeInt(getY());
    }
    // Конструктор не использует никаких причудливых механизмов
    public Foo(int x, int y) { super(x, y); }
    private static final long serialVersionUID = 1856835860954L;
}
```

Внутренние классы (статья 22) редко должны (если вообще должны) реализовывать интерфейс Serializable. Для размещения ссылок на экземпляры контейнера (*enclosing instance*) и значений локальных переменных из окружения они должны пользоваться *искусственные поля* (*synthetic field*), генерируемые компилятором. Как именно эти поля соотносятся с декларацией класса, не конкретизируется. Не конкретизируются также названия анонимных и локальных классов. Поэтому **выбор для внутреннего класса сериализованной формы, предлагаемой по умолчанию, плохое дизайнерское решение**. Однако статический класс — член класса вполне может реализовывать интерфейс Serializable.

Подведем итоги. Легкость реализации интерфейса Serializable является обманчивой. Реализация интерфейса Serializable — се-

рьезное обязательство, которое следует брать на себя с осторожностью, если только не предполагается выбросить класс после недолгого использования. Особое внимание требует класс, предназначенный для наследования. Для таких классов границей между реализацией интерфейса Serializable в подклассах и его запретом является создание доступного конструктора без параметров. Это позволяет реализовать в подклассе интерфейс Serializable, хотя и не является обязательным условием.

Статья
75

Рассмотрите возможность использования специализированной сериализованной формы

Если вы создаете класс в условиях дефицита времени, то, как правило, имеет смысл сконцентрировать усилия на построении самого лучшего API. Иногда это означает создание «одноразовой» реализации, которая в следующей версии поменяется. Обычно это проблем не вызывает, однако, если данный класс реализует интерфейс Serializable и использует при этом сериализованную форму, предоставленную 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 несколькими классами из библиотек для платформы Java, такими как BigInteger.

Нельзя принимать сериализованную форму, предлагаемую по умолчанию, не обдумав как следует, устраивает ли она вас. Ваше решение должно бытьзвешенным, приемлемым с точки зрения гибкости, производительности и правильности приложения. Вообще говоря, вы должны принимать сериализованную форму, используемую по умолчанию, только если она в значительной степени совпадает с той кодировкой, которую вы бы выбрали, если бы проектировали сериализованную форму сами.

Сериализованная форма представления объекта, предлагаемая по умолчанию, — это довольно эффективное *физическое* представление графа объектов, имеющего корнем данный объект. Другими словами, эта форма описывает данные, содержащиеся как в самом объекте, так и во всех доступных из него объектах. Она также отражает топологию взаимосвязи этих объектов. Идеальная же сериализованная форма, описывающая объект, содержит только представляемые им *логические* данные. От физического представления она не зависит.

Сериализованная форма, предлагаемая по умолчанию, по-видимому, будет приемлема в том случае, если физическое представление объекта равно значению его логическому содержанию. Например, сериализованная форма, предлагаемая по умолчанию, будет правильной для следующего класса, который представляет имя человека:

```
// Хороший кандидат для использования формы, предлагаемой
// по умолчанию
public class Name implements Serializable {
    /**
     * Последнее имя (англ.). Не должно быть пустым (non--null)
     * @serial
     */
    private String lastName;
    /**
     * Первое имя. Не должно быть пустым.
     * @serial
     */
    private String firstName;
    /**
     * Средний инициал или '\u0000', если инициал отсутствует
     * @serial
     */
    private char middleInitial;
    ... // Остальное опущено
}
```

Логическое имя человека в английском языке состоит из двух строк, представляющих последнее и первое имя, а также из некоторого символа, соответствующего среднему инициалу. Экземпляры полей в классе Name в точности воспроизводят это логическое содержание.

Если вы решите принять сериализованную форму, предлагаемую по умолчанию, во многих случаях сохранение инвариантов и безопасность требуют реализации метода `readObject`. В случае с классом Name метод `readObject` мог бы гарантировать, что поля `lastName` и `firstName` не будут иметь значения `null`. Эта тема подробно рассматривается в статьях 76 и 78.

Заметим, что поля `lastName`, `firstName` и `middleInitial` сопровождаются комментарием к документации, хотя все они являются закрытыми. Это необходимо, поскольку эти закрытые поля определяют открытый API: сериализованную форму класса, а всякий открытый API должен быть документирован. Наличие тега `@serial` говорит утилите Javadoc о том, что эту информацию необходимо поместить на специальную страницу, где описываются сериализованные формы.

Теперь рассмотрим класс, который представляет набор строк (забудем на минуту о том, что для этого лучше было бы взять в библиотеке одну из стандартных реализаций интерфейса `List`):

```
// Ужасный кандидат на использование сериализованной формы,  
// предлагаемой по умолчанию  
public class StringList implements Serializable  
{  
    private int size = 0;  
    private Entry head = null;  
    private static class Entry implements Serializable {  
        String data;  
        Entry next;  
        Entry previous;  
    }  
    ... // Остальное опущено  
}
```

Логически этот класс представляет последовательность строк. Физически последовательность представлена им как дважды связный список. Если вы примете сериализованную форму, предлагаемую по умолчанию, она старательно отразит каждый элемент в этом связном списке, а также все связи между этими элементами в обоих направлениях.

В случае, когда физическое представление объекта существенно отличается от содержащихся в нем логических данных, сериализованная форма, предлагаемая по умолчанию, 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 `StringList` поменяется, он все равно должен будет получать на входе представление в виде связного списка и генерировать его же на выходе. Этот класс уже никогда не избавится от кода, необходимого для манипулирования связными списками, даже если он ими уже не пользуется.
- **Она может занимать чрезвычайно много места.** В приведенном примере в сериализованной форме без всякой на то надобности представлен каждый элемент связанного списка со всеми его связями. Эти элементы и связи являются всего лишь 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 обрабатываться чрезвычайно долго.** Логика сериализации не содержит информации о топологии графа объекта, а потому приходится выполнять дорогостоящий обход вершин графа. В приведенном примере достаточно было просто идти по ссылкам `next`.

- **Она может вызвать переполнение стека.** Процедура сериализации, реализуемая по умолчанию, выполняет рекурсивный обход графа объектов, что может вызвать переполнение стека даже при обработке графов среднего размера. На моей машине к переполнению стека приводит сериализация экземпляра `StringList` с 1200 элементами. Количество элементов, вызывающее эту проблему, меняется в зависимости от реализации JVM. В некоторых реализациях этой проблемы вообще не существует.

Правильная сериализованная форма для класса `StringList` — это количество строк в списке, за которым следуют сами строки. Это соответствует освобожденным от деталей физической реализации логическим данным, представляемым классом `StringList`. Приведем исправленный вариант `StringList`, содержащий методы `writeObject` и `readObject`, которые реализуют правильную сериализованную форму. Напомним, что модификатор `transient` указывает на то, что экземпляр поля должен быть исключен из сериализованной формы, применяемой по умолчанию:

