Джошуа Блох

Effective Java

Оглавление

От редактора PDF-издания

Шестнадцать лет назад — для нашей профессии это круглое число! — вышло первое издание книги Джошуа Блоха «Effective Java». С тех пор и по сей день она остаётся одной из главных настольных книг для любого Java-разработчика, от начинающего до опытного.

К сожалению, русские переводы англоязычных книг по программированию (да и не только по программированию) зачастую не отличаются качеством, а сами переводчики не отличаются компетентностью в предметной области. Например, я видела перевод двухтомника К. Хорстманна «Core Java», в котором выражение «fork-join architecture» было переведено как «архитектура вилочного соединения». Ещё хуже дело обстоит с Effective Java. Единственный известный мне перевод этой книги (издательство «Лори», 2014, ISBN 978-3-83382-348-6) пестрит стилистическими ошибками, фактическими ошибками, а также неправильными переводами терминов, которые к тому же переведены по-разному в разных главах. Читателю, только освоившему основы языка Java и приступившему к этой книге, будет сложно понимать, что под «комментариями» в русском тексте имеются в виду аннотации, а под «автоматическим созданием контейнеров» — автоупаковка примитивных типов. Более того, некоторые фрагменты текста и примеры кода вообще расходятся с английским оригиналом, потому что, по-видимому, без изменений перекочевали из перевода первого издания.

Когда я начала обучать языку Java свою девушку, я поняла, что на таком переводе мы далеко не уйдём — а английский оригинал был бы достаточно концептуально сложен для человека, только-только прошедшего двухтомник Хорстманна и ещё не набившего руку на практических примерах. Поэтому я взяла перевод издательства «Лори» и прошлась с редактурой по всем статьям книги от первой до последней, 78-й. Сверяясь с английским оригиналом, я привела перевод терминологии в единообразное состояние, убрала расхождения с оригиналом, исправила стилистические ошибки. Результатом этой редакторской работы стало то электронное издание книги, которое вы читаете сейчас.

Но одной только правкой перевода дело не ограничилось. Мне хотелось сделать эту PDF-версию книги настолько хорошей, насколько я могла, и использовать все преимущества электронного формата. Если вы раньше читали Effective Java, то вы помните, что статьи книги часто ссылаются друг на друга. В этом PDF-файле такие ссылки действительно оформлены как рабочие гиперссылки (и выделены цветом, например, так: статья 1).

Кроме того, все листинги кода используют подсветку синтаксиса, а выбранные для этого издания шрифты, хочется верить, сделают книгу более приятной для чтения, чем оригинальный перевод.

Наконец, читая книгу, вы можете заметить разбросанные по тексту книги блоки с комментариями, вроде такого:

> **Комментарий.** В Java 8 эта идиома уже не так актуальна благодаря новым выразительным средствам языка, позволяющим записать то же самое гораздо короче.

Дело в том, что второе и последнее на данный момент издание Effective Java вышло в 2008 году, когда последней версией платформы Java SE была версия 6. Это нужно учитывать при чтении, потому что как сама платформа, так и культура разработчиков с тех пор ушли вперёд. Вышли Java 7 и 8, на подходе Java 9; такие новые конструкции языка, как ромбовидный оператор, инструкция `try` с ресурсами, неабстрактные методы интерфейсов и особенно лямбда-выражения, а также новые API вроде `Objects`, `Optional` и `Stream`, сделали некоторые старые идиомы неактуальными или внесли в них коррективы. Тем не менее подавляющее большинство рекомендаций из этой книги остаются актуальными и поныне, а в тех местах текста, где имеет смысл знать о новых возможностях языка, я и добавила свои комментарии.

Приятного чтения — и приятного программирования на Java!

Апрель 2017 г.

Предисловие

Если бы сослуживец сказал вам: «Супруга моя этим вечером сегодня производит в доме необычный обед. Ты приходишь?», то вам в голову, вероятно, пришли бы сразу три мысли: в-третьих, что вас пригласили на обед; во-вторых, что 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, посвящённых языкам программирования, часто освещаются лишь первые два вопроса, приёмы 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, распределённые вычисления и безопасность. Однако как наилучшим образом использовать этот язык на практике?

Есть и другой аспект. Программы, в отличие от произнесённых 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 ещё 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 трёх 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, посвящённых библиотекам и прикладным интерфейсам, которые связаны с языком Java.

Эта книга посвящена третьей теме: общепринятым и эффективным приёмам работы с языком Java. Джошуа Блох (Joshua Bloch) провёл несколько лет в компании Sun Microsystems, работая с языком программирования Java, занимаясь расширением и реализацией программного кода. Он также прочёл большое количество программного кода, написанного многими людьми, в том числе и мной. Здесь же, приведя в некую систему, он даёт дельные 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 обобщённых типов, перечислимых типов, аннотаций, автоупаковки и циклов for-each в Java 5. Ещё одним новшеством стало добавление новой библиотеки параллельного программирования `java.util.concurrent`, также появившейся в Java 5. Мне повезло, что вместе с Гиладом Брахой я смог возглавить команду, разработавшую новые особенности языка. Мне также повезло в том, что удалось работать в команде, возглавляемой Дагом Ли и разработавшей библиотеку параллельного программирования.

Другим значительным изменением в платформе стало повсеместное распространение интегрированных сред разработки, таких как Eclipse, IntelliJ IDEA и NetBeans, и инструментов статического анализа, 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 API параллельного программирования и коллекций из офисов Google и через 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, требуемый для обновления данной книги. Невозможно было избежать того, что книга стала больше, — и она действительно увеличилась с 57 до 78 статей. Я не просто добавил 23 новые статьи, а тщательно переработал весь изначальный материал — 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 релизах 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 платформы Java. Я занимался проектированием и разработкой множества таких библиотек, занимался их поддержкой, а также давал консультации по многим другим библиотекам. Контроль над этими библиотеками в ходе становления платформы языка Java был возможностью, которая предоставляется только раз в жизни. Не будет преувеличением сказать, что я имел честь работать с великими разработчиками нашего времени. В процессе работы я многое узнал о языке программирования Java: что в нём хорошо, 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 нашел такое оформление необычайно эффективным, и, надеюсь, вы тоже его оцените.

Во многих случаях я осмелился иллюстрировать статьи реальными примерами из библиотек 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 переиспользуемых компонентов, она неизбежно отражает мой опыт в написании таковых, накопленный за последние два десятилетия. Я привык думать в терминах прикладных программных интерфейсов (API) и предлагаю вам делать то же. Даже если вы не занимаетесь разработкой переиспользуемых компонентов, если вы будете пользоваться этими терминами, это может повысить качество написанных вами программ. Более того, нередко случается писать переиспользуемых 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 проектированию 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, какая только сможет работать. Если вы пользуетесь одной из этих методик, то обнаружите, что внимание к разработке API сослужит вам добрую службу в процессе последующего рефакторинга программы (refactoring). Основной задачей рефакторинга является усовершенствование структуры системы, а также исключение дублирующего программного кода. Этой цели невозможно достичь, если у компонентов 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.

Благодарности

Благодарности ко второй редакции

Я 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, организованна 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 Bougher), Синди Блох (Cindy Bloch), Бет Боттос (Beth Bottos), Джо Боубир (Joe Bowbeer), Брайан Гоутс (Brian Goetz), Тим Хэлоран (Tim Halloran), Брайан Кернихэм (Brian Kernigham), Роб Конингсберг Rob Koningsberg), Тим Пирлс (Tim Peierls), Билл Пу (Bill Pough), Иошики Шибата (Yoshiki Shibata), Питер Стаут (Peter Stout), Питер Вайнбергер (Peter Weinberger) и Фрэнк Иеллин (Frank Yellin). Другие рецензенты — Пабло Беллвер (Pablo Bellver), Дэн Блох (Dan Bloch), Дэн Борнстайн (Dan Bornstein), Кевин Буриллион (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 Tereshshenko) и Пол Тима (Paul Tyma). 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 Тиму 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 Хендриксона (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 Гарсиа (Iris Garcia), Константин Кладко (Konstantin Kladko), Иена Литтл (Ian Little), Майк Маклоски (Mike McCloskey) и Марк Рейнхольд (Mark Reinhold). Среди бывших членов группы: Дзенгуа Ли (Zhenghua Li), Билл Мэддокс (Bill Maddox) и Нейвин Санджива (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), Гладу Брахе (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), Питеру 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 Рейнхолду (Mark Reinhold), Кену Расселу (Ken Russell), Биллу Шэннону (Bill Shannon), Питеру Стауту (Peter Stout), Филу Уодлеру (Phil Wadler), ДавидуХолмсу (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), Стефен Грерап (Steffen Grarup), Питер Кесслер (Peter Kessler), Ричард Рода (Richard Roda), Джон Роуз (John Rose) и Дэвид Стаутэмайер (David Stoutamire), давшие полезные разъяснения. Особая благого Б лагодарности к первой редакции 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.

Глава 1. 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 перекрёстных ссылок, поэтому вы можете с лёгкостью построить по этой книге свой 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:

Новая возможность	Глава или статья, где она описана
Обобщённые типы	Глава 5
Перечисления	Статьи 30—34
Аннотации	Статьи 35—37
Циклы for-each	Статья 46
Автоупаковка	Статьи 40, 49
Методы с переменным числом параметров	Статья 42
Статический импорт	Статья 19
`java.util.concurrent`	Статьи 68, 69

Большинство статей иллюстрируются примерами программ. Главной особенностью этой книги является наличие в ней примеров программного кода, кото рые иллюстрируют многие *шаблоны* (design patterns) 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 приёмы, которых следует избегать. Подобные примеры, иногда называемые *«антишаблонами»* (anti-pattern), чётко обозначены комментарием, таким как «`Никогда так не делайте!`». В каждом таком случае в статье даётся 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 выявлять 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 приёмы 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 идёт о написании понятных, правильных, полезных, надёжных, 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.

Комментарий. В 64-битных сборках JRE и JDK поддерживается только Server VM.

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 следующей таблице показано соответствие между названиями версий и их рабочими номерами.

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 Platform, Standard Edition 6	1.6
Java Platform, Standard Edition 7	*1.7*
Java Platform, Standard Edition 8	*1.8*

Комментарий. На момент написания оригинала последней версией Java SE была версия 6. Соответственно, в книге не разбираются новые возможности платформы, появившиеся в версиях 7 и 8.

Данные примеры по возможности являются полными, однако предпочтение отдаётся не завершённости, а удобству чтения. В примерах широко используются классы пакетов `java.util` и `java.io`. Соответственно, 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 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 (extends). 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 сериализованным 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 одноимённой 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 сериализованные формы все вместе называются *элементами* API (API element). Внешний API образуется из элементов API, которые доступны за пределами пакета, где этот API был определён. 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 защищённых (protected) членов, а также конструкторов всех открытых классов и интерфейсов в пакете.

Глава 2. Создание и уничтожение объектов

В этой главе речь идёт 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 фабричными методами

Обычно, чтобы разрешить клиенту получать экземпляр класса, ему предоставляется открытый (public) конструктор. Есть и другой, менее известный приём, который должен быть в арсенале любого программиста. Класс может иметь открытый *статический фабричный метод* (static factory method), который является просто статическим методом, возвращающим экземпляр класса. Простой пример такого метода возьмём 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 фабричный метод и шаблон “фабричный метод” (Factory Method) из Шаблонов проектирования (Gamma95, с. 107) не есть одно и то же.

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

Первое преимущество статического фабричного метода состоит в том, что, в отличие от конструкторов, он имеет название. Тогда как параметры конструктора сами по себе не дают описания возвращаемого объекта, статический фабричный метод 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 фабричный метод с названием `BigInteger.probablePrime`. (В конечном счёте 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 фабричные методы 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 конструкторов с одной и той же сигнатурой, вам следует рассмотреть возможность замены одного или нескольких конструкторов статическими фабричными методами. Тщательно выбранные названия будут подчёркивать их различия.

Второе преимущество статических фабричных методов 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 приём иллюстрирует метод `Boolean.valueOf(boolean)`: он не создаёт 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 фабричных методов при повторных вызовах возвращать тот же самый объект позволяет классам в любой момент времени чётко контролировать, какие экземпляры объекта ещё существуют. Классы, которые делают это, называются *классами с контролем 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 неинстанцируемым (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 даёт такую гарантию, его клиенты вместо метода `equals(Object)` могут использовать оператор `==`, что может привести к существенному повышению производительности программы. Перечислимые типы (статья 30) также дают такую гарантию.

Третье преимущество статического фабричного метода заключается в том, что, в отличие от конструктора, он может возвратить объект, который соответствует не только заявленному типу возвращаемого значения, но и любому его подтипу. Это даёт вам значительную гибкость в выборе класса для возвращаемого объекта.

Например, благодаря такой гибкости интерфейс API может возвращать объект, не декларируя его класс как `public`. Сокрытие реализации классов может привести к созданию очень компактного API. Этот приём идеально подходит для *фреймворков, построенных на интерфейсах* (interface-based frameworks, статья 18), когда эти интерфейсы для статических фабричных методов 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.

> **Комментарий.** В Java 8 интерфейсы могут содержать статические методы, поэтому в новых API фабричные методы, возвращающие интерфейсы, можно включать в сам интерфейс.

Например, архитектура Collections Framework имеет тридцать две полезные реализации интерфейсов коллекций: неизменяемые коллекции, синхронизированные коллекции и т.д. Большинство этих реализаций с помощью статических фабричных методов сводятся в единственный класс (`java.util.Collections`), экземпляр которого создать невозможно. 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 нужды читать дополнительные документы

к этому классу. Более того, использование такого статического фабричного метода даёт клиенту право обращаться к возвращаемому объекту, используя его собственный интерфейс, а не через интерфейс класса реализации, что обычно является хорошим приёмом (статья 52).

Скрытым может быть не только класс объекта, возвращаемого открытым статическим фабричным методом. Сам этот класс может меняться от вызова к вызову, в зависимости от того, какие значения параметров переданы статическому фабричному методу. Это может быть любой класс, который является подтипом 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.

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`, подкреплённый единичным значением `long`. Если же тип перечислений содержит 65 и более элементов, то метод возвращает экземпляр `JumboEnumSet`, подкреплённый массивом `long`. Существование двух реализаций классов невидимо для клиентов. Если экземпляр `RegularEnumSet` перестанет давать преимущество в производительности перечислимым типам с небольшим количеством элементов, то его можно избежать в дальнейшем без каких-либо вредных последствий. Таким же образом, при будущем выполнении могут добавиться третья и четвёртая 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`.

В момент, когда пишется класс, содержащий статический фабричный метод, класс, соответствующий возвращаемому объекту, может даже не существовать. Подобные гибкие статические фабричные методы 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.

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 «гибкая статическая фабрика», составляющая основу системы предоставления услуг.

Есть ещё необязательный четвёртый компонент службы предоставления услуг — интерфейс поставщика службы (service provider interface, SPI), который реализуется поставщиком для создания экземпляров реализации службы. При отсутствии этого интерфейса реализации регистрируются по имени класса, а их экземпляры создаются с помощью рефлексии (статья 53). В случае с JDBC (Java Database Connectivity) `Connection` играет роль интерфейса службы, `DriverManager.registerDriver` — это интерфейс регистрации провайдера, `DriverManager.getConnection` — интерфейс доступа к службе, а `Driver` — интерфейс поставщика службы.

Может быть несколько вариантов шаблона службы предоставления услуг. Например, интерфейс доступа к службе может возвратить более развёрнутый интерфейс службы, чем требуемый от поставщика, 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();
    }
}
```

Четвёртое преимущество статических фабричных методов заключается в том, что они уменьшают многословие при создании экземпляров параметризованных типов. К сожалению, вам необходимо определить параметры типа при вызове конструктора параметризованных классов, 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 определить типы параметров. Это называется *выведением типов* (type inference). Например, предположим, что в реализации `HashMap` присутствует следующий фабричный 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 добавить эти методы в собственный класс параметров. Теперь вы сможете предоставлять статические фабричные методы в собственных параметризованных классах.

> **Комментарий.** Начиная с Java 7, компилятор поддерживает вывод типов при вызове конструкторов параметризованных типов. Например, приведённый выше пример с `HashMap` можно переписать так:
>
> ```
> Map<String, List<String>> m = new HashMap<>();
> ```

Основной недостаток использования только статических фабричных методов заключается в том, что классы, не имеющие открытых или защищённых конструкторов, не могут иметь подклассов. Это же верно и в отношении классов, которые возвращаются

открытыми статическими фабричными методами, но сами открытыми не являются. Например, в архитектуре Collections Framework невозможно создать подкласс ни для одного из классов реализации. Можно, правда, сказать, что это скорее преимущество, а не недостаток, поскольку это поощряет программистов использовать не наследование, а композицию (статья 14).

Второй недостаток статических фабричных методов 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 фабричные методы. Возможно, когда-нибудь в официальной документации по Java будет уделено должное внимание статическим фабричным методам. Указанный недостаток может быть смягчён, если придерживаться стандартных соглашений, касающихся именования. Вот некоторые типичные имена статических фабричных методов:

- `valueOf` — возвращает экземпляр, который, грубо говоря, имеет то же значение, что и его параметры. Статические фабричные методы с таким названием фактически являются операторами преобразования типов.
- `of` — более краткая альтернатива для `valueOf`, популяризированная классом `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` даёт гарантию, что каждый экземпляр отличается от всех остальных.
- `getType` — то же, что и `getInstance`, но используется, когда фабричный метод находится в другом классе. `Type` обозначает тип объекта, возвращённого фабричным методом.
- `newType` — то же, что и `newInstance`, но используется, когда фабричный метод находится в другом классе. `Type` обозначает тип объекта, возвращённого фабричным методом.

Подведём итоги. И статические фабричные методы, и открытые конструкторы имеют свою область применения, имеет смысл разобраться, какие они имеют достоинства

друг перед другом. Часто фабричные методы предпочтительнее, поэтому не надо бросаться создавать конструкторы, не рассмотрев сначала возможность использования статических фабричных методов, поскольку последние часто оказываются лучше.

Статья 2. Используйте шаблон Builder, когда приходится иметь дело с большим количеством параметров конструктора

У конструкторов и статических фабричных методов есть одно общее ограничение: они плохо масштабируются на 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 трансжиров, содержание холестерина, натрия и т.д. У большинства продуктов ненулевыми будут только несколько из этих необязательных значений.

Какой конструктор или какие методы нужно использовать для написания данного класса? Традиционно программисты использовали шаблон *“телескопический конструктор”* (telescoping constructor pattern), при использовании которого вы пишете набор конструкторов: конструктор с одними лишь обязательными параметрами, конструктор с одним необязательным параметром, конструктор с двумя необязательными 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 необязательных 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, **шаблон "телескопический конструктор" нормально работает, но становится трудно писать код программы-клиента, когда имеется много параметров, а ещё труднее этот код читать.** Читателю остаётся только гадать, что означают все эти значения, и нужно тщательно высчитывать позицию параметра, чтобы выяснить, к какому полю он относится. Длинные последовательности 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).

Второе традиционное решение проблемы конструкторов со многими параметрами — это использование шаблона *JavaBeans*, где вы вызываете конструктор без параметров, чтобы создать объект, а затем вызываете сеттеры для установки обязательных и всех интересующих вас необязательных параметров:

```
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 лишён недостатков шаблона телескопических конструкторов. Он прост, хотя и содержит большое количество слов, и 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 лишён серьёзных недостатков. Поскольку его создание разделено между несколькими вызовами методов, **JavaBean может находиться в неустойчивом состоянии во время создания.** У класса нет возможности принудительно обеспечить стабильность простой проверкой действительности параметров конструктора. Попытка использования объекта в то время, когда он находится в неустойчивом состоянии, может привести к ошибкам выполнения даже после удаления ошибки из кода, что создаёт трудности при отладке. Схожим недостатком является то, что **шаблон JavaBeans исключает возможность сделать класс неизменяемым** (статья 15), что требует дополнительных усилий со стороны программиста для обеспечения потокобезопасности.

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 Builder [Gamma 95, с. 97]. Вместо непосредственного создания желаемого объекта клиент вызывает конструктор (или статический метод) со всеми обязательными параметрами и получает *объект-билдер* (builder object). Затем клиент вызывает сеттеры на этом объекте для установки всех интересующих необязательных параметров. Наконец, клиент вызывает метод `build` для генерации объекта, который будет являться неизменяемым. Билдер является статическим внутренним классом в классе (статья 22), который он создаёт. 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 неизменяемым и что все значения параметров по умолчанию находятся в одном месте. Сеттеры билдера возвращают сам этот билдер. 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 ещё важнее, легко читать. **Шаблон билдера имитирует именованные дополнительные параметры**, имеющиеся в таких языках, как Ada и Python.

Как и конструктор, билдер может принуждать к соблюдению инвариантов своих параметров. Метод `build` позволяет проверить эти инварианты. Очень важно, чтобы они были проверены после копирования параметров из билдера в создаваемый объект и чтобы они были проверены на полях объекта, а не на полях билдера (статья 39). Если хоть один инвариант нарушается, то метод `build` должен выбросить

`IIlegalStateException` (статья 60). Детали ошибки в сообщении будут содержать информацию о том, какие инварианты были нарушены (статья 63).

Другой способ соблюдения инвариантов для большого количества параметров состоит в том, чтобы заставить сеттеры принимать сразу целую группу параметров, для которых должен выполняться инвариант. Если условия инвариантов не соблюдаются, то сеттеры выбрасывают `IllegalArgumentException`. В данном случае преимущество в том, что ошибка обнаруживается сразу, как только переданы неверные параметры, вместо того, чтобы ждать вызова метода `build`.

Другое небольшое преимущество использования билдера вместо конструкторов заключается в том, что у билдера может быть несколько методов с переменным числом параметров. У конструкторов, как и у методов, может быть только один переменный список параметров. Поскольку билдеры используют отдельные методы для установки каждого параметра, они могут иметь сколько угодно методов с переменным числом параметров; таковыми могут быть хоть все сеттеры.

Шаблон билдера гибок. Один билдер может использоваться для создания нескольких объектов. Параметры билдера могут быть изменены для того, чтобы создавать различные объекты. Билдер может автоматически заполнить некоторые поля — например, серийный номер, который автоматически увеличивается каждый раз, когда создаётся объект.

Билдер, параметры которого заданы, может служить отличной реализацией шаблона *“абстрактная фаврика”* (abstract factory) [Gamma95, с. 87]. Другими словами, клиент может передать такой билдер методу, чтобы метод мог создавать один или более объектов для клиента. Чтобы сделать возможным такое использование, вам понадобится тип для представления билдера. Если вы используете релиз 1.5 или более поздний, то будет достаточно одного обобщённого типа (статья 26) для всех билдеров, 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 как реализующий интерфейс `Builder<NutritionFacts>`. Методы, которые работают с

экземпляром билдера, обычно накладывают ограничения на его параметры, используя *ограниченный подстановочный тип* (bounded wildcard type) (статья 28).

Комментарий. Термин “wildcard type” в разных русских источниках переводится по-разному: метасимвольные аргументы, подстановочные символы, групповые символы, шаблоны, маски… Здесь использован термин “подстановочный тип”, как в русском переводе двухтомника К. Хорстманна “Core Java” от издательства “Вильямс”.

Например, вот метод, который строит дерево, используя предоставленный клиентом экземпляр билдера для построения каждого узла.

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

Традиционной реализацией абстрактной фабрики в Java является класс `Class`, чей метод `newInstance` играет роль метода `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 натолкнётся на ошибку `InstantiationException` или `IllegalAccessException` в процессе выполнения, что ужасно неприятно. Также метод `newInstance` проталкивает вверх (propagates) любое исключение, брошенное конструктором без параметров, даже если в тексте метода нет соответствующих выражений `throws`. Другими словами. **`Class.newInstance` ломает проверку исключений на этапе компиляции.** Интерфейс Builder, рассмотренный выше, исправляет данный недостаток.

У шаблона Builder есть свои недостатки. Для создания объекта вам надо сначала создать его билдер. Затраты на создание билдера малозаметны на практике на самом деле, но в некоторых ситуациях, где производительность является важным моментом, это может создать проблемы. Кроме того, шаблон билдера более громоздок в использовании, чем шаблон телескопического конструктора, поэтому использовать его нужно при наличии достаточного количества параметров, например четырёх и более. Но имейте в виду, что в будущем вы можете захотеть добавить параметры. Если вы начнёте использовать конструкторы или статические методы, а затем добавите билдер, когда количество параметров в классе выйдет из-под контроля, то уже ненужные конструкторы или статические методы будут для вас словно заноза в пальце. Поэтому часто желательно начинать именно с билдера.

Подведём итоги. **Шаблон Builder — это хороший выбор при проектировании классов, у чьих конструкторов и статических фабричных методов в противном случае имелось бы много параметров**, особенно если большинство из них не являются обязательными. Клиентский код легче читать и писать при использовании билдеров, чем при использовании традиционных шаблонов телескопических конструкторов. Кроме того, билдеры гораздо безопаснее, чем JavaBeans.

Статья 3. Обеспечивайте уникальность синглтонов с помощью закрытого конструктора или перечислимого типа

Синглтон (singleton) — это просто класс, для которого создаётся ровно один экземпляр [Gamma95, с. 127]. Синглтоны обычно представляют некоторые компоненты системы, которые действительно являются уникальными, например, оконный менеджер или файловая система. **Превращение класса в синглтон может создать сложности в тестировании его клиентов**, так как невозможно заменить синглтон на фиктивную реализацию (mock implementation), 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 `final`-полем:

```
// Синглтон с final-полем
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 защищённых (`protected`) конструкторов гарантирует «вселенную с одним Элвисом»: 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 запустить закрытый конструктор с помощью рефлексии, вызвав перед этим метод `AccessibleObject.setAccessible`. Если вы хотите защиту от такого рода атаки, необходимо изменить конструктор так, чтобы он выбрасывал исключение, если поступит запрос на создание второго экземпляра.

Во втором варианте вместо открытого статического `final`-поля создаётся открытый статический фабричный метод:

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

    public void leaveTheBuilding() { ... }
}
```

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 той же вышеупомянутой тонкости).

Основное преимущество первого подхода (с открытым полем) заключается в том, что из объявления класса понятно, что этот класс является синглтоном: открытое статическое поле имеет модификатор `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 (inline) вызовы статического фабричного метода во втором варианте.

Одно из преимуществ второго подхода (использование статического фабричного метода) заключается в том, что он даёт вам возможность отказаться от решения сделать класс синглтоном, не меняя при этом его API. Статический фабричный метод 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 обобщённых типов, подробно описано в статье 27. Зачастую ни одно из этих преимуществ не имеет значения, и подход с использованием открытого поля оказывается проще.

Чтобы класс синглтона, реализованный с помощью одного из указанных подходов, был *сериализуемым*, недостаточно просто добавить к его объявлению `implements Serializable`. Чтобы гарантировать, что синглтон останется синглтоном, вам необходимо объявить все поля экземпляра как несериализуемые (`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 приведёт ко встречам с ложными Элвисами. Чтобы предотвратить это, добавьте в класс `Elvis` следующий метод `readResolve`:

```
// Метод readResolve для сохранения уникальности синглтона
private Object readResolve() {
    // Возвращает единственного истинного Элвиса и даёт возможность
    // сборщику мусора избавиться от Элвиса-самозванца
    return INSTANCE;
}
```

Начиная с релиза 1.5, имеется также третий подход к реализации синглтонов. Достаточно создать перечислимый тип с одним элементом:

```
// Синглтон-перечисление - более предпочтительный подход
public enum Elvis {
    INSTANCE;

    public void leaveTheBuilding() { ... }
}
```

Данный подход функционально эквивалентен подходу с открытым полем, за исключением того факта, что он более краток, бесплатно предоставляет механизм сериализации и даёт железную гарантию, что не будет создано более одного экземпляра, даже в случае замысловатых атак с использованием сериализации или рефлексии. Хотя этот подход на момент написания книги ещё не нашёл широкого распространения, тем не менее **лучшим решением для реализации синглтона является перечислимый тип с одним элементом**.

Статья 4. Используйте закрытый конструктор для предотвращения создания экземпляров класса

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 злоупотребляют ими, чтобы с помощью объектно-ориентированных языков писать процедурные программы. Тем не менее есть ситуации, когда их применение оправданно. 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, в том числе фабричные (статья 1), для объектов, которые реализуют определённый интерфейс, как это сделано в `java.util.Collections`. Можно также использовать их для группировки методов, работающих с некоторым `final`-классом, вместо того, чтобы заниматься расширением этого класса.

> **Комментарий.** Начиная с Java 8, статические методы, работающие с интерфейсом, можно помещать в сам интерфейс.

Подобные *классы утилит* (utility classes) разрабатываются не для того, чтобы создавать их экземпляры — это не имело бы никакого смысла. Однако, если у класса нет явных конструкторов, компилятор по умолчанию сам создаёт для него *конструктор по умолчанию* (default constructor), не имеющий параметров. Для пользователя этот конструктор ничем не будет отличаться от любого другого. В опубликованных API нередко можно встретить классы, непреднамеренно наделённые способностью порождать экземпляры.

Нельзя запретить создавать экземпляры класса, объявив его абстрактным. Можно создать подкласс такого класса и вызывать уже его конструктор. Более того, это вводит пользователя в заблуждение, заставляя думать, что данный класс был спроектирован именно для наследования (статья 17). Есть, однако, простая идиома, гарантирующая отсутствие экземпляров. Конструктор по умолчанию создаётся 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 объявлен как закрытый (`private`), то за пределами класса он будет недоступен. Выбрасывать `AssertionError` не обязательно, но исключение является подстраховкой на случай, если конструктор будет случайно вызван в самом классе. Это гарантирует, что для класса никогда не будет создано никаких экземпляров. Эта идиома несколько контринтуитивна, поскольку конструктор создаётся 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. Все конструкторы должны в конце концов вызывать конструктор суперкласса, явно или неявно. Здесь же у подкласса не было бы доступа к конструктору, который можно было бы вызвать.

Комментарий. Хотя в книге это не упоминается, в таких случаях желательно также объявить класс как `final`, хотя при единственном закрытом конструкторе это излишне. По умолчанию программа Checkstyle, выявляющая типичные нарушения хорошего стиля программирования в Java-коде, предлагает как добавлять в классы утилит закрытые конструкторы, так и объявлять их как `final`.

Статья 5. Избегайте создания ненужных объектов

Вместо того, чтобы создавать новый функционально эквивалентный объект всякий раз, когда в нем возникает необходимость, часто можно просто переиспользовать тот же объект. Переиспользование объектов может быть и изящнее, и быстрее. Если объект

является *неизменяемым* (immutable), его всегда можно использовать повторно (статья 15).

Рассмотрим следующий код, демонстрирующий, как делать не надо:

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

При каждом выполнении этот код создаёт новый экземпляр `String`, причём совершенно без необходимости. 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, даётся 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 фабричные методы (статья 1), вторые предпочесть первым. Например, статический фабричный метод `Boolean.valueOf(String)` почти всегда предпочтительнее, чем конструктор `Boolean(String)`. Конструктор создаёт новый объект при каждом вызове, тогда как от статического фабричного метода 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 распространённый 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, родился ли человек во время «беби-бума», другими словми, родился ли он между 1946 и 1964 годами:

```
import java.util.Calendar;
import java.util.Date;
```

```
import java.util.TimeZone;

public class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        // Defensive copy - see Item 39
        this.birthDate = new Date(birthDate.getTime());
    }

    // Other fields, methods omitted

    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        // Unnecessary allocation of expensive object
        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 создаёт новые экземпляры `Calendar`, `TimeZone` и два экземпляра `Date`. В следующей версии подобная расточительность пресекается с помощью статического инициализатора:

```
class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
```

```
        // Defensive copy - see Item 39
        this.birthDate = new Date(birthDate.getTime());
    }

    // Other fields, methods

    /**
     * The starting and ending dates of the baby boom.
     */
    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 230 раз быстрее. Причём 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`.

Комментарий. Приведённые выше примеры кода используют устаревшие в Java 8 классы `Date` и `Calendar`. В новых проектах желательно использовать новые классы из пакета `java.time`. Эти классы не только более эффективны, но и являются неизменяемыми, из-за чего пропадает необходимость использовать защитные копии. Вот как будет выглядеть исправленный пример при использовании вместо `Date` и `Calendar` класса `LocalDate`:

```
class Person {
  private final LocalDate birthDate;

  public Person(LocalDate birthDate) {
      this.birthDate = birthDate
  }

  // Other fields, methods

  /**
   * The starting and ending dates of the baby boom.
   */
  private static final LocalDate BOOM_START =
      LocalDate.of(1946, Month.JANUARY, 1);
  private static final LocalDate BOOM_END =
      LocalDate.of(1965, Month.JANUARY, 1);

  public boolean isBabyBoomer() {
      return birthDate.compareTo(BOOM_START) >= 0
              && birthDate.compareTo(BOOM_END) < 0;
  }
}
```

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 ленивую инициализацию (lazy initialization) (статья 71), которая бы выполнялась при первом вызове метода

`isBabyBoomer`, однако делать это не рекомендуется. Как часто бывает в случаях с ленивой инициализацией, это усложнит реализацию и вряд ли приведёт к заметному повышению производительности (статья 55).

Во всех примерах, ранее приведённых в этой статье, было очевидно, что рассматриваемые объекты можно использовать повторно, поскольку они не изменялись после инициализации. Есть, однако, другие ситуации, когда это не столь очевидно. Рассмотрим случай с *адаптерами* (adapters) [Gamma95, с. 139], которые известны также как *представления* (views). Адаптер — это объект, который делегирует операции обёрнутому объекту, 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 в словаре. 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 появился и другой способ создания ненужных объектов. Он называется *автоупаковка* (autoboxing) и позволяет программисту смешивать примитивные типы и их классы-обёртки, выполняя упаковку и распаковку автоматически в случае необходимости. Имеются тонкие семантические различия и совсем не тонкие отличия в производительности (статья 49).

Рассмотрим следующую программу, которая рассчитывает сумму всех положительных значений `int`. Для этого используются арифметические операции над типом `long`, так как тип `int` недостаточно велик, чтобы хранить сумму всех положительных значений `int`.

```
// Ужасно медленная программа! Можете определить, на каком этапе
// создаётся 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 даёт 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 создаёт 2^{31} ненужных экземпляров `Long` (грубо говоря, каждый раз, когда `long` добавляется к сумме `Long`). Изменение объявленного типа переменной `sum` с `Long` на `long` уменьшает время выполнения программы на моей машине с 43 до 6,8 секунд. Урок понятен: **предпочитайте примитивные типы классам-обёрткам и избегайте 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 ресурсоёмкими. Классический пример объекта, для которого оправданно создание пула — это соединение с базой данных (database connection). Затраты на установление такого соединения достаточно высоки, и потому многократное использование такого объекта оправданно. Также лицензионная политика вашей СУБД может накладывать ограничения на одновременное 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 лёгкостью превосходят подобные пулы объектов.

В противовес этой статье можно привести статью 39, посвящённую *защитному*

копированию (defensive copying). Если в этой статье говорится: «Не создавайте новый объект, если лучше будет переиспользовать имеющийся», то статья 39 гласит: «Не переиспользуйте имеющийся объект, если вы *обязаны* создать новый». Заметим, что ущерб от повторного использования объекта в случае, когда требуется защитное копирование, значительно превосходит ущерб от бесполезного создания дублирующего объекта. Отсутствие защитных 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 C или C++, на язык со сборкой мусора, ваша работа как программиста существенно упрощается благодаря тому обстоятельству, что ваши объекты автоматически утилизируются, как только вы перестаёте 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];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    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 версию с обобщёнными типами). Вы можете тщательно ее тестировать, и любой тест она пройдёт 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 по исключению `OutOfMemoryError`, хотя подобные отказы встречаются относительно редко.

Так где же происходит утечка? Если стек растёт, 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 references) на эти объекты. Устаревшая ссылка — это такая ссылка, по которой уже никто никогда не обратится к объекту. 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 даёт и другое преимущество: если впоследствии кто-то по ошибке попытается обратиться по какой-либо из этих ссылок, программа незамедлительно завершится по `NullPointerException` вместо того, чтобы продолжать некорректное выполнение. Всегда выгодно обнаруживать ошибки программирования настолько быстро, насколько это возможно.

Когда программисты впервые сталкиваются с подобной проблемой, они порой 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. **Обнуление ссылок на объекты должно быть не нормой, а исключением.** Лучший способ избавиться от устаревшей ссылки — это выйти из области видимости переменной, содержащей ссылку. Это происходит естественным образом, если для каждой переменной вы задаёте максимально ограниченную область видимости (статья 45).

Так когда же вы должны обнулять ссылку? Какая особенность класса `Stack` сделала его восприимчивым к утечке памяти? Дело в том, что класс `Stack` *управляет своей памятью*.

Пул хранения состоит из массива элементов (причём 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 `elements`, 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 озаботиться предотвращением утечек памяти**. Как только элемент массива освобождается, любые ссылки на объекты, имевшиеся в этом элементе, необходимо обнулять.

Другим распространённым 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 ещё долгое время после того, как она стала недействительной. У этой проблемы возможны несколько решений. Если вам посчастливилось создавать кэш, в котором запись остаётся 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 остаётся значимой, чётко 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 распространённый источник утечек памяти — это обработчики событий (listeners) и прочие обратные вызовы. Если вы реализуете API, в котором клиенты регистрируют обратные вызовы, но не отменяют свою регистрацию явно, то они накапливаются, если вы ничего не предпримете. Лучший способ убедиться, что объекты с методами обратного вызова будут оперативно удаляться сборщиком мусора — это

хранить на них только слабые ссылки, например, сохраняя их лишь в качестве ключей в `WeakHashMap`.

> **Комментарий.** Конкретно этот совет мне кажется сомнительным, и мне неизвестны библиотеки, в которых он бы применялся на практике. Как правило, логически обработчики обратных вызовов перестают использоваться одновременно с объектом, к которому они прикрепляются. Например, обработчики событий от кнопки в графическом интерфейсе перестают использоваться приложением вместе с самой кнопкой, если они не были явно удалены до этого. Сборщик мусора позаботится об удалении как самого объекта, так и обработчиков событий. Проблемы могут возникнуть лишь в том случае, если объект, зарегистрировавший обработчики, сохраняет ссылки на них. На практике этого обычно не происходит, да и необходимости в этом нет.

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

Статья 7. 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, но в общем случае их следует избегать.

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 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 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 **в методе `finalize` нельзя выполнять никаких операций, критичных по времени**. Например, будет серьёзной 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 будут 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 сколь угодно долгую 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 даёт гарантий по поводу того, 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 даёт поручительства, что методы `finalize` будут вызваны быстро, она не даёт 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 полагаться на метод `finalize` для обновления**

критического хранимого (persistent) состояния. Например, ставить освобождение хранимой блокировки разделяемого ресурса, такого как база данных, в зависимость от метода `finalize` — верный способ привести всю вашу распределённую систему к сокрушительному краху.

Не соблазняйтесь методами `System.gc` и `System.runFinalization`. Они могут увеличить вероятность запуска методов `finalize`, но не гарантируют ее. Единственные методы, которые гарантируют срабатывание методов `finalize` — это `System.runFinalizersOnExit` и его злой близнец `Runtime.runFinalizersOnExit`. Эти методы имеют неустранимые дефекты по своей сути и объявлены устаревшими.

В том случае, если вы до сих пор не убедились, что методов `finalize` следует избегать, вот ещё одна пикантная новость, стоящая упоминания: если в ходе утилизации возникает необработанное исключение, оно игнорируется, а вызов метода `finalize` прекращается [JLS, 12.6]. Необработанное исключение может оставить объект в испорченном состоянии. И если другой поток попытается воспользоваться таким испорченным объектом, результат может быть непредсказуем. Обычно необработанное исключение завершает поток и выдаёт распечатку стека, однако в методе `finalize` этого не происходит, он даже не даёт предупреждений.

И ещё 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 закрытия, такие как файлы или потоки? Просто создайте **метод явного закрытия** 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 закрыт. Метод явного закрытия должен делать запись в неком закрытом поле о том, что этот объект более не является действительным. Остальные методы класса должны проверять это поле и выбрасывать `IllegalStateException`, если их вызывают после того, как данный объект был закрыт.

Типичный пример метода явного закрытия — 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. Про- блема касается также метода `Image.flush`, который освобождает все ресурсы, связанные с экземпляром Image, но оставляет при этом последний в таком состоянии, что его ещё можно использовать, выделив вновь необходимые ресурсы.

Методы явного закрытия часто используются в сочетании с конструкцией try-finally, чтобы обеспечить гарантированное закрытие. Вызов метода явного закрытия из оператора finally гарантирует, что он будет выполнен, если даже при работе с объектом возникнет исключение:

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

Комментарий. Начиная с Java 7, желательно реализовывать в классах, требующих явного закрытия, интерфейс `AutoCloseable` или один из его подинтерфейсов, например, `Closeable`. Это позволит применять объекты такого класса в блоке `try` с ресурсами, гарантируя вызов метода `close` при любом завершении блока — как нормальном, так и по исключению.