```
// Класс StringList с правильной сериализованной формой
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;
    // Больше нет реализации Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }
    // Добавляет указанную строку в конец списка
    public void add(String s) { }
    /**
     * Сериализует данный экземпляр {@code StringList}.
     */
}
```

```
* @serialData Показывается размер списка (количество содержащихся
* в нем строк) {@code int}, за которым в правильной последовательности
* следуют все элементы списка (каждый в виде {@code String}).
*/
private void writeObject(ObjectOutputStream s)
throws IOException {
    s.defaultWriteObject();
    s.writeInt(size);
    // Выписываем все элементы 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 size = s.readInt();
    // Считываем все элементы и вставляем их в список
    for (int i = 0; i < size; i++)
        add((String)s.readObject());
}
... // Остальное опускаем
}
```

Заметим, что из метода `writeObject` вызывается `defaultWriteObject`, а из метода `readObject` делается вызов `defaultReadObject`, несмотря на то что ни одно из полей класса `StringList` не попадает в сериализованную форму. Если все экземпляры полей имеют модификатор `transient`, то формально можно обойтись без вызова методов `defaultWriteObject` и `defaultReadObject`, но это не рекомендуется. Даже если все экземпляры полей имеют модификатор `transient`, вызов `defaultWriteObject` оказывает влияние на сериализованную форму, в результате чего значительно повышается гибкость сериализации. Полученная форма оставляет возможность в последующих версиях добавлять в форму новые экземпляры полей, сохраняя при этом прямую и обратную совместимость с предыдущими версиями. Так, если сериализовать экземпляр класса в более поздней версии,

а десериализовать в более ранней версии, появившиеся поля будут проигнорированы. Если бы более ранняя версия метода `readObject` не вызывала метод `defaultReadObject`, десериализация закончилась бы инициализированием `StreamCorruptedException`.

Заметим также, что, хотя метод `writeObject` является закрытым, он сопровождается комментариями к документации. Объяснение здесь то же, что и в случае с комментариями для закрытых полей в классе `Name`. Этот закрытый метод определяет сериализованную форму — открытый API, а открытый API должен быть описан в документации. Как и тег `@serial` в случае с полями, тег `@serialData` для методов говорит утилите 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 для исправленной версии `StringList` будет занимать вдвое меньше места, чем в первоначальном варианте. На моей машине сериализация исправленного варианта `StringList` при длине строк в десять символов выполняется примерно в два с половиной раза быстрее, чем сериализация первоначального варианта. И наконец, у исправленного варианта не возникает проблем с переполнением стека, а потому практически нет верхнего ограничения 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, применяемая по умолчанию, не имеет гибкости и работает медленно. Однако она является правильной в том смысле, что в результате сериализации и десериализации экземпляра `StringList` получается точная копия исходного объекта, и все его инварианты будут сохранены. Но для любого объекта, чьи инварианты привязаны к деталям реализации, это не так.

Например, рассмотрим случай с хэш-таблицей. Ее физическим представлением является набор сегментов, содержащих записи `ключ/значение`. Сегмент, куда будет помещена запись, определяется функцией, которая для представленного ключа вычисляет хэш-код.

Вообще говоря, нельзя гарантировать, что в различных реализациях JVM эта функция будет одной и той же. В действительности нельзя даже гарантировать, что она будет оставаться той же самой, если одну и ту же JVM запускать несколько раз. Следовательно, использование для хэш-таблицы сериализованной формы, предлагаемой по умолчанию, может стать серьезной ошибкой: сериализация и десериализация хэш-таблицы могут привести к созданию объекта, инварианты которого будут серьезно нарушены.

Используете вы или нет сериализованную форму, предлагаемую по умолчанию, каждый экземпляр поля, не помеченный модификатором `transient`, будет сериализован при вызове метода `defaultWriteObject`. Поэтому каждое поле, которое можно не заносить в форму, нужно пометить этим модификатором. К таковым относятся избыточные поля, чьи значения можно вычислить по таким «первичным полям данных», как кэшированное значение хэша. Сюда также относятся поля, чьи значения меняются при повторном запуске JVM. Например, это может быть поле типа `long`, в котором хранится указатель на местную (`native`) структуру данных. **Прежде чем согласиться на запись какого-либо поля в сериализованной форме, убедитесь в том, что его значение является частью логического состояния данного объекта.** Если вы пользуетесь специальной сериализованной формой, большинство или даже все экземпляры полей нужно пометить модификатором `transient`, как в примере с классом `StringList`.

Если вы пользуетесь сериализованной формой, предлагаемой по умолчанию, и к тому же пометили одно или несколько полей как `transient`, помните о том, что при десериализации экземпляра эти поля получат значения по умолчанию: `null` для полей ссылок на объекты, нуль для простых числовых полей и `false` для полей типа `boolean` [JLS,<|>4.12.5]. Если для какого-либо из этих полей указанные значения неприемлемы, необходимо предоставить метод `readObject`, который вызывает метод `defaultReadObject`, а затем восстанавливает приемлемые значения в полях, помеченных как `transient` (ст-

тья 76). Альтернативный подход заключается в том, чтобы отложить инициализацию этих полей до первого вызова (статья 71).

Используете вы или нет сериализованную форму по умолчанию, вы должны навязать синхронизацию любой сериализации объекта, какую бы вы навязали любому другому методу, читающему все состояние объекта. Так, например, если у вас есть объект, поддерживающий многопоточность (статья 70), который достиг поддержки многопоточности, синхронизируя каждый метод, используйте следующий метод `writeObject`:

```
// writeObject для синхронизированного класса с сериализованной
// формой по умолчанию
private synchronized void writeObject(ObjectOutputStream s)
throws IOException {
    s.defaultWriteObject();
}
```

Независимо от того, какую сериализованную форму вы выберете, в каждом сериализуемом классе, который вы пишете, явным образом декларируйте `serial version UID`. Тем самым вы исключите этот идентификатор из числа возможных причин несовместимости (статья 74). Это также даст некоторый выигрыш в производительности. Если `serial version UID` не представлен, то при выполнении программы для его генерации потребуется выполнить трудоемкие вычисления.

Для декларации `serial version UID` добавьте в ваш класс строку:

```
private static final long serialVersionUID = randomLongValue;
// Произвольное число типа long
```

Не важно, какое значение вы выберете для `randomLongValue`. Общепринятая практика предписывает генерировать это число, запуская для класса утилиту `serialver`. Однако можно взять число «из воздуха».

Если вы меняете существующий класс, в котором нет серийной версии `UID`, и вы хотите новую версию, чтобы принимать существу-

ющие сериализованные экземпляры, вам необходимо использовать значение, которое было автоматически сгенерировано старой версией класса — той, сериализованные экземпляры которой уже существуют.

Подведем итоги. Если решено, что класс должен быть сериализуемым (статья 74), подумайте над тем, какой должна быть сериализованная форма. Форму, предлагаемую по умолчанию, используйте, только если она правильно описывает логическое состояние объекта. В противном случае создайте специальную сериализованную форму, которая надлежащим образом описывает этот объект. На разработку сериализованной формы для класса вы должны выделить не меньше времени, чем на разработку его методов, предоставляемых клиентам (статья 40). Точно так же, как из последующих версий нельзя изъять те методы класса, которые были доступны клиентам, нельзя изымать поля из сериализованной формы. Чтобы при сериализации сохранялась совместимость, эти поля должны оставаться в форме навсегда. Неверный выбор сериализованной формы может иметь постоянное отрицательное влияние на сложность и производительность класса.

Статья
76

Метод `readObject` должен создаваться с защитой

В статье 39 представлен неизменяемый класс для интервалов времени, который содержит изменяемые закрытые поля даты. Чтобы сохранить свои инварианты и неизменяемость, этот класс создает резервную копию объектов Date в конструкторе и методах доступа. Приведем этот класс:

```
// Неизменяемый класс, который использует резервное копирование
public final class Period {
    private final Date start;
    private final Date end;
    /**
     * @param start начало периода.
```

```

* @param end конец периода, не должен предшествовать началу периода.
* @throws IllegalArgumentException if начало периода указано после конца.
* @throws NullPointerException if начало или конец периода нулевые.
*/
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 + " after " + end);
}
public Date start (){ return new Date(start.getTime()); }
public Date end () { return new Date(end.getTime()); }
public String toString() { return start + " - " + end;
... // Остальное опущено
}

```

Предположим, что вам необходимо сделать этот класс сериализуемым. Поскольку физическое представление объекта `Period` в точности отражает его логическое содержание, вполне можно воспользоваться сериализованной формой, предлагаемой по умолчанию (статья 75). Может показаться, что все, что вам нужно для того, чтобы класс был сериализуемым, — это добавить в его декларацию слова «*implements Serializable*». Если вы поступите таким образом, то гарантировать классу сохранение его критически важных инвариантов будет невозможно.

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 правильность своих аргументов (статья 38) и при необходимости создавать для параметров резервные копии (статья 39). Если метод `readObject` не выполнит хотя бы одно из этих условий, злоумышленник сможет относительно легко нарушить инварианты этого класса.

Метод `readObject` — это конструктор, который в качестве единственного входного параметра принимает поток байтов. В норме

мальных условиях этот поток байтов создается в результате сериализации normally построенного экземпляра. Проблема возникает, когда метод `readObject` сталкивается с потоком байтов, полученным искусственно с целью генерации объекта, который нарушает инварианты этого класса. Допустим, что мы лишь добавили «*implements Serializable*» в декларацию класса `Period`. Следующая уродливая программа генерирует такой экземпляр класса `Period`, в котором конец периода предшествует началу:

```
public class BogusPeriod {  
    // Этот поток байтов не мог быть получен из реального экземпляра Period  
    private static final byte[] serializedForm = new byte[] {  
        (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, 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 с указанной сериализованной формой  
    public static Object deserialize(byte[] sf) {  
        try {  
            InputStream is = new ByteArrayInputStream(sf);  
            ObjectInputStream ois = new ObjectInputStream(is);  
            return ois.readObject();  
        }
```

```

        } catch (Exception e) {
            throw new IllegalArgumentException(e.toString());
        }
    }
}

```

Фиксированный массив байтов, используемый для инициализации массива `serializedForm`, был получен путем сериализации обычного экземпляра `Period` и последующего редактирования потока байтов вручную. Детали построения потока для данного примера значения не имеют, однако, если вам это любопытно, формат потока байтов описан в «*Java™ Object Serialization Specification*» [Serialization, 6]. Если вы запустите эту программу, она напечатает: «Fri Jan 01 12:00:00 PST 1999 — Sun Jan 01 12:00:00 PST 1984». Таким образом, то, что класс `Period` стал сериализуемым, позволило создать объект, который нарушает инварианты этого класса.

Для решения этой проблемы создадим в классе `Period` метод `readObject`, который будет вызывать `defaultReadObject` и проверять правильность десериализованного объекта. Если проверка покажет ошибку, метод `readObject` инициирует исключение `InvalidObjectException`, что не позволит закончить десериализацию:>