```
try (Foo foo = new Foo(...)) {
    // Сделать то, что необходимо сделать с foo
}
// foo.close будет автоматически вызван
```

Зачем же тогда вообще нужны методы `finalize`? Есть два приемлемых применения. Первое — выступать в роли «страховочной сетки» (safety net), на тот случай, если владелец объекта забудет вызвать метод явного закрытия, который вы создаёте по совету, данному в предыдущем абзаце. Хотя нет гарантии, что метод `finalize` будет вызван своевременно, лучше всё-таки освободить ресурс поздно, чем никогда, в тех случаях (будем надеяться, редких), когда клиент не вызовет метод явного закрытия.

Но метод `finalize` обязательно должен вывести в лог 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 ресурсоёмкости.

Четыре класса, представленные как пример использования метода явного закрытия (`FileInputStream`, `FileOutputStream`, `Timer` и `Connection`), имеют также и 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 это может поломать существующий клиентский код.

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

Важно отметить, что методы `finalize` не вызывают друг друга (finalizer chaining) автоматически. Если в классе (за исключением `Object`) есть метод `finalize`, но в подклассе он был переопределён, то метод `finalize` в подклассе должен явно 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 если при завершении подкласса выбросится исключение, и наоборот. Вот как это выглядит. Обратите внимание, что в этом примере используется аннотация `@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 одного 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 создаётся единственный экземпляр анонимного класса, называемый *хранителем утилизации* (finalizer guardian). В этом случае экземпляр основного класса содержит в закрытом поля экземпляра единственную ссылку на хранитель утилизации. Таким образом, хранитель утилизации становится доступен для удаления одновременно с контролируемым им объектом. Когда выполняется метод `finalize` хранителя, он выполняет процедуры, необходимые для ликвидации контролируемого им объекта, 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` подкласса вызван метод `super.finalize` или нет. Возможность использования этой методики следует рассмотреть для каждого открытого расширяемого класса, имеющего метод `finalize`.

Подведём итоги. Не надо использовать методы `finalize`, кроме как в качестве страховочной сетки или для освобождения некритических системных 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 логировать 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`.

Глава 3. Методы, общие для всех объектов

Хотя класс `Object` и является конкретным классом, прежде всего он предназначен для расширения. Поскольку все его методы без модификатора `final` — `equals`, `hashCode`, `toString` и `finalize` — предназначены для переопределения, у них есть явные *общие контракты* (general contracts). Любой класс, в котором эти методы переопределяются, обязан соблюдать эти контракты. Если он этого не делает, то это препятствует правильному функционированию других взаимодействующих с ним классов, работа которых зависит от выполнения контрактов, например, `HashSet` и `HashMap`.

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.

Статья 8. Переопределяя метод equals, соблюдайте его общий контракт

Переопределение метода `equals` кажется простой операцией, однако есть множество способов сделать это неправильно, и последствия 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 уникален по существу. Это утверждение справедливо для таких классов, как `Thread`, которые представляют не значения, 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 клиентам такая возможность не понадобится. В таком случае тот вариант метода `equals`, который наследуется от класса `Object`, вполне приемлем.

- Метод `equals` уже переопределён в суперклассе, и функционал, унаследованный от суперкласса, для данного класса вполне приемлем. Например, большинство реализаций интерфейса `Set` наследуют 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. Можно поспорить, что в таком случае метод `equals` *следует* переопределить, на случай, если кто-то вызовет его случайно:

```
@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 идентичностью объектов, а метод `equals` в суперклассе не был переопределён с том, чтобы реализовать требуемый функционал. Обычно это случается с *классами значений* (value classes), 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 классы, использующие контроль экземпляров (статья 1), чтобы убедиться, что любое значение встречается не более чем в одном объекте. Перечислимые типы (статья 30) также попадают под эту категорию. Поскольку для классов этого типа гарантируется, что каждому значению соответствует не больше одного объекта, то метод `equals`,

унаследованный от `Object`, для этих классов подходит для реализации логической эквивалентности.

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

Метод `equals` реализует *отношение эквивалентности* (equivalence relation). Это означает, что он должен обладать следующими свойствами:

- *Рефлексивность*: Для любой ненулевой ссылки `x` выражение `x.equals(x)` должно возвращать `true`.
- *Симметричность*: Для любых ненулевых ссылок `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 `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 `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 `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 нарушите 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. Взывая на краю. — *Прим. пер.*) Экземпляры одного класса часто передаются другому классу. Работа многих классов, в том числе всех классов коллекций, зависит от того, соблюдают ли передаваемые им объекты контракт метода `equals`.

Теперь, когда вы знаете, с каким ущербом связано нарушение контракта метода `equals`, давайте рассмотрим его внимательнее. Хорошая новость заключается в том,

что, вопреки внешнему виду, соглашения не так сложны. Как только вы их поймёте, 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.

Симметричность. 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 final String s;

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

    // Broken - violates symmetry!
    @Override
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
        if (o instanceof String) // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
    }
}
```

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 `CaseInsensitivenessString` 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. Другая реализация с лёгкостью может вернуть `true` или выбросить исключение. **Как только вы нарушили контракт метода `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 только одну инструкцию 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 создаёт подкласс, придающий своему суперклассу новый аспект. Иными словами, подкласс

привносит немного информации, оказывающей влияние на процедуру сравнения. Начнём 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. Х отя 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 o - обычная точка, выполнить сравнение без проверки цвета
    if (!(o instanceof ColorPoint))
        return o.equals(this);
```

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

Такой подход обеспечивает симметрию, но за счёт 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 отношений эквивалентности в объектно-ориентированных языках. **Не существует способа расширить класс, порождающий экземпляры, и добавить к нему компонент значения, сохранив при этом контракт 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
@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 передаём 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 используемый методом `onUnitCircle` класс `HashSet`, использует метод `equals` для проверки содержимого, и ни один экземпляр `CounterPoint` не равен ни одному экземпляру `Point`. Если тем не менее вы корректным образом реализуете в классе `Point` метод `equals` на основе `instanceof`, тот же самый метод `onUnitCircle` будет работать хорошо, если ему предоставить экземпляр `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: «Предпочитайте композицию наследованию». Вместо того, чтобы классом `ColorPoint` расширять класс `Point`, создайте в `ColorPoint` закрытое поле `Point` и открытый метод-адаптер (статья 5), который возвратил бы обычную точку с теми же координатами, что и данная цветная точка:

```
// Добавляет компонент значения, не нарушая контракт 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;
    }

    /**
     * Returns the point-view of this color point.
     */
    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 платформы Java содержатся некоторые классы, которые являются подклассами для класса, создающего экземпляры, и при этом добавляют ему новый компонент значения. 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 ещё 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.

Комментарий. Это ещё одна причина использовать появившийся в Java 8 пакет `java.time` вместо старых классов даты и времени. Класс `Instant`, реализующий абстракцию “момент времени”, заменяет оба старых класса `Date` и `Timestamp` и при этом свободен от недостатков их обоих.

Заметим, что вы можете добавить аспект в подклассе абстрактного класса, не нарушая при этом контракта 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. Четвёртое требование в контракте метода `equals` гласит, что если два объекта равны, они должны быть равны все время, пока один из них (или оба) не будет изменён. 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 ненадёжных ресурсов**. Очень трудно соблюдать требование непротиворечивости, если вы нарушите этот запрет. Например, метод `equals`, принадлежащий `java.net.URL`, полагается на сравнение IP-адресов для хостов, ассоциирующихся со сравниваемыми URL-адресами. 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 контракт `equals`, и такие случаи встречались на практики. (К сожалению, это поведение невозможно изменить в связи с требованиями совместимости.) За очень небольшим исключением, методы `equals` должны выполнять детерминированные расчёты 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 ни один объект не может быть равен `null`. Хотя трудно себе представить, чтобы в ответ на вызов `o.equals(null)` будет случайно возвращено значение `true`, вовсе нетрудно представить случайное выбрасывание исключения `NullPointerException`. Контракт этого не допускает. 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;
    MyType t = (MyType) o;
    ...
}
```

Если бы эта проверка типа отсутствовала, а метод `equals` получил аргумент неправильного типа, то он бы выбросил исключение `ClassCastException`, что нарушает контракт метода `equals`. Однако здесь используется оператор `instanceof`, и если его первый операнд 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 тот же объект.** Если это так, возвращайте `true`. Это всего лишь оптимизация производительности, имеющая смысл, если процедура сравнения может быть трудоёмкой.

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 контракт 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. **Приведите аргумент к правильному типу.** Поскольку эта операция следует за проверкой `instanceof`, она гарантированно будет выполнена успешно.

4. **Пройдитесь по всем «значимым» полям класса и убедитесь в том, что значение этого 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 определён как интерфейс, вы должны получать доступ к значимым полям аргумента, используя методы самого интерфейса. Если же тип аргумента определён как класс, то, в зависимости от видимости полей класса, вам, возможно, удастся получить к ним прямой доступ. Для полей простых типов, 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 исключения `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)))
```

> **Комментарий.** Начиная с Java 7, вместо этой идиомы можно использовать метод `Objects.equals(a, b)`, который делает то же самое.

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*, чтобы метод `equals` мог выполнять дешёвое и точное сравнение этих канонических форм вместо того, чтобы пользоваться более трудоёмким и неточным вариантом сравнения. Описанный приём особенно 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 очерёдность 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 типа `Lock`, 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`. Если значение избыточного поля является какую-то сводкой по объекту в целом, то сравнение подобных полей позволит вам сэкономить на сравнении всех остальных данных, если будет выявлено расхождение. Пусть, например, у вас есть класс `Polygon`, и вы кэшируете его площадь. Если у двух многоугольников различается площадь, вам уже не понадобится сравнивать их вершины и стороны.

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, а напишите тесты для проверки, соблюдаются ли эти условия. Если ответ окажется отрицательным, 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 `null`).

В качестве конкретного примера метода `equals`, который был выстроен по приведённому 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 контракта 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 символические ссылки (symbolic links), которые относятся к одному и тому же файлу. К счастью, он этого и не делает.
- **Объявляя метод `equals`, не указывайте вместо `Object` другой тип.** Нередко программисты пишут метод `equals` следующим образом, а потом часами ломают голову над тем, почему он работает неправильно:

```
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`, если оба метода возвращают один и тот же результат, но нет никакой причины это делать. Хотя при определённых условиях это может дать минимальный выигрыш в производительности, это не оправдывает дополнительного усложнения программы (статья 55).

Последовательное использование аннотации `@Override` позволит вам предотвратить подобную ошибку. Следующий метод `equals` не скомпилируется, и сообщение об ошибке компиляции в точности объяснит, почему:

```
@Override public boolean equals(MyClass o) {
    ...
}
```

Статья 9. Переопределяя метод equals, всегда переопределяйте hashCode

Распространённым источником ошибок является то, что нет переопределения метода `hashCode`. Вы должны переопределять метод `hashCode` в каждом классе, где переопределён метод `equals`. Невыполнение этого условия приведёт к нарушению общего контракта 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`.

Приведём текст контракта, представленного 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, которое нарушается, если вы не переопределили `hashCode`, 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 контракт.

В качестве примера рассмотрим следующий упрощённый 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(prefix, 999, "prefix");
        rangeCheck(lineNumber, 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` переопределённого 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` почти гарантированно выдаст результат `null`, поскольку у `HashMap` есть оптимизация, которая кэширует 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 допустимый, 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 допустим, 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. Программы, которые выполнялись бы за линейное время, будут выполняться за квадратичное время. Для больших хэш-таблиц такое отличие равносильно переходу от работоспособности к неработоспособности.

Хорошая хэш-функция для неравных объектов стремится генерировать различные хэш-коды. И это именно то, что подразумевает третье условия в контракте `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.

Комментарий. Начиная с Java 7, вычисление хэш-кода набора значений значительно упростилось, поэтому приведённый ниже алгоритм можно не запоминать, хотя полезно понимать, как он работает. Метод `Objects.hash(val1, ..., valN)` вычисляет хэш-код именно по этому алгоритму, с той лишь разницей, что вместо начального значения 17 используется 1.

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 расчёт методом `equals`) выполните следующее:

    a. Вычислите для этого поля хэш-код типа 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.

        Комментарий. Эти алгоритмы повторяют реализацию методов `hashCode` в соответствующих типах-обёртках (`Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` и `Double` соответственно). В Java 8 у каждого из них также появился статический метод `hashCode`, возвращающий хэш-код для примитивного значения.

        vi. Если поле является ссылкой на объект, а метод `equals` данного класса сравнивает это поле, рекурсивно вызывая 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`).

> **Комментарий.** Начиная с Java 7, то же самое делает статический метод `Objects.hashCode(obj)`.

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.

> **Комментарий.** Начиная с Java 5, то же самое делает статический метод `Arrays.hashCode(array)`.

b. Объедините хэш-код c, 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, чьё 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 контракте `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 ноль, 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 создаёт зависимость значения хэш-кода от очерёдности обработки полей, а это даёт 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 путём перестановки букв) будут иметь один и тот же хэш-код. Множитель 31 выбран потому, что является простым нечётным числом. Если бы это было чётное 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;
}
```

Комментарий. Соответственно, в Java 7 можно просто написать:

```
@Override public int hashCode() {
    return Objects.hash(areaCode, prefix, lineNumber);
}
```

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 платформы Java. Он прост, достаточно быстр и правильно разносит неравные телефонные номера по разным сегментам хэш-таблицы.

Если класс является неизменяемым и при этом важно сократить 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 ленивую инициализацию, отложенную до первого обращения к методу `hashCode` (статья 71). Хотя достоинства подобной оптимизации для нашего класса `PhoneNumber` не очевидны, давайте покажем, как это делается:

```
// Ленивая инициализация, кэшируемый hashCode
private volatile int hashCode; // (см. [статью 71](#item71))
@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 может равняться с современными оптимизированными хэш-функциям, хотя библиотеки платформы Java версии 1.6 также не предоставляют таких хэш-функций. Разработка оптимизированных хэш-функций является предметом активных исследований, которые лучше оставить математикам и учёным, работающим в области теории вычислительных машин. Возможно, в последней версии платформы Java для библиотечных классов будут представлены оптимизированные хэш-функции, а также методы-утилиты, которые позволят рядовым программистам самим создавать такие хэш-функции. Пока же приёмы, 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

столкнётся с большим количеством экземпляров, которые существенно разнятся как раз в тех частях, которые вы решили игнорировать. Если это произойдёт, 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, проверяла максимум 16 символов, равномерно распределённых по всей строке, начиная с первого символа. Для больших коллекций иерархических имён, таких как URL-адреса, эта хэш-функция демонстрировала как раз то патологическое поведение, о котором здесь говорится.

У многих классов в библиотеках для платформы Java, таких как `String`, `Integer` или `Date`, в спецификации задокументировано конкретное значение, возвращаемое методом `hashCode`, как функция от значения объекта. Вообще говоря, это не слишком хорошая идея, поскольку она серьёзно ограничивает ваши возможности по улучшению хэш-функций в будущих версиях. Если вы оставите детали реализации хэш-функции неконкретизированными и в ней обнаружится изъян, вы сможете исправить эту хэш-функцию в следующей версии, зная, что клиенты не имеют права полагаться на конкретные значения, возвращаемые хэш-функцией.

Статья 10. Всегда переопределяйте метод toString

Хотя в классе `java.lang.Object` и 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"`. Общий контракт 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 контракте метода `toString` говорится: «Рекомендуется во всех подклассах переопределять этот метод». Это действительно хороший совет!

Хотя эти соглашения можно соблюдать не столь строго, как контракты 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 передаётся 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 является приблизительным аналогом функции `sprintf` в языке C.)

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}`"?

Когда это осуществимо на практике, стоит включать в метод `toString` *всю* полезную информацию, которая содержится в объекте, как это было только что показано на примере с телефонными номерами. Однако это непрактично для больших объектов или объектов, состояние которых трудно представить в виде строки. В таких случаях метод `toString` должен возвращать такие сводки, как `"Manhattan white pages (1487536 listings)"` или `"Thread [main, 5, main]"`. В идеале полученная строка не должна требовать разъяснений. (Последний пример с `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 classes), таких как телефонные номера и таблицы. Задав определённый 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 сохраняемых объектах, таких как документы XML. При задании определённого формата, как правило, полезно бывает создать соответствующий конструктор или статический фабричный метод, принимающий строковое представление объекта, что позволит программисту с лёгкостью 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 синтаксически анализирует данное представление, генерирует его и использует его при записи объектов в базу данных (persistent data). Если в будущих версиях вы поменяете формат представления, они взвоют, поскольку вы поломаете созданный ими код и данные. Отказавшись от спецификации формата, вы сохраняете возможность вносить в него новую информацию и совершенствовать этот формат в последующих версиях.

Будете вы документировать формат или нет, вы должны чётко обозначить свои намерения. Если вы документируете формат, вы должны описать его в точности. В качестве примера представим метод `toString`, который должен сопровождать класс `PhoneNumber` из статьи 9:

```
/**
 * Возвращает представление данного телефонного номера в виде строки.
 * Строка 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 трёх частей телефонного номера слишком мала и
 * не заполняет полностью своё 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=любовное, аромат=скипидар, внешний 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, документируете 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 вы этого не сделаете, то вы вынудите программистов, которым нужна эта информация, делать синтаксический разбор строки. Помимо того, что вы снижаете производительность приложения и заставляете программистов делать ненужную работу, это чревато ошибками и приводит к созданию ненадёжной системы, которая перестанет работать, как только вы поменяете формат. Не предоставив альтернативных методов доступа, вы де-факто превращаете формат строки в элемент API, даже если в документации вы указали, что он может быть изменён.

Статья 11. Соблюдайте осторожность при переопределении метода clone

Интерфейс `Cloneable` проектировался в качестве *интерфейса-примеси* {mixin, статья 18), который позволяет объектам объявлять о том, что они могут быть клонированы. К сожалению, он непригоден для этой цели. Его основной недостаток — отсутствие открытого метода `clone`, в то время как в самом классе `Object` метод `clone` является защищённым (`protected`). Вы не можете, не обращаясь к механизму *рефлексии* (reflection, статья 53), вызывать для объекта метод `clone` лишь на том основании, что он реализует интерфейс `Cloneable`. Даже вызов с помощью рефлексии может закончиться неудачей, поскольку нет гарантии, что у данного объекта есть доступный метод `clone`. Несмотря на этот и остальные недочёты, этот механизм используется достаточно 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`, если он не имеет методов? Он определяет поведение защищённого 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 выброшено исключение `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 защищённого 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 является *внеязыковым*: объект создаётся без использования конструктора.

Общий контракт метода `clone` слаб. Он описан в спецификации класса `java.lang.Object` [JavaSE6]:

Метод создаёт и возвращает копию объекта. Точное значение термина «копия» может

зависеть от класса этого объекта. Общая за дача 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. Хотя обычно предполагается, что выражение

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

будет возвращать `true`, это требование тоже 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.

Такой контракт создаёт 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 ненаследуемым (`final`), метод `clone` может просто вернуть объект, созданный конструктором.

Условие, что `x.clone().getClass()` обычно должно быть равно `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 такое поведение в суперклассе можно только одним способом: вернуть объект, полученный в результате вызова метода `super.clone`. Если метод `clone` вернёт объект, созданный конструктором, это будет объект неверного класса. **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 счёте приведёт к вызову метода `clone` из класса `Object` и созданию экземпляра именно того класса, который нужен. Этот механизм отдалённо 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 ожидается, что в классе, реализующем интерфейс `Cloneable`, должен быть правильно работающий открытый**

метод `clone`. В общем случае выполнить это условие невозможно, если все суперклассы этого класса не будут иметь правильную реализацию метода `clone`, открытую или защищённую.

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 защищённому 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 так можно и нужно делать, потому что *ковариантные возвращаемые типы* появились в версии 1.5 как часть механизма обобщённых типов. Другими словами, теперь допустимо, чтобы типы переопределяющего метода были подклассом типов переопределённого метода. Это позволяет переопределяющему методу давать больше информации о возвращаемом объекте и снимает с клиента необходимость использовать приведение типов. Поскольку `Object.clone` возвращает `Object`, то `PhoneNumber.clone` должен привести тип результата `super.clone()` к `PhoneNumber` до того, как вернуть его, но это гораздо предпочтительнее, чем заставлять каждый вызов клона `PhoneNumber.clone` выполнять приведение типа. Общий принцип здесь: **никогда не заставляйте клиента делать 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 implements Cloneable {
    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; // Eliminate obsolete reference
        return result;
    }

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

    // Ensure space for at least one more element.
    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 вернёт 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 даёт бессмысленные результаты либо выбрасывает `NullPointerException`.

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

```
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

Обратите внимание, что нам нет необходимости приводить результат `elements.clone()` к типу `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 сработало бы, если поле `elements` имело модификатор `final`, поскольку тогда методу `clone` было бы запрещено помещать туда новое значение. Это фундаментальная проблема: **архитектура клонирования несовместима с обычным использованием полей `final`, содержащих ссылки на изменяемые объекты**. Исключение составляют случаи, когда эти изменяемые объекты могут безопасно использовать одновременно 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 упрощённый 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 AssertionError();
        }
    }
```

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

```
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 этой 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 изменён 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 массив `buckets`, выполняя глубокое копирование каждого непустого сегмента. Чтобы скопировать связный список, начинающийся с указанной записи, метод глубокого копирования (`deepCopy`) из класса `Entry` рекурсивно вызывает сам себя. Хотя этот приём 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;
}
```

Последний 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` получило бы при инициализации новый массив сегментов, а затем для каждой пары ключ/значение в клонируемой хэш-таблице был бы вызван 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 переопределённый метод, то этот метод будет выполняться до того, как подкласс, в котором он был определён, 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 непереопределяемым (`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` объявлен как способный выбросить `CloneNotSupportedException`, однако в переопределённых методах `clone` это объявление может быть опущено. Открытые методы `clone` *не должны* быть объявлены как `throws CloneNotSupportedException`, поскольку с методами, не выбрасывающими проверяемых исключений, проще работать (статья 59). Если же метод `clone` переопределяется в расширяемом классе, а особенно в классе, предназначенном для наследования (статья 17), новый метод `clone` должен подражает поведению `Object.clone`: он также должен объявлен как закрытый (protected) и способный выбрасывать `CloneNotSupportedException`, и класс не должен реализовывать интерфейс `Cloneable`. Это даёт подклассу свободу выбора, реализовывать `Cloneable` или нет.

Ещё одна деталь заслуживает внимания. Если вы решите сделать клонируемым потокобезопасный (thread-safe) класс, помните, что метод `clone` нужно должным образом синхронизировать, как и любой другой метод (статья 66). Метод `Object.clone` не синхронизирован, и, хотя это в принципе нормально, вам, возможно, потребуется написать синхронизированный метод `clone`, вызывающий `super.clone()`.

Подведём 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 изменении. Из этого правила есть исключения. Например, поле, предоставляющее серийный номер или иной уникальный идентификатор, а также поле, показывающее время создания объекта, нуждаются в изменении, даже если они имеют простой тип или являются неизменяемыми.

Так ли нужны все эти сложности? Далеко не всегда. Если вы расширяете класс, реализующий интерфейс `Cloneable`, у вас практически не остаётся иного выбора, кроме как реализовать правильно работающий метод `clone`. В противном случае **вам, по-видимому, лучше реализовать другой механизм копирования объектов либо отказаться от 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 копии* (copy constructor) или *фабрики копий* (copy factory). Конструктор копии — 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 фабричный метод:

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

Использование конструктора копии (или, как его вариант, статического фабричного метода) имеет много преимуществ перед механизмом `Cloneable`/`clone`: оно не связано с рискованным внеязыковым 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 определённого типа.

Хотя конструктор копии или статический фабричный метод копирования невозможно поместить в интерфейс, `Cloneable` всё равно не может выполнять функции такого интерфейса, поскольку не имеет открытого метода `clone`. Поэтому нельзя утверждать, что, когда вместо метода `clone` вы используете конструктор копии, вы отказываетесь от возможностей интерфейса.

Более того, конструктор копии (или статический фабричный метод) может иметь аргумент, тип которого соответствует интерфейсу, реализуемому этим классом. Например, все реализации коллекций общего назначения, по соглашению, имеют конструктор копии с аргументом типа `Collection` или `Map`. Конструкторы копии и статические фабричные методы, 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 копии: `new TreeSet(s)`.

После рассмотрения всех проблем, связанных с интерфейсом `Cloneable`, можно с уверенностью сказать, что от него не стоит наследовать другие интерфейсы, а классы, которые предназначены для наследования (статья 17), не должны его реализовывать. Из-за множества недостатков этого интерфейса некоторые опытные программисты предпочитают вообще никогда не переопределять метод `clone` и никогда им не пользоваться, за исключением разве что копирования массивов. Учтите, что, если в классе, спроектированном для наследования, вы не создадите по меньшей мере правильно работающий защищённый метод `clone`, то реализация интерфейса `Cloneable` в подклассах станет невозможной.

Статья 12. Подумайте о том, чтобы реализовать интерфейс Comparable

В отличие от других обсуждавшихся в этой главе методов, метод `compareTo` в классе `Object` не объявлен. Вместо этого он является единственным методом интерфейса `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 порядком* (natural ordering). Сортировка массива объектов, реализующих интерфейс `Comparable`, выполняется просто:

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

Для объектов `Comparable` так же просто выполнять поиск, вычислять экстремальные значения и обеспечивать поддержку 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<String> s = new TreeSet<String>();
        Collections.addAll(s, 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 о том, не реализовать ли этот интерфейс:

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

Общий контракт для метода `compareTo` имеет тот же характер, что и контракт для метода equals:

Выполняет сравнение текущего и указанного объекта и определяет их очерёдность. Возвращает отрицательное целое число, ноль 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, выбрасывается исключение `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 `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` для всех `x` и `y`. (Это подразумевает, что выражение `x.compareTo(y)` должно выбрасывать исключение тогда и только тогда, когда это делает `y.compareTo(x)`.)
- Разработчик должен также гарантировать транзитивность отношения: `(x.compareTo(y) > 0 && y.compareTo(z) > 0)` подразумевает `x.compareTo(z) > 0`.
- Наконец, разработчик должен гарантировать, что из тождества `x.compareTo(y) == 0` вытекает тождество `sgn(x.compareTo(z)) == sgn(y.compareTo(z))` для всех `z`.
- Настоятельно (хотя и не безусловно) рекомендуется выполнять условие `(x.compareTo(y) == 0) == (x.equals(y))`. Вообще говоря, для любого класса, который реализует интерфейс `Comparable`, но нарушает это условие, этот факт должен быть чётко оговорён. Рекомендуется использовать следующую формулировку: «Примечание: данный класс имеет естественный порядок, не согласующееся с `equals`».

Пускай математическая природа этого контракта у вас не вызывает отвращения. Как и контракт метода `equals` (статья 8), контракт `compareTo` не так сложен, как это кажется. Для одного класса любое разумное отношение упорядочения будет соответствовать контракту `compareTo`. Для сравнения разных классов метод `compareTo`, в отличие от метода `equals`, использоваться не должен: если сравниваются две ссылки на объекты различных классов, можно выбросить исключение `ClassCastException`. В подобных случаях метод `compareTo` обычно так и 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 контракта `compareTo`. Первое условие гласит, что, если вы измените порядок сравнения для двух ссылок на объекты, произойдёт 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 даёт равенство, при сравнении с любым третьим объектом должны показывать одинаковый результат.

Из этих трёх условий следует, что проверка равенства, осуществляемая с помощью метода `compareTo`, должна подчиняться тем же самым ограничениям, которые продиктованы контрактом метода `equals`: рефлексивность, симметричность, транзитивность и отличие от `null`. Следовательно, здесь можно дать то же самое предупреждение: невозможно расширить порождающий экземпляры класс, вводя новый компонент значения и не нарушая при этом контракт метода `compareTo` (статья 8). Обходится это ограничение так же. Если вы хотите добавить важное свойство к классу, реализующему интерфейс `Comparable`, не расширяйте его, а напишите новый независимый класс, в котором для исходного класса будет выделено отдельное поле. Затем добавьте метод-адаптер, возвращающий значение этого поля. Это даст вам возможность реализовать во втором классе любой метод `compareTo`, который вам нравится. При этом клиент при необходимости сможет 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`, является *согласованным с проверкой*

равенства (consistent with equals). Если же оно нарушается, то упорядочение называется *несогласованным 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 контрактам соответствующих интерфейсов коллекций (`Collection`, `Set` или `Map`). Дело в том, что общие контракты для этих интерфейсов определяются в терминах метода `equals`, тогда как в отсортированных коллекциях используется проверка равенства, которая реализуется методом `compareTo`, а не `equals`. Если это произойдёт, катастрофы не случится, но это следует иметь в виду.

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. Поскольку тип `Comparable` параметризован, нет необходимости делать проверку или приведение типа. Если аргумент имеет значение `null`, метод `compareTo` должен выбросить `NullPointerException`, что произойдёт сразу же, как только вы 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 путём 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 свой 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`. Именно так нужно поступать, объявляя классы реализующими интерфейс `Comparable`. Обратите вниманиие, что параметром метода `compareTo` является `CaseInsensitiveString`, а не `Object`. Это требуется при параметризованном объявлении класса.

Поля простого типа нужно сравнивать с помощью операторов `<` и `>`. Для сравнения полей с плавающей точкой используйте `Double.compare` или `Float.compare` вместо операторов сравнения, которые не удовлетворяют контракту 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 даёт 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 ноль. Такой приём демонстрирует метод `compareTo` для класса PhoneNumber из статьи 9:

```
public int compareTo(PhoneNumber pn) {
    // Сравниваем коды городов
    if (areaCode < pn.areaCode)
        return -1;
    if (areaCode > pn.areaCode)
        return 1;
```

```
    // Коды городов равны, сравниваем номера АТС
    if (prefix < pn.prefix)
        return -1;
    if (prefix > pn.prefix)
        return 1;
    // Коды городов и номера АТС равны, сравниваем номера линий
    if (lineNumber < pn.lineNumber)
        return -1;
    if (lineNumber > pn.lineNumber)
        return 1;
    return 0; // Все поля равны
}
```

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

```
public int compareTo(PhoneNumber pn) {
    // Сравниваем коды городов
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;
    // Коды городов равны, сравниваем номера АТС
    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 ещё чаще, разность между наименьшим и наибольшим возможными значениями поля меньше или равна значению

`Integer.MAX_VALUE` (2^{31} - 1). Причина, по которой этот приём 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)` произойдёт переполнение и будет возвращено отрицательное значение. Соответственно, полученный нами метод `compareTo` работать не будет: для некоторых аргументов будет возвращён 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.

Комментарий. В Java 8 реализация подобных *цепей сравнения* (comparison chains) упростилась благодаря новым статическим методам, появившимся в тесно связанном с `Comparable` интерфейсе `Comparator` именно для этой цели. Семейство методов `Comparator.comparing` возвращает компараторы, сравнивающие по указанному *извлекателю ключа* (key extractor), а семейство методов `thenComparing` соединяет два компаратора в цепь, применяющую второй компаратор вслед за первым, если первый посчитает сравниваемые значения равными. С помощью этих методов можно реализовать метод `PhoneNumber.compareTo` таким образом:

```
public class PhoneNumber implements Comparable<PhoneNumber> {
    private static final Comparator<PhoneNumber> COMPARATOR =
        Comparator.comparingInt(pn -> pn.areaCode)
            .thenComparingInt(pn -> pn.prefix)
            .thenComparingInt(pn -> pn.lineNumber);

    <...>

    @Override
    public int compareTo(PhoneNumber pn) {
        return COMPARATOR.compare(this, pn);
    }
}
```

Глава 4. 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, надёжными и гибкими.

Статья 13. Сводите к минимуму доступность классов и их 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, чётко 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* (decouple) 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 создаёт 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 них — 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 таковые есть, присутствуют в ее объявлении (`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 ещё 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 открытым, в противном случае он будет доступен только в пределах пакета. Если класс или интерфейс верхнего уровня можно сделать доступным только в пакете, он должен 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, рассмотрите возможность превращения его в закрытый класс (или интерфейс), который будет вложен именно в тот класс, где он используется (статья 22). Тем самым вы ещё 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.
- **защищённый** (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, ещё 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 защищённый (`protected`), уровень доступности этого члена резко возрастает. Защищённый член класса является частью его внешнего API, а потому ему навсегда должна быть обеспечена поддержка. Более того, наличие защищённого члена в классе, передаваемом за пределы пакета, представляет собой открытую передачу деталей реализации (статья 17). Потребность в использовании защищённых 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 суперклассе [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].

Для тестирования вам может захотеться сделать класс, интерфейс или член более доступными. В определённых пределах это допустимо. Допустимо сделать закрытый член открытого класса доступным только в пределах пакета ради его тестирования, но недопустимо увеличивать его доступность сверх того. Другими словами, недопустимо делать класс, интерфейс или член частью внешнего API для целей тестирования. К счастью, это и не нужно, поскольку тесты можно помещать в тот же пакет, классы которого они тестируют, таким образом давая им доступ к элементам программы, доступным только в пределах пакета.

> **Комментарий.** В библиотеке Guava есть аннотация `@VisibleForTesting`, которой можно помечать члены класса, которым предоставлен доступ в пределах пакета для целей тестирования, и которые в противном случае были бы закрытыми.

Поля экземпляра никогда не должны быть открытыми (статья 14). Если поле не имеет модификатора `final` или имеет этот 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 не являются потокобезопасными**. 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 договорённости, названия таких полей состоят из прописных букв, слова в названии разделены символом подчёркивания (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 изменён — с нежелательными последствиями.

Заметим, что массив ненулевой длины всегда является изменяемым. **Поэтому практически никогда нельзя объявлять поле массива как `public static final`; недопустимо также создавать метод доступа, возвращающий ссылку на внутренний массив.** 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, и тем самым 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));
```

Вы также можете заменить открытое поле массива открытым методом, который возвращает копию закрытого массива.

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

Чтобы выбрать между этими двумя альтернативами, подумайте о том, что клиент вероятнее всего будет делать с результатом. Какой тип возвращаемого значения более

удобен? Какой вариант более производителен?

Подведём итоги. Всегда следует снижать уровень доступа, насколько это возможно. Тщательно разработав наименьший открытый API, вы должны не дать возможность каким-либо случайным 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.

Статья 14. 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 обеспечивать никакие инварианты. 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* (accessor methods, getters) и, для изменяемых классов, *мутаторами* (mutators, setters):

```
// 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, создайте методы доступа**, оставляя возможность менять внутреннее представление этого класса. Если открытый класс показал клиенту свои поля данных, то всякая возможность менять это представление будет потеряна, поскольку программный код клиентов открытого класса может использоваться 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 создаёт 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 ещё 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 следовать примеру этих классов, их следует рассматривать как предостережение. В статье 55 рассказывается, как решение раскрыть внутреннее содержание класса

`Dimension` привело к серьёзным проблемам с производительностью, которые нельзя было разрешить, не затрагивая клиентов.

Хотя раскрывать поля открытого класса напрямую — плохая идея, это все же менее опасно, если поля являются неизменяемыми. Невозможно изменить представление такого класса, не поменяв его API, и невозможно выполнять никакие вспомогательные действия во время чтения поля, но по крайней мере можно обеспечивать инварианты. В следующем примере класс будет 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.
}
```

Подведём итоги. Открытые классы никогда не должны раскрывать неизменяемые поля. Когда открытые классы раскрывают неизменяемые поля, это не столь опасно, но тем не менее спорно. Иногда, однако, имеет смысл раскрывать поля классам, открытым в рамках пакета, или закрытым вложенным классам, вне зависимости от их изменяемости.

Статья 15. Предпочитайте неизменяемые классы

Неизменяемый класс — это просто такой класс, экземпляры которого нельзя изменять. Вся информация, которая содержится в любом его экземпляре, записывается в момент его создания и остаётся 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 надёжны. 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 неизменяемости данного класса в небрежном или умышленно плохо написанном подклассе. Защита методов от переопределения обычно осуществляется путём объявления класса в качестве окончательного, однако есть и другие способы (см. ниже).
3. Объявите все поля как `final`. Это ясно выразит ваши намерения, причём в некоторой степени их будет поддерживать сама система. Это также нужно для того, чтобы обеспечить правильное поведение программы в том случае, когда ссылка на вновь созданный экземпляр передаётся от одного потока в другой без выполнения синхронизации, известная как *модель памяти* (memory model) [JLS, 17.3; 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 объекта ко всем его 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. В конструкторах, методах доступа к полям и методах `readObject` (статья 76) создавайте *защитные 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;
    }

    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);

    // Accessors with no corresponding mutators
    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;

        // В [статье 8](#item8) объясняется, почему мы используем
        // compare, а не ==
        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 int hashDouble(double val) {
        long longBits = Double.doubleToLongBits(re);
        return (int) (longBits ^ (longBits >>> 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 распространённый *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 позволяет объектам быть неизменяемыми, 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, надёжное использование изменяемого класса может оказаться сложной или даже невыполнимой задачей.

Неизменяемые объекты по своей сути потокобезопасны (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 на неизменяемый объект со стороны другого потока. 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 приёмов, 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 ещё один шаг в этом направлении. В неизменяемом классе можно предусмотреть статические фабричные методы (статья 1), которые кэшируют часто запрашиваемые экземпляры вместо того, чтобы при каждом запросе создавать новые экземпляры, дублирующие уже имеющиеся. Так делают все обёртки примитивных типов, а также класс `BigInteger`. Использование таких статических фабричных методов заставляет клиентов переиспользовать уже имеющиеся экземпляры вместо того, чтобы создавать новые. Это снижает расход памяти и сокращает работу по освобождению памяти. Выбор статических фабричных методов вместо открытых конструкторов даёт 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 *защитные 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 копии* (copy constructor,

статья 11). Когда платформа Java только появилась, ещё не было чёткого понимания этого обстоятельства, и потому класс `String` сейчас имеет конструктор копии, 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` создаёт 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 хорошие 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». Частным случаем данного принципа является то, что неизменяемые объекты хорошо подходят для использования в качестве ключей словаря (map) и элементов множества (set). При этом вас не должно беспокоить то, что значения, однажды записанные в этот словарь или множество, вдруг поменяются, и это приведёт 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 обратить его младший бит:

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

Метод `flipBit` создаёт 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 за константное время.

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 элементарных. Если многошаговая операция реализована как элементарная (primitive), неизменяемый класс уже не обязан на каждом шаге создавать отдельный объект. Изнутри 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 тяжёлую работу. Описанный приём 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 `StringBuilder` (и практически устаревший `StringBuffer`). Можно сказать, что в некоторых ситуациях класс `BitSet` играет роль изменяемого компаньона для `BigInteger`.

Теперь, когда вы знаете, как создавать неизменяемый класс и каковы 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 объявляется как `final`, но есть и другой, более гибкий вариант добиться этого. Альтернатива объявлению класса как `final` заключается в том, чтобы сделать все его конструкторы закрытыми либо доступными только в пакете и вместо открытых конструкторов использовать открытые статические фабричные методы (статья 1). Для пояснения представим, как бы выглядел класс `Complex`, если бы применялся такой подход:

```
// Неизменяемый класс со статическими фабричными методами
// 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 трёх 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 `final`-классом (effectively final), поскольку они не могут расширить класс, взятый из другого пакета, у которого нет ни открытого, ни защищённого конструктора. Помимо того, что такой подход даёт возможность гибко использовать несколько классов реализации, он даёт возможность повысить производительность класса в последующих версиях путём совершенствования механизма кэширования объектов в статических фабричных методах.

Как показано в статье 1, статические фабричные методы 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(double, double)`. Со статическими фабричными методами все

проще: достаточно просто добавить второй статический фабричный метод с таким названием, которое чётко 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 понимания того, что неизменяемые классы должны быть фактически ненаследуемыми. Поэтому любой метод этих классов можно переопределить. К сожалению, исправить что-либо впоследствии уже было нельзя, не потеряв при этом обратную совместимость версий. Поэтому, если вы пишете класс, безопасность которого зависит от неизменяемости аргумента с типом `BigInteger` или `BigDecimal`, полученного от ненадёжного клиента, вы должны выполнить проверку и убедиться в том, что этот аргумент действительно является «настоящим» классом `BigInteger` или `BigDecimal`, а не экземпляром какого-либо ненадёжного подкласса. Если имеет место последнее, вам необходимо создать защитную копию этого экземпляра, поскольку придётся 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 объявлены как `final`. Эти правила несколько строже, чем это необходимо, и их можно ослабить с целью повышения производительности программы. Действительно, ни один метод не может произвести такие изменение 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 трудоёмких вычислений. Если в дальнейшем потребуется произвести те же самые вычисления, то будет возвращено ранее сохранённое значение, ненужные вычисления выполняться не будут. Такая уловка работает надёжно 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 *ленивой 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 не-совсем-неизменяемого класса. Эта тема детально раскрывается в статье 76.

Подведём итоги. Не стоит для каждого метода `get` писать метод `set`. **Классы следует оставлять неизменяемыми, если нет уж совсем веской причины сделать их изменяемыми.** Неизменяемые классы имеют массу преимуществ, единственный же их недостаток — возможные проблемы с производительностью при определённых условиях. Небольшие объекты значений, такие как `PhoneNumber` или `Complex`, всегда следует делать неизменяемыми. (В библиотеках платформы Java есть несколько классов, например, `java.util.Date` и `java.awt.Point`, которые должны были бы стать неизменяемыми, но таковыми не являются.) Следует также серьёзно подумать, не сделать ли неизменяемыми более крупные объекты значений, такие как `String` или `BigInteger`. Создавать для вашего неизменяемого класса открытый изменяемый класс-компаньон следует *только* тогда, когда вы убедитесь в том, что это необходимо для получения приемлемой производительности (статья 55).

Есть классы, для которых обеспечивать неизменяемость непрактично. **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. Поэтому **объявляйте все поля как `final`, если только у вас нет веской причины сделать поле изменяемым**.

Конструктор такого класса должен создавать полностью инициализированный объект,

у которого все инварианты уже установлены. 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 даёт (если вообще даёт) лишь небольшой выигрыш в производительности за счёт увеличения сложности приложения.

Перечисленные правила иллюстрирует класс `TimerTask`. Он является изменяемым, однако пространство его состояний намеренно оставлено небольшим. Вы создаёте экземпляр, задаёте порядок его выполнения и, возможно, отменяете это решение. Как только задача, контролируемая таймером, была запушена 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, которые не округляют результаты должным образом и плохо справляются со случаями комплексных значений `NaN` и бесконечности [Kahan91, Smith62, Thomas94],

Статья 16. Предпочитайте композицию наследованию

Наследование (inheritance) — это мощный способ добиться многократного использования кода, но не всегда лучший инструмент для работы. При неправильном применении наследование приводит к появлению ненадёжных 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 обыкновенных не абстрактных классов за пределами пакета сопряжено с риском. Напомним, что в этой книге слово «наследование» (inheritance) используется для обозначения *наследования 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 настройки производительности программы 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 счётчик количества попыток добавления элемента и предоставляет метод доступа к этому счётчику. 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> {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
```

```
    @Override public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

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<String>();
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 влечёт вызов метода `add`, переопределённого в классе `InstrumentedHashSet` — по одному разу для каждого элемента. Каждый из этих трёх вызовов добавлял к значению `addCount` ещё 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`. Хотя 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 и не поменяется при переходе от одной версии к другой. Соответственно, полученный класс `InstrumentedHashSet` может быть ненадёжен. 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 приём не решает всех наших проблем. Он подразумевает повторную реализацию методов суперкласса, которые могут приводить, а могут не приводить к использованию классом самого себя. Этот вариант сложен, трудоёмок 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.