```

private void readObject(ObjectInputStream s)
throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // Проверим правильность инвариантов
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}

```

Это решение не позволит зломуышленнику создать неправильный экземпляр класса `Period`. Однако 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` в этом

экземпляре. Злоумышленник может прочесть экземпляр Period из ObjectInputStream и получить «неконтролируемые ссылки на объекты», прилагаемые к этому потоку. Имея указанные ссылки, злоумышленник получает доступ к объектам, на которые есть ссылки в закрытых полях Date объекта Period. Меняя эти экземпляры Date, он может менять и сам экземпляр Period. Следующий класс демонстрирует атаку такого рода:

```
public class MutablePeriod {  
    // Экземпляр интервала времени  
    public final Period period;  
    // Поле начала периода, к которому мы не должны иметь доступ  
    public final Date start;  
    // Поле конца периода, к которому мы не должны иметь доступ  
    public final Date end;  
    public MutablePeriod() {  
        try {  
            ByteArrayOutputStream bos =  
                new ByteArrayOutputStream();  
            ObjectOutputStream out =  
                new ObjectOutputStream(bos);  
            // Сериализуем правильный экземпляр Period  
            out.writeObject(new Period(new Date(), new Date()));  
            /*  
             * Добавляем в конец неконтролируемые «ссылки на предыдущие объекты»  
             * для внутренних полей Date в экземпляре Period. Подробнее см.  
             * «Java Object Serialization Specification», раздел 6.4.  
             */  
            byte[] ref = {0x71, 0, 0x7e, 0, 5}; // Ссылка № 5  
            bos.write(ref); // Поле start  
            ref[4] = 4; // Ссылка № 4  
            bos.write(ref); // Поле end  
            // Десериализация экземпляра Period и «украденных» ссылок  
            // на экземпляры Date  
            ObjectInputStream in = new ObjectInputStream(  
                new ByteArrayInputStream(bos.toByteArray()));
```

```

        period = (Period) in.readObject();
        start = (Date) in.readObject();
        end    = (Date) in.readObject();
    } catch (Exception e) {
        throw new RuntimeException(e.toString());
    }
}
}

```

Чтобы увидеть описанную атаку в действии, запустим 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 шестидесятые!
    pEnd.setYear(69);
    System.out.println(p);
}

```

Запустив эту программу, имеем на выходе следующее:

```

Wed Apr 02 11:04:26 PDT 2008 - Sun Apr 02 11:04:26 PST 1978
Wed Apr 02 11:04:26 PDT 2008 - Wed Apr 02 11:04:26 PST 1969

```

Хотя экземпляр `Period` создается с неповрежденными инвариантами, при желании его внутренние компоненты можно поменять извне. Завладев изменяемым экземпляром класса `Period`, злоумышленник может причинить массу вреда, передав этот экземпляр классу, чья безопасность зависит от неизменяемости класса `Period`. И это не такая уж надуманная тема. Существуют классы, чья безопасность зависит от неизменяемости класса `String`.

Причина этой проблемы кроется в том, что метод `readObject` класса `Period` не выполняет необходимого резервного копирования. При

десериализации объекта крайне важно создать резервные копии для всех полей, содержащих ссылки на те объекты, которые не должны попасть в распоряжение клиентов. Поэтому каждый сериализуемый неизменяемый класс, содержащий закрытые изменяемые компоненты, должен в своем методе `readObject` создавать резервные копии для этих компонентов. Следующий метод `readObject` достаточен для того, чтобы объект `Period` оставался неизменяемым и сохранялись его инварианты:

```
// Метод readObject с резервным копированием и проверкой правильности
private void readObject(ObjectInputStream s)
throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // Резервное копирование наших изменяемых компонент
    start = new Date(start.getTime());
    end = new Date(end.getTime());
    // Проверка ваших инвариантов
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

Заметим, что резервное копирование осуществляется перед проверкой корректности и что для резервного копирования не используется метод `clone` из класса `Date`. Указанные особенности реализации необходимы для защиты объекта `Period` (статья 39). Заметим также, что выполнить резервное копирование для полей `final` невозможно. Следовательно, для того чтобы можно было воспользоваться методом `readObject`, мы должны сделать поля `start` и `end` неокончательными. Это огорчает, но приходится выбирать из двух зол меньшее. Разместив в классе метод `readObject` и удалив модификатор `final` из полей `start` и `end`, мы обнаруживаем, что класс `MutablePeriod` потерял свою силу. Приведенная выше программа выводит теперь следующие строки:

```
Wed Apr 02 11:05:47 PDT 2008 - Thu Apr 02 11:05:47 PDT 2008
Wed Apr 02 11:05:47 PDT 2008 - Thu Apr 08 11:05:47 PDT 2008
```

В версии 1.4 методы `writeUnshared` и `readUnshared` добавились к `ObjectOutputStream` с целью воспрепятствовать неконтролируемым атакам ссылок на объекты за счет резервного копирования [Serialization]. К сожалению, эти методы уязвимы к сложным атакам, похожим по своей природе на атаку ElvisStealer, описанную в статье 77. Не используйте методы `writeUnshared` и `readUnshared`. Они обычно быстрее, чем резервное копирование, но они не обеспечивают должных гарантий.

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 резервные копии, как это требуется от конструктора.

Между конструкторами и методами `readObject` существует еще одно сходство, касающееся расширяемых сериализуемых классов: метод `readObject` не должен вызывать переопределяемые методы ни прямо, ни косвенно (статья 17). Если это правило нарушено и вызываемый метод переопределен, то он будет вызван прежде, чем будет десериализовано состояние соответствующего подкласса. Скорее всего, это приведет к сбою программы.

Подведем итоги. Всякий раз, когда вы пишете метод `readObject`, относитесь к нему как к открытому конструктору, который должен создавать правильный экземпляр независимо от того, какой поток байтов был ему передан. Не надо исходить из того, что полученный поток байтов действительно представляет сериализованный экземпляр. Мы рассмотрели примеры классов, использующих сериализованную форму, предлагаемую по умолчанию. В той же степени все это относится к классам со специальными сериализованными фор-

мами. Приведем в кратком изложении рекомендации по написанию «пуленепробиваемого» метода `readObject`:

- Для классов, где есть поля, которые хранят ссылки на объект и при этом должны оставаться закрытыми, для каждого объекта, который должен быть помещен в такое поле, необходимо создавать резервную копию. В эту категорию попадают также изменяемые компоненты неизменяемых классов.
- Для классов, где есть инварианты, выполняйте проверку этих инвариантов и в случае ошибки инициируйте исключение `InvalidObjectException`. Проверка должна производиться после создания всех резервных копий.
- Если после десериализации необходимо проверить целый граф объектов, следует использовать интерфейс `ObjectInputValidation`. Порядок применения этого интерфейса выходит за рамки данной книги. Пример можно найти в *The Java Class Libraries, Second Edition, Volume 1*[Chan98, с. 1256].
- Ни прямо, ни косвенно не используйте в этом классе переопределемых методов.



**Для контроля над экземплярами
предпочитайте использование
перечислимых типов методу `readResolve`**

В статье 3 описывается шаблон `Singleton` и приводится следующий пример класса-синглтона. В этом классе есть ограничения на доступ к конструктору с тем, чтобы гарантировать создание только одного экземпляра:

```
public class Elvis {  
    public static final Elvis INSTANCE = new Elvis();  
    private Elvis() {
```

```

    }
public void leaveTheBuilding { ... }
}

```

Как отмечалось в статье 3, этот класс перестает быть синглтоном, если в его декларацию были добавлены слова «`implements Serializable`». Не имеет значения, использует ли этот класс сериализованную форму, предлагаемую по умолчанию, или использует специальную форму (статья 75), предоставляемую пользователю в этом классе явный метод `readObject` (статья 76). Любой метод `readObject`, явный или используемый по умолчанию, возвращает вновь созданный экземпляр, а не тот самый экземпляр, который был создан в момент инициализации класса.

Функция `readResolve` позволяет нам заменить другой экземпляр на первый, созданный с помощью `readObject` [Serialization, 3.7]. Если класс десериализуемого объекта имеет метод `readResolve` с соответствующей декларацией, то по завершении десериализации этот метод будет вызван для вновь созданного объекта. Клиенту вместо ссылки на вновь созданный объект передается ссылка, возвращаемая этим методом. В большинстве случаев, когда используется этот механизм, ссылка на новый объект не сохраняется, а сам объект фактически является «мертворожденным» и немедленно становится объектом для сборки мусора.

Если класс `Elvis` создан так, чтобы реализовать интерфейс `Serializable`, то для того, чтобы обеспечить свойство синглтона, достаточно будет создать следующий метод `readResolve`:

```

//readResolve для контроля над экземплярами
private Object readResolve(){
    // Возвращает только истинный экземпляр Elvis и дает возможность
    // сборщику мусора позаботиться об Elvis--самозванце
    return INSTANCE;
}

```

Этот метод игнорирует десериализованный объект, просто возвращая уникальный экземпляр Elvis, который был создан во время инициализации класса. По этой причине необходимо, чтобы в сериализованной форме экземпляра Elvis не содержалось никаких реальных данных, а все поля экземпляра были помечены как `transient`. На самом деле, если вы полагаетесь на метод `readResolve` для контроля, все поля со ссылками на объект должны быть объявлены как `transient`. В противном случае для определенного атакующего возможно обезопасить ссылку на десериализованный объект до того, как он запустит метод `readResolve`, используя прием, который частично похож на атаку `MutablePeriod` из статьи 76.

Атака немного сложнее, но основная идея проста. Если синглтон содержит непереходное (*nontransient*) поле со ссылкой на объект, содержимое этого поля будет десериализовано до того, как будет запущен метод синглтона `readResolve`. Это позволяет осторожно созданному потоку «украсть» ссылку на исходный десериализуемый синглтон, в то время как 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 скрыт. В потоке сериализации происходит замена непереходного поля синглтона экземпляром класса-вора. Теперь у вас появилась цикличность: синглтон содержит класс-вор и класс-вор ссылается на синглтон.

Поскольку синглтон содержит класс-вор, то метод этого класса `readResolve` выполняется сначала тогда, когда синглтон десериализован. В результате, когда метод `readResolve` выполняется, поле его экземпляра все еще ссылается на частично десериализованный синглтон.

Метод класса-вора `readResolve` копирует ссылку из поля своего экземпляра в статическое поле, чтобы она была доступна после выполнения метода `readResolve`. Затем метод возвращает значение корректного типа для поля, в котором он прячется. Если бы этого

не делалось, то виртуальная машина вывела бы ClassException при попытке системы сериализации сохранить украденную ссылку в это поле.

Для конкретности рассмотрим следующий прием взломанного синглтона:

```
// Взломанный синглтон - у него есть непереходное (nontransient)
// поле со ссылкой 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;
    }
}
```

А вот класс-вор, созданный как описано выше:

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;
    private Object readResolve() {
        // Сохраняет ссылку в экземпляр Elvis
        impersonator = payload;
        // Возвращает объект правильного типа для поля favoriteSongs
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
```

И наконец, ужасная программа, которая десериализует поток, сделанный руками для производства двух различных экземпляров

испорченного синглтона. Метод десериализации опущен из программы, так как он идентичен такому же методу на странице 303:

```
public class ElvisImpersonator {  
    // Битовый поток не может прийти из реального экземпляра Elvis!  
    private static final byte[] serializedForm = new byte[] {  
        (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, 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) {  
        // Инициализирует ElvisStealer.impersonator и возвращает  
        // реальный Elvis (которым является Elvis.INSTANCE)  
        Elvis elvis = (Elvis) deserialize(serializedForm);  
        Elvis impersonator = ElvisStealer.impersonator;  
        elvis.printFavorites();  
        impersonator.printFavorites();  
    }  
}
```

Запуск этой программы приводит к следующему результату, доказывающему, что можно создать два различных экземпляра Elvis (с различными вкусами в музыке):

```
[Hound Dog, Heartbreak Hotel]  
[A Fool Such as I]
```

Вы можете решить проблему, объявив поле favoriteSong как transient но лучше проблему решить, сделав Elvis единственным элементом перечислимого типа (статья 3). Исторически метод readResolve использовался для всех сериализуемых классов, контролируемых экземплярами. Но с версии 1.5 это более не является лучшим способом поддерживать контроль экземпляров в сериализуемом классе. Как нам продемонстрировала атака ElvisStealer, данный приём хрупок и требует огромной осторожности.