Ещё одна причина ненадёжности подклассов связана с тем, что в новых версиях суперкласс может обзавестись новыми методами. Предположим, безопасность программы зависит от того, чтобы все элементы, помещённые в некую коллекцию, соответствовали некоему 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 произойдёт, 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 вы 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 вы переопределите последний и опять столкнётесь 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 ещё не были сформулированы.

К счастью, есть способ устранить 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 *переадресацией* (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(); }
    @Override public String toString(){ return s.toString(); }
}
```

Комментарий. Похожий класс `ForwardingSet` реализован в библиотеке Guava, которая также предоставляет классы `ForwardingCollection`, `ForwardingList`, `ForwardingQueue` и `ForwardingMap`.

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 отдельный 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 шаблон Decorator (декоратор) [Gamma95, с. 175], поскольку 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 передаёт себя «обёрнутому» 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 передаёт другому объекту ссылку на самого себя для последующего вызова (callback — «обратный вызов»). Поскольку «обёрнутый» объект не знает о своей оболочке, он передаёт 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-оболочек 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 класс на каждый интерфейс; кроме того, классы переадресации могут быть уже предоставлены пакетом, содержащим интерфейс.

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

В библиотеках для платформы Java имеется множество очевидных нарушений этого принципа. Например, стек не является вектором, соответственно, класс `Stack` не должен быть расширением класса `Vector` (синхронизированного предшественника `ArrayList`). Точно так же список свойств не является хэш-таблицей, а потому класс `Properties` 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 серьёзно то, что, раскрывая внутренние элементы класса, вы позволяете клиенту обращаться к ним напрямую. Это как минимум может привести к запутанной семантике. Например, если `p` ссылается на экземпляр класса `Properties`, то `p.getProperty(key)` может давать совсем другие результаты, чем `p.get(key)`: первый метод учитывает значения по умолчанию, тогда как второй метод, унаследованный от класса `Hashtable`, этого не делает. И самое серьёзное: 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 распространятся на API вашего класса? Наследование копирует любые дефекты в API суперкласса, тогда как композиция позволяет вам разработать новый API, который эти недостатки скрывает.

Подведём итоги. Наследование является мощным инструментом, но оно же создаёт 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 ненадёжной, особенно если подкласс и суперкласс принадлежат к разным пакетам, а сам суперкласс не предназначен для расширения. Для устранения этой ненадёжности вместо наследования используйте композицию и переадресацию, особенно когда для реализации класса-оболочки есть подходящий интерфейс. Классы-оболочки не только надёжнее подклассов, но и более мощны.

Статья 17. Проектируйте и документируйте классы для наследования 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 чётко 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 защищённого 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 *переопределяемыми* (overridable) методами здесь мы подразумеваем методы, которые не объявлены как `final` и являются либо открытыми (`public`), либо защищёнными (`protected`).) 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 вызывает переопределяемые методы, должен содержать описание этих обращений в конце своего Javadoc-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. Приведём 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 выбросит исключение `UnsupportedOperationException`.

Комментарий. В Java 8 для спецификации реализации метода по умолчанию появился новый тег Javadoc `@implSpec`, который форматирует соответствующий раздел комментария под заголовком “Implementation Requirements”.

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`, возвращённого методом `iterator`, будет влиять на работу метода `remove`. Сравните это с ситуацией, рассматриваемой в статье 16, когда программист, создающий 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 настройки своей собственной внутренней деятельности в виде правильно выбранных защищённых методов** или, в редких случаях, защищённых 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 `(toIndex - 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 (`subList`). 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 удалён указанный диапазон. **Примечание: если `ListIterator.remove` выполняется за линейное время, то время работы данной реализации будет квадратичным.**

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 лёгких!

Как же решить, которые из защищённых 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 защищённых методов и полей, поскольку каждый из них связан с деталями реализации. С другой стороны, их количество не должно быть слишком малым, поскольку отсутствие защищённого метода может сделать класс практически негодным для наследования.

***Единственный* способ протестировать класс, спроектированный для наследования — это написать подклассы.** Если при написании подкласса вы пренебрежёте ключевыми защищёнными членами, то данное пренебрежение негативно проявит себя. И наоборот — если вы напишете несколько подклассов и им не понадобится какой-то защищённый член, то, вероятнее всего, лучше сделать его закрытым. Один или несколько таких подклассов должны быть написаны кем-либо со стороны, а не автором суперкласса.

Готовя к наследованию класс, который, по-видимому, получит широкое распространение, учтите, что вы *навсегда* задаёте схему использования классом самого себя, а также реализацию, неявно представленную этими защищёнными 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 путём 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. На момент написания этого текста практически не было инструментов или соглашений о комментировании, способных отделить документацию обычного API от той информации, которая представляет интерес лишь для тех программистов, которые создают подклассы.

> **Комментарий.** Как уже было сказано выше, тег `@implSpec` в версии Javadoc для Java 8 (а также тег `@implNote` с неформальными замечаниями о реализации) позволяет визуально отделить документацию реализации от документации API.

Есть ещё 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 неправильной работе программы. Конструктор суперкласса выполняется прежде конструктора подкласса, а потому переопределяющий метод в подклассе будет вызываться перед запуском конструктора этого подкласса. И если переопределённый метод зависит от инициализации, которую осуществляет конструктор подкласса, то этот метод будет работать совсем не так, как ожидалось. Для ясности приведём 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 `overrideMe`, неправомерно вызываемый единственным конструктором класса `Super`:

```
public final class Sub extends Super {
    private final Date date; // Blank final, set by constructor
```

```
    Sub() {
        date = new Date();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        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 `overrideMe` вызван конструктором `Super` прежде, чем конструктор `Sub` получает возможность инициализировать поле `date`. Отметим, что данная программа видит поле `final` в двух разных состояниях. Заметим, что если бы `overrideMe` вызвал любой метод у ссылки `date`, то при запуске выбросилось бы исключение `NullPointerException` в момент, когда конструктор `Super` вызвал бы `overrideMe`. Единственная причина, почему программа не выводит сообщение об ошибке, как это должно быть, заключается в том, что метод `println` может работать с аргументом `null`.

Реализация интерфейсов `Cloneable` и `Serializable` при проектировании наследования создаёт 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 приёмы, которые можно использовать с тем, чтобы дать возможность подклассам реализовать эти интерфейсы без реализации их в суперклассе. Эти приёмы 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` переопределённый метод будет выполняться перед десериализацией состояния подкласса. Что касается метода `clone`, то переопределённый 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. Это может произойти, например, в случае, если переопределяемый метод предполагает, что изменяет копию глубокого состояния объекта в клоне, но это состояние ещё не было скопировано.

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 защищёнными. Если эти методы будут закрытыми, то подклассы будут молча их игнорировать. Это ещё 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 лёгкостью. В ряде ситуаций это необходимо делать, например, когда речь идёт об абстрактных классах, включая *скелетные реализации* интерфейсов (skeletal implementations, 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 объявляются как `final`, 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 наследуемом неабстрактном классе, не спроектированном 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 `final`. Другой подход заключается в том, чтобы сделать все конструкторы класса закрытыми или доступными лишь в пределах пакета, а вместо них создать открытые статические фабричные методы. 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, создаёт превосходную альтернативу наследованию, используемому всего лишь для изменения функциональности.

Если же 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 механически исключить использование классом собственных переопределяемых методов, оставив его поведение без изменений. Переместите тело каждого переопределяемого метода в закрытый вспомогательный метод (helper method), а затем поместите в каждый переопределяемый метод вызов соответствующего закрытого вспомогательного метода. Наконец, каждый вызов переопределяемого метода в классе замените прямым вызовом соответствующего ему закрытого 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, определённого 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 серьёзно 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 ещё нет, и добавить интерфейс в раздел `implements` объявления класса. 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 придётся 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 примесей (mixins). Помимо своего «первоначального типа» класс может реализовать некий дополнительный тип, объявив о том, что в этом классе реализован дополнительный функционал. Например, `Comparable` является дополнительным интерфейсом, который даёт классу возможность декларировать, что его экземпляры упорядочены по отношению к другим, сравнимым с ними объектам. Такой интерфейс называется *примесью* (mixin), поскольку позволяет к первоначальным функциям некоего типа примешивать (mix 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 также являются авторами песен. Поскольку для определения этих типов мы использовали не абстрактные классы, а интерфейсы, то одному классу никак не запрещается реализовывать оба интерфейса `Singer` и `Songwriter`. Более того, 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 отдельный класс для каждой поддерживаемой ею комбинации атрибутов. Если в системе имеется N атрибутов, то существует 2^N возможных сочетаний, которые, возможно, придётся 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* (wrapper class), описанную 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 надёжными, 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 со скелетной реализацией (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`. Возможно, стоило назвать их `SkeletalCollection`, `SkeletalSet`, `SkeletalList` и `SkeletalMap`, но префикс `Abstract` уже прочно укоренился.

При правильном проектировании скелетная реализация позволяет программистам без труда создавать свои собственные реализации ваших интерфейсов. В качестве примера приведём статический фабричный метод, содержащий завершённую, полнофункциональную реализацию интерфейса `List`:

```
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, этот 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 эта функциональность реализована в виде статического фабричного метода, сам же класс является недоступным *анонимным классом* (статья 22), спрятанным внутри метода.

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 приём, 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 приведём скелетную реализацию интерфейса `Map.Entry`.

```
public abstract class AbstractMapEntry<K, V>
        implements Map.Entry<K, V> {
    // Primitive operations
    public abstract K getKey();

    public abstract V getValue();

    // Entries in modifiable maps must override this method
    public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of 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 equals(getKey(), arg.getKey())
                && equals(getValue(), arg.getValue());
    }

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

    // Implements the general contract of Map.Entry.hashCode
```

```
    @Override public int hashCode() {
        return hashCode(getKey()) ^ hashCode(getValue());
    }

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

Комментарий. Начиная с Java 7, вместо закрытых статических методов `equals` и `hashCode`, объявленных в приведённом классе `AbstractMapEntry`, можно было бы использовать стандартные статические методы `Objects.equals` и `Objects.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.

Вариацией на тему скелетной реализации является *простая реализация*, такая как, например, `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 свой подкласс.

Комментарий. Сведения, приведённые ниже, отчасти устарели с появлением в Java 8 возможности добавлять в интерфейсы неабстрактные методы.

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 реализаций данного абстрактного класса. Для интерфейсов этот приём 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, изменить 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 серьёзные 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 *до того*, как он будет «заморожен». Это позволит вам наити все ошибки, пока у вас ещё есть возможность их исправить.

Комментарий. С появлением Java 8 даже эти преимущества абстрактных классов перед интерфейсами уже не столь велики. Ограничения на эволюцию интерфейсов значительно ослабли с появлением возможности объявлять в интерфейсах неабстрактные *методы по умолчанию* (default methods), что позволяет добавлять в интерфейсы новые методы, не ломая его клиентов. Множество таких методов (например, `stream` и `forEach`) было добавлено в стандартные интерфейсы коллекций. Более того, благодаря методам по умолчанию интерфейсы теперь сами могут служить своими скелетными реализациями, оставив абстрактными только примитивные операции. Например, расширенный в Java 8 интерфейс `Iterable` реализует методы `forEach` и `spliterator` поверх абстрактного метода `iterator`.

Единственными ограничениями интерфейсов, из-за которых иногда всё-таки имеет смысл использовать абстрактные классы, является невозможность объявлять в интерфейсах поля экземпляра и конструкторы, а также объявлять методы как `final`, `private` или `protected`. Даже ограничение на `private`-методы легко обойти, вынеся их в отдельный недоступный извне класс. А в Java 9 в интерфейсах можно объявлять и `private`-методы, хотя все остальные ограничения остаются в силе.

Подведём итоги. Интерфейс обычно является наилучшим способом определения типа, который допускает несколько реализаций. Исключением из этого правила является случай, когда лёгкости совершенствования придаётся 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 путём написания многочисленных реализаций.

Статья 19. Используйте интерфейсы только для определения типов

Если класс реализует интерфейс, то этот интерфейс должен служить неким *типом*, 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`, экспортирующие константы. Классы, в которых эти константы используются, реализуют данный интерфейс для того, чтобы исключить необходимость в добавлении к названию констант название класса. Приведём пример:

```
// Антишаблон интерфейса констант - не используйте его!
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 двоичной совместимости (binary compatibility). Если же интерфейс констант реализуется наследуемым классом, константами из этого интерфейса будет засорено пространство имён всех его подклассов.

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

Для экспорта 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 объявлять их нужно с помощью *перечислимого типа* (enum type, статья 30). В остальных случаях вы должны экспортировать константы с помощью *класса утилит* (utility class), не имеющего экземпляров (статья 4). Представим вариант класса утилит для предыдущего примера `PhysicalConstants`:

```
// Класс утилит для констант
package com.effectivejava.science;

public class PhysicalConstants {
    // Предотвращает создание экземпляров
    private PhysicalConstants() { }

    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 классу утилит требуется, чтобы клиенты квалифицировали названия констант именем класса, например, `PhysicalConstants.AVOGADROS_NUMBER`. Если вы часть используете константы, объявленные в классе утилит, то вы можете отказаться от необходимости квалификации констант именем класса, используя возможности *статического импорта*, добавленные в версии 1.5.

```
// Используем статический импорт для избежания необходимости
// квалификации констант
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }

    // Если PhysicalConstants используется ещё много где,
    // то использование статического импорта оправданно
}
```

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

Статья 20. Предпочитайте иерархии классов классам с метками

Иногда вам может встретиться класс, экземпляры которого бывают двух и более видов и могут содержать поле метки (tag field), которое определяет вид экземпляра. Например, рассмотрим класс, который может представлять круг или прямоугольник:

```
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 объединённых классов множество недостатков. Они загромождены шаблонным кодом (boilerplate), включающим в себя объявление перечислимых типов, полей метки и инструкций `switch`. Читаемость кода ещё больше страдает, потому что большое количество реализаций смешивается в один класс. Растёт расход памяти, потому что экземпляры нагружены ненужными полями другого вида. Поля не могут быть объявлены как `final`, если только конструкторы не инициализируют ненужные поля, что приводит к созданию большего объёма шаблонного кода. Конструкторы должны задавать поле метки и инициализировать корректные поля данных без помощи компилятора: если будут инициализированы не те поля, то программа даст сбой при выполнении. Вы не можете добавить ещё одну разновидность к объединённому классу, если вы не измените его исходный файл. Если вы действительно добавите новую разновидность, то должны не забыть добавить новый блок `case` к каждой инструкции `switch`, в противном случае класс вы получите ошибку времени выполнения. Наконец, тип экземпляра класса сам по себе не даёт понятия о его разновидности. Короче говоря, **классы с метками слишком многословны, чреваты ошибками и неэффективны**.

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

Чтобы преобразовать класс с меткой в иерархию классов, определите абстрактный класс, содержащий метод для каждой операции, работа которой зависит от значения метки. 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 метки, представьте ее как неабстрактный метод корневого класса. Точно так же, если в классе с меткой есть какие-либо поля данных, используемые всеми его разновидностями, поместите их в корневой класс. В приведённом классе `Figure` подобных операций и полей данных, не зависящих от разновидности экземпляра, нет.

Далее, для каждой разновидности, которая может быть представлена классом с меткой, определите неабстрактный подкласс корневого класса. В предыдущем примере такими типами являются круг и прямоугольник. В каждый подкласс поместите те поля данных,

которые характерны для соответствующей разновидности. 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 класса с меткой:

```
// 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 классов с метками, отмеченные ранее. Код прост и ясен, и не содержит шаблонного кода. Реализация каждой разновидност выделена в свой собственный класс, и ни один из этих классов не нагружен ненужными полями данных. Все поля объявлены как `final`. 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 из-за отсутствия нужного блока `case` в инструкции `switch`. 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.

Ещё 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 класс с меткой в исходном примере допускает также построение квадратов. В иерархии классов можно показать, что квадрат — это частный случай прямоугольника (при условии, что оба класса являются неизменяемыми):

```
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 классы с метками редко приемлемы. Если у вас будет искушение написать такой класс, подумайте, можно ли его заменить иерархией. Если вам попадётся уже существующий класс, подумайте над возможностью преобразования его в иерархию.

Статья 21. Используйте объекты-функции для представления стратегий

Некоторые языки поддерживают *указатели на функции* (function pointer), *делегаты* (delegates), *лямбда-выражения* (lambda expressions) и другие возможности, что даёт

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 передаёт ей указатель на другую функцию. Например, функция `qsort` из стандартной библиотеки C 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, ноль, 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 “Стратегия” (Strategy) из [Gamma95, с. 315], функция-компаратор представляет алгоритм сортировки элементов.

В языке Java указатели на функции отсутствуют, но те же самые возможности можно получить с помощью ссылок на объекты. Вызов метода у объекта обычно производит некоторое действие над *самим этим объектом*. Между тем можно построить объект, чьи методы выполняют действия над *другими объектами*, непосредственно предоставленным этим методам. 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 передаёт 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 задаёт упорядочение строк по длине. Ссылка на объект `StringLengthComparator` служит «указателем на функцию», что

позволяет его использовать для любой пары строк. Иными словами, экземпляр класса `StringLengthComparator` — это *конкретная стратегия* (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 стратегии* (strategy interface), который должен соответствовать классу конкретной стратегии. Представим этот интерфейс:

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

Интерфейс `Comparator` определён в пакете `java.util`, хотя никакого волшебства в этом нет и вы могли точно так же определить его сами. Интерфейс `Comparator` является *обобщённым типом* (generic type, статья 26), что позволяет применять его для компараторов объектов, не являющихся строковыми. Его метод `compare`

принимает два параметра типа `T` (его *формальный параметр типа*) вместо `String`. Класс `StringComparator` можно объявить как реализующий этот интерфейс:

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

Классы конкретных стратегий часто объявляются 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 приведёт к созданию нового экземпляра каждый раз при выполнении вызова. Если выполнение должно происходить постоянно, рассмотрите возможность хранения объекта функции в поле `private static final` и повторного его использования. Другое преимущество этого заключается в том, что вы можете дать полю понятное имя для объекта функции.

Комментарий. Начиная с Java 8, интерфейсы с единственным абстрактным методом (функциональные интерфейсы) можно реализовывать лямбда-выражениями:

```
Arrays.sort(stringArray, (s1, s2) -> s1.length() - s2.length());
```

или даже, ещё лучше:

```
Arrays.sort(stringArray, Comparator.comparingInt(String::length));
```

JVM оптимизирует обращения к лямбда-выражениям без состояния, переиспользуя один и тот же объект на каждый вызов, так что нет необходимости в ручном кэшировании объекта, как в случае с обычным или анонимным классом.

Поскольку интерфейс стратегии используется как тип для всех экземпляров конкретных стратегий, то для того, чтобы предоставить конкретную стратегию, нет необходимости делать соответствующий класс открытым. Вместо этого класс-хозяин (host) может передать открытое статическое поле (или статический фабричный метод), тип которого соответствует интерфейсу стратегии, сам же класс стратегии может оставаться закрытым

классом, вложенным в класс-хозяин. В следующем примере вместо анонимного класса используется статический класс-член, что позволяет реализовать в классе стратегии второй интерфейс — `Serializable`:

```
// Предоставление конкретной стратегии
class Host {
    private static class StrLenCmp
            implements Comparator<String>, Serializable {
        public int compare(String s1, String s2) {
            return s1.length() - s2.1ength();
        }
    }

    // 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` экспортировать компаратор строк, не зависящий от регистра.

Комментарий. В Java 8 можно сделать лямбда-выражение сериализуемым, явно указав интерфейс `Serializable` в числе реализуемых им интерфейсов. Кроме того, семейство статических методов `Comparator.comparingXXX`, использованное выше, возвращает сериализуемый компаратор, если переданный ему извлекатель ключа (key extractor) тоже реализует `Serializable`.

```
Arrays.sort(stringArray, (Comparator<String> & Serializable)
        (s1, s2) -> s1.length() - s2.length());
Arrays.sort(stringArray, Comparator.comparingInt(
        (ToIntFunction<String> & Serializable) String::length));
```

Подведём итоги. Указатели на функции обычно используются для реализации шаблона Strategy. Для того, чтобы реализовать этот шаблон в языке Java, необходимо создать интерфейс, представляющий стратегию, 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 передаётся для многократного использования, ее класс обычно становится закрытым статическим классом-членом и передаётся через поле `public static final`, чей тип соответствует интерфейсу стратегии.

Статья 22. Предпочитайте статические классы-члены нестатическим

Класс называется *вложенным* (nested), если он определён 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). 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 объявлен 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 объявлен как закрытый, доступ к нему имеет лишь окружающий его класс, и т.д.

Одним из распространённых видов статических классов-членов является открытый вспомогательный класс (public helper class), который пригоден для применения только в сочетании с внешним классом. Например, рассмотрим перечисление, описывающее операции, которые может выполнять калькулятор (статья 30). Перечислимый тип `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`.

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 внешнего класса* (enclosing instance). Из метода в экземпляре нестатического класса-члена можно вызывать методы содержащего его экземпляра, либо, используя специальную конструкцию *квалифицированного this* (qualifying 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 *обязан* быть статическим: 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 задаётся автоматически путём вызова конструктора нестатического класса-члена из экземпляра метода во внешнем классе. Иногда имеет смысл установить связь вручную, используя выражение `enclosingInstance.new MemberClass(args)`, хотя потребность в этом возникает редко. 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 views), 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<E> {
        ...
    }
}
```

Если вы объявили класс-член, которому не нужен доступ к экземпляру содержащего его класса, *обязательно* поместите в его объявление модификатор `static`, 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 часто используются для представления составных частей 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 создаётся внутренний объект `Entry`. Хотя каждая такая запись ассоциируется с содержащим её словарём, методам этой записи (`getKey`, `getValue` и `setValue`) доступ к самому словарю не требуется. Соответственно, использовать нестатические классы-члены для представления отдельных записей в объекте `Map` было бы расточительно, самое лучшее решение — закрытый статический класс-член. Если в декларации этой записи вы случайно пропустите модификатор `static`, словарь будет работать, но каждая запись будет содержать ненужную ссылку на содержащий её словарь, напрасно тратя время и место в памяти.

Вдвойне важно правильно сделать выбор между статическим и нестатическим классом-членом, когда этот класс является открытым или защищённым 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 двоичной совместимости.

Анонимные классы не похожи ни на какие другие классы в языке Java. Анонимный

класс не имеет имени. Он не является членом содержащего его класса. Вместо того, чтобы объявляться вместе с остальными членами класса, он одновременно объявляется и порождает экземпляр в момент использования. Анонимный класс можно поместить в любом месте программы, где разрешается применять выражения. В зависимости от местоположения анонимный класс ведёт себя как статический либо как нестатический класс-член: в нестатическом контексте появляется окружающий его экземпляр. Но даже если анонимные классы появляются в статическом контексте, у них не может быть статических членов.

Применение анонимных классов имеет несколько ограничений. Их экземпляр нельзя создать нигде, кроме как в точке объявления класса. Анонимный класс не имеет имени, поэтому с ним нельзя использовать оператор `instanceof` и никакие другие конструкции, требующие имени класса. Нельзя объявить анонимный класс как реализующий несколько интерфейсов либо как одновременно расширяющий класс и реализующий интерфейс. Клиенты анонимного класса не могут вызывать никакие его методы, кроме тех, которые он наследует от своего супертипа. 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). Например, вызов метода `sort` может отсортировать строки массива по их длине при передаче ему подходящего объекта `Comparator`. Другой распространённый случай использования анонимного класса — создание *объекта процесса* (process object), такого как `Thread`, `Runnable` или `TimerTask`. Третий вариант — использование анонимных классов в статических фабричных методах (см. метод `intArrayAsList` в статье 18).

> **Комментарий.** В Java 8 применять анонимные классы приходится значительно реже, поскольку самый часто встречающийся сценарий их использования — реализация интерфейсов с единственным абстрактным методом — теперь покрывается лямбда-выражениями.

Локальные классы используются реже всего среди этих четырёх категорий вложенных классов. Локальный класс можно объявлять везде, где разрешается объявлять локальную переменную, и он подчиняется тем же самым правилам видимости. Локальный класс имеет несколько признаков, объединяющих его с каждой из трёх

других категорий вложенных классов. Как и классы-члены, локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они содержат ссылку на экземпляр внешнего класса тогда и только тогда, когда применяются в нестатическом контексте. Кроме того, как и анонимные классы, они должны быть достаточно короткими, чтобы не мешать удобству чтения.

Подведём итоги. Существует четыре категории вложенных классов, каждая из которых занимает своё 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, делайте его нестатическим, в остальных случаях он должен быть статическим. Если класс находится внутри метода, то анонимным классом его можно сделать тогда, когда вам нужно создавать экземпляры этого класса только в одном месте программы и уже есть тип, который этот класс характеризует. В противном случае используйте локальный класс.

Глава 5. Средства обобщённого программирования (generics)

В релизе 1.5 в языке Java появились средства *обобщённого программирования* (generics). До их появления необходимо было выполнять приведение типа (type cast) всех объектов, прочитанных из коллекций. Если кто-то случайно вставлял объект неправильного типа, приведение типа завершалось с ошибкой во время выполнения. С помощью средств обобщённого программирования вы указываете компилятору, какие типы объектов разрешены в каждой коллекции. Компилятор автоматизирует приведение типов 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 определёнными 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 Аангера [Langer08] или книгу Нафталина и Уодлера [Naftalin07].

Статья 23. Не используйте сырые типы в новом коде

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

Каждый обобщённый тип определяет набор *параметризованных типов* (parameterized types), которыt состоит из наименования класса или интерфейса, после которого следует в угловых скобках список *фактических параметров типа* (actual type parameters), соответствующий формальным параметрам типа, относящимся к обобщённому типу [JLS, 4.4, 4.5]. Например, `List<String>` (читается «`List` от `String`») — это параметризованный тип, представляющий список, элементы которого принадлежат типу `String`. (`String` — это фактический параметр типа, соответствующий формальному параметру типа `E`.)

Наконец, каждый обобщённый тип определяет *сырой тип* (raw type) , который является наименованием обобщённого типа без каких-либо фактических параметров типа [JLS, 4.8]. Например, сырым типом, соответствующим `List<E>`, будет `List`. Сырые типы ведут себя так, словно вся информация об обобщённом типе оказалась стёрта при его объявлении. На практике сырой тип `List` ведёт себя аналогично тому, как вёл себя интерфейс `List` до добавления к платформе Java средств обобщённого программирования.

До релиза 1.5 типичным объявлением коллекции было бы следующее:

```
// Сырой тип коллекции - не делайте так!
/**
 * Моя коллекция марок (stamp). Содержит только экземпляры Stamp.
 */
private final Collection stamps = ... ;
```

Если вы случайно добавите в 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:

```
// Сырой тип итератора - не делайте так!
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 как можно раньше, в идеале на этапе компиляции. В нашем же случае вы не найдёте ошибку до момента выполнения программы, а найдёте ее намного позднее, чем она сделана, и совершенно не в том коде, в котором она была допущена. Как только вы увидите исключение `ClassCastException`, вам придётся искать по всему коду и смотреть, где вызываются методы, которые помещают монету в коллекцию марок. Компилятор тут вам не сможет помочь, потому что он не понимает комментарий, в котором сказано «Содержит только экземпляры `Stamp`».

С использованием средств обобщённого программирования вы заменяете комментарий объявлением параметризованного типа, который даст компилятору дополнительную информацию, недоступную ранее в комментарии:

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

Благодаря такому объявлению 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` объявляется с использованием параметризованных типов, 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 приводить тип вручную при извлечении элементов из коллекции. Компилятор добавляет невидимые операции приведения типа за вас и гарантирует, что они обязательно сработают (опять-таки при условии, что весь ваш код откомпилирован компилятором, который знает о существовании обобщённых средств программирования, и не скрывает никаких предупреждений). Это утверждение верно вне зависимости от того, используете ли вы цикл for-each (статья 46):

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

или обычный цикл:

```
// Цикл for с использованием параметризованного итератора -
// 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 обобщённые типы без объявления параметров, но делать этого не рекомендуется. **Если вы используете сырые типы, то вы теряете все преимущества обобщённого программирования.** В то же время, если вам не следует использовать сырые типы, почему тогда создатели языка оставили вам такую возможность? Для совместимости. Платформа Java существовала два десятилетия до появления средств обобщённого программирования, и теперь в мире существует огромное количество кода, который их не использует. Критически важно, чтобы этот старый код продолжал компилироваться мог взаимодействовать с новым кодом, использующим средства обобщённого программирования. Необходимо было разрешить передавать экземпляры параметризованных типов методам, которые были созданы для использования только обычгых типов, и наоборот. Это требование, известное как *миграционная совместимость* (migration compatibility) , привело к решению сохранить поддержку сырых типов.

Хотя сырые типы наподобие `List` не следует использовать в новом коде, совершенно нормально использовать типы с параметрами, разрешающими добавление произвольных объектов, например, `List<Object>`. В чем же разница между сырым типом `List` и параметризованным типом `List<Object>`? Грубо говоря, в первом случае не выполняется никакой проверки типов, в то время как последний точно говорит компилятору, что он может содержать объект любого типа. В то время как вы можете передать объект типа `List<String>` параметру с типом `List`, вы не можете передать его параметру с типом `List<Object>`. Существуют правила образования подтипов в обобщённом программировании. `List<String>` является подтипом сырого типа `List`, но не является таковым для параметризованного типа `List<Object>` (статья 25). Как следствие, **вы теряете в безопасности типов в случае использования сырого типа, такого как `List`, но сохраняете ее при использовании параметризованного типа**

`List<Object>`.

Для конкретного примера рассмотрим следующую программу:

```
// Использует сырой 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 сырой тип `List`, компилятор выводит 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 исключение `ClassCastException`, когда программа пытается привести результат вызова `strings.get(0)` к типу `String`. Это приведение типов сгенерировано компилятором, поэтому обычно гарантируется, что оно успешно завершится, но в данном случае мы проигнорировали предупреждение компилятора и заплатили за это.

Если вы замените сырой тип `List` параметризованным типом `List<Object>` в объявлении `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 сырой тип для коллекции, типы которой неизвестны или не имеют для вас значения. Например, предположим, что вы хотите написать метод, который берет два множества и возвращает количество общих для них элементов. Так будет выглядеть метод, если вы незнакомы со средствами обобщённого программирования:

```
// Использование сырых типов для неизвестных типов
// элементов - не делайте так!
static int numElementsInCommon(Set s1, Set s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1))
            result++;
    return result;
}
```

Этот метод работает, но он использует сырые типы, что опасно. Релиз Java 1.5 предоставляет безопасную альтернативу, известную под названием *«неограниченные подстановочные типы»* (unbounded wildcard types). Если вы хотите использовать обобщённые типы, но не знаете фактические параметры типов либо они вас не интересуют, то вы можете использовать вместо этого знак вопроса. Например, неограниченным подстановочным типом для обобщённого типа `Set<E>` будет `Set<?>` (читается как «`Set` какого-то типа»). Это наиболее общий параметризованный тип `Set`, способный описать *любое* множество. Вот пример того, как метод `numElementsInCommon` выглядит с использованием неограниченного подстановочного типа:

```
// 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 неограниченным подстановочным типом `Set<?>` и сырым типом

`Set`? Даёт ли нам что-либо знак вопроса? Если вкратце, то подстановочный тип безопасен, а сырой тип — нет. Вы можете добавить в коллекцию любой элемент с использованием сырого типа, с лёгкостью повредив инварианты типа коллекции (как показано на примере метода `unsafeAdd`), но **вы не можете добавить никакой элемент (кроме `null`) 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");
      ^
```

Честно говоря, это сообщение об ошибке оставляет желать лучшего, но компилятор выполнил свою задачу, защитив инварианты коллекции от повреждения. Вы не только не можете добавить никакой элемент (кроме `null`) в `Collection<?>`, но также не можете ничего предположить относительно типа объекта, который извлечёте. Если эти ограничения неприемлемы, то можно использовать *обобщённые методы* (generic methods) (статья 27) или *ограниченные подстановочные типы* (bounded wildcard types) (статья 28).

Есть два небольших исключения из правила о неиспользовании сырых типов в новом коде, и оба они основаны на том факте, что информация об обобщённых типах стирается во время выполнения (статья 25). **Вы обязаны использовать сырые типы в классовых литералах.** Спецификация языка не разрешает использование параметризованных типов в данном случае, хотя и разрешает типы массивов и примитивные типы [JLS 15.8.2]. Другими словами, `List.class`, `String[].class` и `int.class` использовать разрешено, но `List<String>.class` и `List<?>.class` — нет.

Второе исключение из правила связано с оператором `instanceof`. Поскольку информация об обобщённом типе стирается при выполнении, нельзя применять оператор `instanceof` к параметризованным типам, за исключением неограниченных подстановочных типов. Использование неограниченных подстановочных типов вместо сырых типов никогда не влияет на поведение оператора `instanceof`. В данном случае угловые скобки и знаки вопроса только загромождают код. **Вот как предпочтительнее использовать оператор `instanceof` с обобщёнными типами:**

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

Обратите внимание, что, как только вы определите, что `o` является экземпляром типа `Set`, вы должны привести его к подстановочному типу `Set<?>`, а не к сырому типу `Set`. Это проверяемое приведение типов не приведёт к предупреждениям со стороны компилятора.

Подводём итоги. Использование сырых типов может привести к исключениям во время выполнения. Не используйте их в новом коде. Они существуют лишь для совместимости и взаимодействия с кодом, написанным до появления средств обобщённого программирования. В качестве небольшого резюме: `Set<Object>` — это параметризованный тип, который может содержать объекты любого типа, `Set<?>` — подстановочный тип, который может содержать только объекты некоторого неизвестного типа, а `Set` — это сырой тип, который лишает нас возможности использовать систему обобщённых 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
Параметризованный тип (parameterized type)	`List<String>`	Статья 23
Фактический параметр типа (actual type parameter)	`String`	Статья 23
Обобщённый тип (generic type)	`List<E>`	Статьи 23, 26
Формальный параметр типа (formal type parameter)	`E`	Статья 23
Неограниченный подстановочный тип (unbounded wildcard type)	`List<?>`	Статья 23

Термин	Пример	Статья
Сырой тип (raw type)	`List`	Статья 23
Ограниченный параметр типа (bounded type parameter)	`<E extends Number>`	Статья 26
Рекурсивная граница типа (recursive type bound)	`<T extends Comparable<T>>`	Статья 27
Ограниченный подстановочный тип (bounded wildcard type)	`List<? extends Number>`	Статья 28
Обобщённый метод (generic method)	`static <E> List<E> asList(E[] a)`	Статья 27
Метка типа (type token)	`String.class`	Статья 29

Статья 24. Избавляйтесь от предупреждений о непроверенных операциях с типами

Если вы программируете с использованием обобщённых средств, то столкнётесь со множеством предупреждений компилятора: предупреждения о непроверенном приведении типов, предупреждение о непроверенном вызове метода, предупреждение о непроверенном создании обобщённого массива и о непроверенном преобразовании. По мере приобретения опыта работы с обобщённым программированием таких предупреждений будет все меньше и меньше, но не стоит быть уверенными, что новый написанный код с использованием обобщённых средств будет абсолютно безошибочным.

Многих таких предупреждений можно легко избежать. Например, предположим, вы случайно написали такое объявление:

```
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>();
```

Комментарий. А начиная с Java 7, можно и не повторять параметры типа:

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

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. **Избавляйтесь от 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 исключение `ClassCastException` при выполнении, и увеличит вашу уверенность в том, что программа ведёт 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 подавить предупреждения с помощью аннотации `@SuppressWarnings("unchecked")`. Если вы скроете ошибки, не убедившись, что код безопасен, то вы тем самым лишь даёте себе ложное ощущение безопасности. Код может откомпилироваться без предупреждений, но при выполнении все равно выбросить `ClassCastException`. Если тем не менее вы просто проигнорируете предупреждения о непроверенных операциях, зная при этом, что они безопасны (вместо того, чтобы подавить эти предупреждения), то тогда вы не заметите, когда появится новое предупреждение, представляющее реальную проблему. Новое предупреждение потеряется среди ложных тревог, которые вы не подавили.

Аннотация `SuppressWarnings` может использоваться на любом уровне гранулярности, от объявления локальных переменных до целых классов. **Всегда используйте аннотацию `SuppressWarnings` на как можно меньшей области видимости.** Обычно это будет объявление переменной, очень короткий метод или конструктор. Никогда не

используйте `SuppressWarnings` на целом классе. Это может спрятать действительно критические предупреждения.

Если вы используете аннотацию `SuppressWarnings` для метода или конструктора длиной более чем в одну строку, то, возможно, вы сможете свести его до применения на одном объявлении локальной переменной. Возможно, вам понадобится объявить новую локальную переменную, но это стоит того. Например, рассмотрим это на методе `toArray` из класса `ArrayList`:

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
    if (a.length < size)
        return (T[]) Arrays.copyOf(elementData, size, a.getClass());
    System.arraycopy(elementData, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

Если вы уберёте с метода аннотацию `@SuppressWarnings("unchecked")` и затем скомпилируете класс `ArrayList`, метод выведет следующее предупреждение:

```
ArrayList.java:407: warning: [unchecked] unchecked cast
found   : Object[], required: T[]
    return (T[]) Arrays.copyOf(elementData, size, a.getClass());
                               ^
```

Не разрешено применять аннотацию `SuppressWarnings` на выражении возврата, потому что это не объявление [JLS, 9.7]. Вы можете захотеть поместить аннотацию на весь метод, но делать этого не надо. Вместо этого объявите локальную переменную для хранения возвращаемого значения и аннотацию поместите на объявление, 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(elementData, size, a.getClass());
        return result;
    }
    System.arraycopy(elementData, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
}
```

Этот метод компилируется чисто, а подавление предупреждений ограничено минимально возможной областью. 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 изменён 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.

Подводём итоги. Предупреждения о непроверенных операциях важны. Не стоит их игнорировать. Каждое такое предупреждение представляет собой потенциальную предпосылку возникновения исключения `ClassCastException` при выполнении. Сделайте все возможное, чтобы избежать их. Если же избежать не удаётся и вы уверены, что код безопасен, подавите предупреждения аннотацией `@SuppressWarnings("unchecked")` на минимальной возможной области действия. Опишите в комментарии причину, по которой вы приняли решение подавить предупреждения.

Статья 25. Предпочитайте списки массивам

Массивы отличаются от обобщённых типов в двух важных аспектах. Во-первых, массивы *ковариантны* (covariant). Это жуткое слово значит просто, что если `Sub` является подтипом `Super`, тогда тип массива `Sub[]` является подтипом `Super[]`. Средства обобщённого программирования, напротив, *инвариантны* (invariant): для любых двух отдельных типов `Type1` и `Type2` тип `List<Type1>` не является ни подтипом, ни

супертипом для `List<Type2>` [JLS, 4.10; Naftalin07, 2.5]. Вы можете подумать, что это недостаток средств обобщённого программирования, но можно поспорить, что недостатком, напротив, обладают именно массивы.

Этот фрагмент кода разрешён:

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

А этот — нет:

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

В любом случае вы не можете вставить `String` в контейнер объектов `Long`, но в случае массива вы обнаружите ошибку только при выполнении, в то время как использование списка приведёт к ошибке на этапе компиляции.

Вторым важным отличием массивов от обобщённых типов является то, что массивы *реифицированы* (reified) [JLS, 4.7]. Это значит, что массивы знают и проверяют тип своих элементов при выполнении. Как выше было сказано, если вы попытаетесь сохранить `String` в массив `Long`, вы получите исключение `ArrayStoreException`. Обобщённые типы, напротив, реализуются *стиранием* (erasure) [JLS, 4.6]. Это значит, что они проверяют свои ограничения типов только на этапе компиляции и затем выбрасывают (или стирают) информацию о типах элементов при выполнении. Стирание позволяет обобщённым типам легко взаимодействовать со старым кодом, который не использует средства обобщённого программирования (статья 23).

По причине этих фундаментальных различий массивы и обобщённые типы не могут применяться одновременно. Например, нельзя создавать массив обобщённых типов, параметризованных типов или параметров типа. Ни одно из этих выражений создания массивов не является разрешённым: `new List<E>[]`, `new List<String>[]`, `new E[]`. Все выражения приведут к *ошибкам создания обобщённых массивов* на этапе компиляции.

Почему нельзя создавать обобщённые массивы? Потому что это небезопасно. Если бы это было разрешено, приведение типов, генерируемое компилятором в правильно

написанной программе, могло бы вызвать исключение `ClassCastException`. Это бы нарушило фундаментальные гарантии, которые даёт система обобщённых типов.

Чтобы подкрепить эти рассуждения конкретным примером, рассмотрим следующий фрагмент кода:

```
// Почему создание обобщённых массивов не разрешено - не компилируется!
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 обобщённый массив, разрешена. Строка 2 создаёт и инициализирует `List<Integer>`, содержащий единственный элемент. Строка 3 сохраняет массив `List<String>` в переменную массива `Object`, что разрешено, потому что массивы ковариантны. Строка 4 сохраняет `List<Integer>` в единственный элемент массива `Object`, что тоже закончится удачно, потому что обобщённые типы реализуются стиранием: типом времени выполнения экземпляра `List<Integer>` является просто `List`, а типом времени выполнения экземпляра `List<String>[]` является `List[]`, поэтому такое присваивание не приводит к исключению `ArrayStoreException`. Теперь у нас проблема. Мы сохранили экземпляр `List<Integer>` в массив, который объявлен как хранящий только экземпляры `List<String>`. В строке 5 мы извлекаем один элемент из списка в этом массиве. Компилятор автоматически приведёт извлечённый элемент к типу `String`, но он на самом деле `Integer`, так что мы получим при выполнении исключение `ClassCastException`. Чтобы такого не случилось, строка 1 (создающая обобщённый массив) выдаёт ошибку при компиляции.

Такие типы, как `E`, `List<E>` и `List<String>`, известны под техническим названием *нереифицируемых* (nonreifiable) типов [JLS, 4.7]. Говоря интуитивно, нереифицируемые типы — это такие типы, представление которых содержит меньше информации при выполнении, чем при компиляции. Единственными реифицируемыми параметризованными типами являются неограниченные подстановочные типы, такие как `List<?>` и `Map<?,?>` (статья 23). Разрешено, хотя это и редко бывает нужно, создание массивов неограниченных подстановочных типов.