Если вы напишете вместо этого сериализуемый класс, контролируемый экземплярами, как перечислимый тип, у вас будет железная гарантия, что никаких экземпляров, кроме объявленных констант, не появится. Виртуальная машина Java даст вам такую гарантию, и вы можете на неё положиться. Но с вашей стороны потребуется особая забота. Вот как будет выглядеть пример с Elvis, если его сделать как перечислимый тип:

```
// Синглтон как перечислимый тип - более 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 контроля над экземплярами не устарело. Если вам нужно написать сериализуемый класс с управляемыми экземплярами, экземпляры которого не известны на момент компиляции, вы не сможете представить класс как перечислимый тип.

Доступ метода readResolve очень важен. Если вы поместите метод readResolve в завершённый класс, он должен быть закрытым. Если вы поместите метод readResolve в незавершённый класс, вам нужно внимательно отнестись к его доступу. Если он закрытый, то его действие не будет распространяться на подклассы. Если он открытый

только в рамках пакета, его действие будет распространяться только на подклассы внутри одного и того же пакета. Если он защищенный или открытый, то его действие будет распространяться на все подклассы, не переопределяющие его. Если метод `readResolve` защищен или является открытым и подклассы его не переопределяют, то десериализация и сериализация экземпляра подкласса произведет экземпляр суперкласса, который, скорее всего, приведет к ошибке `ClassCastException`.

Подведем итоги. Вам нужно использовать перечислимые типы, чтобы установить контроль экземпляров над инвариантами, насколько это возможно. Если это невозможно и вам нужен класс, который был бы и сериализуемым, и с контролем экземпляров, то вам нужен метод `readResolve`, чтобы убедиться, что все поля экземпляров класса являются либо примитивными, либо переходными (*transient*).

Страница
78

Рассмотрите использование агентов сериализации вместо сериализованных экземпляров

Как уже говорилось в статье 74 и обсуждалось на протяжении всей главы, применение `Serializable` увеличивает вероятность ошибок и проблем с безопасностью, так как он приводит к созданию экземпляров, использующих механизмы, за пределами языка вместо обычных конструкторов. Есть тем не менее прием, который существенно снижает подобный риск. Этот прием известен как *шаблон агента сериализации*.

Шаблон агента сериализации довольно прямолинеен. Во-первых, создайте закрытый статический вложенный класс сериализуемого класса, который точно представляет собой логическое состояние экземпляра окружающего класса. Этот вложенный класс, известный как агент сериализации, должен иметь один конструктор, типом параметров которого является окружающий класс. Этот конструктор просто копирует данные из его аргумента: ему не требуется проверка целостности или резервное копирование. С точки зрения дизайна сериализованная форма агента сериализации по умолчанию является

совершенной сериализованной формой окружающего класса. И окружающий класс, и его агент сериализации должны быть декларированы так, чтобы реализовывать `Serializable`.

Например, рассмотрим неизменяемый класс `Period`, написанный в статье 39 и сериализованный в статье 76. Вот агент сериализации для этого класса. `Period` настолько прост, что его агент сериализации содержит в точности те же самые поля, что и класс:

```
// Агент сериализации 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; // Any number will do (Item 75)
}
```

Затем добавьте следующий метод `writeReplace` к окружающему классу. Этот метод можно дословно скопировать в любой класс с агентом сериализации:

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

Присутствие этого метода заставляет систему сериализации выдать экземпляр `SerializationProxy` вместо экземпляра окружающего класса. Другими словами, метод `writeReplace` переводит экземпляр окружающего класса в его агент сериализации до самой сериализации.

При использовании метода `writeReplace` система сериализации никогда не выдаст сериализованный экземпляр окружающего класса, но его можно сфабриковать при попытке злонамеренно разрушить

инварианты класса. Для того чтобы гарантировать, что такая атака не удастся, просто добавьте метод `readObject` к окружающему классу:

```
// Метод readObject для шаблона агента сериализации
private void readObject(ObjectInputStream stream)
throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

Наконец, предоставьте метод `readResolve` классу `SerializationProxy`, который возвратит логически эквивалентный экземпляр окружающего класса. Присутствие этого метода приводит к тому, что система сериализации переводит агента сериализации обратно в экземпляр окружающего класса при десериализации.

Этот метод `readResolve` создает экземпляр окружающего класса, используя только открытый API, и в этом и заключается вся красота шаблона. Он в большей степени избегает неязыковой сути сериализации, потому что создается десериализованный экземпляр с использованием тех же самых конструкторов, методов статической генерации, как при создании любого другого экземпляра. Это освобождает вас от необходимости отдельно проверять, чтобы десериализованные экземпляры подчинялись инвариантам класса. Если методы статической генерации класса или конструкторы устанавливают эти инварианты и методы его экземпляра поддерживают их, то у вас есть гарантия, что сериализация будет также их поддерживать.

Вот пример метода `readResolve` для вышеупомянутого `Period.SerializationProxy`

```
// Метод readResolve для Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end); // Uses public constructor
}
```

Как и подход с использованием резервного копирования (статья 76), подход с использованием агента сериализации препятствует тяжелым бит-потоковым атакам (статья 76) и атакам кражи вну-

треннего поля (статья 76). В отличие от двух предыдущих подходов этот позволяет полю, принадлежащему к `Period`, быть завершенным, что требуется для того, чтобы класс `Period` был на самом деле неизменяемым (статья 15). И в отличие от двух предыдущих подходов он не требует длительного размышления. Вам не нужно думать о том, какие поля могут быть дискредитированы при атаках сериализации, вам также не нужно явно выполнять проверку действительности в качестве части десериализации.

Рассмотрим случай с `EnumSet` (статья 32). У этого класса нет открытого конструктора, только методы статической генерации. С точки зрения клиента, она возвращает экземпляры `EnumSet`, но на самом деле она возвращают один или два подкласса, в зависимости от размера основного перечислимого типа (статья 1). Если у основного перечислимого типа 64 или менее элементов, то методы статической генерации выдают `RegularEnumSet`; в противном случае они возвращают `JumboEnumSet`. Теперь посмотрим, что происходит, если вы сериализуете набор перечислимых типов, в котором перечислимый тип содержит 60 элементов, затем добавите к нему еще пять элементов и десериализуете набор. При сериализации это был экземпляр `RegularEnumSet`, но лучше бы ему быть `JumboEnumSet` при десериализации. На самом деле это точно то, что и происходит, потому что `EnumSet` использует шаблон агента сериализации. Если вам интересно, вот как выглядит агент сериализации `EnumSet`. Он действительно настолько прост:

```
// Агент сериализации EnumSet
private static class SerializationProxy <E extends Enum<E>>
    implements Serializable {
    // Тип элементов данного набора перечислимых типов
    private final Class<E> elementType;
    // Элементы, содержащиеся в данном наборе перечислимых типов
    private final E[] elements;
    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(EMPTY_ENUM_ARRAY); // (Item 43)
```

```
}

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 агента сериализации есть два ограничения. Он несовместим с классами, расширяемыми своими клиентами (статья 17). Он также несовместим с некоторыми классами, диаграммы объектов которых содержат цикличности: если вы попытаетесь запустить метод на объекте в рамках метода `readResolve` агента сериализации, то получите ошибку `ClassCastException`, так как у вас еще нет объекта, а только его агент сериализации.

Наконец, дополнительные возможности и безопасность шаблона агента сериализации не свободны. Но моей машине на 14% более затратно сериализовывать и десериализовывать экземпляры `Period` с помощью агента сериализации, чем с помощью резервного копирования.

Подведем итоги. Рассмотрите шаблон агента сериализации каждый раз, когда вам приходится писать методы `readObject` или `writeObject` на классе, который нерасширяем для своих клиентов. Этот шаблон, возможно, является простейшим путем сериализации объектов с нетривиальными инвариантами.

Список литературы

- [Arnold05] Arnold, Ken, James Gosling, and David Holmes. *The Java™ Programming Language, Fourth Edition*. Addison-Wesley, Boston, 2005. ISBN: 0321349806.
- [Asserts] *Programming with Assertions*. Sun Microsystems. 2002. <<http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html>>
- [Beck99] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416.
- [Beck04] Beck, Kent. *JUnit Pocket Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2004. ISBN: 0596007434.
- [Bloch01] Bloch, Joshua. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, 2001. ISBN: 0201310058.
- [Bloch05] Bloch, Joshua, and Neal Gafter. *Java™ Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, Boston, 2005. ISBN: 032133678X.
- [Bloch06] Bloch, Joshua. Collections. In *The Java™ Tutorial: A Short Course on the Basics, Fourth Edition*. Sharon Zakhour et al. Addison-Wesley, Boston, 2006. ISBN: 0321334205. Pages 293–368. Also available as <<http://java.sun.com/docs/books/tutorial/collections/index.html>>.

- [Bracha04] Bracha, Gilad. *Generics in the Java Programming Language*. 2004.
<<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>>
- [Burn01] Burn, Oliver. *Checkstyle*. 2001–2007.
<<http://checkstyle.sourceforge.net>>
- [Collections] *The Collections Framework*. Sun Microsystems. March 2006.
<<http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html>>
- [Gafter07] Gafter, Neal. *A Limitation of Super Type Tokens*. 2007.
<<http://gafter.blogspot.com/2007/05/limitation-of-super-type-tokens.html>>
- [Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN: 0201633612.
- [Goetz06] Goetz, Brian, with Tim Peierls et al. *Java Concurrency in Practice*. Addison-Wesley, Boston, 2006. ISBN: 0321349601.
- [Gong03] Gong, Li, Gary Ellison, and Mary Dageforde. *Inside Java™ 2 Platform Security, Second Edition*. Addison-Wesley, Boston, 2003. ISBN: 0201787911.
- [HTML401] *HTML 4.01 Specification*. World Wide Web Consortium. December 1999.
<<http://www.w3.org/TR/1999/REC-html401-19991224/>>