Запрет на создание обобщённых массивов может доставлять неудобства. Он означает,

что обобщённые типы не могут вернуть массив этого типа элемента (но см. статью 29, где приводится частичное решение этой проблемы). Это также означает, что вы можете получить запутанные предупреждения при использовании методов с переменным числом параметров (статья 42), так как каждый раз при запуске такого метода создаётся массив для хранения переданных ему параметров. Если тип элемента в этом массиве не является реифицируемым, то вы получите предупреждение. Мало что можно сделать, чтобы избежать этих предупреждений, — остаётся только их подавить (статья 24) и избегать одновременного использования средств обобщённого программирования и методов с переменным числом параметров в одном и том же API.

Комментарий. В Java 7 появилась аннотация `@SafeVarargs`, позволяющая сказать компилятору, что метод с переменным числом параметров не производит никаких небезопасных операций над массивом параметров. Применение этой аннотации предпочтительнее, чем скрытие предупреждений с помощью `@SuppressWarnings(unchecked)`.

Например:

```
@SafeVarargs
public static <T> List<T> concat(
        Collection<? extends T>... collections) {
    return Stream.of(collections)
                 .flatMap(Collection::stream)
                 .collect(Collectors.toList());
}
```

Когда вы получаете ошибку создания обобщённого массива, часто 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 более широкие возможности взаимодействия с другим кодом.

Например, предположим, что у вас есть синхронизированный список (наподобие возвращаемого методом `Collection.synchronizedList`) и функция, которая берет два значения из типов, хранящихся в списке, и возвращает третье. Теперь предположим, что вы хотите написать метод для свёртки списка с помощью функции. Если список содержит целые числа и функция складывает значения двух чисел, то метод `reduce` вернёт сумму всех значений в списке. Если функция умножает два целых числа, метод вернёт произведение значений списка. Если список содержит строковые значения

и функция объединяет две строки, то метод вернёт строку, содержащую обе строки соединённые последовательно. Помимо списков и функций метод `reduce` принимает начальное значение свёртки, которое возвращается, если список пуст. (Начальное значение обычно является *нейтральным элементом* (identity element) для функции — например, 0 для сложения, 1 для умножения и `""` для объединения строк.) Вот как может выглядеть код без использования средств обобщённого программирования:

```
// Метод reduce без использования средств обобщённого
// 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 чужие методы (alien methods) из синхронизированного блока. Так что вы модифицируете метод `reduce` так, чтобы он копировал содержимое списка во время удержания блокировки, чтобы затем применить операцию свёртки к копии списка. До релиза 1.5 мы бы использовали метод `List.toArray`, который захватывает блокировку внутри себя:

```
// Метод reduce без средств обобщённого
// программирования и без недостатков параллелизма.
static Object reduce(List list, Function f, Object initVal) {
    Object[] snapshot = list.toArray(); // Locks list internally
    Object result = initVal;
    for (Object o : snapshot)
        result = f.apply(result, o);
```

```
    return result;
}
```

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

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

Комментарий. В Java 8 вам уже не нужно объявлять такой интерфейс самим: он есть в стандартной библиотеке под названием `BinaryOperator`.

А вот наивная попытка использовать средства обобщённого программирования в пересмотренной версии метода `reduce`. Это обобщённый метод (статья 27). Не обращайте внимания, если вам непонятно его объявление. В данном случае надо изучить содержимое метода:

```
// Наивная реализация метода reduce с использованием
// обобщённого программирования - не компилируется!
static <E> E reduce(List<E> list, Function<E> f, E initVal) {
    E[] snapshot = list.toArray(); // Locks list
    E result = initVal;
    for (E e : snapshot)
        result = f.apply(result, e);
    return result;
}
```

Если вы попытаетесь откомпилировать этот метод, то получите следующую ошибку:

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

«Не беда, — подумаете вы, — я приведу тип массива `Object` к массиву `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(); // Locks list
                                             ^
```

Компилятор говорит, что он не может проверить безопасность приведения типов во время выполнения, так как не представляет, чем во время выполнения будет `E`. 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 вы получите исключение `ClassCastException` в строке, которая не содержит явного приведения типов. Во время компиляции `E[]` соответствует типу, который может быть `String[]`, `Integer[]` или любым другим типом массива. Во время выполнения его типом является `Object[]`, и именно это опасно. Приведение типа для массивов нереифицируемых типов должно использоваться только при особых 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 обобщённого
// 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.

> **Комментарий.** Это объявление метода `reduce` должно рассматриваться только как учебный пример. В реальном API параметр `list` следовало бы объявить как `List<? extends E>`, а не `List<E>`, по причинам, объяснённым в статье 28.

Подедём итоги. Массивы и обобщённые типы подчиняются разным правилам. Массивы ковариантны и реифицируемы, обобщённые типы — инвариантны и используют механизм стирания. Следствием этого становится то, что массивы обеспечивают безопасность типов во время выполнения, но не во время компиляции. С обобщёнными типами все наоборот. Вообще говоря, массивы и обобщённые типы не стоит использовать вместе. Если вам все же придётся сделать это и получить при этом ошибки или предупреждения при компиляции, то в первую очередь постарайтесь заменить массивы списками.

Статья 26. Предпочитайте обобщённые типы

Обычно не так сложно задать параметры в объявлении коллекции и использовать обобщённые типы и методы, поставляемые JDK. Написание же своего собственного обобщённого типа несколько сложнее. Однако это стоит того, и вам следует этому научиться. Рассмотрим простую реализацию стека из статьи 6:

```
// Коллекция на основе типа Object - 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; // Убираем устаревшую ссылку
        return result;
    }

    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 обобщённого программирования. В том виде, в каком оно сейчас есть, вы должны выполнять приведение типа для объектов, вытолкнутых из стека, и эти приведения типов могут закончиться ошибкой во время выполнения. Первый шаг состоит в добавлении параметров типа в объявление класса. В нашем случае есть только один параметр, представляющий тип элемента стека, и стандартным наименованием для этого параметра будет `E` (статья 56).

Следующий шаг — надо заменить все использования типа `Object` соответствующим параметром типа и попытаться скомпилировать полученную программу:

```
// Начальная попытка обобщить стек - не компилируется!
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
}
```

Обычно вы получите по крайней мере одну ошибку или предупреждение при компиляции, 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 нереифицируемых типов, таких как `E`. Эта проблема появляется каждый раз, когда вы пишете обобщённый тип, реализованный поверх массива. Есть два варианта решения проблемы. Первое решение обходит запрет на создание обобщённых массивов: создайте массив для `Object` и приведите его к массиву обобщённого типа. Теперь вместо ошибки компилятор выдаст предупреждение. Такое использование разрешено, но в целом не является типобезопасным (typesafe):

```
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 убедиться, что непроверяемое приведение типов не нарушит типобезопасность программы. Массив `elements` хранится в закрытом поле и никогда не возвращается клиенту и не передаётся никакому другому методу. Все элементы этого массива добавляются в него методом `push`, который гарантированно кладёт в него элементы типа `E`, так что непроверяемое приведение типов не может причинить вреда.

После того, как вы убедились, что непроверяемое приведение типов безопасно, подавите предупреждение в насколько возможно узкой области (статья 24). В этом случае конструктор содержит только непроверенное создание массива, так что вполне нормально подавить предупреждения во всем конструкторе. После добавления аннотации класс `Stack` компилируется без предупреждений, и вы можете использовать его, не боясь явных приведений типа и исключений `ClassCastException`:

```
// Элементы массива будут содержать только экземпляры E из push(E).
// Этого достаточно для типобезопасности, 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 обобщённых 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, приводя тип извлекаемого элемента массива от `Object` к `E`:

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

Поскольку `E` является нереифицируемым типом, то компилятор не может проверить это приведение типа при выполнении. Снова вы можете легко доказать себе, что непроверяемое приведение типа безопасно, поэтому нормально будет подавить предупреждения. Следуя совету из статьи 24, мы подавляем предупреждения только в той части метода, которая содержит непроверяемое приведение типа, а не во всем методе `pop`:

```
// Допустимое подавление предупреждений о непроверяемой операции
public E pop() {
    if (size == 0)
        throw new EmptyStackException();

    // push requires elements to be of type E, so cast is correct
    @SuppressWarnings("unchecked")
    E result = (E) elements[--size];

    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

Какую из двух технологий выбрать для обхода ошибок создания обобщённых массивов — это вопрос вашего вкуса. При прочих равных более рискованно скрывать предупреждения о непроверяемых приведениях типов при использовании типов массивов, нежели при использовании скалярных типов, что делает второе решение предпочтительнее. 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 приведений типа к `E` вместо однократного приведения типа к `E[]`, поэтому первое решение является более общепринятым [Naftalin07, 6.7].

Следующая программа демонстрирует использование обобщённого класса `Stack`. Программа печатает свои аргументы командной строки в обратном порядке и с переводом в верхний регистр. Не требуется явных приведений типа для вызова метода `String.toUpperCase` у вытолкнутых из стека элементов, и автоматически сгенерированное приведение типа гарантированно выполнится успешно:

```
// Маленькая программа, использующая на практике обобщённый 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 обобщённых типов. Java не поддерживает списки на уровне языка, так что некоторые обобщённые типы, такие как `ArrayList`, *приходится* реализовывать поверх массивов. Другие обобщённые типы, такие как `HashMap`, реализуются поверх массивов для улучшения производительности.

Подавляющее большинство обобщённых 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>` приведёт к ошибке компиляции. Это фундаментальное ограничение, накладываемое системой обобщённого программирования в языке Java. Вы можете обойти ограничения, используя упакованные (boxed) примитивные типы (статья 49).

Есть несколько обобщённых типов, которые ограничивают разрешённые значения своих параметров. Например, рассмотрим класс `java.util.concurrent.DelayQueue`, объявление которого выглядит следующим образом:

```
class DelayQueue<E extends Delayed> implements BlockingQueue<E>;
```

Список параметров типа (`<E extends Delayed>`) требует, чтобы фактический параметр типа `E` был подтипом `java.util.concurrent.Delayed`. Это позволит реализации `DelayQueue` и ее клиентам использовать методы типа `Delayed` на элементах `DelayQueue` без необходимости явного приведения типа или риска ошибки `ClassCastException`. Параметр типа `E` называется *ограниченным параметром типа* (bounded type parameter). Обратите внимание, что отношение подтипа определено таким образом, что каждый тип является подтипом самого себя [JLS, 4.10], так что разрешается создавать `DelayQueue<Delayed>`.

Подведём итоги. Обобщённые типы более безопасны и легки в применении, чем типы, требующие приведения типов в клиентском коде. Когда вы проектируете новые типы, убедитесь, что они могут быть использованы без подобного рода приведений типов. Зачастую это будет означать, что их следует сделать обобщёнными. Обобщайте существующие типы, насколько позволит время. Это облегчит жизнь новым пользователям этих типов, не сломав при этом существующий клиентский код (статья 23).

Статья 27. Предпочитайте обобщённые методы

Методы, подобно классам, выигрывают от использования обобщённых типов. Статические служебные методы — хорошие претенденты для этого. Все алгоритмические методы класса `Collections` (такие, как `binarySearch` и `sort`) обобщены. Написание обобщённых методов схоже с написанием обобщённых типов. Рассмотрим метод, который возвращает объединение двух множеств:

```
// Использует сырые типы - недопустимо! ([статья 23](#item23))
public static Set union(Set s1, Set s2) {
    Set result = new HashSet(sl);
    result.addAll(s2);
    return result;
}
```

Этот метод компилируется, 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);
                     ^
```

Чтобы исправить эти предупреждения и сделать метод типобезопасным, добавьте в объявление метода объявление типа параметров, представляющих тип элемента для трёх множеств (двух аргументов и возвращаемого значения), и используйте этот тип параметра в методе. **Список параметров типа лежит между модификаторами метода и его возвращаемым типом.** В этом примере списком параметров типа является `<E>`, а возвращаемым типом — `Set<E>`. Условия наименований для параметров для типа те же, что и для обобщённых методов и для обобщённых типов (статьи 26, 56):

```
// Обобщённый 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 в том, что касается простых обобщённых методов, это все, что следует знать. Теперь метод компилируется без каких-либо предупреждений и обеспечивает типобезопасность и простоту использования. Вот простая программа для того, чтобы попрактиковаться с нашим методом. Программа не содержит приведений типа 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 `[Моё, Harry, Tom, Curly, Larry, Dick]`. Порядок элементов зависит от реализации JDK.

Ограничение метода `union` заключается в том, что типы всех трёх наборов (как и входные параметры, так и возвращаемые значения) должны быть одинаковы. Вы можете сделать методы более гибкими, используя *ограниченные подстановочные типы* (bounded wildcard types, статья 28).

Ещё одна особенность обобщённых методов, которую стоит отметить, заключается в том, что вам нет необходимости явно задавать значение параметра для типа, как это необходимо делать при запуске обобщённых 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 `E` должен быть `String`. Этот процесс называется *выводом типов* (type inference).

Комментарий. Приведённый ниже фрагмент текста устарел с появлением Java 7.

Как уже говорилось в статье 1, вы можете использовать вывод типов, предоставляемый механизмом обобщённых методов, чтобы облегчить процесс создания экземпляров параметризованных типов. Как вы помните, необходимость явно передавать значения параметров для типов при запуске обобщённых конструкторов может раздражать. Параметры приходится задавать два раза, слева и справа от объявления переменной:

```
// Создание экземпляра типа с параметрами с помощью конструктора
Map<String, List<String>> anagrams =
    new HashMap<String, List<String>>();
```

Чтобы избежать этой избыточности, напишите обобщённый статический фабричный метод, соответствующий каждому конструктору, который вы хотите использовать. Например, вот фабричный метод, соответствующий конструктору без параметров класса `HashMap`:

```
// Обобщённый статический фабричный метод
public static <K,V> HashMap<K,V> newHashMap() {
    return new HashMap<K,V>();
}
```

С этим обобщённым фабричным методом вы можете заменить вышеописанное объявление переменной более кратким:

```
// Создание экземпляра параметризованного типа с помощью
// статического фабричного метода
Map<String, List<String>> anagrams = newHashMap();
```

Было бы замечательно, если бы язык выполнял вывод типа при запуске конструктора на обобщённых типах, как он делает это при запуске обобщённых методов. Когда-нибудь такое будет возможно, а сейчас, в релизе 1.6, этого нельзя сделать.

Комментарий. “Когда-нибудь” наступило с выходом Java 7 и появлением *ромбовидного оператора* (diamond operator), который позволяет использовать при вызове конструкторов параметризованных типов тот же механизм вывода типов, что и при вызове обобщённых методов.

```
Map<String, List<String>> anagrams = new HashMap<>();
System.out.println(new ArrayList<>(anagrams.keySet()));
```

Схожий шаблон — *обобщённая фабрика-синглтон* (generic singleton factory). Допустим, вам надо создать объект, который неизменяем, но применим ко многих различным типам. Поскольку обобщённые типы реализуются стиранием (erasure, статья 25), вы можете использовать единственный объект для параметризации на все необходимые типы, но вам нужно будет написать статический фабричный метод, чтобы отдавать один и тот же объект для каждой запрошенной параметризации. Этот шаблон зачастую используется для объектов-функций (статья 21), таких как `Collections.reverseOrder`, но он также используется и для коллекций, например, `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 идентичности* (identity function), которая просто возвращает свой аргумент. Будет напрасной тратой времени создавать новую функцию каждый раз, когда она потребуется. Если бы обобщённые типы были реифицируемыми, то вам бы понадобилась своя функция идентичности для каждого типа, но поскольку обобщённые средства «стираются», то вам будет достаточно синглтона. Это выглядит так:

```
// Шаблон обобщённой фабрики-синглтона
private static UnaryFunction<Object> IDENTITY_FUNCTION =
    new UnaryFunction<Object>() {
        public Object apply(Object arg) { return arg; }
    };

// IDENTITY_FUNCTION не имеет состояния, и ее параметры не являются
// ограниченными, так что безопасно
// использовать общий экземпляр для всех типов
@SuppressWarnings("unchecked")
public static <T> UnaryFunction<T> identityFunction() {
    return (UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

Приведение типа `IDENTUTY_FUNCTION` к `(UnaryFunction<T>)` выдаёт предупреждение о непроверяемом приведении типов, поскольку `UnaryFunction<Object>` не является `UnaryFunction<T>` для любого другого типа `T`. Но функция идентичности особая: она возвращает свои аргументы неизменными, так что мы знаем, что безопасно использовать этот же объект как `UnaryFunction<T>` вне зависимости от значения `T`. Следовательно, мы можем с уверенностью подавить предупреждение, сгенерированное приведением типов. После того, как мы сделаем это, код скомпилируется без ошибок или предупреждений.

Комментарий. В Java 8 аналогичный интерфейс включён в стандартную библиотеку под названием `UnaryOperator`, а статический метод `UnaryOperator.identity` на самом деле реализован лямбда-выражением:

```
    static <T> UnaryOperator<T> identity() {
        return t -> t;
    }
```

Приведём пример программы, использующей наш обобщённый синглтон как `UnaryFunction<String>` и `UnaryFunction<Number>`. Как обычно, он не содержит приведений типов и компилируется без ошибок и предупреждений:

```
// Sample program to exercise generic singleton
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 это и встречается относительно редко, чтобы параметр типа был ограничен неким выражением, включающим в себя сам этот параметр. Это явление известно под названием *рекурсивное ограничение типа* (recursive type bound). 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.

Комментарий. В англоязычной литературе встречаются также термины *curiously recurring template pattern* (слово “template” здесь обозначает конструкцию языка C++ для поддержки обобщённого программирования) и *F-bound polymorphism*.

Есть много методов работы со списками элементов, которые реализуют `Comparable` для сортировки списка, поиска внутри его, расчёта 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 сопоставимыми. Вот как можно выразить это ограничение:

```
// Используем рекурсивное ограничение типа для выражения взаимной
// сравнимости элементов списка.
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 сравнён с самим собой», что более или менее точно передаёт явление взаимной сравнимости.

Приведём реализацию объявленного выше метода. Этот метод 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;
}
```

Комментарий. По-хорошему параметр типа `T` здесь должен быть объявлен как `<T extends Comparable<? super T>>` по причинам, объясняемым в статье 28. Кроме того, этот метод выбросит исключение `NoSuchElementException` в случае, если список пуст. По историческим причинам так же поступает стандартный метод `Collections.max`. Начиная с Java 8, по-хорошему следовало бы объявить метод как возвращающий `Optional<T>` и возвращать пустое значение для пустого списка. Именно так делает метод `max` интерфейса `Stream`.

Рекурсивное ограничение типа может быть намного сложнее, чем представлено здесь, но, к счастью, оно используется не слишком часто. Если вы понимаете идиому и ее вариант с подстановочным типом (статья 28), то сможете справиться со многими рекурсивными ограничениями типов, которые встретятся на практике.

Подведём итоги. Обобщённые методы, как и обобщённые типы, более безопасны и более легки в использовании, чем методы, требующие от клиентов приведения типов входных параметров и возвращаемых значений. Как и с типами, вы должны быть уверены, что ваши новые методы можно будет использовать без приведений типа, что зачастую будет означать, что их нужно будет объявить как обобщённые. Подобно работе с типами, вам следует сделать обобщёнными существующие методы, чтобы облегчить жизнь новым пользователям и не сломать код существующих клиентов (статья 23).

Статья 28. Используйте ограниченные подстановочные типы для увеличения гибкости API

Как было отмечено в статье 25, параметризованные типы являются *инвариантными* (invariant). Другими словами, для любых двух различных типов `Type1` и `Type2` тип `List<Type1>` не является ни подтипом, ни супертипом для `List<Type2>`. Хотя то, что `List<String>` не является подтипом `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. Ч тобы освежить вашу память, вот его открытый API:

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Предположим, что мы хотим добавить метод, который берет последовательность элементов и вставляет их в стек. Сделаем первую попытку:

```
// Метод pushAll без использования подстановочных типов - 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 вызываете `push(intVal)`, где `intVal` принадлежит к типу `Integer`. Это работает, потому что `Integer` является подтипом `Number`. Так что, логически, 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, параметризованные типы 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 даёт нам специальный вид параметризованных типов, называемый *ограниченными подстановочными типами* (bounded wildcard types), чтобы

справляться с подобными ситуациями. Тип входных параметров для `pushAll` должен быть не «`Iterable` от `E`» , а «`Iterable` некоего подтипа `E`», и есть подстановочный тип, означающий именно это: `Iterable<? extends E>`. (Использование ключевого слова `extends` слегка запутывает: вспоминаем из статьи 26, что подтип определяется таким образом, что каждый тип является подтипом самого себя.) Давайте изменим `pushAll` так, чтобы он использовал этот тип:

```
// Подстановочный тип для параметра, служащего производителем E
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

С этими изменениями без предупреждений компилируется не только класс `Stack`, но также и любой клиентский код, который бы не компилировался с первым объявлением метода `pushAll`. Поскольку и `Stack` и его клиенты компилируются чисто, то вы знаете, что они типобезопасны.

Теперь предположим, что вы хотите написать метод `popAll` для симметрии 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 подстановочного типа - имеет недостатки!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

Опять-таки, всё компилируется чисто и работает прекрасно, если тип элементов коллекции назначения точно совпадает с типом элементов стека. 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. Поэтому разве не должно быть возможности написать и вот так?

```
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 подстановочный тип даёт нам выход. Тип входного параметра для popAll должен быть не «коллекцией E», а «коллекцией некоего супертипа E» (где супертип определён таким образом, что E является супертипом для самого себя [JLS, 4.10]). И снова есть подстановочный тип, обозначающий именно это: `Collection<? super E>`. Давайте изменим метод `popAll` так, чтобы использовать его:

```
// Подстановочный 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 подстановочные типы для входных параметров, представляющих производителей или потребителей. Если входные параметры являются одновременно и производителем и потребителем, тогда подстановочные 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 подстановочный тип использовать:

PECS значит производитель — `extends`, потребитель — `super`
(producer — `extends`, consumer — `super`).

Другими словами, если параметризованный тип представляет производителя `T`, используйте `<? extends T>`, а если он представляет потребителя `T`, используйте `<? super T>`. В нашем примере с классом `Stack` параметр `src` метода `pushAll` производит экземпляры `E` для использования стеком, поэтому подходящим типом для `src` будет `Iterable<? extends E>`. Параметр `dst` метода `popAll` потребляет экземпляры `E` из стека, поэтому соответствующим типом для `dst` будет `Collection<? super E>`. Схема PECS демонстрирует фундаментальный принцип, который управляет использованием подстановочных типов. Naftalin и Wadler называют это *Get and Put Principle* [Naftalin07,

2.4].

Имея в голове эту схему, давайте посмотрим на объявление некоторых методов из предыдущих статей. Метод `reduce` в статье 25 объявлен так:

```
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 параметр `list` только как производителя `E`, так что его объявление должно использовать подстановочный тип с `extends E`. Параметр `f` представляет функцию, которая как потребляет, так и производит экземпляры `E`, так что ее подстановочный тип не подойдёт для этого. Приведём пример получившегося объявления метода:

```
// Подстановочный 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 свернуть его с помощью `Function<Number>`. С оригинальным объявлением это компилироваться не будет, однако если вы добавите ограниченный подстановочный тип, то такой код скомпилируется.

Давайте рассмотрим метод `union` из статьи 27. Вот яго объявление:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Оба параметра s1 и s2 являются производителями E, так что схема PECS говорит нам, что объявление должно иметь вид:

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

Обратите внимание, что возвращаемый тип — по-прежнему `Set<E>`. **Не используйте подстановочный тип в качестве возвращаемого типа.** Вместо того, чтобы давать больше гибкости пользователям, он будет заставлять их использовать подстановочные типы в клиентском коде.

При правильном использовании подстановочные типы 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 подстановочных типах, 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. Если вы посмотрите на пересмотренное объявление метода `union`, можно подумать, что можно сделать 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);
                              ^
```

Комментарий. Со временем компилятор Java всё-таки становится умнее. Как минимум начиная с Java 8 этот код уже компилируется.

К счастью, есть способ справиться с подобными ошибками. Если компилятор не выводит типы, которые вы хотите, тогда скажем ему, какие типы использовать, с помощью *явного параметра типа* (explicit type parameter). Это придётся делать не так часто, что хорошо, так как явные параметры типа не очень красивы. С явным параметром программа компилируется без ошибок:

```
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Теперь обратим внимание на метод `max` из статьи 27. Вот его оригинальное объявление:

```
public static <T extends Comparable<T>> T max(List<T> list)
```

Вот переделанное объявление, использующее подстановочные типы:

```
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list)
```

Для получения изменённого объявления мы дважды применили схему PECS. Сначала мы использовали ее для параметра `list`. Он производит экземпляры `T`, поэтому мы заменили тип с `List<T>` на `List<? extends T>`. Рассмотрим более сложное использование этой схемы на параметре типа `T`. Здесь мы впервые увидели пример использования подстановочного типа на параметре типа. Изначально `T` расширял `Comparable<T>`, но `Comparable<T>` потребляет экземпляры `T` (и производит целые числа, отражающие порядковые отношения). Следовательно, параметризованный тип `Comparable<T>` заменяется ограниченным подстановочным типом `Comparable<? super T>`. Объекты `Comparable` всегда являются потребителями, так что **всегда следует использовать Comparable<? super T>** вместо `Comparable<T>`. То же самое относится и к компараторам, так что **вы всегда должны использовать `Comparator<? super T>` вместо `Comparator<T>`**.

> **Комментарий.** Вообще говоря, *чистые функции* (pure functions) являются потребителями своих аргументов и производителями своих возвращаемых значений. Поэтому, например, API, использующие функциональные интерфейсы `Supplier<T>`, `Consumer<T>` и `Function<T, R>`, обычно объявляют типы соответствующих параметров как `Supplier<? extends T>`, `Consumer<? super T>` и `Function<? super T, ? extends R>`.

Изменённое объявление метода `max`, вероятно, является самым сложным объявлением метода во всей книге. Даёт ли нам что-то эта увеличенная сложность? Да, даёт. Вот простой пример списка, который не будет работать с оригинальным объявлением метода, но будет с изменённым:

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

Причина, почему вы не можете применить оригинальное объявление метода к данному списку, состоит в том, что `java.util.concurrent.ScheduledFuture` не реализует `Comparable<ScheduledFuture>`. Вместо этого он является подинтерфейсом `Delayed`, который расширяет `Comparable<Delayed>`. Другими словами, экземпляр `ScheduledFuture` не просто сравним с другими экземплярами `ScheduledFuture`; он сравним с любым экземпляром `Delayed`, и этого достаточно, чтобы оригинальный метод `max` не смог работать с такими объектами.

Есть одна небольшая проблема с изменённым объявлением метода `max`: оно не даёт методу компилироваться. Вот метод с изменённым объявлением:

```
// Не компилируется - подстановочным типам 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 произойдёт 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? Оно значит, что `list` не является экземпляром `List<T>`, поэтому его метод `iterator` не возвращает `Iterator<T>`. Он возвращает `iterator` некоего подтипа `T`, так что нам нужно заменить объявленный тип переменной `i` на ограниченный подстановочный тип:

```
Iterator<? extends T> i = list.iterator();
```

Это единственное изменение, которое нужно сделать в теле метода. Элементы, возвращаемые методом `next` итератора, являются некими подтипами `T`, так что они могут быть безопасно сохранены в переменной типа `T`.

Есть ещё одна заслуживающая обсуждения тема, связанная с подстановочными типами. Есть некоторая двойственность между параметрами для типов и подстановочными типами, так как многие методы могут объявляться с использованием либо одного, либо

другого. Например, вот два возможных объявления статического метода перестановки двух индексированных пунктов в списке. Первый использует неограниченный параметр типа (статья 27), а второй использует неограниченный подстановочный тип:

```
// Два возможных объявления метода swap
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Какое из двух объявлений предпочтительнее, и почему? В открытом API второе лучше, потому что оно проще. Вы передаёте список — любой список — и метод переставляет элементы по указанным индексам. Не надо волноваться о параметрах. Как правило, **если параметр типа появляется только один раз в объявлении метода, следует заменить его подстановочным типом**. Если это неограниченный параметр типа, замените его неограниченным подстановочным типом, если же это ограниченный параметр типа, замените его ограниченным подстановочным типом.

Есть одна проблема со вторым объявлением метода `swap`, использующим подстановочный тип вместо параметра типа: его прямолинейная реализация не будет компилироваться.

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Попытка компиляции приведёт 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 странным, что мы не можем поместить элемент обратно в список, из которого мы его только что взяли. Проблема в том, что типом переменной `list` является `List<?>`, и вы не можете поместить в `List<?>` никакое значение, кроме `null`. К счастью, есть способ реализовать метод, не прибегая ни к небезопасному приведению типов, ни к сырым типам. Идея в том, чтобы написать вспомогательный обобщённый метод, чтобы *захватить* (capture) подстановочный 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 подстановочного типа
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i))):
}
```

Метод `swapHelper` знает, что `list` — это `List<E>`. Следовательно, он знает, что любое значение, которое он получает из этого списка, является экземпляром типа `E` и что безопасно поместить любое значение типа `E` в список. Эта несколько запутанная реализация `swap` компилируется без ошибок. Она позволяет экспортировать метод `swap` на основе подстановочных типов, одновременно пользуясь более сложным обобщённым методом изнутри. Клиентам метода `swap` не придётся сталкиваться с более сложным методом `swapHelper`.

Подведём итоги. Использование подстановочных типов в вашем API, хотя и замысловато, делает API более гибким. Если вы напишете библиотеку, которая будет широко использоваться, то правильное использование подстановочных типов должно быть в ней обязательным. Помните основное правило: схема PECS (производитель — `extends`, потребитель — `super`). И помните, что все объекты `Comparable` и `Comparator` являются потребителями.

Статья 29. Рассмотрите возможность использования типобезопасных неоднородных контейнеров

Средства обобщённого программирования чаще всего используются для коллекций, таких как `Set` и `Map`, или для одноэлементных контейнеров, таких как `ThreadLocal` и `AtomicReference`. Во всех этих случаях параметры типа задаются именно контейнерам. Это ограничивает вас до определённого количества параметров на один контейнер. Обычно это именно то, что нам нужно. У типа `Set` есть один параметр типа, представляющий его элементы; у типа `Map` есть два, представляющие типы его ключей и 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 типобезопасный доступ. К счастью, есть лёгкий способ это осуществить. Идея в том, чтобы параметры типа задать *ключу*, а не *контейнеру*. Затем нужно представить параметризованный ключ контейнеру, чтобы добавить или извлечь значение. Система обобщённых 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 «избранные» экземпляры произвольно большого количества классов. Объект `Class` будет играть роль параметризованного ключа. Причина, по которой это работает, состоит в том, что класс `Class` был обобщён в версии 1.5. Тип литерала класса теперь не просто `Class`, а `Class<T>`. Например, `String.class` относится к типу `Class<String>`, а класс `Integer.class` относится к типу `Class<Integer>`. Когда литерал класса передаётся между методами, чтобы обращаться к информации во время компиляции и во время выполнения, то это называется *метка типа* (type token) [Bracha04].

API класса `Favorites` прост. Он выглядит как простой словарь, только параметризован не сам словарь, а ключ. Клиент представляет объект `Class` при установке и получении элементов словаря. Вот как выглядит API:

```
// Шаблон типобезопасного неоднородного контейнера - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
```

Приведём пример программы, которая использует класс `Favorites`, сохраняя, извлекая и печатая экземпляры `String`, `Integer` и `Class`:

```
// Шаблон типобезопасного 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 класса `Favorites` является *типобезопасным*: он никогда не вернёт `Integer`, если вы просите его вывести String. Он также является *неоднородным*: все ключи являются ключами разных типов, в отличие от обычного словаря. Следовательно, мы можем назвать `Favorites` *типобезопасным неоднородным контейнером*.

Реализация класса `Favorites` на удивление коротка. Вот она полностью:

```
// Шаблон типобезопасного 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 есть несколько тонкостей. Каждый экземпляр `Favorites` поддерживается закрытым объектом `Map<Class<?>, Object>`, который называется `favorites`. Вы можете подумать, что здесь невозможно поместить что-либо в `Map` из-за неограниченного подстановочного типа, но на самом деле все наоборот. Дело в

том, что подстановочный тип является вложенным: подстановочным является не тип самого объекта `Map`, а тип его ключа. Это значит, что у каждого ключа могут быть разные параметризованные типы: один может быть `Class<String>`, другой `Class<Integer>` и т.д. Вот откуда берётся неоднородность.

Также необходимо отметить, что типом значения `favorites` является просто `Object`. Другими словами, в данном случае объект `Map` не является гарантом того, 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 пользуемся этим при получении сохранённых в словаре объектов.

Реализация метода `putFavorite` довольно тривиальна: она просто помещает в `favorites` связь между объектом `Class` и экземпляром соответствующего класса. Как было отмечено, это устраняет «связь типов» между ключом и значением; утрачивается информация о том, что значение есть экземпляр ключа. Но это нормально, потому что метод `getFavorites` может и должен восстанавливать эту связь.

Реализация метода `getFavorite` более замысловата, чем у метода `putFavorite`. Во-первых, она берет из словаря `favorites` значение, соответствующее данному объекту `Class`. Это именно та ссылка на объект, которую мы должны вернуть, но у неё неверный тип времени компиляции. Ее тип — просто `Object` (тип значения словаря `favorites`), а нам нужно возвратить `T`. Таким образом, реализация метода `getFavorite` выполняет *динамическое приведение типа* (dynamic type cast) к типу, представленному объектом `Class`, с использованием метода `cast` класса `Class`.

Метод `cast` — это динамический аналог оператора приведения типа языка Java. Он просто проверяет, что его аргумент является экземпляром типа, представленного объектом `Class`. Если так, он возвращает аргумент; в противном случае он выбрасывает `ClassCastException`. Реализация метода `getFavorite` гарантирует, что он никогда не выбросит `ClassCastException`, если клиентский код компилируется без предупреждений, потому что мы знаем, что значения в карте `favorites` всегда соответствуют типам ключей.

Итак, что даёт нам метод `cast`, если он просто возвращает свой аргумент? Объявление (сигнатура) метода `cast` максимально использует преимущества того факта, что класс `Class` является обобщённым. Его возвращаемый тип — это параметр типа объекта `Class`:

```
public class Class<T> {
    T cast(Object obj);
}
```

Это в точности, что нам нужно от метода `getFavorite`. Это то, что позволит нам сделать типобезопасным класс `Favorites`, не прибегая к непроверяемому приведению к типу T.

У класса Favorites есть два ограничения, которые стоит упомянуть. Во-первых, злонамеренные клиенты могут легко сломать типобезопасность экземпляра `Favorites`, просто используя объект `Class` в его сырой форме. Но получившийся клиентский код выведет предупреждения о непроверяемых передачах при компиляции. Так же ведут себя обычные реализации коллекций, такие как `HashSet` и `HashMap`. Вы можете легко поместить `String` в `HashSet<Integer>`, используя сырой тип `HashSet` (статья 23). Тем не менее вы можете обеспечить типобезопасность при выполнении, если вы согласны заплатить за неё. Единственный способ убедиться, что `Favorites` никогда не нарушит инварианты своих типов, — это заставить метод `putFavorite` проверить, что объект `instance` действительно является экземпляром типа, представленным объектом `type`. И мы уже знаем, как это сделать: использовать динамическое приведение типа.

```
// Типобезопасность обеспечивается динамическим приведением типа
public <T> void putFavorite(Class<T> type, T instance) {
    favorites.put(type, type.cast(instance));
}
```

В классе `java.util.Collections` имеются обёртки для коллекций, которые делают то же самое. Они называются `checkedSet`, `checkedList`, `checkedMap` и т.д. Их статические фабричные методы берут объект `Class` (или два объекта) плюс коллекцию (или словарь). Эти статические фабрики являются обобщёнными и гарантируют, что все элементы коллекции будут соответствовать типу объекта `Class` во время компиляции, то есть они добавляют реификацию типов к коллекции, которую они оборачивают. Например, возвращённая таким методом обёртка выбрасывает `ClassCastException` во время выполнения, 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 обобщённый код и старый код, не работающий с обобщёнными типами.

Второе ограничение класса `Favorites` заключается в том, что он не может

использоваться с нереифицируемыми типами (статья 25). Другими словами, вы можете хранить избранный экземпляр `String` или `String[]`, но 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>`: выражение `List<String>.class` синтаксически неверно, но, с другой стороны, это и хорошо. `List<String>` и `List<Integer>` используют один общий объект `Class`, а именно `List.class`. Он бы натворил немало бед внутренностям объекта `Favorites`, 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 *суперметками типа* (super type tokens), которая обходит это ограничение по-своему, однако она сама имеет свои ограничения [Gafter07].

Комментарий. Исторически разные библиотеки обходили это ограничение по-разному. Наиболее популярным подходом является задание меток типа с помощью объявления анонимных классов, сохраняющих во время выполнения информацию о параметрах типа для своего супертипа. Такой подход используют классы `TypeToken` в библиотеке Guava, `TypeLiteral` в Guice и `TypeReference` в Jackson.

В библиотеке Guava есть контейнер `ClassToInstanceMap`, являющийся расширенной версией вышеприведённого класса `Favorites` и страдающий от того же ограничения при использовании параметризованных типов в качестве ключей, а также контейнер `TypeToInstanceMap`, который принимает в качестве ключей как объекты типа `Class`, так и объекты типа `TypeToken`, и потому он способен корректно работать и с параметризованными типами.

```
TypeToInstanceMap map = new TypeToInstanceMap();
map.putInstance(String.class, "Java");
// Создание TypeToken с помощью анонимного класса
map.putInstance(new TypeToken<List<String>>() {},
    Arrays.asList("Effective", "Java"));
```

Метки типа, используемые классом `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* (bounded type token), которая просто является меткой типа, устанавливающей ограничения на то, какие типы могут быть представлены, используя ограниченный параметр типа (статья 27) или ограниченный подстановочный

тип (статья 28).

API аннотаций (статья 35) широко использует ограниченные метки типа. Например, вот метод для чтения аннотаций во время выполнения. Этот метод объявлен в интерфейсе `AnnotatedElement`, который реализуется типами из API рефлексии (reflection), представляющими классы, методы, поля и другие элементы программы:

```
public <T extends Annotation>
    T getAnnotation(Class<T> annotationType);
```

Аргумент `annotationType` является ограниченной меткой типа, представляющей тип аннотации. Метод возвращает аннотацию данного типа, если таковая имеется, или `null`, если нет. В сущности, аннотированные элементы являются безопасными неоднородными контейнерами, ключи которых являются типами аннотаций.

Предположим, что у вас есть объект типа `Class<?>` и вы хотите передать его методу, которому требуется ограниченная метка типа, такому как `getAnnotation`. Вы можете привести тип объекта к `Class<? extends Annotation>`, но это приведение типов непроверяемо, поэтому будет выведено предупреждение во время компиляции (статья 24). К счастью, класс `Class` даёт нам метод экземпляра, который безопасно (и динамически) решает эту задачу. Метод называется `asSubclass` и приводит объект `Class`, у которого он вызван, к типу, представляющему подкласс класса, представленного его аргументом. Если приведение типов удаётся, то метод возвращает свой аргумент, если нет, то выбрасывается `ClassCastException`.

Вот как метод `asSubclass` используется 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 (ReflectiveOperationException ex) {
        throw new IllegalArgumentException(ex);
    }
```

```
    return element.getAnnotation(
        annotationType.asSubclass(Annotation.class));
}
```

Подведём итог. Нормальное использование средств обобщённого программирования, представленное API коллекций, ограничивает вас фиксированным количеством параметров типа на контейнер. Вы можете обойти это ограничение, поместив параметр типа в ключ вместо контейнера. Вы можете использовать объекты `Class` в качестве ключей для таких типабезопасных неоднородных контейнеров. Объект `Class`, используемый таким образом, называется меткой типа. Вы можете также использовать свой собственный тип ключа. Например, вы можете представить типом `DatabaseRow` строку базу данных (которая и будет контейнером) и использовать обобщённый тип `Column<T>` в качестве её ключа.

Глава 6. Перечислимые типы и аннотации

В версии 1.5 в язык Java были добавлены две группы ссылочных типов: новый вид класса, называемый *перечислимым типом* (enum type), и новый вид интерфейса, называемый *аннотационным типом* (annotation type). В этой главе обсуждаются наилучшие способы использования этих двух групп типов.

Статья 30. Используйте перечислимые типы вместо констант int

Перечислимый тип — это тип, разрешённые 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 имитировались шаблоном группы именных констант `int`, с одной константой на каждый элемент перечисления:

```
// Шаблон перечисления int (int enum) - 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 *шаблон перечисления `int`* (int enum pattern), 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 и того хуже:

```
// Вкусный яблочный соус, приправленный цитрусом!
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 имён для перечислимых групп `int`. Префиксы препятствуют совпадению имён, если две перечислимые группы `int` будут иметь константы с одинаковыми названиями.

Программы, использующие шаблон перечисления `int`, довольно хрупки. Поскольку перечисления `int` являются константами на время компиляции, их значения встраиваются в клиентские классы, 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 лёгкого 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 надёжного 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 *шаблон перечисления `String`* (String enum pattern), ещё хуже. В то время как он даёт печатаемые строки для своих констант, он может привести к проблемам с производительностью, потому что полагается на сравнение строк. Что ещё хуже, он может привести неопытных пользователей к непосредственному объявлению (hardcode) строковых значений в клиентском коде вместо использования наименований полей. Если такие скопированные в клиентский код строковые константы содержат типографическую ошибку, то она будет пропущена при компиляции и в результате проявится при выполнении.

К счастью, в версии 1.5 язык даёт нам альтернативу, которая помогает избежать недостатков шаблонов перечислений `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 завершённое поле. Перечислимые типы являются де факто `final` (effectively final) благодаря тому, что у них нет доступных извне конструкторов. Клиенты не могут ни создавать экземпляры перечислимых типов, ни расширять их, поэтому не может существовать никаких других экземпляров перечислимого типа, кроме объявленных констант. Другими словами, перечислимые типы используют контроль экземпляров. Они являются обобщением синглтона (статья 3), которые являются по сути перечислимым типом из одного элемента. Для читателей, знакомых с первой редакцией этой книги, перечислимые типы предоставляют поддержку шаблона “типобезопасное перечисление” (typesafe enum) на уровне языка [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 трёх разрешённых значений `Apple`. Попытка передать значение неверного типа приведёт 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 сосуществуют «мирно» потому, что у каждого типа имеется своё пространство имён. 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 клиентские классы, как в случае с шаблоном перечисления `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 шаблона перечисления `int`, они позволяют нам добавлять произвольные методы и поля, а также реализовывать произвольные интерфейсы. Они предлагают высококачественные реализации всех методов `Object` (глава 3), реализуют `Comparable`

(статья 12) и `Serializable` (глава 11), а их сериализованная форма спроектирована так, чтобы быть устойчивой к большей части 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, возвращающего цвет фрукта или его изображение. Вы можете добавить к перечислимому типу любой нужный метод. Перечислимый тип может начать жизнь, как простая коллекция перечислимых констант, и развиться 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 принимает данные и сохраняет их в полях.** Перечислимые типы по своей природе неизменяемы, так что все поля должны быть объявлены как `final` (статья 15). Они могут быть открытыми, но лучше сделать их закрытыми и снабдить открытыми методами 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 объявленные названия каждого значения перечислимого типа, давая возможность их легко распечатать методами `println` и `printf`. Если вас не устраивает 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 своего аналога в C в том, что использует `%n` там, где в C используют `\n`.

Комментарий. Это связано с тем, что язык C гарантирует, что при записи строк в текстовый поток вывода символ `\n` преобразуется в системный разделитель строк. В Java же символ `\n` имеет чётко установленное значение `U+000A` (*перевод строки*, line feed, LF). В Unix-подобных системах разделитель строк состоит из одного символа LF (`\n`), а в Windows — из двух символов CR+LF (`U+000D U+000A`, либо `\r\n`), где CR означает *возврат каретки* (carriage return). Спецификация `%n` указывает методу `printf` и его близкому родственнику `String.format` вставить разделитель строк, соответствующий платформе, на которой выполняется Java-программа. Вне этих методов его можно получить, вызвав метод `System.lineSeparator`.

Кстати, сигнатура формата PNG специально спроектирована так, чтобы выявить ошибки при автоматическом преобразовании разделителей строк в C-программах. Каждый PNG-файл начинается с определённых восьми байт:
`89 50 4E 47 0D 0A 1A 0A`
Если C-программа под Windows случайно откроет PNG-файл как текстовый, то `0D 0A` будет прочитано как `0A` (`\r\n` схлопнется в `\n`). Если же она случайно запишет PNG-файл в текстовом режиме вместо двоичного, то, наоборот, последнее `0A` запишется как `0D 0A` (`\n` развернётся до `\r\n`). В обоих случаях сигнатура файла повредится, и программы и библиотеки, работающие с PNG-файлами, отбракуют такой файл.

Некоторые особенности, связанные с перечислимыми константами, возможно необходимо использовать только в рамках класса или пакета, в котором определён перечислимый тип. Такие особенности лучше всего реализованы как закрытые или доступные в пределах пакета методы. В этом случае каждая константа позволит классу или пакету, содержащему перечислимый тип, должным образом реагировать на появление этой константы. Как и с другими классами, если у вас нет причины открывать перечислимый метод, объявите его как закрытый или, если необходимо, доступный только в пределах пакета (статья 13).

Если перечислимый тип полезен в отрыве от других типов, то он должен быть классом верхнего уровня; если его использование привязано к классу верхнего уровня, то он должен быть классом-членом класса верхнего 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`. Сделав `RoundingMode` перечислимым типом верхнего уровня, разработчики библиотеки подтолкнули программистов, которым потребуется задание режимов округления, переиспользовать этот перечислимый тип, что приводит к улучшению согласованности между разными API.

Технология, показанная в примере с `Planet`, достаточна для большинства перечислимых типов, но иногда вам требуется большее. С каждой константой `Planet` связаны различные данные, но иногда требуется связать фундаментально различное *поведение* с каждой константой. Например, предположим, вы пишете перечислимый тип, представляющий операции на основе калькулятора четырёх функций, и вы хотите снабдить его методом для выполнения арифметических операций, представленным

каждой коллекцией. Одним из способов достижения этого является инструкция `switch` по значению перечислимого типа:

```
// Перечислимый тип, использующий switch на своё 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 ещё хуже, этот код довольно хрупкий. Если вы добавите новую перечислимую константу, но забудете добавить соответствующий блок `case` к `switch`, перечислимый тип все равно откомпилируется, но даст ошибку при выполнении, когда вы попытаетесь выполнить новую операцию.

К счастью, есть более удачный способ ассоциации различных реакций с каждой перечислимой константой: объявите абстрактный метод `apply` в перечислимом типе и переопределите его конкретным методом для каждой константы в *константоспецифичном теле класса* (constant-specific class body). Такие методы известны как *константоспецифичные реализации методов* (constant-specific method implementations):

```
// Перечислимый тип с константоспецифичными реализациями методов
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 любым объявлением константы. В случае, если все же забудете, компилятор вам об этом напомнит, поскольку абстрактные методы в перечислимом типе должны быть переопределены конкретными методами во всех его константах.

Реализация методов, привязанная к константам, может сочетаться с данными, привязанным к константам. Например, вот версия `Operation`, которая переопределяет метод `toString` для возвращения символа, обычно связанного с операцией:

```
// Перечислимый тип с константоспецифичными телами класса и данными
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 своих собственных строковых представлений обратно в соответствующие перечислимые типы. Следующий код (с изменёнными должным образом именами типов) будет выполнять эту операцию для любого перечислимого типа до тех пор, пока у каждой константы есть уникальное строковое представление:

```
// Implementing a fromString method on an enum type
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 словарь `stringToEnum` из блока `static`, который запускается после создания констант. 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 исключению `NullPointerException`. Конструкторам перечислимых типов не разрешён доступ к статическим полям перечислимых типов, кроме полей констант времени компиляции. Это ограничение необходимо, потому что эти статические поля ещё не инициализированы на момент запуска конструктора.

Недостаток константоспецифичных реализаций методов 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, рассчитывающий 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, в том числе в выходные, должно генерировать повышенную оплату. С выражением `switch` легко выполнить расчёт, применяя различные метки `case` 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 расчёта оплаты труда (статья 48):

```
// Перечислимый тип, переключающийся на своё 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 сверхурочные часы
        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 блок `case` 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 расчёты оплаты с помощью реализации зависимого от констант метода, вам нужно дублировать расчёт оплаты переработки для каждой константы или переместить расчёт в два вспомогательных метода (один для рабочих дней, другой для выходных) и запустить подходящий вспомогательный метод для каждой константы. Любой подход приведёт к большому объёму кода, существенно уменьшая его читаемость и увеличивая возможность ошибки.

Шаблонный код можно уменьшить, заменив абстрактный метод `overtimePay` перечислимого типа `PayrollDay` конкретным методом, который выполняет расчёт переработки в выходные дни. Но это приведёт к тому же недостатку, что и выражение `switch`: если вы добавите ещё день без переопределения метода `overtimePay`, то расчёт будет вестись неправильно, как оплата в рабочие дни.

Что вам действительно нужно — это чтобы компилятро *заставлял* вас выбирать стратегию оплаты переработки каждый раз при добавлении перечислимой константы. К счастью, есть способ, как этого достичь. Суть в том, чтобы переместить расчёт оплаты переработки в закрытый вложенный перечислимый тип и передать экземпляр этого *перечислимого типа стратегии* (strategy enum) в конструктор перечислимому типу `PayrollDay`. Перечислимый тип `PayrollDay` затем передаст расчёт оплаты переработки перечислимому типу стратегии, избегая необходимости в выражении `switch` или использовании константоспецифичных реализаций методов в `PayrollDay`. Хотя этот шаблон менее краток, чем выражение `switch`, он более безопасен и гибок:

```
// Шаблон "перечислимый тип стратегии"
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);
    }

    // The strategy enum type
    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 инструкции `switch` для перечислимых типов не являются хорошим выбором для реализации реакций в зависимости от констант, для чего же их имеет смысл использовать? **Инструкция `switch` на перечислимых типах хорошо подходит для задания константоспецифичного поведения для *внешних* перечислимых типов**. Например, предположим, что перечислимый тип `Operation` не находится под вашим контролем, и вы хотели бы, чтобы у него был метод экземпляра для возврата противоположности каждой операции. Вы можете смоделировать данный эффект следующим статическим методом:

```
// switch 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 сравнимы по производительности 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 «естественные перечислимые типы», такие как планеты, дни недели и шахматные фигуры. Но это также включает в себя другие наборы, для которых вы знаете все во зможные значения во время компиляции, такие как выбор пунктов меню, кодов операций и флагов командной строки. Нет необходимости, чтобы набор констант в перечислимом типе оставался неизменным все время. Функционал перечислимых типов специально был разработан с тем расчётом, чтобы позволять им развиваться с сохранением двоичной совместимости.

Подведём итоги. Преимущества перечислимых типов по сравнению с константами `int` сложно переоценить. Перечислимые типы хорошо читаемы, безопасны и имеют больше возможностей. Многим перечислимым типам не требуются явные конструкторы и члены, но многие другие извлекают пользу от ассоциации данных с каждой из констант и предоставления методов, на поведение которых влияют данные. Намного меньшему числу перечислимых типов требуется реализация разного поведения одного и того же метода. В этом относительно редком случае предпочитайте константоспецифичные методы. Рассмотрите возможность использования шаблона “перечисляемые тип стратегии”, если у нескольких перечислимых констант есть общее поведение.

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 порядковый номер каждой

перечислимой константы своего типа. Вы можете столкнуться с искушением вывести ассоциированное значение `int` из порядкового номера:

```
// Злоупотребление методом ordinal для выведения
// 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, поддержка такого решения может стать кошмаром. Если поменять порядок констант, метод `numberOfMusicians` будет сломан. Вы не сможете добавить ещё одну константу с тем же значением `int`, что и одна из уже имеющихся. Например, нет способа добавить к такому перечислению константу для *двойного квартета*, который, как и октет, состоит из восьми музыкантов.

Если вы захотите добавить вторую константу перечисления, связанную со значением `int`, которую вы уже использовали, то вам снова не повезёт. Например, может быть прекрасно, если вы захотите добавить константу, представляющую *тройной квартет*, состоящий из 12 музыкантов. Нет стандартного термина для ансамбля, состоящего из 11 музыкантов, так что вам придётся добавить константу для неиспользованного значения `int` (11). В лучшем случае это ужасно выглядит. Если неиспользованными остаются много значений `int`, то это непрактично.

К счастью, есть простое решение этих проблем. **Никогда не выводите значение, связанное с перечислимым типом, из его порядкового номера; вместо этого храните его в поле экземпляра:**

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
```

```
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

Спецификация класса `Enum` говорит о методе `ordinal`: «Большинству программистов этот метод не понадобится. Он создан для использования структурами данных общего назначения на основе перечислимых типов, такими как `EnumSet` и `EnumMap`». Если 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 во множествах (set), то традиционно используется шаблон перечисления `int` (статья 30) с присвоением каждой константе своей степени двойки:

```
// 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 во множество, известное 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 новые сверх того. 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 ещё используют битовые поля, если им необходимо обойти множества констант. Нет нужды так делать — есть лучшая альтернатива. Пакет `java.util` даёт нам класс `EnumSet` для эффективного представления множества значений, извлечённых из одного перечислимого типа. Этот класс реализует интерфейс `Set`, давая вам всю мощность, типобезопасность и совместимость, которые вы можете получить от 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` представляется одним значением `long`, так что его производительность сравнима с производительностью битового поля. Групповые операции, такие как `removeAll` и `retainAll`, реализуются с использованием побитовой арифметики, так же, 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 за 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 передаёт экземпляр `EnumSet` методу `applyStyles`. `EnumSet` даёт богатый набор статических фабричных методов для упрощения создания множеств, один из которых приведён в данном коде:

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

Обратите внимание, что метод `applyStyles` принимает `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 даёт возможность необычному клиенту передать в метод другую реализацию `Set` и не обладает никакими недостатками.

Подведём итоги. **Нет необходимости представлять перечислимые типы битовыми полями только потому, что они будут использоваться во множествах.** 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.

Комментарий. В библиотеке Guava есть статический фабричный метод `Sets.immutableEnumSet`, возвращающий неизменяемое множество, реализованное поверх `EnumSet`.

Статья 33. Используйте EnumMap вместо индексирования по порядковому номеру

Вам может попасться код, использующий метод `ordinal` (статья 31) для индексирования внутри массива. Например, рассмотрим простой класс, который должен представлять вид растения:

```
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, двулетние или многолетние).

Для того чтобы это сделать, вы создаёте три множества, по одному для каждого типа, и выполняете итерацию по саду, помещая каждую траву в соответствующее множества. Некоторые программисты сделают это, помещая множества в массив, индексированный по порядковому номеру типа:

```
// ordinal() используется 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 приём работает, но в нем много проблем. Так как массив несовместим с средствами обобщённого программирования (статья 25), программе потребуется приведение к сырому типу и она будет компилироваться с предупреждениями. Поскольку массиву неизвестно, что представляет собой его индекс, то необходимо

пометить результат вручную. Но самая серьёзная проблема состоит в том, что, когда вы получите доступ к массиву, индексированному порядковым перечислимым типом, то на вас ляжет ответственность по использованию 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 повезёт — выбросит `ArrayIndexOutOfBoundsException`.

К счастью, есть намного более удачный способ добиться того же эффекта. Массив здесь, по сути, выступает в роли словаря между перечислимым типом и значением, так что вместо него можно можно использовать `Map`. Заметим, что есть быстрая реализация `Map` для использования с перечислимыми ключами, известная как `java.util.EnumMap`. Вот как выглядит та же программа, переписанная с использованием `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 приведений типа, нет необходимости вручную помечать результат, поскольку ключи словаря — это элементы перечислимого типа, которые знают, как перевести самих себя в печатаемые строки, и нет места для ошибок в расчёте индексов массивов. Причина, по которой `EnumMap` сравним по скорости с массивом, индексированным по порядковому номеру, заключается в том, что `EnumMap` использует такой массив внутри. Но он скрывает детали этой реализации от программиста, объединяя мощность и типобезопасность `Map` со скоростью массивов. Обратите внимание, что конструктор `EnumMap` принимает объект `Class` для типа ключа: это *ограниченная метка типа* (bounded type token), которая предоставляет информацию о типе во время выполнения (статья 29).

Иногда используется массив массивов, индексированный (дважды) по порядковому номеру и используемый для представления словаря с двумя ключами перечислимых

типов. Например, эта программа использует такой массив для представления переходов между агрегатными состояниями вещества (переход из жидкого состояния в твёрдое называется замерзанием, из жидкого в газообразное — кипением и т.д.):

```
// Использует ordinal() для индексирования массива массивов -
// 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 об отношениях между порядковыми номерами 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 принять вид исключения `ArrayIndexOutOfBoundsException` или `NullPointerException`, или, что ещё хуже, программа будет тихо работать некорректно. При этом размер таблицы равен квадрату количества фаз, даже если количество ненулевых значений в таблице будет меньше.

И опять вы можете улучшить этот код с помощью `EnumMap`. Поскольку каждый переход из одной фазы в другую индексируется *парой* объектов `Phase`, то лучше всего представить

отношения в виде словаря от первого перечислимого типа (исходная фаза) в словарь от второго перечислимого типа (фаза, в которую осуществляется переход) к результату (сам переход). Две фазы, связанные с переходом, лучше всего записывать, связывая данные с элементами перечисления `Transition` и затем используя их для инициализации вложенного объекта `EnumMap`:

```
// Использование вложенного объекта 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;
        }

        // Initialize the phase transition map
        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 инициализации словаря может выглядеть сложным, но он не так уж непонятен. Типом этого словаря является `Map<Phase, Map<Phase, Transition>>`, что означает «словарь от исходной фазы к словарю от фазы назначения к переходу». Первый цикл в блоке статической инициализации инициализирует внешний словарь, содержащий три пустых внутренних словаря. Второй цикл в блоке инициализирует внутренние словари, используя информацию об источнике и назначении, ассоциированную с константой перечисления `Transition`.

Теперь предположим, что вы хотите добавить в систему новое агрегатное состояние: *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 придётся добавить одну новую константу к `Phase` и две к `Phase.Transition` и заменить оригинальныи 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 повезёт: программа откомпилируется, но выведет ошибку при выполнении. Для обновления версии на основе `EnumMap` все, что вам надо сделать, — это добавить константу `PLASMA` к перечислению `Phase`, а также константы `IONIZE(GAS, PLASMA)` и `DEIONIZE(PLASMA, GAS)` к перечислению `Transition`. Программа обо всем остальном сама позаботится, и у вас не будет возможности сделать ошибку. Внутри словарь словарей реализуется как массив массивов, так что вы мало что теряете теряете в плане затрат и времени выполнения, взамен получая большую ясность, безопасность и простоту поддержки.

Подведём итоги. **Редко возникает ситуация, когда имеет смысл использовать `ordinal` для индексирования массивов: вместо этого надо использовать `EnumMap`.** Если взаимоотношения, которые вы хотите представить, являются многомерными, используйте `EnumMap<..., EnumMap<...>>`. Это особый случай общего принципа, говорящего, что программистам приложений редко, если вообще, имеет смысл использовать метод `Enum.ordinal` (статья 31).

Статья 34. Имитируйте расширяемые перечислимые типы с помощью интерфейсов

Перечислимые типы почти во всех отношениях превосходят *шаблон типобезопасного перечисления* (typesafe enum pattern), описанный в первой редакции этой книги [Bloch01]. Однако при всем этом, казалось бы, имеется одно исключение. Оно касается расширяемости, которая была возможна в оригинальном шаблоне, но не поддерживается конструкцией `enum`. Другими словами, при использовании шаблона можно было сделать так, чтобы один перечислимый тип расширял другой; при использовании же конструкции `enum` это невозможно. Это не случайно. По большей части расширение перечислимых типов оказывается плохой идеей. Сбивает с толку то, что элементы расширяющего типа являются экземплярами основного типа, но не наоборот. Нет хорошего способа перечислить все элементы основного типа и его расширения. Наконец, расширяемость усложняет многие аспекты дизайна и реализации перечислимых типов.

Тем не менее есть случай, когда имеет смысл использовать расширяемые перечислимые типы: это *коды операций* (operation codes, opcodes). Код операции — это перечислимый тип, элементы которого представляют операции на некой «машине», вроде типа `Operation` в статье 30, который представляет функции простого калькулятора. Иногда желательно позволить пользователям API предоставлять свои собственные операции, фактически расширяя набор операций, предоставляемых API.

К счастью, есть прекрасный способ достичь такого эффекта с использованием перечислимых типов. Основная идея здесь — воспользоваться тем, что перечислимые типы могут реализовывать произвольные интерфейсы, и определить интерфейс для типа кодов операций и перечислимый тип, являющйсся стандартной реализацией этого интерфейса. Например, вот пример расширяемой версии типа `Operation` из статьи 30:

```
// Имитация расширяемого перечислимого типа 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 операций возведения в степень и получения остатка от деления. Все, что вам нужно сделать, — это написать перечислимый тип для реализации интерфейса `Operation`:

```
// Имитация расширенного перечислимого типа
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 тип реализации (`BasicOperation`). Обратите внимание, что вам не нужно объявлять абстрактный метод `apply` в перечислимом типе, как вы бы делали это в нерасширяемом перечислимом типе с релизацией метода, специфичного для экземпляра. Дело в том, что абстрактный метод (`apply`) является членом интерфейса (`Operation`).

Комментарий. В API `java.nio.file`, появившемся в Java 7, этот шаблон используется для опций открытия файлов (интерфейс `OpenOption`) и копирования файлов (интерфейс `CopyOption`). Большинство пользователей будет использовать стандартные реализации

этих интерфейсов (перечислимые типы `StandardOpenOption` и `StandardCopyOption` соответственно), но библиотеки, предлагающие собственные реализации файловых систем, имеют возможность определить и собственные, специфичные для их файловых систем опции открытия и копирования файлов.

Есть возможность не только передать один экземпляр расширенного перечислимого типа везде, где ожидается основной перечислимый тип, — можно передать метку типа для расширенного перечислимого типа и использовать его элементы в дополнение или вместо элементов основного типа. Например, вот версия программы, которая выполняет все операции расширения, определённые 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 литерал класса для расширенного операционного типа (`ExtendedOperation.class`) передаётся из `main` в `test` для описания набора расширенных операций. Литерал класса служит здесь *ограниченной меткой типа* (статья 29). Довольно сложное объявление параметра `opSet` как `<T extends Enum<T> & Operation> Class<T>` гарантирует, что объект `Class` представляет и перечислимый тип, и подтип интерфейса `Operation`, что требуется для обхода элементов перечисления и выполнения операций, связанных с каждым из них.

Второй альтернативой будет использование типа `Collection<? extends Operation>`, который является *ограниченным подстановочным типом* (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 test2(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, а метод `test` более гибок: он позволяет вызывающему объединить операции из нескольких типов реализации. С другой стороны, вы не сможете использовать `EnumSet` (статья 32) и `EnumMap` (статья 33) для переданных операций, так что вам лучше воспользоваться ограниченной меткой типа, если только вам не требуется гибкость для объединения операций из нескольких типов реализации.

Обе вышеупомянутые программы выдадут один и тот же 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 `Operation` 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 объем общего функционала был больше, вы могли бы инкапсулировать его во вспомогательный класс или статический вспомогательный метод для избежания дублирования кода.

Подведём итог. **Хотя вы не можете написать расширяемый перечислимый тип, вы можете имитировать его, написав интерфейс с базовым перечислимым типом,**

реализующим этот интерфейс. Это позволяет клиентам писать свои собственные перечислимые типы, реализующие этот же интерфейс. Эти перечислимые типаs могут потом использоваться везде, где может быть использован базовый перечислимый тип, при условии, что внешние API работают именно с интерфейсом.

Статья 35. Предпочитайте аннотации шаблонам именования

До версии 1.5 было общепринятым использование *шаблонов именования* (naming patterns) для определения некоторых элементов программы как требующих специального подхода со стороны внешнего инструмента или фреймворка. Например, среда тестирования JUnit изначально требовала от пользователей начинать имена методов тестирования с символов `test` [Веск04]. Этот приём работает, но у него есть несколько недостатков. Во-первых, типографические ошибки могут привести к невидимым проблемам. Например, предположим, что вы случайно назвали метод тестирования `tsetSafetyOverride` вместо `testSafetyOverride`. JUnit не сообщит вам об ошибке, но и не будет выполнять тестирование, что приведёт к ложному чувству безопасности.

Второй недостаток именования шаблонов заключается в том, что нет способа гарантировать, что они используются только на тех элементах программы, на каких нужно. Например, предположим, вы дали *классу* имя `testSafetyMechanisms`, надеясь, что JUnit автоматически проверит все методы этого класса 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 выбросит определённое исключение. Тип исключения является по сути параметром теста. Вы можете закодировать название исключения типа в названии метода тестирования, используя готовые шаблоны, но это будет работать ненадёжно 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 это 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 названием `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 {
}
```

Объявление типа аннотации `Test` само аннотировано аннотациями `Retention` и `Target`. Такие аннотации, которые применяются к объявлениям типов аннотаций же, называются *мета-аннотациями* (meta annotations). Мета-аннотация `Retention(RetentionPolicy.RUNTIME)` означает, что аннотация `Test` будет доступна в программе во время выполнения. Без этого аннотация `Test` была бы невидима инструменту тестирования. Мета-аннотация `@Target(ElementType.METHOD)` обозначает, что аннотация `Test` разрешена только на объявлениях методов: она не может применяться при объявлении классов, полей и других элементов.

Обратите внимание на комментарий при декларации аннотации `Test`, который говорит «Используйте только на статических методах без параметров». Было бы хорошо, если бы компилятор мог наложить это ограничение, но он не может. Есть определённые 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 *маркерной аннотацией* (marker annotation), потому что у неё нет параметров, она только маркирует аннотированные элементы. 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 маркерные аннотации
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`, бросают исключения, и два, `m1` и `m5`, не бросают. Но один из аннотируемых методов, который не бросает исключения, `m5`, является методом экземпляра, поэтому нельзя использовать аннотацию в данном случае. В итоге `Sample` содержит четыре теста: один будет успешным, другой нет, и один не является разрешённым. 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 заинтересованных программ. Говоря более общо, аннотации никогда не меняют семантики аннотируемого кода, но создают возможность для специальной обработки этого кода инструментами, такими как эта простая программа, запускающая тест:

```
// Программа для обработки маркерных аннотаций
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 (ReflectiveOperationException exc) {
                    System.out.println("INVALID @Test: " + m);
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

Этот инструмент принимает полное имя класса в командной строке и запускает все аннотированные аннотацией `Test` методы класса с помощью рефлексии, путём вызова `Method.invoke`. Метод `isAnnotationPresent` говорит инструменту, какие именно методы запускать. Если метод тестирования выбрасывает исключение, то механизм рефлексии заворачивает его в оболочку `InvocationTargetException`. Инструмент фиксирует это исключение и печатает отчёт об ошибке, содержащий оригинальное исключение, выброшенное методом тестирования, которое извлекается

из `InvocationTargetException` с помощью метода `getCause`.

Если попытка запуска метода тестирования отражением выбрасывает какое-либо исключение, кроме `InvocationTargetException`, это означает неправильное использование аннотации `Test`, которое не было отловлено во время компиляции. Такое использование включает в себя аннотацию на методе экземпляра, на методе с одним или более параметром или на недоступном методе. Второй блок `catch` в программе запуске теста фиксирует ошибки использования анннотации `Test` и выводит соответствующее сообщение об ошибке. Вот результат, который выводится, когда программа `RunTests` запускается на классе `Sample`:

```
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
```

Комментарий. Библиотека JUnit тоже использует аннотацию `Test` для маркировки методов тестирования, начиная с версии 4. Однако тестовые методы являются не статическими, а методами экземпляра, для чего экземпляр класса, в котором они объявлены, сначала создаётся с помощью конструктора без параметров.

Теперь добавим поддержку тестов, которые завершаются успешно, только если они выбрасывают определённое исключение. Для этого нам понадобится новый тип аннотации:

```
// 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 подстановочный тип довольно громоздкий. Он означает «объект `Class` некоего класса, расширяющего `Exception`», и позволяет пользователю аннотации задать любой тип исключения. Такое использование является примером *ограниченного подстановочного типа* (статья 29). Вот как это выглядит на практике. Обратите внимание, что литералы класса используются здесь как значения параметров аннотации:

```
// Программа, содержащая аннотации с параметрами
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 он обрабатывал новую аннотацию. Нам нужно добавить следующий код 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 wrappedExc) {
        Throwable exc = wrappedExc.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 (ReflectiveOperationException exc) {
            System.out.println("INVALID @Test: " + m);
        }
    }
```

Этот код похож на тот, который мы использовали при обработке аннотации `Test`, с одним отличием: код извлекает значение параметра аннотации и использует его для проверки правильности ш а выводимого исключения. Явного приведения типов нет, поэтому исключение `ClassCastException` не представляет опасности. Тот факт, что программа тестирования 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 лределенный тип исключения, больше нет при выполнении. В этом, будем надеяться, редком случае программа, запускающая тест, выбросит исключение `TypeNotPresentException`.

Продвинем пример с проверкой исключений на один шаг вперёд. Вполне возможно представить тест, который проходит успешно, если выводится одно из нескольких определённых исключений. Механизм аннотаций имеет функционал, который облегчает поддержку подобного его использования. Предположим, что мы изменим тип параметра аннотации `ExceptionTest` так, чтобы он был массивом объектов `Class`:

```
// Тип аннотации с параметром-массивом
@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 использования аннотации `ExceptionTest` все ещё действительны в новой версии `ExceptionTest` с параметром-массивом, и в результате получается одноэлементный массив. Для определения многоэлементного массива необходимо окружить элементы фигурными скобками и отделить их запятыми:

```
// Код, содержащий аннотацию с параметром-массивом
@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(s, null);
}
```

Относительно просто изменить программу, запускающую тесты, для обработки новой версии аннотации `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);
    }
}
```

Комментарий. В Java 8 мы могли бы обойтись без переменной `oldPassed`, переписав код следующим образом:

```
Class<? extends Exception>[] excTypes = ...;

if (Stream.of(excTypes).anyMatch(
        excType -> excType.isInstance(exc))) {
    passed++;
} else {
    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 возможности обработки аннотаций. Если вы пишете инструмент, который требует от программистов добавлять информацию к файлам исходного кода, определите подходящий набор типов аннотаций. **Нет никакой причины для использования шаблонов именования теперь, когда есть аннотации.**

Как уже говорилось, за исключением авторов инструментов обработки кода, большей части программистов не потребуется определять типы аннотаций. **Все программисты должны, тем не менее, использовать предопределённые типы аннотаций, входящие в состав платформы Java** (статьи 36 и 24). Также рассмотрите возможность использования любых аннотаций, предоставляемых вашей IDE или инструментами статического анализа. Такие аннотации могут улучшить качество диагностической информации, предоставляемой этими инструментами. Обратите внимание, однако, что эти аннотации ещё не стандартизированы, так что вам придётся проделать некоторую работу, если вы захотите сменить используемые инструменты или если стандарт, наконец, появится.

Комментарий. К таким инструментам относятся, например, FindBugs и Error Prone, которые позволяют находить распространённые ошибки в написании кода на этапе компиляции. Некоторая поддержка статического анализа на основе аннотаций имеется также в IDE Eclipse и IntelliJ IDEA.

Статья 36. 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 объявлении методов, и она может определить, что объявление аннотируемого метода переопределяет объявление 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());
    }
}
```

Эта программа в цикле добавляет ко множеству 26 биграмм, каждая из которых состоит из двух идентичных символов нижнего регистра. Затем она выводит размер множества. Вы ожидаете, что программа напечатает 26, так как элементы множества не могут дублироваться. Если вы попытаетесь запустить программу, то увидите, что она распечатает не 26, а 260. Что не так?

Вполне понятно, что автор класса `Bigram` намеревался переопределить метод `equals` (статья 8) и даже не забыл переопределить метод `hashCode` (статья 9). К сожалению, наш неудачливый программист не переопределил `equals`, а перегрузив его (статья 41). Для переопределения `Object.equals` вы должны определить метод `equals`, параметр которого принадлежит типу `Object`, но параметр метода `equals` класса `Bigram` не принадлежит 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`, и это объясняет, почему программа пишет 260.

К счастью, компилятор может помочь обнаружить эту ошибку, но только если вы поможете ему, сказав, что вы собираетесь переопределить `Object.equals`. Чтобы это сделать, необходимо аннотировать `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 скомпилировать программу, компилятор выдаст сообщение об ошибке, вроде этого:

```
Bigram.java:10: method does not override or implement a method
from a supertype
```

```
    @Override public boolean equals(Bigram b) {
    ^
```

Вы сразу же поймёте, 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 объявления каждого метода, который должен переопределять метод 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, возможно, захотите 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 ещё одну причину, по которой необходимо последовательно использовать аннотации `Override`. У таких IDE есть автоматические проверки, известные как *инспекции кода* (code inspections). 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. Благодаря сообщениям от IDE и компилятора 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 ещё больше помощи в поиске ошибок. В версии 1.6 стало разрешено использование

аннотации `Override` на объявлениях методов, которые переопределяют методы интерфейсов, а не только 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. Опять-таки вы можете, если хотите, добавить эти аннотации просто для привлечения внимания к методам 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 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 объявлениях методов, чтобы гарантировать, что он случайно не добавит новые методы к интерфейсу `Collection`.

Подведём итоги. Компилятор может защитить вас от большого количества ошибок, если вы используете аннотацию `Override` для всех объявлений методов, которые должны переопределять объявление супертипов, за одним исключением. В конкретных классах вам не нужно аннотировать методы, которые должны переопределять абстрактные методы (хотя делать так не вредно).

Статья 37. Используйте маркерные интерфейсы для определения типов

Маркерный интерфейс (marker interface) — это такой интерфейс, который не содержит деклараций методов, но просто определяет (или «маркирует») класс, который реализует интерфейс как имеющий определённое свойство. Например, рассмотрим интерфейс `Serializable` (глава 11). Реализуя этот интерфейс, класс сообщает, что его экземпляры могут быть записаны в `ObjectOutputStream` (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 определённого интерфейса. Если вы определите его как маркерный интерфейс, то вы сможете расширить единственный интерфейс, к которому он применим, гарантируя, что все маркированные типы являются также подтипами единственного интерфейса, к которому он применим.

Есть спорное утверждение, что интерфейс `Set` как раз является *ограниченным маркерным интерфейсом* (restricted marker interface). Он применим только к подтипам `Collection`, но не добавляет никаких методов, кроме определённых в `Collection`. Он обычно не рассматривается как маркерный интерфейс, потому что он уточняет контрактц нескольких методов `Collection`, которые включают `add`, `equals` и `hashCode`. Но легко представить маркерный интерфейс, который применим только к подтипам определённого интерфейса и не уточняет контракты никаких методов интерфейса. Такой маркерный интерфейс может описать некоторые инварианты всего объекта или показать своим наличием, что экземпляры подходят для обработки методом какого-либо другого класса (таким же образом, как и интерфейс `Serializable` показывает, что экземпляры типа 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 путём добавления одного или более элементов типа аннотации к уже имеющимся по умолчанию [JLS, 9.6], что даёт 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).

> **Комментарий.** В Java 8 появилась возможность расширять интерфейсы с сохранением обратной совместимости благодаря методам по умолчанию.

Другим преимуществом маркерных аннотаций является то, что они являются частью более широкого механизма аннотаций. Следовательно, маркерные аннотации обеспечивают последовательность во фреймворках, разрешающих применение аннотаций на большом количестве программных элементов.

Итак, когда же мы должны использовать маркерную аннотацию, а когда — маркерный интерфейс? Определённо, 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 предпочтительнее использовать маркерный интерфейс вместо аннотации. Это даст вам возможность использовать интерфейс в качестве типа параметра для методов в вопросе, который приведёт к реальной выгоде при проверке типов на этапе компиляции.

Если вы ответили на первый вопрос, задайте себе ещё один: хочу ли я ограничить использование этого маркера элементами определённого интерфейса навсегда? Если так, то разумно определить маркер как подинтерфейс этого интерфейса. Если вы ответили «нет» на оба вопроса, то вам, вероятнее всего, стоит использовать маркерную аннотацию.

Подведём итоги. И маркерные интерфейсы, и маркерные аннотации имеют свои области применения. Если вы хотите определить тип, который не имеет никаких новых методов, связанных с ним, то лучший способ — использовать маркерный интерфейс. Если вы хотите маркировать элементы программы, не являющиеся классами или интерфейсами, либо дать возможность добавления информации к маркеру в будущем, либо использовать маркер с фреймворком, который уже использует типы аннотаций, тогда верным выбором будет использование маркерной аннотации. **Если вы пишете тип маркерной аннотации, цель применимой только к `ElementType.TYPE`, потратьте время, чтобы определить, действительно ли это должен быть тип аннотации, или же более**

подходящим решением будет маркерный интерфейс.

В некотором смысле эта статья — противоположность статьи 19, в которой говорится: «Если вы не хотите определять тип, не используйте интерфейс». В первом приближении эта статья говорит: «Если вы действительно хотите определить тип, используйте интерфейс».

Глава 7. Методы

В этой 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 правильность 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 чётко 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 случай обнаружение ошибки станет менее вероятным, а определение источника ошибки — более трудоёмким.

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 нормально, без возражений вычислив неверный результат. Но самое худшее, если метод завершится нормально, 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 выбрасываться, 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 выбрасываться в случае нарушения ограничений, установить эти ограничения для метода не составит труда. Приведём типичный пример:

```
/**
 * Возвращает объект BigInteger, значением которого
 * является (this mod 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, если m <= 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus not positive");
    ... // Вычисления
}
```

Если метод не экспортируется, 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. Поэтому методы, не являющиеся открытыми, должны проверять свои параметры, используя *утверждения* (assertions), 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* истинными, вне зависимости от того, как пакет используется своими клиентами. В отличие от обычных проверок корректности, утверждения выбрасывают ошибку `AssertionError` в случае неудачного запуска. Также, в отличие от обычных проверок, они не имеют никакого эффекта и ничего не стоят, пока не включены явно. Это можно сделать, передав флаг

`-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 фабричный метод из статьи 16, который получает массив целых чисел и возвращает представление этого массива в виде экземпляра `List`. Если клиент метода передаст значение `null`, метод выбросит исключение `NullPointerException`, поскольку содержит явную проверку. Если бы проверка отсутствовала, метод возвращал бы ссылку на вновь сформированный экземпляр `List`, который будет выбрасывать 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 выброшено 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, выбрасывается совсем не то исключение. Другими словами, исключение, которое выбрасывается 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* (exception translation), 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 реализуемыми абстракциями.

Подведём 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 защитные 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 methods) 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 C и C++. 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 чаще всего вашему классу придётся иметь дело с непредвиденным поведением 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 неизменяемого 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 лёгкостью 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 *защитную копию* (defensive copy)** и использовать именно эти копии, а не оригинал, как составные части экземпляра `Period`:

```
// Исправленный конструктор: для представленных параметров создаёт
// защитные 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 + " after " + end);
}
```

С новым конструктором описанная ранее атака уже не может воздействовать на экземпляр `Period`. Заметим, что защитные 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 такая атака называется атакой типа «время проверки/время использования» (time of check/time of use, TOCTTOU) [Viega01].

Заметим также, что для создания защитных копий мы не пользовались методом `clone` из класса `Date`. Поскольку `Date` не объявлен как `final`, нет гарантии, что метод `clone` возвратит объект именно класса `java.util.Date`, — он может вернуть экземпляр ненадёжного 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 защитной копии параметра, который имеет тип, позволяющий ненадёжным партнёрам создавать подклассы.**

Обновлённый конструктор успешно защищает от вышеописанной атаки, однако все равно остаётся 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 защитные копии изменяемых внутренних полей.

```
// Исправленные методы доступа: создаются защитные копии
// внутренних полей
public Date start() {
    return new Date(start.getTime());
)

public Date end() {
    return new Date(end.getTime());
}
```

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.

Новые методы доступа, в отличие от конструктора, могли бы использовать метод `clone` для создания защитных копий. 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 ненадёжному подклассу. Тем не менее лучше всё-таки использовать конструктор или статический фабричный метод по причинам, указанным в статье 11.

Защитное 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 защитную 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 множества или словаря могут быть нарушены, если после добавления в них объект вдруг поменяется.

То же самое справедливо и в отношении защитного копирования внутренних компонентов перед возвращением их клиенту. Вы должны дважды подумать, является ли ваш класс изменяемым или нет, прежде чем передавать клиенту ссылку на внутренний компонент, который можно изменить. Есть большая вероятность, что вам все же следует возвращать защитную копию. Помните, что массивы ненулевой длины всегда являются изменяемыми. Поэтому для внутреннего массива вы всегда должны делать защитную копию, прежде чем возвращать его клиенту. Как альтернатива, вы можете возвращать пользователю неизменяемое представление (immutable view) этого массива. Оба этих приёма показаны в статье 13.

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 защитном копировании (статья 13). 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.

> **Комментарий.** Конкретно в этом случае, начиная с Java 8, имело бы смысл вместо устаревшего класса `Date` использовать в реализации класса `Period` один из классов пакета `java.time`, например, `Instant`. Все эти классы являются неизменяемыми, поэтому они не требуют защитного копирования.

Не всегда можно создать защитную копию изменяемого параметра перед его включением в объект. Если класс доверяет тому, кто его вызывает, и не будет изменять внутреннее содержимое, (например, поскольку класс и его клиенты являются частями одного и того же пакета), то можно обойтись без защитного копирования. При подобных обстоятельствах в документации к классу должно быть чётко объяснено, что вызывающий не должен менять задействованные параметры или возвращаемые значения.

Даже за рамками одного пакета не всегда стоит использовать защитное копирование изменяемых параметров перед интегрированием их в объект. Существуют такие методы и конструкторы, чей вызов означает явную *передачу управления* (ownership transfer) для объекта, на который указывает параметр-ссылка. Вызвав такой метод, клиент даёт обещание, что он не будет напрямую менять этот объект. Для метода или конструктора, предполагающего, что ему будет полностью передано управление полученным от клиента изменяемым объектом, это обстоятельство должно быть чётко 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 определённом 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.

Подведём итоги. Если у класса есть изменяемые компоненты, которые он получает от клиента или возвращает ему, то необходимо защитное копирование всех компонентов класса. Если затраты на копирование слишком высоки и класс доверяет своим клиентам, зная, что они не изменят неподходящим образом компоненты, тогда защитное копирование можно заменить документированием, отражающим, что клиенты не должны менять задействованные компоненты.

Статья 40. 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 должны соответствовать стандартным соглашениям по именованию (статья 56). Вашей главной целью должен быть выбор понятных имён, которые будут согласоваться с остальными названиями в том же пакете. Второй целью должен быть выбор имён, отвечающих 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 оправдывать своё существование. Избыток методов делает класс слишком сложным для изучения, использования, описания, тестирования и сопровождения. В отношении интерфейсов это верно вдвойне: большое количество методов усложняет жизнь и разработчикам, и пользователям. Для каждого действия, поддерживаемого вашим типом, создайте полнофункциональный метод. Сокращённый вариант операции рассматривайте лишь в том случае, если она будет использоваться часто. **Если есть сомнения, забудьте об этом варианте.**

Избегайте длинного списка 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 документации. Современные IDE могут здесь помочь, но всё равно более короткие списки параметров предпочтительнее. **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 слшком длинных списков параметров можно использовать три приёма. 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 приём помогает *сократить* количество методов путём 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 (view) 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, оперирующими экземплярами `List`, чтобы выполнять самые разные операции для подсписков. Полученный API имеет очень высокое соотношение мощности и размера.

Второй приём сокращения чрезмерно длинных перечней параметров заключается в создании *вспомогательных классов* (helper classes), обеспечивающих агрегирование параметров. Обычно эти вспомогательные классы являются статическими классами-членами (статья 22). Этот приём 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 передаётся 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 приём, сочетающий в себе аспекты первых двух, состоит в том, чтобы вставить шаблон «построитель» (builder, статья 2) между созданием объекта и запуском

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 или маленькую связанную группу параметров. Как только все нужные параметры заданы, клиент запускает «исполняемый» метод на объекте, который выполняет окончательные проверки параметров и производит сам расчёт.

Выбирая 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 `HashMap`, лучше использовать `Map`. Это позволит вам передавать этому методу `Hashtable`, `HashMap`, `TreeMap`, подсловарь `TreeMap` и вообще любую, даже ещё не написанную реализацию интерфейса `Map`. Применяя же вместо интерфейса класс, вы навязываете клиенту конкретную реализацию и вынуждаете использовать ненужное и потенциально трудоёмкое копирование в том случае, если входные данные будут представлены в какой-либо иной форме.

> **Комментарий.** Пример того, как делать не надо — методы `appendReplacement` и `appendTail` класса `java.util.regex.Matcher`. Эти методы принимают объект класса `StringBuffer` и тем самым навязывают клиентскому коду накладные расходы на синхронизацию, а возможно, и на копирование в объект более подходящего типа. Здесь уместнее было бы использовать интерфейс `Appendable`, что позволяло бы использовать `StringBuffer`, `StringBuilder`, `StringWriter`, `PrintWriter` и многие другие классы. Правда, интерфейс `Appendable` и класс `StringBuilder` появились только в Java 5, то есть уже после внедрения API регулярных выражений в Java 1.4, но это не объясняет, почему эти методы не были перегружены с использованием нового интерфейса.
> Разработчики Java 9 «исправили» класс `Matcher`, добавив перегрузку этих двух методов для класса `StringBuilder`. По-прежнему непонятно, однако, что мешало им вместо этого использовать интерфейс `Appendable` и тем самым позволить использовать эти методы с классами, отличными от `StringBuffer` и `StringBuilder`.

Предпочитайте использовать двухэлементные перечислимые типы вместо параметров `boolean`. Код легче читать и писать, особенно если вы используете IDE, которая поддерживает автозаполнение. Также это облегчает добавление новых опций. Например, пусть у вас есть тип `Thermometer` со статическим фабричным методом, который принимает значение перечислимого типа:

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Вызов `Thermometer.newInstance(TemperatureScale, CELSIUS)` не только намного понятнее, чем использование `Thermometer.newInstance(true)`, но также позволяет вам в будущей версии добавить к перечислению `ThermometerScale` элемент `KELVIN` без необходимости добавлять новый метод к классу `Thermometer`. Также вы можете вынести код, зависимый от температурной шкалы, в методы констант перечисления (статья 30). Например, у каждой константы шкалы может быть метод, который берет значение `double` и нормализует его к шкале Цельсия.

Статья 41. Соблюдайте осторожность, перегружая методы

Приведём пример попытки классифицировать коллекции по признаку того, являются ли они множествами, списками или каким-то другим видом коллекций:

```
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` *перегружен* (overloaded), и **выбор варианта перегрузки осуществляется на стадии компиляции**. Для всех трёх 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 контринтуитивно потому, что **выбор перегруженных методов осуществляется статически, тогда как выбор переопределённых методов — динамически**. Правильный вариант переопределённого метода выбирается при выполнении программы, исходя из того, какой тип в этот момент имел объект, для которого этот метод был вызван. Напомним, что переопределение (override) метода осуществляется тогда, когда подкласс имеет объявление метода с точно такой же сигнатурой, что и у декларации метода предка. Если в подклассе метод был переопределён и затем данный метод был вызван для экземпляра этого подкласса, то выполняться будет уже *переопределённый 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 объявлен в классе `Wine` и переопределён 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 переопределённого метода: всегда выполняется «самый точный» переопределяющий метод. Сравните это с перегрузкой, когда тип объекта на стадии выполнения уже не влияет на то, какой вариант перегрузки 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 перегруженный метод на основании того, какой тип имеет параметр на стадии выполнения. Именно это делает метод `name` в примере с `Wine`. Перегрузка метода не имеет такой возможности. Предполагая, что тут требуется статический метод, исправить программу можно, заменив все три варианта перегрузки 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 перегрузка — исключением, именно переопределение задаёт, что люди ожидают увидеть при вызове метода. Как показал пример `CollectionClassifier`, перегрузка может не оправдать эти ожидания. Не следует писать код, поведение которого имеет все шансы запутать программистов. Особенно это касается API. Если рядовой пользователь API не знает, какой из перегруженных методов будет вызван для указанного набора параметров, то работа с таким API, вероятно, будет сопровождаться ошибками. Причём 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 перегрузки**.