- [Jackson75] Jackson, M. A. *Principles of Program Design*. Academic Press, London, 1975. ISBN: 0123790506.
- [Java5-feat] *New Features and Enhancements J2SE 5.0*. Sun Microsystems. 2004.
<<http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html>>

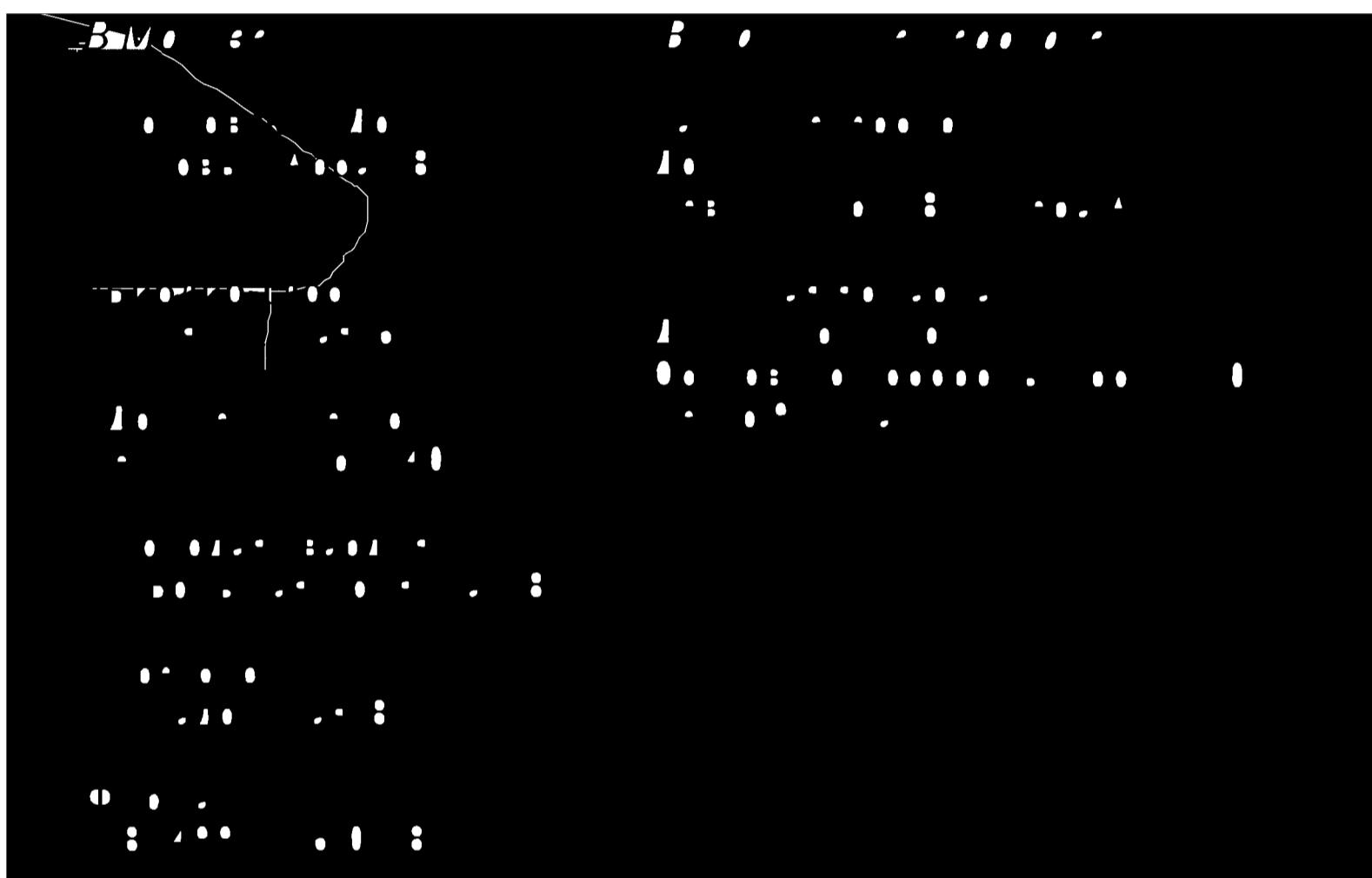
- [Java6-feat] *Java™ SE 6 Release Notes: Features and Enhancements*. Sun Microsystems. 2008.
<<http://java.sun.com/javase/6/webnotes/features.html>>
- [JavaBeans] *JavaBeans™ Spec.* Sun Microsystems. March 2001.
<<http://java.sun.com/products/javabeans/docs/spec.html>>
- [Javadoc-5.0] *What's New in Javadoc 5.0.* Sun Microsystems. 2004.
<<http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/whatsnew-1.5.0.html>>
- [Javadoc-guide] *How to Write Doc Comments for the Javadoc Tool.* Sun Microsystems. 2000–2004.
<<http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>>
- [Javadoc-ref] *Javadoc Reference Guide.* Sun Microsystems. 2002–2006.
<<http://java.sun.com/javase/6/docs/technotes/tools/solaris/javadoc.html>>
<<http://java.sun.com/javase/6/docs/technotes/tools/windows/javadoc.html>>
- [JavaSE6] *Java™ Platform, Standard Edition 6 API Specification.* Sun Microsystems. March 2006.
<<http://java.sun.com/javase/6/docs/api/>>
- [JLS] *Gosling, James, Bill Joy, and Guy Steele, and Gilad Bracha.*
The Java™ Language Specification, Third Edition. Addison-Wesley, Boston, 2005. ISBN: 0321246780.
- [Kahan91] Kahan, William, and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic.* UCB/CSD-91-667, University of California, Berkeley, 1991.
- [Knuth74] Knuth, Donald. *Structured Programming with go to Statements.* In Computing Surveys 6 (1974): 261–301.
- [Langer08] Langer, Angelika. *Java Generics FAQs — Frequently Asked Questions.* 2008.

- <<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>>
- [Lea00] Lea, Doug. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310090.
- [Lieberman86] Lieberman, Henry. 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.
- [Liskov87] Liskov, B. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of OOPSLA '87 and SIGPLAN Notices*, Vol. 23, No. 5: 17–34, May 1988.
- [Meyers98] Meyers, Scott. *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, MA, 1998. ISBN: 0201924889.
- [Naftalin07] Naftalin, Maurice, and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., Sebastopol, CA, 2007. ISBN: 0596527756.
- [Parnas72] Parnas, D. L. *On the Criteria to Be Used in Decomposing Systems into Modules*. In *Communications of the ACM* 15 (1972): 1053–1058.
- [Posix] 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.
- [Pugh01] *The “Double-Checked Locking is Broken” Declaration*. Ed. William Pugh. University of Maryland. March 2001.
<<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>>

- [Serialization] *Java™ Object Serialization Specification*. Sun Microsystems. March 2005.
[<http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html>](http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html)
- [Sestoft05] Sestoft, Peter. *Java Precisely, Second Edition*. The MIT Press, Cambridge, MA, 2005. ISBN: 0262693259.
- [Smith62] Smith, Robert. Algorithm 116 Complex Division. In *Communications of the ACM*, 5.8 (August 1962): 435.
- [Snyder86] Snyder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, 38–45, 1986. ACM Press.
- [Thomas94] Thomas, Jim, and Jerome T. Coonen. *Issues Regarding Imaginary Types for C and C++*. In *The Journal of C Language Translation*, 5.3 (March 1994): 134–138.
- [ThreadStop] *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?* Sun Microsystems. 1999.
[<http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html>](http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html)
- [Viega01] Viega, John, and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, Boston, 2001. ISBN: 020172152X.
- [W3C-validator] *W3C Markup Validation Service*. World Wide Web Consortium. 2007.
 [<http://validator.w3.org/>](http://validator.w3.org/)
- [Wulf72] Wulf, W. A Case Against the GOTO. In *Proceedings of the 25th ACM National Conference* 2 (1972): 791–797.

Книги издательства «ЛОРИ»

Вы можете приобрести:



В Украине:

Интернет-магазин
www.books.ua

Книжные интернет-магазины:

Магазин «ОЗОН» www.ozon.ru

Магазин «Колибри» www.colibri.ru

Магазин «Чакона» www.chaconne.ru

Магазин «10 Книг» www.10books.ru



Издательство
«ЛОРИ»
www.lory-press.ru