Стоит обсудить, что же именно сбивает людей с толку при использовании перегрузки. **Безопасная, умеренная политика предписывает никогда не предоставлять два варианта перегрузки с одним и тем же числом параметров.** Если метод использует переменное число параметров, 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 перегрузки соответствует тому или иному набору параметров. Это ограничение не слишком обременительно, поскольку, вместо того чтобы использовать перегрузку, 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 перегружать метод `write`, они применяют такие сигнатуры, как `writeBoolean(boolean)`, `writeInt(int)` и `writeLong(long)`. Дополнительное преимущество такой схемы именования по сравнению с перегрузкой 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 перегрузке. Правда, в отдельных ситуациях вы можете вместо конструктора предоставлять статический фабричный метод (статья 1). Кроме того, при применении конструкторов вам не нужно беспокоиться о взаимосвязи между перегрузкой и переопределением, так как конструкторы нельзя переопределять. Поскольку вам, вероятно, придётся предоставлять несколько

конструкторов с одним и тем же количеством параметров, полезно знать, в каких случаях это безопасно.

Предоставление нескольких перегруженных методов с одним и тем же количеством параметров вряд ли запутает программистов, если всегда понятно, какой вариант перегрузки соответствует заданному набору реальных параметров. Это как раз тот случай, когда у каждой пары вариантов перегрузки есть хотя бы один формальный параметр с «радикально отличным» (radically different) типом. Два типа считаются радикально отличными, если экземпляр одного из этих типов невозможно привести к другому типу. В этих условиях выбор варианта перегрузки 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 автоупаковки, и оно 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 множеству и к списку, затем делает три идентичных обращения к `remove` и в наборе, и в списке. Скорее всего, вы ожидаете, что программа удалит неотрицательные значения (0, 1 и 2) из множества и списка и напечатает `[-3, -2, -1] [-3, -2, -1]`. На самом деле программа удаляет неотрицательные значения из множества и нечётные значения из списка и выводит `[-3, -2, -1] [-2, 0, 2]`. Будет преуменьшением назвать такое поведение программы запутанным.

Итак, вот что происходит: Обращение к `set.remove()` выбирает перегруженный метод `remove(E)`, где `E` — тип элемента множества (`Integer`), и преобразует `i` из `int` в `Integer`. Мы ожидаем такое поведение программы, следовательно, программа удаляет положительные значения из набора. Обращение к `list.remove()`, с другой стороны, выбирает вариант перегрузки `remove(int i)`, который удаляет элементы из определённых позиций в списке. Если вы начнёте со списка `[-3, -2, -1, 0, 1, 2]` и удалите нулевой элемент, затем первый и затем второй, у вас останется `[-2, 0, -2]` и загадка будет разгадана. Для решения проблемы приведите аргумент `list.remove` к типу `Integer`, заставив тем самым компилятор выбрать верный вариант перегрузки. 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))
}
```

Запутанное 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, в которой он был обобщён, было обобщено, у интерфейса `List` был метод `remove(Object)` вместо `remove(E)`, и соответствующие типы параметров `Object` и `int` радикально отличались. Но с появлением средств обобщённого программирования и преобразований эти два типа параметров больше не являются радикально отличными. Другими словами, появление средств обобщённого

программирования и автоупаковки в языке нанесло ущерб интерфейсу `List`. К счастью, лишь незначительное количество API в библиотеках Java пострадало подобным образом, но эта история разъясняет нам (что более важно), что перегрузку надо использовать осторожно, так как автоупаковка и средства обобщённого программирования теперь тоже являются частью языка.

Комментарий. В платформе Microsoft .NET автоупаковка числовых типов присутствовала изначально, поэтому её разработчики сразу разрешили эту проблему, вообще отказавшись в данном случае от перегрузки. Интерфейс списков `IList` в .NET использует для удаления элементов методы с разными именами: `Remove` для удаления по значению и `RemoveAt` для удаления по индексу.

Типы массивов и классы, кроме `Object`, радикально отличаются. Также типы массивов и интерфейсы, кроме `Serializable` и `Cloneable`, радикально отличаются. Два отдельных класса считаются *неродственными* (unrelated), если ни один из классов не является потомком другого [JLS, 5.5]. Например, `String` и `Throwable` — неродственные классы. Ни один объект не может быть экземпляром двух неродственных классов, потому что неродственные 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 будет применена к определённому набору фактических 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 были расширены для поддержки этого интерфейса. В то же время, как в платформу был добавлен интерфейс `CharSequence`, класс `String` получил перегрузку метода `contentEquals`, принимающую `CharSequence`. Получившаяся перегрузка очевидно нарушает инструкции данной статьи, но она не причиняет вреда, если оба перегруженных метода делают одно и то же при запуске на одной и той же ссылке на объект. Программисты могут не знать, какая именно перегрузка 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 платформы Java в основном следуют приведённым здесь советам, все же можно найти несколько мест, где они нарушаются. Например, класс `String` имеет два перегруженных статических фабричных метода `valueOf(char[])` и `valueOf(Object)`, которые, получив ссылку на один и тот же объект, выполняют совершенно разную работу. Этому нет чёткого объяснения, и относиться к данным методам следует как к аномалии, способной вызвать настоящую неразбериху.

Подведём итоги. То, что вы можете осуществлять перегрузку методов, ещё не означает, что вам стоит это делать. Обычно лучше воздерживаться от перегрузки методов, которые имеют несколько сигнатур с одинаковым количеством параметров. Но иногда, особенно в случае конструкторов, невозможно следовать этому совету. Тогда постарайтесь избежать ситуации, при которой благодаря приведению типов один и тот же набор параметров может использоваться разными вариантами перегрузки. 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 перегрузки, получая одни и те же параметры, будут вести себя одинаковым образом. Если же вы этого не сделаете, программистам будет трудно эффективно использовать перегруженный метод или конструктор, и они не смогут понять, почему он не работает.

Статья 42. Соблюдайте осторожность при использовании методов с переменным числом параметров

В версии 1.5 были добавлены методы с переменным числом параметров (varargs, variable arity methods) [JLS, 8.4.1]. Такие методы принимают ноль или более аргументов определённого типа. При вызове метода сначала создаётся массив, размер которого

равен числу аргументов, переданных в месте вызова, затем помещает значения аргументов в массив и, наконец, передаёт массив методу.

Например, вот метод, 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, а не *ноль* или более. Например, предположим, вы хотите найти минимальный элемент в последовательности аргументов `int`. Эта функция не определена, если клиент не передаёт ни одного аргумента. 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 серьёзная заключается в том, что если клиент вызовет этот метод без аргументов, то ошибка произойдёт при выполнении, а не при компиляции. Другая проблема заключается в том, что оно ужасно выглядит. Вам необходимо включить явную проверку корректности для массива `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 определённого типа и один переменный список параметров 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. Механизм varargs был создан для метода `printf`, который появился в платформе версии 1.5, и для средств рефлексии (статья 53), которые были модифицированы для использования преимуществ varargs в этой версии. И `printf`, и рефлексия очень много выиграли от появления varargs.

Вы можете изменить существующий метод, берущий массив как последний параметр, с тем, чтобы он вместо массива принимал переменный список параметров, и это не поломает совместимость с существующими клиентами. Однако то, что вы можете так сделать, вовсе не значит, что это следует делать. Рассмотрим случай с методом `Array.asList`. Этот метод никогда не разрабатывался для того, чтобы собирать несколько аргументов в список, но разработчики сочли хорошей идеей изменить объявление метода так, чтобы он вместо массива принимал переменный список параметров. 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 существовала общая идиома для распечатывания содержимого массива:

```
// Устаревшая идиома для печати массива!
System.out.println(Arrays.asList(myArray));
```

Она была необходима, потому что массивы наследуют свою реализацию `toString` от `Object`, из-за чего вызов `toString` непосредственно на массиве выдаёт бесполезную строку, такую 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 с переменным числом параметров 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» так, чтобы он мог использовать varargs, кладёт ссылку на массив целых чисел `digits` в одноэлементный массив массивов и заворачивает его в экземпляр `List<int[]>`. Распечатка этого списка приводит к тому, что запускается `toString` на его 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 куда более надёжна. В версии 1.5 в класс `Arrays` был добавлен полный набор методов `Arrays.toString` (без переменного числа параметров!), 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 `Arrays.asList` было бы лучше добавить в класс `Collections` метод специально с целью сбора аргументов в список:

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

Такой метод даёт возможность собрать элементы в список, не причиняя вреда проверке типов в уже существующем методе `Arrays.asList`.

Урок понятен. **Не стоит модифицировать каждый метод, у которого последний параметр имеет тип массива; используйте varargs, только когда вызов оперирует 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 с переменным числом параметров приводит к созданию и инициализации массива. Если вы понимаете, что не можете позволить себе такие расходы, но вам нужна гибкость методов varargs, есть шаблон, который позволит совместить преимущества обоих подходов. 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 трёх, 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 трёх. Как и в большинстве случаев оптимизации производительности, этот приём обычно нет смысла использовать, но там, где он нужен, он незаменим.

Класс `EnumSet` использует данный приём для своих методов статической генерации для уменьшения затрат на создание наборов перечислимых типов до абсолютного минимума. Было решено использовать данный приём, потому что класс `EnumSet` проектировался как сопоставимый по производительности с битовыми полями (статья 32).

Подведём итоги. Механизм varargs удобен для определения методов, которые требуют переменного количества аргументов, но не стоит переусердствовать с ним. Результаты выполнения таких методов могут сбить пользователя с толку при неправильном их использовании.

Статья 43. Возвращайте массивы и коллекции нулевой длины, а не null

Нередко встречаются методы, имеющие следующий вид:

```
private List<Cheese> 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 многословность необходима почти при каждом вызове метода, который вместо массива или коллекции 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 более нуля объектов. Следует ещё упомянуть о том, что возврат `null` вместо массива приводит к усложнению самого метода, возвращающего массив или коллекцию.

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 является 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 передаётся 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 метода `Collection.toArray(T[])` даёт гарантию, что, если входной массив будет достаточно вместителен, чтобы содержать коллекцию, возвращён 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);
    }
```

Подведём итоги. **Нет никаких причин для того, чтобы работающий с массивами или коллекциями метод возвращал значение `null`, а не массив или коллекцию нулевой длины.** Такая идиома, по-видимому, проистекает из языка C, где длина массива возвращается отдельно от самого массива. В языке C бесполезно выделять память под массив нулевой длины.

Статья 44. Пишите комментарии Javadoc для всех открытых элементов API

Чтобы сделать API удобным для использования, его нужно документировать. Традиционно документация к API писалась вручную, и поддержание соответствия между документацией и программным кодом было весьма неприятной рутинной работой. 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 документации (documentation comments), которые чаще называют doc-комментариями (doc comments). Утилита Javadoc предлагает простой и эффективный способ документирования API и используется повсеместно.

Если вы ещё не знакомы с соглашениями для doc-комментариев, то вам следует их изучить. Эти соглашения не являются частью языка программирования Java, но де-факто они образуют свой API, который следует знать каждому программисту. Эти соглашения описаны на веб-странице Sun “How to Write Doc Comments” [Javadoc-guide]. Хотя эта страница и не обновлялась с момента выхода версии 1.4, она все ещё остаётся бесценным ресурсом. Два важных тега были добавлены в Javadoc с версией 1.5, `{@literal}` и `{@code}` [Javadoc-5.0]. Эти теги также обсуждаются в данной статье.

Чтобы должным образом документировать API, следует предварять doc-комментарием каждое видимое пользователями объявление класса, интерфейса, конструктора, метода и поля. Если класс является сериализуемым, нужно документировать также его сериализованную форму (статья 75). Если doc-комментарий отсутствует, самое лучшее, что может сделать Javadoc, — это воспроизвести объявление элемента API как единственно возможную для него документацию. Работа с API, у которого нет комментариев к документации, чревата ошибками. Чтобы создать программный код, приемлемый для сопровождения, вам следует также написать doc-комментарии для большинства классов, интерфейсов, конструкторов, методов и полей, которые невидимы для пользователей.

Doc-комментарий для метода должен лаконично описывать контракт между этим методом и его клиентами. Контракт должен 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 (preconditions), т.е. утверждения, которые должны быть истинными для того, чтобы клиент мог вызвать этот метод, и постусловия (postconditions), т.е. утверждения, которые будут истинными после успешного завершения вызова. Обычно предусловия неявно описываются тегами `@throws` для непроверяемых исключений. Каждое непроверяемое 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 видимое извне изменение состояния системы, которое не является очевидным требованием для достижения постусловия. Например, если метод запускает фоновый поток, это должно быть отражено в документации. Наконец, комментарии к документации должны описывать *потокобезопасность* (thread safety) класса, которая обсуждается в статье 70.

В целях полного описания контракта doc-комментарий метода должен включать в себя: тег `@param` для каждого параметра, тег `@return`, если только метод не объявлен как `void`, и тег `@throws` для каждого исключения, выбрасываемого этим методом, как проверяемого, так и непроверяемого (статья 62). По соглашению, текст, который следует за тегом `@param` или `@return`, представляет собой фразу, описывающую значение данного параметра или возвращаемое значение. Текст, следующий за тегом `@throws`, должен состоять из слова “if” и фразы, описывающей условия, при которых инициируется данное исключение. Иногда вместо словесных описаний используются арифметические выражения. По соглашению содержимое тегов `@param` и `@return` не заканчивается точкой. Все эти соглашения иллюстрирует следующий краткий doc-комментарий из интерфейса `List`:

```
/**
 * Возвращает элемент по заданной позиции в данном списке.
 * <p>
 * Этот метод не даёт гарантии выполнения за константное время. В
 * некоторых реализациях он выполняется за время, пропорциональное
 * позиции элемента.
 *
 * @param index индекс элемента, который нужно возвратить; индекс
```

```
     *              должен быть меньше размера списка и неотрицательным.
     * @return элемент, занимающий в списке указанную позицию
     * @throws IndexOutOfBoundsException, если 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 содержавшиеся 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 `@throws`. Это необходимо для двух целей: фрагмент кода внутри тега отображается шрифтом разметки кода (code font), а также подавляется обработка HTML разметки и вложенных Javadoc-тегов во фрагменте кода. Последнее свойство — это то, что нам даёт возможность использования знака “меньше чем” (`<`) во фрагменте кода, даже несмотря на то, что это метасимвол языка HTML. До версии 1.5 фрагменты кода включались в doc-комментарии с помощью 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» в doc-комментарии всегда ссылается на тот объект, для которого вызывается метод.

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. Тег `{@literal}` работает подобно тегу `{@code}`, за исключением того, что он использует для текста

шрифт кода. Например, фрагмент Javadoc:

```
/**
 * Неравенство треугольника - это {@literal |x + y| < |x| + |y|}.
 */
```

генерирует следующую документацию: «Неравенство треугольника - это |x + y| < |x| + |y|.» Тег `{@literal}` мог быть помещён только вокруг знака `<` вместо того, чтобы заключать в него все неравенство, причём 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 читаемыми не только в сгенерированной документации, но и в исходном коде.

Если и того и другого достичь не удаётся, 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. Например, в вышеприведённом doc-комментарии метода `get` общим описанием является предложение “Возвращает элемент по заданной позиции в данном списке”. Во избежание путаницы **никакие два члена или конструктора в одном классе или интерфейсе не должны иметь одинакового общего описания**. Особое внимание обращайте на перегруженные методы, описание которых часто хочется начать с одного и того же предложения.

Внимательно следите за первыми предложениями doc-комментариев, содержащими точку. В противном случае общее описание будет завершено раньше времени. Например, комментарий к документации, который начинается с фразы «`A college degree, such as B.S., M.S. or Ph.D.`», приведёт к появлению в общем описании фразы «A college degree, such as B.S., M.S.» Проблема заключается в том, что общее описание считается законченным при появлении первой точки, за которой следует пробел, символ табуляции или перевод строки (или при появлении первого блочного тега) [Javadoc-ref]. В данном предложении за второй точкой в аббревиатуре M.S. следует пробел. Лучшим решением будет заключить точку и ассоциированный с ней текст в тег `{@literal}`, чтобы за точкой в исходном коде больше не следовал пробел:

```
/**
 * 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 неопределённо-личное предложение, которое описывает операцию, осуществляемую этим методом. Например:

- `ArrayList(int initialCapacity)` — создаёт пустой список с заданной начальной ёмкостью.
- `Collection.size()` — возвращает количество элементов в данной коллекции.

Общее описание классов, интерфейсов и полей должно быть описанием сущности, 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 обобщённого программирования, перечислимые типы и аннотации. Когда вы пишете документацию по обобщённым типам или методам, обязательно документируйте все параметры типа:

```
/**
 * Объект, отображающий ключи на значения. Словарь не может
 * содержать повторяющиеся ключи; каждому ключу соответствует
 * не более одного значения.
 * (Остальное опущено)
```

```
 *
 * @param <K> тип ключей, содержащихся в словаре
 * @param <V> тип значений, соответствующих ключам
 */
public interface Map<K, V> {
... // Остальное опущено
}
```

При документировании перечислимого типа убедитесь, что документированы константы, тип и все открытые методы. Обратите внимание, что вы можете поместить весь doc-комментарий в одну строку, если он краткий:

```
/**
 * Инструментальная секция симфонического оркестра.
 */
public enum OrchestraSection {
    /** Деревянные духовые инструменты, например,
        флейта, кларнет и гобой. */
    WOODWIND,
    /** Медные духовые инструменты, например, труба и валторна. */
    BRASS,
    /** Ударные инструменты, например, литавры и тарелки. */
    PERCUSSION,
    /** Струнные инструменты, например, скрипка и виолончель. */
    STRING;
}
```

При документировании аннотационных типов убедитесь, что документирован не только сам тип, но и все его члены. Документируйте члены фразой с существительным, как будто они являются полями. Для общего описания типа используйте фразу с глаголом, которая говорит, что означает присутствие аннотации на элементе программы:

```
/**
 * Указывает, что аннотированный метод является методом
 * тестирования, который для успешного прохождения теста
```

```
     * должен выбросить указанное исключение.
     */
    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public @interface ExceptionTest {
        /**
         * Исключение, которое должен выбросить аннотированный метод,
         * чтобы тест считался пройденным. (Тест может выбросить
         * исключение любого подтипа указанного типа.)
         */
        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 файл `package-info.java` может (но не обязан) содержать объявление пакета и аннотации уровня пакета.

Двумя аспектами экспортируемого API класса, которые зачастую игнорируются, являются потокобезопасность и возможность сериализации. Уровень потокобезопасности класса должен быть документирован в соответствии с указаниями в статье 70. Если класс является сериализуемым, нужно документировать его сериализованную форму, как описано в статье 75.

Утилита Javadoc имеет возможность «наследовать» комментарии к методам. Если метод не имеет doc-комментария, Javadoc находит копирует в него ближайший doc-комментарий из супертипов, 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 от супертипов, используя тег `{@inheritDoc}`. Это означает в том числе то, что классы 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 неё есть несколько ограничений. Подробности выходят за рамки этой книги.

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 проверки кода HTML доступны в Интернете для загрузки, либо вы можете воспользоваться веб-приложением валидатора W3C [W3C-validator].

Следует добавить ещё 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 этого API. Если такой документ существует, то комментарии к документации в соответствующем классе или пакете должны на него ссылаться.

Соглашения, описанные в данной статье, покрывают основы написания комментариев Javadoc. Более детальное руководство содержится в статье How To Write Doc Comments [Javadoc-guide], выпущенной компанией Sun. Имеются плагины для IDE, которые проверяют следование многим из этих правил [Burn01].

Подведём итоги. Комментарии Javadoc — самый лучший, самый эффективный способ документирования API. Написание doc-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, причём метасимволы HTML необходимо маскировать escape-последовательностями или тегами `{@code}` и `{@literal}`.

Глава 8. Общие вопросы программирования

Эта глава посвящена обсуждению основных элементов языка Java. В ней рассматриваются интерпретация локальных переменных, использование библиотек и различных типов данных, а также две внеязыковые возможности: *рефлексия* (reflection) и *машинозависимые методы* (native methods). Наконец, обсуждаются оптимизация и соглашения по именованию.

Статья 45. Сводите к минимуму область видимости локальных переменных

Эта статья по своей сути схожа со статьёй 13 «Сводите к минимуму доступность классов и членов». Сужая область видимости локальных переменных, вы повышаете удобство чтения и сопровождения своего кода и сокращаете вероятность возникновения ошибок.

Старые языки программирования, такие как C, обязывали объявлять локальные переменные 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 инструкция.

> **Комментарий.** Это замечание насчёт языка C верно только для версии C89 и выше. С появлением стандарта C99 появилась возможность объявлять локальную переменную в любом месте блока, как и в C++ и в Java.

Самый сильный приём сужения области видимости локальной переменной заключается в объявлении ее в том месте, где она впервые используется. Объявление переменной до ее использования только засоряет программу: появляется ещё 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 объявление, если та находится далеко от места первого использования переменной.

Слишком раннее объявление локальной переменной может привести к нежелательному расширению её области видимости не только вверх, но и вниз. Область видимости локальной переменной начинается в месте её объявления и заканчивается с завершением блока, содержавшего эту объявление. Если переменная объявлена за пределами блока, в котором она используется, то она остаётся 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 каждое объявление локальной переменной должна содержать инициализатор. Если у вас недостаточно информации для правильной инициализации переменной, вы должны отложить объявление до той поры, пока она не появится. Исключение из этого правила связано с использованием инструкций `try/catch`. Если для инициализации переменной применяется метод, выбрасывающий проверяемое исключение, то инициализация переменной должна осуществляться внутри блока `try`. Если переменная должна использоваться за пределами блока `try`, объявлять ее следует перед блоком `try`, там, где она ещё не может быть «правильно инициализирована» (статья 53).

> **Комментарий.** Начиная с Java 7, это ограничение стало менее актуальным для типов, реализующих интерфейс `AutoCloseable`. Локальные переменные этих типов теперь часто объявляются и инициализируются в объявлении блока `try` с ресурсами, что позволяет обрабатывать выбрасываемые инициализатором исключения в блоке `catch`.

Циклы предоставляют ещё больше возможностей для сужения области видимости переменных. Цикл `for` позволяет объявлять переменные цикла (loop variables), 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 обхода коллекции (статья 46):

```
// Предпочтительная идиома для обхода коллекции
for (Element e : c) {
    doSomething(e);
}
```

До версии 1.5 предпочтительной была другая идиома (и у неё до сих пор есть

оправданные применения):

```
// До версии 1.5 не было циклов for-each и обобщённых типов
for (Iterator i = c. iterator(); i.hasNext(); ) {
    doSomething((Element) i.next());
}
```

Для пояснения, почему эти циклы `for` предпочтительнее 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 переменная `i`, которая, к сожалению, остаётся в области видимости. Полученный код компилируется без замечаний и выполняется без исключений, 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, она может оставаться незамеченной долгое время.

Если бы аналогичная ошибка копирования была допущена при применении цикла `for`, полученный код не скомпилировался бы. Для той области, где располагается второй цикл, переменная первого цикла уже была бы вне области 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 переиспользование даже считается более красивым.

Идиома цикла `for` имеет ещё 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.

Приведём ещё одну идиому цикла, 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 неё есть две переменные цикла: `i` и `n`, и обе имеют абсолютно правильную область видимости. Вторая переменная, `n`, используется для хранения границы значений первой, тем самым избегая дополнительных затрат на расчёты на каждом шаге цикла. Как правило, вам следует использовать эту идиому, если проверка цикла включает в себя запуск метода, гарантирующий возврат одного и того же результата на каждом шаге цикла.

Последний приём, позволяющий уменьшить область видимости локальных переменных, заключается в **создании небольших методов с чётким назначением**. 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 46. Предпочитайте циклы for-each традиционным циклам for

До версии 1.5 для обхода коллекции предпочтительнее было использовать следующую идиому:

```
// Более не является предпочтительной идиомой для обхода коллекции!
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next()); // (No generics before 1.5)
}
```

Так выглядела наиболее предпочтительная идиома для обхода массива:

```
// Более не является предпочтительной идиомой для обхода 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 предоставят возможность для ошибки. Итератор встречается трижды в каждом цикле, а переменная индекса — четыре раза, что даёт большую возможность использовать неверную переменную. Если вы так сделаете, нет никакой гарантии, что компилятор сможет заметить проблему.

Цикл for-each, введённый 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 обхода коллекций или массивов
for (Element e : elements) {
    doSomething(e);
}
```

Двоеточие следует читать как «в». Следовательно, цикл будет читаться как «для каждого элемента `e` в `elements`. Обратите внимание, что использование цикла for-each не наносит никакого ущерба производительности, даже для массивов. На самом деле такой

цикл может предложить небольшое преимущество в производительности над простым циклом `for` в некоторых обстоятельствах, так как он подсчитывает границу 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 традиционным циклом `for` даже больше, когда речь заходит о вложенных обходах 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. Её делают даже многие эксперты в программировании. 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 масти карт, цикл выбросит `NoSuchElementException`.

Если вам особенно не повезёт и размер внешней коллекции кратен размеру 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());
```

Эта программа не выводит ошибку, но она печатает только шесть двойных значений (от `"ONE ONE"` до `"SIX SIX"`) 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<Face> i = faces.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
}
```

Если вместо этого использовать вложенный цикл for-each, то проблема просто исчезает. Получившийся в результате код настолько краток, насколько возможно:

```
// Предпочтительная 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 даёт 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.

Подведём итоги. Цикл for-each обладает сильнейшими преимуществами по сравнению с традиционным циклом `for` по части ясности и профилактики 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 элементов, тогда вам нужен итератор списка или индекс массива, чтобы задать значение элемента.
3. **Параллельная итерация** — если вам необходимо обойти несколько коллекций одновременно, то вам нужен явный контроль над итератором или переменной индекса, чтобы все итераторы или переменные индекса могли продвинуться к следующему значению одновременно (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.

Статья 47. Изучите библиотеки и пользуйтесь ими

Предположим, вам нужно генерировать случайные целые числа в диапазоне от нуля до некоторой верхней границы. Столкнувшись с такой распространённой 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 начнёт 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 чётко. Графически это демонстрируется 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 полумиллиону, однако, запустив эту программу, вы обнаружите, что она печатает число, близкое к 666 666. Две трети чисел, сгенерированных методом `random`, попадает в нижнюю половину диапазона!

Третий недостаток представленного метода `random` заключается в том, что он может, хотя и редко, вернуть абсолютно неприемлемый результат, выходящий за пределы указанного диапазона. Это происходит потому, что метод пытается преобразовать значение, возвращённое 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 (`%`) вернёт 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 того, 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 провёл 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. Большинству программистов хотелось бы 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, причём 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. За годы развития платформы Java многие библиотеки были переписаны для улучшения производительности, порой несколько раз.

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 серьёзной 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.** С каждой новой версией платформы Java компания Sun публикует веб-страницу, описывающую её новые возможности. С этими страницами следует ознакомиться. Библиотеки слишком объемны, чтобы просматривать всю документацию, однако **каждый программист должен быть знаком с содержимым пакетов `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 работать с коллекциями независимо от деталей их реализации. Она сокращает объёмы 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, прочитайте документацию на веб-сайте Sun [Collections] или учебник [Bloch06].

В версии 1.5 появился пакет `java.util.concurrent` с утилитами многопоточности. Этот пакет содержит как высокоуровневые утилиты совместимости для упрощения многопоточного программирования, так и низкоуровневые примитивы многопоточности, что позволяет экспертам писать свои собственные высокоуровневые многопоточные 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.

Подведём итоги. Не изобретайте велосипед. 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 ещё лучше. Мы не ставим под сомнение ваши способности как программиста, однако библиотечному коду уделяется гораздо больше внимания, чем может позволить себе большинство разработчиков при реализации той же функциональности.

Статья 48. Если требуются точные ответы, избегайте использования типов float и double

Типы `float` и `double` в первую очередь предназначены для научных и инженерных расчётов. Они реализуют *двоичную 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 их нельзя использовать там, где нужны именно точные результаты. **Типы `float` и `double` не подходят для денежных расчётов**, 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? Приведём фрагмент наивной программы, которая пытается ответить на этот вопрос:

```
System.out.println(1.03 - 0.42);
```

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 * 0.10);
```

Если верить этому фрагменту программы, то вы получите `$0.09999999999999998`.

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 ещё есть возможность взять следующую конфету. Сколько конфет вы купите и сколько получите сдачи? Решим эту задачу следующим образом:

```
// Ошибка: использование плавающей точки для денежных расчётов!
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 ещё `$0.3999999999999999`. Но это неправильный ответ! Правильный путь решения задачи заключается в том, чтобы **применять для денежных расчётов типы `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 производителен, как 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 расчёты 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) {
        itemsBought++;
        funds -= price;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: " + funds + " cents");
}
```

Подведём итоги. Не используйте типы `float` и `double` для вычислений, требующих точного результата. 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 ещё то преимущество, что он даёт вам полный контроль над округлением: для любой операции, завершающейся округлением, предоставляется на выбор восемь режимов округления. Это пригодится, если вы будете выполнять экономические расчёты с жёстко 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 придётся работать с `BigDecimal`.

Статья 49. Предпочитайте примитивные типы упакованным примитивным типам

Система типов языка Java состоит из двух частей: *примитивные типы* (primitive types), такие как `int`, `double` и `boolean`, и *ссылочные типы* (reference types), такие как `String` и `List`. Каждому примитивному типу соответствует ссылочный тип, называемый *упакованным примитивным типом* (boxed primitive). Упакованными примитивными типами, соответствующими обычным примитивным типам `int`, `double` и `boolean`, являются `Integer`, `Double` и `Boolean` соответственно.

В версии 1.5 к языку добавились *автоупаковка* (autoboxing) и *автораспаковка* (auto-unboxing). 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 ещё 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 польностью функциональные значения, в то время как упакованные примитивные типы имеют 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 значений `Integer`. (Вспоминаем, что метод компаратора `compare` возвращает

число — отрицательное, положительное или ноль, в зависимости от того, является ли первый аргумент меньшим, равным или большим второго.). На практике вам не нужно писать компаратор, так как он реализует естественный порядок типа `Integer`, который можно получить и без компаратора, но пример получается интересный:

```
// Нерабочий компаратор - можете ли вы найти ошибку?
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 пройдёт 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 `naturalOrder.compare(new Integer(42), new Integer(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 нормально. 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. Предположим, что нет. Тогда следующая проверка проверяет выражение `first == second`, который выполняет *сравнение идентичностей* (identity comparison) двух ссылок на объект. Если `first` и `second` ссылаются на различные экземпляры `Integer`, представляющие различные значения `int`, то это сравнение вернёт значение `false` и компаратор ошибочно выведет значение 1, которое означает, что первое значение `Integer` больше второго. **Использование оператора `==` на упакованных примитивных типах почти всегда является ошибкой.**

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 так же странно. Она выбрасывает исключение `NullPointerException` при вычислении выражения `(i == 42)`. Проблема в том, что `i` — это `Integer`, а не `int`, и, как и у всех полей со ссылками на объекты, его начальным значением является `null`. Когда программа проверяет выражение `(i == 42)`, она сравнивает `Integer` и `int`. Почти всегда, **когда вы смешиваете обычные и упакованные примитивные типы одной операцией, упакованный примитивный тип автоматически распаковывается**, и данный случай — не исключение. Если распаковываемая ссылка на объект равна `null`, то вы получите `NullPointerException`. То, что демонстрирует данная программа, может случиться где угодно. Исправить программу просто: объявите `i` как `int`, а не `Integer`.

Наконец, рассмотрим программу (статья 5):

```
// Ужасно медленная программа! Можете найти, где создаётся объект?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

Эта программа намного медленнее, чем должна быть, потому что она случайно объявила локальную переменную (`sum`) как имеющую упакованный примитивный тип `Long`, а не обычный примитивный тип `long`. Программа компилируется без ошибок или предупреждений, и переменная постоянно упаковывается и распаковывается, снижая производительность.

У всех трёх 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 — серьёзному снижению производительности.

Итак, когда же следует использовать упакованные примитивные типы? Их использование оправданно в нескольких случаях. Во-первых, в качестве элементов, ключей и значений коллекций. Вы не можете поместить обычные примитивные типы в коллекцию, поэтому вам придётся использовать упакованные примитивные типы. Это частный случай более общего правила, говорящего, что вы обязаны использовать упакованные примитивные типы в качестве параметров типа для параметризованных типов (глава 5), потому что язык не позволяет вам использовать обычные примитивные типы. Например, вы не можете объявить переменную как `ThreadLocal<int>`, вместо этого вам нужно использовать `ThreadLocal<Integer>`. Наконец, вы обязаны использовать упакованные примитивные типы при вызове методов с помощью рефлексии (статья 53).

Подведём 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 расчёты и с обычными и с упакованными примитивными типами, она выполняет распаковку, а **когда ваша программа выполняет распаковку, она может выбросить `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 другой 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 оправданно 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), вам следует им воспользоваться. Если такового нет, вы должны его написать. Этот совет кажется очевидным, но его часто нарушают.

Строки — плохая замена перечислениям. Как говорилось в статье 30, перечислимые типы представляют константы перечисления значительно лучше, чем строки.

Строки — плохая замена составным типам (aggregate types). Если некая сущность имеет несколько составных частей, то попытка представить ее одной строкой — обычно неподходящее решение. Для примера приведём строку кода из реальной системы (идентификатор изменён, 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, трудоёмкая 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* (capabilities). Иногда строки используются для обеспечения доступа к неким функциональным возможностям. Например, рассмотрим механизм создания переменных, привязанных к определённому потоку. Этот механизм позволяет создавать переменные, в которых каждый поток может хранить собственное значение. В библиотеках Java с версии 1.2 есть возможность использования потоко-локальных переменных (thread-local variables), 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 мандата
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

имён. 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, злоумышленник может намеренно воспользоваться тем же ключом, что и клиент.

API можно исправить, если эту строку заменить неподделываемым ключом, который иногда называют мандатом (capability):

```
public class ThreadLocal {
    private ThreadLocal() { }  // Не порождает экземпляров

    public static class Key {  // (Мандат)
        Key() { }
    }

    // Генерирует уникальный, неподделываемый ключ
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

Это решает обе проблемы с API, использующим строки, но можно сделать ещё 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 избавиться, а вложенный класс переименовать в `ThreadLocal`:

```
public final class ThreadLocal {
    private ThreadLocal();
    public void set(Object value);
```

```
    public Object get();
}
```

Этот API не является типобезопасным, потому что вам приходится приводить тип значения от `Object` к его настоящему типу, когда вы извлекаете его из привязанной к потоку локальной переменной. Невозможно сделать типобезопасным оригинальный API на основе строки, и сложно сделать типобезопасным API на основе ключа, но очень просто сделать типобезопасным новый API, обобщив класс `ThreadLocal` (статья 26):

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

Примерно такой API реализует `java.util.ThreadLocal`. Помимо того, что этот интерфейс разрешает проблемы API, применяющего строки, он быстрее и элегантнее любого API, использующего ключи.

Подведём 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 примитивных, перечислимых и составных типов.

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 он не масштабируется. **Время, которое необходимо оператору конкатенации для объединения *n* строк в цикле, 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 создаёт строковое представление для выписываемого счета, последовательно объединяя строки для каждого пункта в счёте:

```
// Неуместное применения оператора +, 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 незавершённого выражения. (Класс `StringBuilder`, появившийся в версии 1.5, является несинхронизированной заменой класса `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 ещё 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.

Комментарий. В Java 8 мы также могли бы воспользоваться потоковыми операциями с коллектором `Collectors.joining`, который использует `StringBuilder` внутри себя.

```
  return IntStream.range(0, numItems())
      .mapToObj(this::lineForItem)
      .collect(Collectors.joining());
```

Мораль проста: не пользуйтесь оператором конкатенации для объединения большого числа строк, если производительность имеет важное значение. Лучше применять метод `append` класса `StringBuilder`. В качестве альтернативы можно использовать массив символов или обрабатывать строки по одной, не объединяя их.

Статья 52. Используйте интерфейсы для ссылок на объекты

В статье 40 даётся совет: в качестве типа параметра указывать интерфейс, а не класс. В более общей формулировке этот совет звучит так: ссылаясь на объект, вы должны отдавать предпочтение не классу, а интерфейсу. **Если есть подходящие типы интерфейсов, то параметры, возвращаемые значения, переменные и поля следует объявлять с их использованием.** Единственным случаем, когда вам нужно ссылаться на класс объекта, является его создание. 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, — это 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 это объявление, ничего не знал о прежнем типе, который реализовывал интерфейс. Поэтому он не должен заметить изменение объявления.

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 первое объявление, 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 объявлении переменной.

Тогда зачем менять реализацию? Возможно, потому, что новая реализация предлагает повышенную производительность, либо потому, что она обеспечивает необходимую дополнительную функциональность. В качестве примера из реальной жизни возьмём класс `ThreadLocal`. Внутри этот класс реализован через поле `Map` в классе `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 `ThreadLocal` стал работать быстрее.

Будь это поле объявлевано 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. Объявление 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 расчёте на несколько реализаций. Как правило, это ненаследуемые (`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 является частью фреймворка, построенного 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 `LinkedHashMap`. Тогда этот класс нужно использовать для ссылки на его экземпляры только тогда, когда программа использует дополнительные методы. Случаи, когда такой класс имеет смысл использовать как тип параметра, редки (статья 40).

Приведённые варианты не исчерпывают всех случаев, а лишь дают представление о множестве ситуаций, в котороых 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 интерфейсы рефлексии

Механизм *рефлексии* (reflection, пакет `java.lang.reflect`) предоставляет программный доступ к информации о загруженных классах. Имея экземпляр класса `Class`, вы можете получить экземпляры классов `Constructor`, `Method` и `Field`, соответствующие

конструкторам, методам и полям данного объекта `Class`. Эти объекты обеспечивают программный доступ к названиям методов в классе, к типам полей, сигнатурам методов и т.д.

Более того, классы `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` даёт возможность вызвать любой метод из любого объекта любого класса (при соблюдении обычных ограничений, связанных с безопасностью). Механизм рефлексии разрешает одному классу использовать другой, даже если последний во время компиляции первого ещё не существовал. Однако такая мощь обходится недёшево:

- **Вы лишаетесь всех преимуществ проверки типов на этапе компиляции**, в том числе проверки исключений. Если программа попытается вызвать несуществующий или недоступный метод с помощью рефлексии, то в случае отсутствия специальных мер предосторожности произойдёт сбой программы во время выполнения.
- **Программный код, необходимый для рефлексии классов, неуклюж и многословен.** Его тяжело писать и трудно читать.
- **Страдает производительность.** Рефлексивный 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 рефлексии классов первоначально создавался для средств разработки, обеспечивающих построение приложения из компонентов. Как правило, такие инструменты осуществляют загрузку классов по требованию и используют механизм рефлексии 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 рефлексивный доступ. Отражение применяется только на *этапе проектирования приложения* (design time). **Как правило, обычные приложения на стадии выполнения не должны пользоваться доступом к объектам с помощью рефлексии.**

Есть лишь несколько сложных приложений, которым необходим механизм рефлексии. В их число входят визуализаторы классов, инспекторы объектов, анализаторы программного кода и интерпретирующие встроенные системы. Рефлексию можно также использовать в системах удалённого вызова процедур (remote procedure call, RPC) с целью снижения потребности в компиляторах объектов-прослоек (stub). 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 рефлексии, если будете применять его в ограниченном масштабе. 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 создать экземпляры с помощью рефлексии, 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 приведём программу, которая создаёт экземпляр интерфейса `Set`, чей класс задан первым аргументом командной строки. Остальные аргументы командной строки программа вставляет в полученное множество и затем распечатывает его. При выводе аргументов программа уничтожает дубликаты. 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 `TreeSet` сортируются.

```
// Рефлексивное 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);
        }

        // Создаёт 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 instantiable.");
            System.exit(1);
        }

        // Использует созданное множество
        s.addAll(Arrays.asList(args).subList(1, args.length));
        System.out.println(s);
    }
```

Хотя эта программа является всего лишь игрушкой, она показывает мощный приём. Этот код легко превратить в универсальную программу проверки множеств, 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 полноценного фреймворка поставщиков услуг (service provider framework) (статья 1). В большинстве случаев описанный приём — это все, что нужно знать о рефлексии.

Этот пример иллюстрирует два недостатка системы рефлексии. Во-первых, во время его выполнения могут возникнуть три ошибки, которые, если бы не использовался механизм рефлексии, были бы обнаружены ещё 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 создаёт 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 рефлексии.

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 обобщённых типов, но не отображает реальной проблемы. Как скрыть предупреждения, описано в статье 24.

Ещё один отклоняющийся от темы вопрос, который заслуживает упоминания, — это использование программой `System.exit`. Очень редко возникает ситуация, когда стоит использовать этот метод, завершающий работу всей виртуальной машины. Тем не менее он подходит при ненормальном завершении утилиты командной строки.

> **Комментарий.** Применение `System.exit` является особенно спорным в случае, когда метод `main` контролируете не вы: например, если вы пишете библиотеку, которая будет использоваться в самых разных программах. Кроме того, не следует применять `System.exit` в серверных приложениях, исполняющихся внутри контейнеров сервлетов или серверов приложений Java EE, поскольку при вызове `System.exit` завершит работу весь контейнер вместе со всеми исполняющимися в нём приложениями. Ключевое правило здесь такое: если процесс инициализировали не вы, не вам его и завершать.

Приемлемым, хотя и редким вариантом использования рефлексии является разрешение 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. Приём 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 рефлексии. 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 усечённого функционала.

Подведём итоги. Рефлексия — это мощный инструмент, который необходим для решения определённых 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 рефлексии 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) даёт возможность приложениям на языке Java делать вызов машинозависимых методов (native methods), т.е. специальных методов, написанных на машинозависимом языке программирования, таком как C или C++. 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 файлов. Они обеспечивали доступ к библиотекам унаследованного кода (legacy code), 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 унаследованному коду.

Использование машинозависимых методов для повышения производительности редко оправдывает себя. В ранних версиях платформы (до 1.3) 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 включён пакет `java.math`, класс `BigInteger` был реализован поверх быстрой библиотеки арифметических операций с произвольной точностью, написанной на языке C. В то время это было необходимо для получения приемлемой производительности. В версии 1.3 класс `BigInteger` был полностью переписан на языке Java и тщательно оптимизирован. Даже тогда новая версия работала быстрее старой, и за прошедшие с тех пор годы виртуальные машины стали только быстрее.

Применение машинозависимых методов имеет серьёзные 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. Приложения, использующие машинозависимый код, труднее отлаживать. С переходом на машинозависимый код и с возвратом в Java связаны фиксированные 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.

Подведём 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. Одна-единственная ошибка в машинозависимом коде может полностью разрушить все ваше приложение.

Статья 55. Соблюдайте осторожность при оптимизации

Есть три афоризма, посвящённые оптимизации, которые обязан знать каждый. Возможно, они пострадали от слишком частого цитирования, однако приведём их на тот случай, если вы с ними не знакомы:

- Во имя эффективности (но при этом не обязательно достигая её) программисты

допускают больше грехов, чем по каким-либо иным причинам, включая непроходимую тупость.
— *Уильям Вульф (William A. Wulf) [Wulf72]*

- Нам нужно забыть о мелких усовершенствованиях, скажем, в 97% случаев: преждевременная оптимизация — корень всех зол.
  — *Дональд Кнут (Donald E. Knuth) [Knuth74]*

- При оптимизации мы следуем двум правилам:
  Правило 1. Не делайте оптимизаций.
  Правило 2 (только для экспертов). Не делайте оптимизаций до поры до времени — то есть пока у вас нет абсолютно чёткого неоптимизированного решения.
  — *М. А. Джексон (M. A. Jackson) [Jackson75]*

Эти афоризмы на два десятилетия опередили язык программирования Java. В них отражена сущая правда: при оптимизации 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. Старайтесь писать хорошие программы, а не быстрые. Если хорошая программа работает недостаточно быстро, ее архитектура позволит осуществить оптимизацию. Хорошие программы 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 путём 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, сетевые протоколы 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 защитных копий (статья 39). Точно так же использование наследования в открытом классе, для которого уместнее была бы композиция, навсегда 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 определённой реализации этого интерфейса, даже несмотря на то, что в будущем, возможно, будут написаны ещё 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 создаёт новый экземпляр `Dimension`. И хотя, начиная с версии 1.3, создание небольших объектов обходится относительно дёшево, бесполезное создание миллионов объектов может нанести производительности приложения реальный ущерб.

В данном случае имеется несколько альтернатив. В идеале класс `Dimension` должен был изначально быть неизменяемым (статья 15). Кроме того, 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 API 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. **Искажать API ради улучшения производительности — очень плохая идея.** Проблемы с производительностью, которые заставили вас переделать API, могут исчезнуть с появлением новой платформы или других базовых программ, а вот искажённый API и связанная с ним головная боль останутся с вами навсегда.

После того как вы тщательно спроектировали программу и выстроили чёткую, краткую и хорошо структурированную ее реализацию, можно подумать об оптимизации, если, конечно, вы ещё не удовлетворены производительностью программы. Напомним два правила Джексона: «не делайте этого» и «не делайте этого пока что (для экспертов)». Он мог бы добавить ещё одно: измеряйте производительность до и после попытки ее оптимизации. Нередко попытки оптимизации не оказывают поддающегося измерению влияния на производительность, иногда они даже ухудшают ее. Основная причина заключается в том, что сложно догадаться, где именно происходят потери производительности. 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 ещё худшей) зависимостью, никакие настройки эту проблему не решат. Следовательно, вам придётся 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 чёткой *модели производительности* (performance model). Нет чёткого 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 надёжное 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 оптимизации для каждой из этих реализаций.

Иногда в отношении производительности вам придётся идти на компромисс между различными реализациями JVM.

Подведём 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, сетевыми протоколами 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 имён придерживайтесь общепринятых соглашений

Платформа Java обладает хорошо устоявшимся набором соглашений, касающихся выбора имён (naming conventions). Многие из них приведены в «The Java Language Specification» [JLS, 6.8]. Соглашения об именовании делятся на две категории: типографические и грамматические. Типографических соглашений, касающихся выбора имён 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 программистов, работающих с этим кодом, а также способствовать появлению ложных допущений, приводящих к ошибкам. В данной статье приведён обзор этих соглашений.

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.emu`, `com.sun`, `gov.nsa`.

Исключение из этого правила составляют стандартные библиотеки и расширения платформы Java, чьи названия пакетов 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].

Остаток названия пакета должен 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 аббревиатуры, 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 поддержки на уровне языка.

Названия классов и интерфейсов состоят из одного или нескольких слов, причём в каждом слове первая буква должна быть заглавной, например, `Timer` или `TimerTask`. Необходимо избегать сокращений, кроме стандартных аббревиатур и нескольких общепринятых сокращений, таких как `max` и `min`. Нет полного единодушия по поводу того, должны ли аббревиатуры 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 аббревиатур. Какое название класса вы предпочли бы увидеть: `HTTPURL` или `HttpUrl`?

Комментарий. На данный момент сложилось неформальное соглашение о том, что аббревиатуры из четырёх букв и более (например, `Http`) записываются со всеми строчными буквами, кроме первой (например, `Http`), а из двух — прописными (например, `DB`). В трёхбуквенных же сокращениях в разных библиотеках встречаются оба варианта (например, `URL` и `Url`), хотя в стандартной библиотеке они пишутся прописными буквами (например, `URLConnection` или `XMLEventReader`).

Названия методов и полей подчиняются тем же самым типографическим 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 аббревиатура, она вся 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 отделённых друг от друга символом подчёркивания, например, `VALUES` или `NEGATIVE_INFINITY`. Поле-константа — это поле `static final`, значение которого является неизменяемым. Если поле `static final` имеет примитивный тип или неизменяемый ссылочный тип (статья 15), то это поле-константа. Например, константы перечислимых типов являются полями-константами. Если поле `static final` имеет изменяемый ссылочный тип, оно все ещё может быть полем-константой, если объект, на который оно ссылается, является неизменяемым. Заметим, что поля-константы — это единственное место, где допустимо использование символа подчёркивания.

Названия локальных переменных подчиняются тем же типографическим 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 параметров типа 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 значений словаря, `X` для исключений. Последовательность произвольных типов может быть `T`, `U`, `V` или `T1`, `T2`, `T3`. Примеры типографических соглашений приведены в таблице:

Тип идентификатора	Примеры
Пакет	`com.google.inject`, `org.joda.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`
Параметр типа	`T`, `E`, `K`, `V`, `X`, `T1`, `T2`

По сравнению с типографическими 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`, `BufferedWriter` или `ChessPiece`. Интерфейсы именуются так же, как классы, например, `Collection` и `Comparator`, либо применяются названия-прилагательные с окончаниями `-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 идёт слово `is`, а потом существительное, именная конструкция или любое слово (фраза), играющее роль прилагательного, например `isDigit`, `isProbablePrime`, `isEmpty`, `isEnabled`, `isRunning`.

Для именования методов, возвращающих значение не типа `boolean`, 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 bean-классу [JavaBeans]. Ее можно также рекомендовать, если в будущем вы собираетесь превратить свой класс в bean. Наконец, серьёзные 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* (view, 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 фабричных методов 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 спроектированные API практически не имеют открытых полей. 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 ещё менее обязательно.

Подведём итоги. Изучите стандартные соглашения по именованию и доведите их использование до автоматизма. Типографические соглашения просты и практически однозначны; грамматические соглашения более сложны и свободны. 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.

Глава 9. Исключения

Если исключения (exceptions) используются наилучшим образом, они способствуют написанию понятных, надёжных 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 повезёт, 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 выбрасыванием исключения `ArrayIndexOutOfBoundException`, его 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 была непродуманная попытка улучшить производительность, исходящая из ложного умозаключения о том, 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, выброс и перехват исключения дорого обходятся системе.
- Размещение кода внутри блока `try-catch` препятствует выполнению определённых 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, не просто снижает производительность и делает непонятным программный код. Нет гарантий, что она вообще будет работать! При появлении непредусмотренной разработчиком ошибки эта идиома может замаскировать настоящую ошибку, тихо подавив её, 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 идиомы цикла с использованием исключений то исключение, которое было вызвано настоящей ошибкой, 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 нормальном выполнении программы 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 поглощён неуклонным совершенствованием реализаций JVM. А вот коварные ошибки и сложность поддержки, вызываемые чересчур 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 определённых непредсказуемых условий, то в этом же классе, как правило, должен присутствовать отдельный метод проверки состояния (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, приведённый 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 приём не годится, поскольку `null` является допустимым значением для метода `next`.

Приведём 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 приём с особым возвращаемым значением, поскольку состояние объекта может поменяться в период между вызовом метода проверки состояния и вызовом соответствующего метода, который зависит от состояния объекта.

Комментарий. Как мы помним, подобная разновидность условия гонки (race condition) называется ошибкой TOCTTOU (time to check to time of use). Кстати, это соображение подсказывает и необходимое исключение из общего правила: при операциях с таким разделяемым ресурсом, как файловая система, вы *обязаны* обрабатывать исключения, поскольку состояние файловой системы могло измениться между временем проверки на допустимость операции и временем её применения. Например, такой код будет неверным:

```
if (!Files.exists(path)) {
    Files.write(path, Arrays.asList("Blank file"));
}
```

Если в файл нужно записать только в том случае, когда его не существует, то этот код перезапишет существующий файл, если он был создан другим процессом или потоком во время между вызовом `Files.exists` и `Files.write`. Единственным способом корректно обработать такую ситуацию будет отлов исключения:

```
try {
    // CREATE_NEW гарантирует, что мы не попытаемся
    // перезаписать существующий файл
    Files.write(path, Arrays.asList("Blank file"),
        StandardOpenOption.CREATE_NEW,
        StandardOpenOption.WRITE);
} catch (FileAlreadyExistsException e) {
    // обработать ситуацию существования файла
}
```

> Подобным образом может потребоваться поступать с любыми разделяемыми ресурсами, для которых нет возможности синхронизации с другими процессами или потоками, совместно использующими этот ресурс.

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.

Подведём 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 проверяемые исключения для восстановления, для программных ошибок используйте исключения времени выполнения

В языке программирования Java предусмотрены три типа объектов `Throwable`: проверяемые исключения (checked exceptions), исключения времени выполнения (runtime exceptions) и ошибки (errors). 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 проверяемым и непроверяемым исключениями гласит: используйте проверяемые исключения в тех случаях, когда есть основания полагать, что вызывающая сторона способна их обработать. Выбрасывая проверяемое исключение, вы принуждаете вызывающую сторону обрабатывать его в блоке `catch` или передавать дальше. Каждое проверяемое исключение, которое, согласно объявлению, может выбросить некий метод, является, таким образом, серьёзным предупреждением для пользователя API о том, что при вызове данного метода могут возникнуть соответствующие условия.

Предоставляя пользователю API проверяемое исключение, разработчик API передаёт

ему право осуществлять восстановление после наступления условия, породившего исключение. Пользователь может пренебречь этим правом, перехватив исключение и проигнорировав его. Однако, как правило, это оказывается плохим решением (статья 65).

Есть два типа непроверяемых объектов `Throwable`: исключения времени выполнения и ошибки. Поведение у них одинаковое: ни тот ни другой не нужно и, вообще говоря, нельзя перехватывать. Если программа выбрасывает непроверяемое исключение или ошибку, то, как правило, это означает, что восстановление невозможно и дальнейшее выполнение программы принесёт больше вреда, чем пользы. Если программа не перехватывает такой объект, его появление вызовет остановку текущего потока 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, 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.

X отя в спецификации языка Java это не оговорено, существует строго соблюдаемое соглашение о том, что ошибки зарезервированы в JVM для сигнализации о дефиците ресурсов, нарушении инвариантов и других условияях, делающие невозможным дальнейшее выполнение программы [Chan98, HorstmanOO]. Поскольку эти соглашения признаны практически повсеместно, лучше вообще не создавать новых подклассов класса `Error`. Все реализуемые вами непроверяемые исключения должны прямо или косвенно наследовать класс `RuntimeException`.

Можно определить подкласс класса `Throwable`, который не наследует ни от одного из 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 проверяемые исключения (которые являются подклассами класса `Exception`, но не `RuntimeException`). Когда же имеет смысл объявлять такие классы? Если одним словом, то никогда. Не имея никаких преимуществ перед обычными проверяемыми исключениеми, они только запутают пользователей вашего API.

Подведём итоги. Для ситуаций, когда можно обработать ошибку и продолжить

исполнение, используйте проверяемые исключения, для программных ошибок применяйте исключения времени выполнения. Разумеется, ситуация не всегда будет настолько однозначной. 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 временным увеличением спроса, после такой ситуации вполне возможно восстановиться. Именно разработчик API принимает решение, возможно ли восстановление работоспособности программы в конкретном случае исчерпания ресурсов. Если вы считаете, что работоспособность можно восстановить, используйте проверяемое исключение. В противном случае применяйте исключение времени выполнения. Если неясно, возможно ли восстановление, то по причинам, описанным в статье 59, лучше остановиться на непроверяемом исключении.

Разработчики API часто забывают, что исключения — это полноценные 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 придётся разбирать строковое представление этого исключения, выуживая из него дополнительную информацию. Эта крайне плохая практика (статья 10). Классы редко указывают какие-либо детали в своём 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 ненадёжным.

Поскольку проверяемые 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 проверяемое исключение выбрасывается 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 ненужного использования проверяемых исключений

Проверяемые исключения — замечательная особенность языка программирования Java. В отличие от кодов возврата, они заставляют программиста отслеживать условия возникновения исключений, что значительно повышает надёжность приложения. Это означает, что злоупотребление проверяемыми исключениями может сделать API менее удобным для использования. Если метод выбрасывает одно или несколько проверяемых 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 объявлено, что этот код сам выбрасывает исключения и передаёт их дальше. В любом случае перед программистом стоит нелёгкая задача.

Такое решение оправданно, если даже при надлежащем применении API невозможно предотвратить возникновение условий для исключительной ситуации, однако пользующийся этим API программист, столкнувшись с таким исключением, мог бы предпринять какие-либо полезные действия. Если не выполняются оба этих условия, лучше пользоваться непроверяемым исключением. Роль лакмусовой бумажки в данном случае играет вопрос: как программист будет обрабатывать исключение? Является ли это решение лучшим:

```
} catch (TheCheckedException e) {
    throw new AssertionError(); // Этого случиться не может!
}
```

А что скажете об этом:

```
} catch (TheCheckedException e) {
    e.printStackTrace();        // Что ж, не повезло.
    System.exit(1);
}
```

Если программист, применяющий API, не может сделать ничего лучшего, то больше подходит непроверяемое исключение. Примером исключения, не выдерживающего подобной проверки, является `CloneNotSupportedException`. Оно выбрасывается методом `Object.clone`, который должен использоваться лишь для объектов,

реализующих интерфейс `Cloneable` (статья 11), и практически всегда имеет характер невозможного исключения. В этом случае проверяемое исключение не даёт программисту преимуществ, но требует от последнего дополнительных усилий и усложняет программу.

Дополнительные действия со стороны программиста, связанные с обработкой проверяемого исключения, значительно увеличиваются, если это единственное исключение, выбрасываемое данным методом. Если есть другие исключения, метод будет стоять в блоке `try`, так что для этого исключения понадобится всего лишь ещё один блок `catch`. Если же метод выбрасывает только одно проверяемое исключение, оно будет требовать, чтобы вызов соответствующего метода был помещён в блок `try`. В таких условиях имеет смысл подумать, не существует ли какого-либо способа избежать проверяемого исключения.

Один из приёмов, позволяющих превратить проверяемое исключение в непроверяемое, состоит в разбиении метода, выбрасывающего исключение, на два метода, первый из которых будет возвращать булево значение, указывающее, выбросилось ли бы исключение, если бы был вызван второй метод. Таким образом, в результате преобразования API последовательность вызова

```
// Вызов с проверяемым исключением
try {
    obj.action(args);
} catch (TheCheckedException e) {
    // Обработать исключительную ситуацию
}
```

принимает следующий вид:

```
// Вызов с использованием метода проверки состояния
// и непроверяемого исключения
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    // Обработать исключительную ситуацию
}
```

Комментарий. Именно такой характер имеют методы `hasNext` и `next` интерфейса `Iterator`.

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 упрощённую последовательность вызова:

```
obj.action(args);
```

Если вы предполагаете, что применение упрощённой последовательности вызова будет нормой, то описанное преобразование API приемлемо. API, полученный в результате этого преобразования, в сущности, получается тем же, 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 своё 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 методу `actionPermitted` придётся дублировать работу метода action, то от преобразования, возможно, придётся 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 платформы Java реализован основной набор непроверяемых исключений, покрывающий большую часть потребностей в исключениях для API. В этой статье обсуждаются наиболее часто применяемые исключения.

Повторное использование уже имеющихся исключений имеет несколько преимуществ. Прежде всего, 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 выбрасывается, когда вызываемому методу передаётся аргумент с неправильным значением. Например, `IllegalArgumentException` может выбрасываться в случае, если для параметра, указывающего количество повторов для некоей процедуры, передано отрицательное значение.

Другое часто используемое исключение — `IllegalStateException`. Оно обычно выбрасывается, если в соответствии с состоянием объекта вызов метода является неправомерным. Например, это исключение может выбрасываться, когда делается попытка использовать некий объект до его инициализации надлежащим образом.

В принципе можно утверждать, что все неправильные вызовы методов сводятся к неправильным аргументам или неправильному состоянию, но для определённых типов неправильных аргументов и состояний стандартно используются 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 выбрасываться 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, выбрасываться должно исключение `IndexOutOfBoundsException`, а не `IllegalArgumentException`.

Ещё одно универсальное исключение, о котором необходимо знать: `ConcurrentModificationException`. Оно должно выбрасываться, 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 — это `UnsupportedOperationException`. Оно выбрасывается, 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 выбрасывать это исключение, если кто-то попытается удалить элемент. В таблице ниже собраны самые распространённые из повторно используемых исключений.

Исключение	Повод для использования
`IllegalArgumentException`	Неправильное значение параметра
`IllegalStateException`	Состояние объекта неприемлемо для вызова метода
`NullPointerException`	Значение параметра равно `null`, а это запрещено
`IndexOutOfBoundsException`	Значение параметра, задающего индекс, выходит за пределы диапазона
`ConcurrentModificationException`	Обнаружена параллельная модификация объекта из разных потоков, а это запрещено
`UnsupportedOperationException`	Объект не поддерживает вызываемый метод

Помимо перечисленных исключений, при определённых обстоятельствах могут применяться и другие исключения. Например, при реализации таких арифметических объектов, как комплексные и рациональные числа, уместно пользоваться исключениями `ArithmeticException` и `NumberFormatException`. Если исключение отвечает вашим потребностям, пользуйтесь им, но только тогда, когда условия, при которых вы будете его выбрасывать, не вступают 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», приведённые в таблице, не являются взаимоисключающими. Рассмотрим, например, объект, соответствующий колоде карт. Предположим, что для него есть метод, осуществляющий выдачу карт из

колоды, причём в качестве аргумента ему передаётся 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. Выбрасывайте исключения, соответствующие абстракции

Если метод выбрасывает исключение, не имеющее видимой связи с решаемой задачей, это сбивает с толку. Часто это происходит, когда метод передаёт выше исключение, выброшенное 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 выбрасываемые 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, выбрасывать исключения, которые можно объяснить в терминах абстракции верхнего уровня**. Описываемая идиома, которую мы называем *трансляцией исключений* (exception translation), выглядит следующим образом:

```
// Трансляция исключения
try {
    // Используем абстракцию нижнего уровня
    // для выполнения нашей задачи
    ...
} catch (LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

Приведём 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<E>`:

```
/**
* Возвращает элемент, находящийся в указанной позиции в этом списке.
* @throws IndexOutOfBoundsException, если индекс находится
* за пределами диапазона (index < 0 || index >= size()).
*/
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

В тех случаях, когда исключение нижнего уровня может быть полезно при анализе ситуации, вызвавшей исключение, лучше использовать особый вид трансляции исключений, называемый *сцеплением исключений* (exception chaining). При этом исключение нижнего уровня (*причина*, cause) передаётся при создании исключения верхнего уровня, и метод доступа `Throwable.getCause` позволяет извлечь исключение нижнего уровня:

```
// Сцепление исключений
try {
    // Используем абстракцию нижнего уровня
    // для выполнения нашей задачи
    ...
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

Конструктор исключений верхнего уровня передаёт параметр `cause` конструктору

суперкласса, *учитывающему сцепление* (chaining-aware), так что в конце концов он передаётся конструктору класса `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 таких конструкторов, вы можете воспользоваться методом `initCause` класса `Throwable`. Сцепление исключений не только даёт вам программный доступ к причине (с помощью метода `getCause`), но и печатает стек вызовов исключения-причины после стека вызовов исключения верхнего уровня при вызове `printStackTrace`.

Хотя трансляция исключений лучше, чем бессмысленная передача наверх исключений с нижних уровней, злоупотреблять ею не следует. Самый хороший способ работы с исключениями 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 путём 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 логировать исключения, используя какой-либо механизм логирования, например, пакет `java.util.logging`, появившийся в версии 1.4. Это даёт возможность администратору исследовать возникшую проблему и в то же время изолирует от неё программный код клиента и конечного пользователя.

Подведём итоги. 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

выбрасываемые им исключения будут приемлемы для абстракции верхнего уровня. Сцепление исключений позволяет выбросить исключение, подходящее для верхнего уровня, и при этом сохранить нижележащую причину этого исключения для анализа ошибки (статья 63).

Статья 62. Для каждого метода документируйте все выбрасываемые им исключения

Описание выбрасываемых методом исключений составляет важную часть документации, которая необходима для правильного применения метода. Поэтому крайне важно, чтобы вы уделили время тщательному описанию всех исключений, выбрасываемых каждым методом.

Всегда объявляйте проверяемые исключения, выбрасываемые методом, по отдельности и чётко описывайте условия, при которых каждое из них выбрасывается, с помощью тега Javadoc `@throws`. Не пытайтесь сократить себе мысленную работу, объявив метод как выбрасывающий некий суперкласс исключений, вместо того чтобы объявлять несколько классов возможных исключений. Например, никогда не объявляйте метод как `throws Exception` или, что ещё хуже, `throws Throwable`. Помимо того, что такая формулировка не даёт программисту никакой информации о том, какие исключения могут быть выброшены данным методом, она значительно затрудняет работу с методом, поскольку перекрывает любое другое исключение, которое может быть выброшено в этом же месте.

Хотя язык Java не требует, чтобы программисты объявляли непроверяемые исключения, которые могут быть выброшены методом, имеет смысл документировать их столь же тщательно, как и проверяемые исключения. Непроверяемые исключения обычно представляют ошибки программирования (статья 40), и ознакомление программиста со всеми этими ошибками может помочь ему избежать их. Хорошо составленный перечень непроверяемых исключений, которые может выбросить метод, фактически описывает *предусловия* (preconditions) для его успешного выполнения. Важно, чтобы в документации к каждому методу были описаны его предусловия, а описание непроверяемых исключений как раз и является наилучшим способом выполнения этого требования.

Для методов интерфейса особенно важно, чтобы в документации были описаны непроверяемые исключения, которые могут быть ими выброшены. Такая документация является частью основного контракта интерфейса и обеспечивает единообразное поведение различных его реализаций.

Для описания каждого непроверяемого исключения, которое может быть выброшено методом, используйте тег Javadoc `@throws`, но *не* включайте непроверяемые исключения в объявление метода с помощью ключевого слова `throws`. Программист, пользующийся вашим API, должен знать, какие из исключений являются проверяемыми, а какие — нет, поскольку в первом и втором случаях на него возлагается различная ответственность. Наличие в документации описания, соответствующего тегу `@throws`, при отсутствии исключения в объявлении `throws` в заголовке метода создаёт мощный визуальный сигнал, помогающий программисту отличить проверяемые исключения от непроверяемых.

Следует отметить, что документирование всех непроверяемых исключений, которые могут быть выброшены каждым методом — это идеал, который не всегда достижим в реальности. Когда производится пересмотр класса и предоставляемый пользователю метод меняется так, что начинает выбросить новые непроверяемые 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 непроверяемые исключения, выбрасываемые каждым методом. Однако, если второй класс был изменён так, что теперь он выбрасывает дополнительные непроверяемые исключения, первый класс (не претерпевший изменений) тоже будет передавать наверх эти новые непроверяемые исключения, хотя он их и не объявлял.

Если одно и то же исключение по одной и то же причине выбрасывается 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`. Вполне нормально добавить в комментарий к классу предложение «все методы этого класса выбрасывают исключение `NullPointerException`, если с каким-либо параметром была передана нулевая ссылка на объект», или другие слова с тем же смыслом.

Подведём итоги. Необходимо документировать все исключения, которые могут быть выведены каждым написанным вами методом. Это относится как к проверяемым, так и к

непроверяемым исключениям, как к абстрактным, так и к конкретным методам. Каждое проверяемое исключение должно перечисляться отдельно в объявлении `throws`, однако же не надо использовать это выражение для непроверяемых исключений. Если вы не будете документировать исключения, которые выбрасывают ваши методы, то другим людям будет сложно или даже невозможно 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 имени класса исключения и *сообщения с подробностями* (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 ещё 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 трёх значений или все они вместе могут быть неправильными. Представленный индекс может оказаться на единицу меньше нижней границы или быть равен верхней границе (ошибка на единицу) либо может иметь несуразное значение, как слишком маленькое, так и слишком большое. Нижняя граница может быть больше верхней (серьёзная 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. Трассировка 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 приёмов, 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 с параметром типа `String` исключение `IndexOutOfBoundsException` могло бы иметь следующий конструктор:

```
/**
 * Создаёт экземпляр IndexOutOfBoundsException
 *
 * @param lowerBound - самое меньшее из разрешённых значений индекса
 * @param upperBound - самое большее из разрешённых 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 платформы Java используется не слишком интенсивно. С ее помощью программист, выбрасывающий исключение, может с лёгкостью зафиксировать обстоятельства сбоя. Вместо того чтобы заставлять каждого пользующегося классом генерировать своё 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 проверяемых исключений ещё важнее, чем для непроверяемых, поскольку информация об обстоятельствах сбоя может быть полезна для восстановления работоспособности программы. Программный доступ к деталям непроверяемого исключения редко интересует программистов (хотя это и не исключено). Однако, согласно общему принципу (статья 10), такие методы доступа имеет смысл создавать даже для непроверяемых исключений.

Статья 64. Добивайтесь атомарности методов по отношению к сбоям

После того, как объект выбрасывает исключение, обычно необходимо, чтобы он оставался во вполне определённом, пригодном для дальнейшей обработки состоянии, даже несмотря на то, что сбой произошёл непосредственно в процессе выполнения

операции. Особенно это касается проверяемых 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 сбоям* (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 неопределённом 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 путём проверки правильности параметров перед выполнением операции (статья 38). Благодаря этому любое исключение будет инициироваться до того, как начнётся 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 выбрасывать исключение при попытке получить элемент из пустого стека. Однако при этом он будет оставлять поле `size` в неопределённом (отрицательном) состоянии. А это приведёт к тому, что сбоем будет завершаться вызов любого метода в этом объекте. Кроме того, само исключение, выбрасываемое методом `pop`, не будет соответствовать текущему уровню абстракции (статья 61).

Другой приём, 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 приём 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 произойдёт в процессе поиска этого элемента в дереве, но до того, как в этом дереве что-либо будет изменено.

Третий, редко встречающийся приём 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 приём используется главным образом для структур, записываемых в базу данных.

Наконец, последний приём, позволяющий добиться атомарности метода, заключается в том, чтобы выполнять операцию на временной копии объекта и, как только операция будет завершена, замещать содержимое объекта содержимым его временной копии. Такой приём естественным образом 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.

Комментарий. Начиная с Java 8, в общем случае это уже неверно. Теперь метод `Collections.sort` является лишь тонкой обёрткой для метода `sort` интерфейса `List`, реализация которого по умолчанию работает вышеописанным образом, но для некоторых реализаций списков (в частности, `ArrayList`) этот метод переопределён с реализацией сортировки на месте (in place), то есть сортируется непосредственно внутренний массив, поверх которого реализован `ArrayList`, без лишнего копирования в новый массив и обратно.

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 неопределённом состоянии. А потому после перехвата исключения `ConcurrentModificationException` нельзя полагаться на то, что объект все ещё 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 чётко указано, в каком состоянии будет оставлен объект. К сожалению, множество имеющейся документации к API не отвечает этому идеалу.

Статья 65. Не игнорируйте исключения

Этот совет кажется очевидным, но он нарушается настолько часто, что заслуживает повторения. Когда разработчики API объявляют, что некий метод выбрасывает исключение, этим они пытаются что-то вам сказать. Не игнорируйте это! Игнорировать исключения легко: необходимо всего лишь окружить вызов метода инструкцией `try` с пустым блоком `catch`:

```
// Пустой блок catch игнорирует исключение - крайне
// подозрительный код!
try {
    ...
} catch (SomeException e) {
}
```

Пустой блок `catch` лишает смысла механизм исключений, ведь смысл как раз и состоит в том, чтобы заставить вас обрабатывать 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 проверяемым, так и к непроверяемым исключениям. Вне зависимости от того, представляет ли исключение предсказуемое условие или программную ошибку, если оно игнорируется и используется пустой блок `catch`, то в результате программа, столкнувшись с ошибкой, будет работать дальше, никак на неё не реагируя. Затем в любой произвольный момент времени программа может завершиться с ошибкой, и программный код, где это произойдёт, не будет иметь никакого отношения к действительному источнику проблемы. Должным образом обработав исключение, вы можете избежать отказа. Даже простая передача непроверяемого исключения вовне вызовет, по крайней мере, быстрый останов программы, при котором будет сохранена информация, полезная при устранении сбоя.

Комментарий. Совет в последнем абзаце является частным случаем философии обработки ошибок, называемой *fail-fast*. Суть её в том, чтобы обнаружить ошибку как можно раньше, и если восстановление после ошибки невозможно, как можно раньше завершить работу аварийно вместо того, чтобы потом сломаться позже, в месте, никаким очевидным образом не связанном с настоящим источником ошибки.

Глава 10. Многопоточность

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 его использование 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 многопоточные программы.

Комментарий. Для краткости и читаемости примеры из этой главы, которые в оригинале использовали анонимные классы, переписаны с использованием лямбда-выражений. Даже если вы до сих пор не перешли на Java 8, вам всё равно стоит привыкнуть к новому синтаксису, потому что вы всё чаще будете видеть его в документации.

Статья 66. Синхронизируйте доступ потоков к совместно используемым изменяемым данным

Использование ключевого слова `synchronized` даёт 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 создаётся 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 даёт потоку возможность наблюдать объект в промежуточном состоянии, но также даёт гарантию, что объект

будет переходить из одного согласованного состояния в другое в результате чёткого 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 даёт гарантию, что чтение и запись отдельной переменной, если это не переменная типа `long` или `double`, являются *атомарными* (atomic) операциями [JLS, 17.4.7]. Иными словами, гарантируется, 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 надёжного взаимодействия между ними. Это является следствием сугубо технического аспекта языка программирования Java, который называется *моделью памяти* (memory model) [JLS, 17, Goetz06 16].

Отсутствие синхронизации для доступа к совместно используемой переменной может иметь серьёзные последствия, даже если переменная имеет свойство атомарности как при чтении, так и при записи. Рассмотрим задачу остановки одного потока из другого. В библиотеке 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 приём, 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(() -> {
            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 приведёт 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 (!done)
    i++;
```

в такой:

```
if (!done)
    while (true)
        i++;
```

Эта оптимизация называется *поднятием* (hoisting), и это именно то, что делает серверная версия виртуальной машины HotSpot. Результат — *ошибка живучести* (liveness failure): программе не удаётся 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(() -> {
            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 **синхронизация не имеет эффекта, если не синхронизированы операции как записи, так и чтения**.

Действия синхронизированных методов класса `StopThread` были бы атомарными даже без синхронизации. Другими словами, синхронизация этих методов используется *исключительно* для коммуникации, а не для взаимного исключения. Хотя затраты на синхронизацию каждой итерации цикла невелики, есть корректная альтернатива,

которая не будет настолько нагружена текстом и производительность которой будет лучше. Блокировку во второй версии `StopThread` можно опустить, объявив переменную `stopRequested` как изменчивую (`volatile`). Хотя модификатор `volatile` не выполняет взаимного исключения, он гарантирует, что любой поток, который прочитает поле, увидит самое последнее записанное в него значение:

```
// 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(() -> {
            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 2^{32} 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 инкремента (`++`) 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 прочитает значение после того, как первый прочитает старое значение, но до того, как он запишет новое, второй поток увидит то же старое значение и инкрементирует его, и один из инкрементов будет потерян. Это *ошибка безопасности* (safety failure): программа считает неверные результаты.

Одним из способов исправления метода `generateSerialNumber` сводится к простому добавлению в его объявление слова `synchronized`. 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 следует удалить модификатор `volatile` из объявления поля `nextSerialNumber`. Чтобы сделать этот метод «железобетонным», возможно, имеет смысл заменить `int` на `long` или выбрасывать какое-либо исключение, если значение `nextSerialNumber` будет близко к переполнению.

Ещё 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 делайте общими неизменяемые данные (статья 15), либо вообще не разделяйте никаких данных между потоками. Другими словами, **не допускайте видимости изменяемых данных несколькими потоками**. Если вы примените данную политику, необходимо её документировать, чтобы при развитии программы это

её свойство 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 публикации ссылки на объект для других потоков. Другие потоки тогда смогут читать объект без дальнейшей синхронизации до тех пор, пока он снова не изменится. Такие объекты называются *де факто неизменяемыми* (effectively immutable) [Goetz06, 3.5.4]. Перенос такой ссылки на объект от одного потока на другой называется *безопасной публикацией* (safe publishing) [Goetz06, 3.5.3]. Существует много способов безопасной публикации ссылки на объект: вы можете хранить ее в статическом поле как часть инициализации класса; вы также можете хранить ее в поле `volatile`, в поле `final` или в поле, доступ к которому осуществляется с помощью обычной блокировки; или же вы можете поместить его в параллельную коллекцию (concurrent collection, статья 69).

Подведём итоги. **Когда несколько потоков совместно работают с изменяемыми данными, каждый поток, который читает или записывает эти данные, должен пользоваться синхронизацией.** 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, а не взаимное исключение потоков, модификатор `volatile` является подходящей формой синхронизации, но использовать его корректно может быть довольно сложно.

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 ошибок живучести и безопасности никогда не передавайте управление клиенту внутри синхронизированного метода или блока. Иными словами, из области синхронизации не следует вызывать открытые или защищённые 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* (alien). 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 наблюдаемую (observable) обёртку вокруг множества. Он позволяет клиентам подписываться на уведомления каждый раз, когда к набору добавляется элемент. Это шаблон Observer [Gamma95, стр. 293]. Для краткости класс не предоставляет уведомлений, когда элементы удаляются из множества, но настроить такие уведомления не составит труда. Этот класс реализуется поверх переиспользуемого класса `ForwardingSet` из статьи 16:

```
// 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, вызывая метод `addObserver`, и отписываются с помощью метода `removeObserver`. В обоих случаях методу передаётся экземпляр интерфейса обратного вызова (callback):

```
public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
```

```
}
```

На первый взгляд класс `ObservableSet` работает нормально. 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((s, 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 передаёт наблюдателя, печатающего только что добавленное ко множеству значение `Integer` и отписывающего себя от уведомлений, 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 отпишется и программа тихо завершит работу. Но на самом деле она выводит значения от 0 до 23, после чего выбрасывает исключение `ConcurrentModificationException`. Проблема в том, что метод `notifyElementAdded` находится в процессе итерации по списку `observers`, когда запускается метод для наблюдателя `added`. Метод `added` вызывает метод `removeObserver` класса `ObservableSet`, который в свою очередь вызывает `observers.remove`. Теперь у нас проблема. Мы пытаемся удалить из списка элемент во время итерации по этому списку, что недопустимо. Итерация в методе `notifyElementAdded` находится

в синхронизированном блоке, чтобы воспрепятствовать параллельному изменению списка, но он не запрещает самому итерирующему потоку сделать обратный вызов к объекту `ObservableSet` и изменить его список `observers`.

Попробуем теперь нечто странное: пусть наблюдатель попробует отписаться, но вместо непосредственного вызова метода `removeObserver` мы для этого воспользуемся услугами другого потока. Наблюдатель использует объект `ExecutorService` (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 SetObserver<Integer> observer = this;
            try {
                executor.submit(() -> {
                    s.removeObserver(observer);
                }).get();
            } catch (ExecutionException ex) {
                throw new AssertionError(ex.getCause());
            } catch (InterruptedException ex) {
                throw new AssertionError(ex.getCause());
            } finally {
                executor.shutdown();
            }
        }
    }
});
```

На этот раз ошибка исключения не выводится — мы получаем взаимную блокировку. Фоновый поток вызывает метод `s.removeObserver`, который пытается заблокировать список `observers`, но он не может захватить блокировку, потому что её уже захватил главный поток. Тем временем главный поток ждёт, пока фоновый поток завершит удаление наблюдателя, что объясняет появление взаимной блокировки.

Этот пример неестественен, так как нет причин для наблюдателя использовать фоновый поток, но проблема реальна. Запуск чужих методов из синхронизированных областей вызвал много блокировок в реальных системе, таких, как библиотеки GUI.

В обоих предыдущих примерах (исключение и блокировка) нам повезло. Ресурс, защищённый синхронизированной областью (список `observers`), находился в согласованном состоянии на момент вызова чужого метода (`added`). Предположим, вы запустили чужой метод из синхронизированного потока в тот момент, когда инвариант, защищённый синхронизированной областью, был временно недействителен. Поскольку блокировки в языке Java являются *реентерабельными* (reentrant), такие вызовы не приведут к блокировке. Как и в первом примере, который завершился исключением, вызываемый поток уже захватил блокировку, так что поток сможет повторно захватить блокировку, даже в этот момент он находится в процессе выполнения другой, концептуально не связанной операции над данными, защищёнными блокировкой. Последствия такого сбоя могут быть катастрофическими. По сути, блокировка перестаёт выполнять свою функцию. Реентерабельные блокировки упрощают создание многопоточных объектно-ориентированных программ, но также они могут превратить ошибки живучести в ошибки безопасности.

К счастью, обычно не слишком сложно решить такую проблему путём перемещения вызова чужого метода за пределы синхронизированных блоков. Для метода `notifyElementAdded` это подразумевает копирование 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 перемещён за пределы синхронизированного блока -
// открытые вызовы
private void notifyElementAdded(E element) {
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<SetObserver<E>>(observers);
    }
    for (SetObserver<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 содержат параллельную коллекцию (concurrent collection, статья 69), известную как `CopyOnWriteArrayList` и подходящую для нашей цели. Это вариант `ArrayList`, в котором все операции записи реализуются копированием всего нижележащего 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 подойдёт идеально, так как он редко меняется, но итерация по нему выполняется часто.

Методы `add` и `addAll` набора `ObservableSet` нет необходимости менять, если реализация списка заменится на `CopyOnWriteArrayList`. Вот как выглядит оставшаяся часть класса. Обратите внимание, что здесь нет явной синхронизации:

```
// Безопасная реализация ObservableSet 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, вызванный за пределами синхронизированной области, известен как *открытый вызов* (open call) [Lea00 2.4.1.3]. Помимо того, что открытые вызовы предотвращают сбои, они могут существенно улучшить параллелизацию. Чужой метод может выполняться в течение произвольно длительного периода. Если же чужой метод запущен из синхронизируемой области, другие потоки будут впустую простаивать, ожидая доступа к защищённым ресурсам.

Как правило, вам нужно выполнять как можно меньше действий внутри синхронизируемой области. Захватите блокировку, просмотрите общие данные, преобразуйте их как надо и затем снимите блокировку. Если вам требуется выполнение длительных действий, найдите способ переместить эти действия за пределы синхронизируемой области, не нарушая инструкций из статьи 66.

Первая часть этой статьи была посвящена корректности. Теперь вкратце рассмотрим вопрос производительности. Хотя затраты на синхронизацию уменьшились по сравнению с ранними версиями языка Java, тем не менее важно не переусердствовать с синхронизацией. В многоядерной среде реальные затраты на излишнюю синхронизацию — это не время, затрачиваемое процессором на получение блокировок, а утерянные возможности параллелизации и задержки, вызванные необходимостью убедиться в том, что все ядра имеют согласованное представление о содержимом памяти. Другие скрытые затраты от излишней синхронизации заключаются в том, что она может ограничить возможность виртуальной машины оптимизировать код при выполнении.

Вам следует сделать неизменяемый класс потокобезопасным (статья 70), если предполагается его параллельное использование и вы можете достичь большей параллелизации, выполняя синхронизацию изнутри, чем путём блокирования всего объекта извне. В противном случае не синхронизируйте объект изнутри. Пусть клиенты синхронизируются извне, где это приемлемо. Вначале, при появлении платформы Java, многие классы нарушали эти рекомендации. Например, экземпляры `StringBuffer` почти всегда используются одним потоком, но все-таки выполняют синхронизацию изнутри. Именно поэтому в версии 1.5 он был заменён на `StringBuilder`, который представляет собой несинхронизованную версию `StringBuffer`. Если вы сомневаетесь, то не синхронизируйте свой класс, а укажите в документации, что он не потокобезопасен (статья 70).

Если вы синхронизируете класс изнутри, вы можете пользоваться различными приёмами для достижения высокой параллелизации, такими, как разделение блокировки, распределение блокировки и контроль параллельности без блокировки. Эти приёмы выходят за рамки этой книги, но описываются другими [Goetz06, Lee00].

Если метод изменяет статическое поле, он *обязан* иметь внутреннюю синхронизацию, даже если обычно применяется только с одним потоком. В этом случае 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).

Подведём итоги. Во избежание взаимной блокировки потоков и разрушения данных никогда не вызывайте чужие методы из синхронизированной области. Говоря более общо, постарайтесь ограничить объем работы, выполняемой вами в синхронизированных областях. Проектируя изменяемый класс, подумайте о том, не должен ли он иметь свою собственную синхронизацию. В современном многопроцессорном мире как никогда важно не злоупотреблять синхронизацией. Синхронизируйте класс изнутри, только если для этого есть хорошая причина, и чётко документируйте своё решение (статья 70).

Статья 68. Предпочитайте Executor Framework непосредственному использованию потоков

В первой редакции книги содержался код для простой *рабочей очереди* (work queue) [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 требовалась целая страница очень тонкого кода, к которому нужно подходить очень основательно, чтобы избежать ошибок безопасности и живучести. К счастью, более нет причин писать такой код.

В версии 1.5 платформы появился пакет `java.util.concurrent`. Этот пакет содержит инфраструктуру Executor Framework, предоставляющую гибкий, основанный на интерфейсах механизм выполнения задач. Создание рабочей очереди, реализованной лучше, чем та, которая описана в первой редакции книги, требует всего лишь одной строки кода:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Вот как можно запустить задачу на выполнение:

```
executor.execute(runnable);
```

А вот как корректно завершить службу исполнения (если вы этого не сделаете, то, скорее всего, ваша виртуальная машина не закроется):

```
executor.shutdown();
```

С помощью службы исполнения вы можете сделать куда больше. Например, вы можете подождать, пока завершится определённая 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 исполнения корректно завершится (используя метод `awaitTermination`). Можно вывести результаты задач один за другим по мере их завершения (с помощью `ExecutorCompletionService`) и т.д.

Если вы хотите, чтобы запросы из очереди обрабатывались более чем одним потоком, просто вызовите другой статический фабричный метод, который создаёт другой вид службы исполнения, называемый *пулом потоков* (thread pool). Вы можете создавать пул потоков с фиксированным или переменным количеством потоков. Класс `java.util.concurrent.Executors` содержит статические фабричные методы, предоставляющие большинство видов служб исполнения, которые вам когда-либо понадобятся. Если вы захотите что-то необычное, вы можете использовать непосредственно класс `ThreadPoolExecutor`. Этот класс позволяет вам контролировать почти каждый аспект работы пула потоков.

Выбор службы исполнения для определённого приложения может быть довольно запутанным. Если вы пишете небольшую программу или легко нагруженный сервер, использование метода `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 создаётся 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 даёт 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. Ключевой абстракцией многопоточности больше не является класс `Thread`, который служил ранее в качестве и рабочей единицы, и механизма её выполнения. Теперь рабочая единица и механизм выполнения разделены. Теперь ключевой абстракцией является рабочая единица, которая называется *задачей* (task). 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 исполнения* (executor service). Если вы будете мыслить в терминах задач и позволяете службам исполнения выполнять их за вас, вы получите огромную выгоду в гибкости в плане выбора подходящей политики выполнения. По сути, Executor Framework делает для выполнения задач то, что Collections Framework делает для хранения множественных объектов.

У Executor Framework есть замена для класса `java.util.Timer`, которой является `ScheduledThreadPoolExecutor`. Хотя использовать `Timer` легче, класс `ScheduledThreadPoolExecutor` является более гибким. `Timer` использует только один поток для выполнения задач. Если единственный поток таймера выбросит необработанное исключение, таймер прекратит работу. `ScheduledThreadPoolExecutor` поддерживает использование нескольких потоков для исполнения задач и корректно восстанавливается после необработанного исключения.

Полное описание Executor Framework выходит за рамки данной книги, но заинтересованный читатель может найти его в книге «Java Concurrency in Practice» [Goetz06].

Статья 69. Предпочитайте утилиты параллельности методам wait и notify

В первой редакции этой книги была статья, посвящённая корректному использованию `wait` и `notify` (Bloch01, статья 50). Содержащиеся в ней советы все ещё 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 высокого уровня, которые делают те вещи, которые вы раньше писали вручную поверх `wait` и `notify`. **С учётом трудности корректного использования `wait` и `notify` вам следует использовать вместо них высокоуровневые утилиты параллельности.**

Эти утилиты, содержащиеся в `java.util.concurrent`, делятся на три категории: Executor Framework, который был кратко описан в статье 68, а также 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 даст эффекта**, а только затормозит выполнение программы.

Это значит, что клиенты не могут атомарно использовать композицию вызовов методов на параллельных коллекциях. Некоторые из этих интерфейсов коллекций расширены *операциями изменения, зависящими 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 потокобезопасных канонизирующих словарей. Например, этот метод имитирует поведение `String.intern`:

```
// Параллельный канонизирующий словарь поверх ConcurrentMap -
// не оптимален
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<String, String>();

public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

В действительности можно сделать лучше. `ConcurrentHashMap` оптимизирован для операций извлечения, таких, как `get`. Следовательно, имеет смысл запускать `get` вначале и вызывать `putIfAbsent`, если только `get` покажет, что это необходимо:

```
// Параллельный канонизирующий словарь поверх ConcurrentMap -
// 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;
}
```

Комментарий. В Java 8 эти классы были расширены большим количеством атомарных методов, принимающих экземпляры функциональных интерфейсов, например, `ConcurrentMap.merge(key, value, remappingFunction)`. Приведённый выше пример теперь можно записать ещё и так:

```
public static String intern(String s) {
    return map.merge(s, s, (oldValue, newValue) -> oldValue);
}
```

Конечно, следует помнить о советах из статьи 67, касающихся чужих методов, и не вызывать напрямую операции, изменяющие объект `ConcurrentMap`, в передаваемых в него блоках кода.

Кроме отличной параллельности, `ConcurrentHashMap` ещё 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 слабые ссылки, чтобы избежать утечек памяти с течением времени). Если у вас нет серьёзной причины сделать по-другому, **используйте `ConcurrentHashMap` вместо `Collections.synchronizedMap` или `Hashtable`**. Простая замена старых синхронизированных словарей параллельными словарями может серьёзно увеличить производительность параллельных приложений. Говоря более общо, предпочитайте использование параллельных коллекций синхронизируемым извне коллекциям.

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, блокируясь, если очередь пуста. Это позволяет использовать блокирующие очереди для *рабочих очередей* (work queues, также известные как *очереди типа производитель-потребитель* — producer-consumer queues), в которые один или несколько *производящих потоков* (producer threads) ставят рабочие задания и из которых один или более *потребляющих потоков* (consumer threads) 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 свою деятельность. Наиболее часто используемые синхронизаторы — это `CountDownLatch` и `Semaphore`. Реже используются `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` принимает `int`, который задаёт количество вызовов метода `countDown`, которые должны произойти, прежде чем всем ожидающим потокам будет разрешено продолжить.

Удивительно легко создавать полезные вещи поверх этого простого примитива. Например, предположим, что вы хотите создать простой фреймворк для подсчёта времени параллельного выполнения действия. Эта структура состоит из одного метода, который берет объект `Executor` для выполнения действия, уровень параллельности, представляющий собой количество действий, которые должны параллельно выполняться, и объект `Runnable`, представляющий собой само 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` и `notify` была бы, мягко говоря, запутанной, но поверх `CountDownLatch` она будет удивительно проста:

```
// Простой фреймворк для подсчёта 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(() -> {
            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(); // 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 объекта `CountDownLatch`. Первый, `ready`, используется рабочими потоками, чтобы сказать потоку таймера, когда он готов. Рабочие потоки затем ждут второй защёлки, которой является `start`. Последний рабочий поток запускает `ready.countDown`, позволяя всем рабочим потокам продолжать. Поток таймера ждёт третьей защёлки, `done`, пока последние рабочие потоки не закончат выполнение действия и не вызовут `done.countDown`. Как только это

произойдёт, поток таймера пробуждается и фиксирует время окончания.

Стоит упомянуть ещё несколько деталей. Объект `Executor`, который передаётся методу `time`, должен позволить создание по крайней мере такого количества потоков, которое определено уровнем параллельности, иначе тест никогда не завершится. Такая ситуация называется *thread starvation deadlock* [Goetz06, 8.1.1]. Если рабочий поток наталкивается на `InterruptionException`, он устанавливает флаг прерывания, используя идиому `Thread.currentThread().interrupt()`, и выходит из метода `run`. Это позволяет объекту `Executor` поступать с прерыванием так, как он считает нужным, что и является правильным решением. Наконец, обратите внимание, что вместо `System.currentTimeMillis` для записи времени выполнения используется `System.nanoTime`. **Для измерения интервалов времени всегда лучше использовать `System.nanoTime`, чем `System.currentTimeMillis`.** Метод `System.nanoTime` и более точен, и обладает большей разрешающей способностью; кроме того, на него не влияют настройки системного времени.

Эта статья лишь поверхностно освещает утилиты параллельности. Например, три объекта `CountDownLatch` из предыдущего примера могут быть заменены одним объектом `CyclicBarrier`. Код получится даже короче, но его будет труднее понять. За подробностями обращайтесь к книге «Java Concurrency in Practice» [Goetz06].

Хотя вам всегда следует предпочитать утилиты параллельности методам `wait` и `notify`, возможно, вам придётся столкнуться с унаследованным кодом, использующим эти методы напрямую. Метод `wait` используется, чтобы заставить поток ждать наступления определённого условия. Он должен запускаться в синхронизированной области, которая блокируется на объекте, для которого вызывается метод `wait`. Вот стандартная идиома использования метода `wait`:

```
// Стандартная идиома использования метода 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 ожидания для вызова метода `wait`; никогда не

вызывайте его вне цикла. Цикл служит для проверки условий до и после ожидания.

Проверка условия перед ожиданием и пропуск ожидания, если условие уже выполняется, необходимы для гарантии живучести. Если условие уже выполняется и метод `notify` (или `notifyAll`) уже был вызван перед тем, как поток перешёл к ожиданию, нет гарантии, что поток вообще когда-либо прекратит ожидание и проснётся.

Проверка условия после ожидания и возвращение в состояние ожидания, если условие не будет выполняться, необходимы для обеспечения безопасности. Если поток продолжает работу, когда условие не выполняется, он может уничтожить инвариант, охраняемый блокировкой. Есть несколько причин, по которым поток может проснуться, даже если условие не соблюдается:

- Другой поток получил блокировку и изменил защищённое состояние в период между моментом вызова `notify` и моментом пробуждения ожидающего потока.
- Другой поток мог запустить notify случайно или злоумышленно, 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 чересчур 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. Такая ситуация называется *ложным пробуждением* (spurious wakeup) [Posix, 11.4.3.6.1: JavaSE6].

Рассмотрим связанный вопрос о том, следует ли для пробуждения ожидающих потоков использовать `notify` или `notifyAll`. (Вспомните, что `notify` пробуждает один ожидающий поток, если таковой существует, а `notifyAll` будит все ожидающие потоки.) Часто советуют использовать `notifyAll`. Это разумный консервативный совет. Он всегда приведёт к корректному поведению, потому что он гарантирует, что вы разбудите потоки, которые должны быть разбужены. Вы можете также разбудить некоторые другие потоки, но это не повлияет на корректность выполнения программы. Эти потоки проверят условия, на 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 находиться в ожидании, ждут одного и того же условия и только один поток в одно время может извлечь выгоду от того, что условие станет истинным.

Даже если эти условия верны, могут быть другие причины для использования `notifyAll` вместо `notify`. Так же, как помещение вызова `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 состоянии бесконечного ожидания.

Подведём итоги. Использование методов `wait` и `notify` непосредственно подобно программированию на ассемблере по сравнению с высокоуровневым программированием, предоставленным пакетом `java.util.concurrent`. **Необходимость использовать `wait` и `notify` в новом коде возникает редко, если она вообще когда-нибудь возникает.** Если вы поддерживаете старый код, использующий `wait` и `notify`, убедитесь, что `wait` всегда запускается из цикла `while` с использованием стандартной идиомы. В общем случае предпочтительнее использовать `notifyAll`, нежели `notify`. Если используется `notify`, необходимо серьёзно позаботиться о возможных ошибках живучести.

Статья 70. Документируйте уровень потокобезопасности

То, как класс работает, когда его экземпляры и статические методы одновременно используются в нескольких потоках, является важной частью контракта, устанавливаемого классом для своих клиентов. Если вы не отразите эту сторону поведения класса в документации, использующие его программисты будут вынуждены делать предположения, и если эти предположения окажутся неверными, полученная программа может иметь либо недостаточную (статья 66) либо избыточную (статья 67) синхронизацию. В любом случае это способно привести к серьёзным ошибкам.

Иногда говорится, что пользователи могут сами определить, потокобезопасен ли метод, если проверят, присутствует ли модификатор `synchronized` в документации, генерируемой утилитой Javadoc. Это неверно по нескольким причинам. При нормальном

выполнении утилита Javadoc не указывает в создаваемом документе модификатор `synchronized`, и неспроста. **Наличие в объявлении метода модификатора `synchronized` — это деталь реализации, а не часть внешнего API.** Присутствие модификатора не является надёжной гарантией того, что метод потокобезопасен. От версии к версии ситуация может меняться.

Более того, само утверждение о том, что наличия ключевого слова `synchronized` достаточно для того, чтобы говорить о потокобезопасности, содержит в себе распространённое заблуждение о том, что потокобезопасность — это характеристика из серии «всё или ничего». На самом деле есть несколько уровней потокобезопасности. **Чтобы класс можно было безопасно использовать в многопоточной среде, в документации к нему должно быть чётко указано, какой уровень потокобезопасности он поддерживает.**

В следующем списке приводятся уровни потокобезопасности, которых может придерживаться класс. Этот список не претендует на полноту, однако в нем представлены самые распространённые 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).
- **Безусловно потокобезопасный** (unconditionally thread-safe). Экземпляры такого класса могут изменяться, однако все методы имеют довольно надёжную внутреннюю синхронизацию, чтобы эти экземпляры могли параллельно использовать несколько потоков безо всякой внешней синхронизации. Примеры: `Random` и `ConcurrentHashMap`.
- **Условно потокобезопасный** (conditionally thread-safe). То же, что и с поддержкой многопоточности, за исключением того, что некоторые методы требуют внешней синхронизации для безопасного использования в многопоточной среде. Примеры: `Hashtable` и `Vector`, чьи итераторы требуют внешней синхронизации.
- **Не потокобезопасный** (not thread-safe). Экземпляры такого класса изменяемы, и чтобы их можно безопасно использовать в нескольких потоках одновременно, каждый вызов метода (а в некоторых случаях каждую последовательность вызовов) необходимо окружить внешней синхронизацией. Среди примеров можно назвать 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 такая несовместимость с модификацией статических данных без синхронизации. Никто не пишет несовместимые с многопоточностью классы намеренно; такие классы появляются из-за того, что их разработчик не задумывался о многопоточном использовании класса. 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) примерно соответствуют аннотациям потокобезопасности (thread safety annotations), приведённым в книге «Java Concurrency in Practice», которыми являются `Immutable`, `ThreadSafe` и `NotThreadSafe` [Goetz06, Appendix A]. Аннотация `ThreadSafe` покрывает как условно, так и безусловно многопоточные классы.

Комментарий. Аннотации `Immutable`, `ThreadSafe` и `NotThreadSafe` входят в состав пакета `javax.annotation.concurrent` в библиотеке аннотаций, разработанной в рамках проекта JSR-305. Кроме того, в этом пакете есть аннотация `GuardedBy`, тоже описанная в вышеназванной книге. Она позволяет документировать имя объекта блокировки, по которой класс осуществляет внутреннюю синхронизацию.

Документированию условно потокобезопасного класса 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 по самому экземпляру класса, 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 возвращаемом словаре при итерации по любому из его коллекций-представлений (collection views):

```
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 недетерминированное поведение.

Описание потокобезопасности класса обычно содержится в комментариях к документации на него, но методы с особыми настройками потокобезопасности должны описывать эти свойства в своих собственных комментариях к документации. Нет необходимости документировать неизменяемость перечислимых типов. Еак продемонстрировано на примере выше с `Collections.synchronizedMap`, статические фабричные методы должны документировать потокобезопасность возвращаемого объекта, если только она не очевидна из типа возвращаемого значения.

Если класс синхронизируется по объекту блокировки, доступному извне, он даёт клиентам возможность выполнять последовательный запуск методов атомарно, но у этой гибкости есть цена. Такое решение несовместимо с высокопроизводительным внутренним контроле параллельности, который используется в таких классах, 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 *закрытый объект блокировки* (private lock object) вместо использования синхронизированных методов (которые подразумевают открытый доступ к объекту блокировки):

```
// Идиома закрытого объекта блокировки - 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. Фактически мы применили совет из статьи 13, инкапсулировав объект блокировки в рамках объекта, который он синхронизирует.

Обратите внимание, что поле `lock` объявляется как `final`. Это не позволяет вам по неосторожности переприсвоить его, что может привести к катастрофическим последствиям, в частности, к несинхронизированному доступу к содержащемуся объекту (статья 66). Мы применили совет из статьи 15, сводя к минимуму изменяемость поля `lock`.

Ещё раз повторим: идиома закрытого объекта блокировки может использоваться только в классах с безусловной поддержкой многопоточности. Классы с условной поддержкой многопоточности не могут использовать эту идиому, потому что им надо документировать, какую именно блокировку должны получить их клиенты при выполнении определённых последовательностей вызовов методов.

Использование внутренних объектов для блокировки особенно подходит классам, которые предназначены для наследования (статья 17). Если бы такой класс использовал для блокировки собственные экземпляры, то подкласс мог бы легко и непреднамеренно вмешаться в операции суперкласса, 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`, как показано в «Java Puzzlers» [Bloch05, задача 77].

Подведём итоги. Для каждого класса необходимо чётко документировать возможность работы с несколькими потоками. Единственная возможность сделать это — представить аккуратно составленный текст описания или аннотацию потокобезопасности. К документированию того, как метод работает в условиях многопоточности, наличие модификатора `synchronized` отношения не имеет. Для условно потокобезопасных классов в документации важно указывать, какие последовательности вызовов методов требуют внешней синхронизации, и какой объект блокировки при этом нужно захватывать. Если вы пишете безусловно потокобезопасный класс, рассмотрите

возможность использования закрытой блокировки вместо синхронизированных методов. Это защищает вас от вмешательства клиентов и подклассов в синхронизацию и даёт вам гибкость в применении более сложного подхода к контролю параллельности в последующих версиях.

Статья 71. Соблюдайте осторожность при использовании ленивой инициализации

Ленивая инициализация (lazy initialization) — это задержка инициализации поля до тех пор, пока его значение не потребуется. Если значение не потребуется вовсе, поле никогда не инициализируется. Этот приём применяется как для статических полей, так и для полей экземпляров. Хотя в основном ленивая инициализация является оптимизацией, она также может разрушить вредоносные циклические зависимости при инициализации класса или экземпляра [Bloch05, задача 51].

Как и в случае с большинством оптимизаций, лучшим советом для ленивой инициализации будет «не используйте ее до тех пор, пока она вам действительно не понадобится» (статья 55). Ленивая инициализация — это палка о двух концах. Она уменьшает затраты на инициализацию класса или создание экземпляра за счёт 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, ленивая инициализация может (как и любая «оптимизация») повредить производительности.

Как уже говорилось, ленивая инициализация имеет свою область применения. Если к полю получает доступ только часть экземпляра класса и инициализация поля требует затрат, тогда, возможно, стоит использовать ленивую инициализацию. Единственный способ узнать наверняка — измерить производительность класса с ленивой инициализацией и без неё.

При наличии нескольких потоков ленивая инициализация может быть довольно сложна. Если два или более потока используют общее инициализируемое поле, то важно, чтобы применялась какая-либо форма синхронизации, иначе могут возникнуть серьёзные ошибки (статья 66). Все приёмы инициализации, обсуждаемые в этой статье,

потокобезопасны.

В большинстве случаев нормальная инициализация предпочтительнее ленивой. Вот типичный пример объявления нормально инициализируемого поля экземпляра. Обратите внимание на использование модификатора `final` (статья 15):

```
// Нормальная инициализация экземпляра поля
private final FieldType field = computeFieldValue();
```

Если вы используете ленивую инициализацию, чтобы разорвать цикл в зависимостях инициализации, используйте синхронизированный метод доступа, так как он является наиболее простой и понятной альтернативой:

```
// Ленивая 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 ленивая инициализация с синхронизированным методом доступа) остаются без изменений, когда используются для статического поля, кроме того, что вы добавляете модификатор `static` к объявлениям поля и метода доступа.

Если ленивая инициализация статического поля нужна вам для повышения производительности, используйте идиому *класса-держателя ленивой инициализации* (lazy initialization holder class idiom). Эта идиома (также известная как идиома класса-держателя инициализации по запросу, initialize-on-demand holder class idiom) использует гарантию того, что класс не будет инициализирован до тех пор, пока не будет использован [JLS 12.4.1]. Вот как она выглядит.

```
// Идиома класса-держателя ленивой инициализации статического поля
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
```

```
}
static FieldType getField() { return FieldHolder.field; }
```

Когда метод `getField` вызывается впервые, 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 ленивая 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 для повышения производительности вам требуется ленивая инициализация поля экземпляра, используйте *идиому двойной проверки* (double-check idiom). Эта идиома снимает затраты 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 блокировки не происходит, если поле уже инициализировано, то важно, чтобы поле было объявлено как `volatile` (статья 6 6). Вот как выглядит эта идиома:

```
// Идиома двойной проверки для ленивой 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 нужна для того, чтобы значение поля `field` читалось только один раз в типичном случае, где она уже инициализирована. Хотя в ней и нет острой необходимости, она может улучшить производительность и более элегантна с точки зрения стандартов низкоуровневого потокового программирования. На моей машине метод работал на 25% быстрее, чем очевидная версия без локальной переменной.

До релиза 1.5 идиома двойной проверки не работала надёжно, потому что семантика модификатора `volatile` не была достаточно сильна для обеспечения надёжности идиомы [Pugh01]. Модель памяти, представленная в версии 1.5, решила проблему [JLS, 17; Goetz06, 16]. Сегодня идиома двойной проверки является хорошим приёмом для ленивой инициализации поля. Хотя идиому двойной проверки можно применить и к статическому полю, нет причин так поступать: идиома класса-держателя ленивой инициализации больше подходит для этого.

Два варианта идиомы двойной проверки заслуживают внимания. В некоторых случаях лениво инициализируемое поле экземпляра допускает повторную инициализацию. Если вы окажетесь в такой ситуации, то можно использовать вариант идиомы двойной проверки, который обходится без второй проверки. Этот вариант известен под названием *«идиома однократной проверки»* (single-check idiom). 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();
        return result;
}
```

Все приёмы инициализации, обсуждённые в этой статье, относятся и к примитивным

полям, и к полям со ссылками на объекты. Когда идиома двойной или однократной проверки применяется для примитивного числового поля, значение поля сравнивается с 0 (значение по умолчанию для примитивных числовых переменных), а не с `null`.

Если для вас не имеет значения, будет ли *каждый* поток пересчитывать значение поля, и если тип поля является примитивным, но не `long` или `double`, то можно убрать модификатор `volatile` из объявления поля в идиоме однократной проверки. Этот вариант известен как *идиома однократной проверки с гонкой* (racy single-check idiom). Она ускоряет доступ к полю в некоторых архитектурах, платя за это вероятностью дополнительных инициализаций (максимум до одной на поток, получающий доступ к полю). Это определённо экзотический приём, не для каждодневного использования. Но, тем не менее, он используется экземплярами класса `String` для кэширования хэш-кодов.

Подведём итоги. Большинство полей имеет смысл инициализировать с помощью нормальной инициализации, а не ленивой. Если требуется ленивая инициализация поля с целью повышения производительности или для разрыва циклической зависимости, используйте подходящий приём ленивой инициализации. Для полей экземпляров это идиома двойной проверки; для статических полей — идиома инициализации содержащего класса. Для полей экземпляров, которые переносят повторную инициализацию, вы можете рассматривать вариант с идиомой однократной проверки.

Статья 72. Не полагайтесь на планировщик потоков

Когда в системе выполняется несколько потоков, соответствующий планировщик определяет, какие из них будут выполняться и в течение какого времени. Любая хорошо написанная операционная система при этом будет пытаться добиться какой-то справедливости, однако конкретная стратегия диспетчеризации 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, при которых в любой момент времени среднее число доступных

для выполнения (runnable) потоков ненамного больше общего числа процессоров. В этом случае планировщику потоков остаётся совсем небольшой выбор: он просто передаёт управление выполняемым потокам, пока те ещё могут выполняться. Как следствие, поведение программы не будет сильно меняться даже при выборе совершенно других алгоритмов диспетчеризации потоков. Обратите внимание, что число доступных для выполнения потоков — это не то же самое, что общее число потоков. Потоки в состоянии ожидания не являются доступными для выполнения.

Основной приём, 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 правильно задать размер пула потоков [Goetz06. 8.2] и делать так, чтобы задачи были небольшими и не зависели друг от друга. Правда, задачи не должны быть *слишком* маленькими, иначе затраты на диспетчеризацию 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 произойдёт. Помимо того, что программа при этом становится чувствительной к причудам планировщика, активное ожидание может значительно повысить нагрузку на процессор, соответственно уменьшая количество полезной работы, которую на той же машине могли бы выполнить остальные потоки. В качестве примера того, как не надо поступать, рассмотрим неправильную реализацию `CountDownLatch`:

```
// Ужасная реализация CountDownLatch - необоснованно
// применяет активное ожидание!
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 на защёлке 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 использовании `SlowCountDownLatch`, но производительность и переносимость, скорее всего, пострадают.

Столкнувшись с программой, которая едва работает из-за того, что некоторым потокам уделяется недостаточно процессорного времени по сравнению с другими потоками, **не поддавайтесь искушению «починить» программу добавлением вызова `Thread.yield`**. Вам, может, удастся заставить программу в какой-то мере работать, но она не будет переносимой. Один и тот же вызов `yield` может увеличить производительность на одной реализации виртуальной машины, на другой — ухудшить, а на третьей вообще не будет иметь никакого эффекта. **У метода `Thread.yield` нет никакой экспериментально проверяемой семантики.** Лучшим подходом будет изменить структуру приложения для снижения количества параллельно выполняемых потоков.

Аналогичный приём, к которому также относится похожий недостаток — это настройка приоритетов потоков. **Приоритеты потоков — одна из наименее переносимых возможностей платформы Java.** Вполне допустимо задать несколько приоритетов потоков, чтобы улучшить время отклика приложения, но это редко бывает необходимо, и такая техника непереносима. Совсем неразумно использовать приоритетность потоков

для решения серьёзных проблем с живучестью. Проблема, скорее всего, будет по-прежнему обнаруживать себя до тех пор, пока вы не найдёте и не устраните основную её причину.

В первой редакции этой книги говорилось, что единственное применение метода `Thread.yield`, которое будет иметь смысл использовать большинству программистов — это искусственное увеличение параллельности для тестирования. Идея была в том, чтобы выявить ошибки путём исследования большой части пространства состояния программы. Этот приём когда-то был довольно эффективен, но его корректная работа никогда не гарантировалась. Для реализации `Thread.yield` вполне допустимо ничего не делать, просто возвращая управление вызывающему методу. Некоторые современные виртуальные машины так и делают. Следовательно, вам необходимо использовать `Thread.sleep(1)` вместо `Thread.yield` для тестирования параллельности. Не используйте `Thread.sleep(0)`, поскольку этот вызов может немедленно вернуть управление.

Подведём итоги. Корректность вашей программы не должна зависеть от планировщика потоков, иначе программа не будет устойчивой и переносимой. Как следствие, не надо полагаться на метод `Thread.yield` и приоритеты потоков. Этот функционал даёт планировщику лишь рекомендации, которые он вправе проигнорировать. Их можно дозированно использовать для улучшения отзывчивости уже работающей программы, но ими никогда нельзя пользоваться для «исправления» программы, которая едва работает.

Статья 73. Не используйте класс ThreadGroup

Помимо потоков, блокировок и мониторов, система многопоточности Java предлагает ещё одну базовую абстракцию: *группа потоков* (thread group). Первоначально группы потоков рассматривались 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 стандартной работе, описывающей модель безопасности платформы Java [Gong03], они даже не упоминаются.

Но если группы потоков не несут никакой функциональной нагрузки в системе безопасности, то какие же функции они выполняют? Немногие. Они дают вам

возможность применять некоторые примитивы класса `Thread` сразу к нескольким потокам. Некоторые из этих примитивов уже устарели, остальные используются нечасто.

По иронии судьбы, API класса `ThreadGroup` слаб с точки зрения потокобезопасности. Чтобы для некоей группы получить перечень активных потоков, вы должны вызвать метод `enumerate`. В качестве параметра ему передаётся 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 создаёте массив и передаёте его методу `enumerate`. Если указанный массив окажется слишком мал, метод `enumerate` без каких-либо предупреждений игнорирует потоки, не поместившиеся в массив.

Таким же дефектом обладает API для получения списка подгрупп, входящих в группу потоков. И хотя указанные проблемы можно было бы решить, добавив в класс `ThreadGroup` новые методы, этого не было сделано из-за отсутствия в этом реальной потребности. **Группы потоков устарели.**

До появления версии 1.5 существовал небольшой функционал, который был доступен только чере API `ThreadGroup`: метод `ThreadGroup.uncaughtException` был единственным способом получения контроля в случаях, когда поток выбрасывал необработанное исключение. Этот функционал полезен, например, для перенаправления трассировки стека в систему логирования, выбранную приложением. Тем не менее, начиная с релиза 1.5 тот же самый функционал доступен с методом `setUncaughtExceptionHandler` класса `Thread`.

Подведём итоги. Группы потоков практически не имеют сколько-нибудь полезной функциональности, и большинство предоставляемых им возможностей имеют дефекты. Группы потоков следует рассматривать как неудачный эксперимент, а их существование можно игнорировать. Если вы проектируете класс, который работает с логическими группами потоков, вам нужно, вероятнее всего, использовать пулы потоков из Executor Framework (статья 68).

Глава 11. Сериализация

В этой главе описывается API *сериализации объектов* (object serialization), который предоставляет механизм для записи 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 удалёнными 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 многих ловушек, связанных с сериализацией объектов.

Статья 74. Соблюдайте осторожность при реализации интерфейса Serializable

Чтобы сделать экземпляры класса сериализуемыми, достаточно добавить в его объявление слова `implements Serializable`. Поскольку это так легко, широкое распространение получило неправильное представление, что сериализация требует от программиста совсем небольших усилий. На самом деле все гораздо сложнее. Хотя немедленные затраты на включение класса в механизм сериализации невелики, долговременные расходы на сопровождение этого решения могут стать значительными.

Значительная доля затрат на реализацию интерфейса `Serializable` связана с тем, что это решение уменьшает 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 придётся поддерживать соответствующую сериализованную форму точно так же, как вы обязаны поддерживать все остальные части интерфейса, предоставляемого клиентам. Если вы не приложите

усилий к построению *специализированной 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 `ObjectOutputStream.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 сопровождать класс в долгосрочной перспективе (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 оставить его искалеченным навсегда.

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, объявив поле `private static final long` с названием `serialVersionUID`, система генерирует его автоматически, используя для класса сложную схему расчётов. При этом на автоматически генерируемое значение оказывают влияние название класса, названия реализуемых им интерфейсов, а также все открытые и защищённые 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 объявлять этот идентификатор, совместимость с предыдущими версиями будет потеряна, и результатом будет исключение `InvalidClassException` во время выполнения.

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 *внеязыковой* (extralinguistic) механизм создания объектов. 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 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 своим внутренностям и разрушению своих инвариантов (статья 76).

Третье неудобство реализации интерфейса `Serializable` связано с тем, что выпуск новой версии класса сопряжён 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. Это тестирование нельзя автоматизировать, 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 тестировании уменьшается, но полностью не исчезает.

Решение о реализации интерфейса `Serializable` не должно приниматься сгоряча. У этого интерфейса есть реальные преимущества: его реализация играет важную роль, если класс должен участвовать в каком-либо фреймворке, которая использует сериализацию для передачи объекта или его сохранения на долговременный носитель (persistence). Более того, это упрощает применение класса как составной части другого класса, реализующего интерфейс `Serializable`. Однако с реализацией интерфейса `Serializable` связано и множество неудобств. Реализуя класс, соотносите неудобства

с преимуществами. Практическое правило таково: классам значений, такие как `Date` и `BigInteger`, и большинству классов коллекций следует реализовывать этот интерфейс. Классы, представляющие активные сущности, например, пул потоков, будут реализовывать `Serializable` крайне редко.

Классы, предназначенные для наследования (статья 17), обычно не должны реализовывать `Serializable`, а интерфейсы обычно не должны 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 некотором фреймворке, требующем, чтобы все используемые в нём классы реализовывали интерфейс `Serializable`, то лучше всего, чтобы этот класс (интерфейс) реализовывал (расширял) `Serializable`. Классы, предназначенные для наследования и реализующие интерфейс `Serializable`, включают в себя `Throwable`, `Component` и `HttpServlet`. `Throwable` реализует `Serializable` для того, чтобы исключения из механизма *удалённого вызова методов* (remote method invocation, RMI) могли передаваться от сервера к клиенту. `Component` реализует `Serializable`, чтобы можно было передать, сохранить и восстановить состояние 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 типа `boolean`, `null` для ссылочных типов), 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 решения *не* реализовывать 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 добавления к нему конструктора без параметров**. Часто это не требует особых усилий, поскольку многие классы, предназначенные для наследования, не имеют состояния. Но так бывает не всегда.

Лучше всего устанавливать все инварианты объекта при его создании (статья 15). Если для установки инвариантов необходима информация от клиента, это будет препятствовать использованию конструктора без параметров. Наивное добавление конструктора без параметров и метода инициализации в класс, остальные конструкторы которого устанавливают инварианты, усложняет пространство состояний этого класса и увеличивает вероятность появления ошибки.

Приведём 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 защищённый конструктор без параметров и отдельный метод инициализации. Причём 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 завершёнными, так как они установлены методом initialize:

```
// Несериализуемый класс с состоянием, для которого можно
// создать подклассы
public abstract class AbstractFoo {
    private int x, y; // Our state

    // 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;
    }

    // Must call from all public and protected instance methods
    private void checkInit() {
        if (init.get() != State.INITIALIZED)
            throw new IllegalStateException("Uninitialized");
    }
    // Остальное опущено
```

```
}
```

Все методы класса `AbstractFoo`, прежде чем выполнять свою работу, должны вызывать `checkInit`. Тем самым гарантируется быстрое и чистое аварийное завершение этих методов в случае, если неудачно написанный подкласс не инициализировал данные суперкласса. Обратите внимание, что поле `init` реализовано как атомарная ссылка (atomic reference). Это необходимо для защиты целостности класса от злонамеренного использования. Без этой предосторожности, если бы один поток вызвал `initialize` на экземпляре в тот момент, когда его попытался бы использовать другой поток, второй поток мог бы увидеть этот экземпляр в несогласованном состоянии. Имея этот механизм взамен прежнего, вы можете 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();

        // Manually deserialize and initialize superclass state
        int x = s.readInt();
        int y = s.readInt();
        initialize(x, y);
    }

    private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();

        // Manually serialize superclass state
        s.writeInt(getX());
        s.writeInt(getY());
    }

    // Constructor does not use the fancy mechanism
    public Foo(int x, int y) {
```

```
        super(x, y);
    }

    private static final long serialVersionUID = 1856835860954L;
}
```

Внутренние классы (статья 22) не должны реализовывать интерфейс `Serializable`. Для размещения ссылки на экземпляр внешнего класса (enclosing instance) и значений локальных переменных из окружения они используют синтетические поля (synthetic fields), генерируемые компилятором. Как именно эти поля соотносятся с объявлением класса, не конкретизируется. Не конкретизируются также названия анонимных и локальных классов. Поэтому **сериализованная форма по умолчанию для внутренних классов не определена чётко**. Однако *статический класс-член* (static member class) вполне может реализовывать интерфейс `Serializable`.

Подведём итоги. Лёгкость реализации интерфейса `Serializable` является обманчивой. Реализация интерфейса `Serializable` — серьёзное обязательство, которое следует брать на себя с осторожностью, если только не предполагается выбросить класс после недолгого использования. Особого внимания требует класс, предназначенный для наследования. Для таких классов компромиссом между реализацией интерфейса `Serializable` в подклассах и его запретом является создание доступного конструктора без параметров. Это позволяет реализовать в подклассе интерфейс `Serializable`, но оставляет создателю подкласса возможность его не реализовывать.

Статья 75. Рассмотрите возможность использования специализированной сериализованной формы

Если вы создаёте 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 быть взвешенным, 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 тем представлением, 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 равнозначно 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 {
    /**
    * Фамилия. Не может быть null.
    * @serial
    */
    private final String lastName;

    /**
    * Личное имя. Не может быть null.
    * @serial
    */
    private final String firstName;
```

```
    /**
     *
     * Среднее имя или null, если оно отсутствует
     * @serial
     */
    private final char middleName;

    ... // Остальное опущено
}
```

Логически полное имя человека в английском языке состоит из двух строк, представляющих фамилию, личное имя и среднее имя. Поля экземпляра в классе `Name` в точности воспроизводят это логическое содержание.

Даже 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 на полях `lastName`, `firstName` и `middleInitial` присутствуют комментарии документации, хотя все они являются закрытыми. Это необходимо, поскольку эти закрытые поля определяют открытый API — сериализованную форму класса. 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 список 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 собой последовательность строк. Физически последовательность представлена им как двусвязный 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 приведённом 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 приведённом 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 ей приходится выполнять дорогостоящий обход вершин графа. В приведённом 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 1258 элементами. Количество элементов, вызывающее эту проблему, меняется в зависимости от реализации JVM. В некоторых реализациях этой проблемы вообще не существует.

Разумная сериализованная форма для класса `StringList` — это количество строк в списке, за которым следуют сами строки. Это соответствует освобождённым от деталей физической реализации логическим данным, представляемым классом `StringList`. Приведём исправленный вариант `StringList`, содержащий методы `writeObject` и `readObject`, которые реализуют такую сериализованную форму. Напомним, что модификатор `transient` указывает на то, что экземпляр поля должен быть исключён из сериализованной формы, применяемой по умолчанию:

```
// Класс StringList с разумной сериализованной формой
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;

    // No longer Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // Appends the specified string to the list
    public final void add(String s) {
        // Implementation omitted
    }

    /**
     * Serialize this {@code StringList} instance.
```

```
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted ({@code int}), followed by all of
     * its elements (each a {@code String}), in the proper sequence.
     */
    private void writeObject(ObjectOutputStream s) throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s) throws IOException,
            ClassNotFoundException {
        s.defaultReadObject();
        int numElements = s.readInt();

        // Read in all elements and insert them in list
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }

    ... // Остальное опущено
}
```

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 поля экземпляра имеют модификатор `transient`, то формально можно обойтись без вызова методов `defaultWriteObject` и `defaultReadObject`, но это не рекомендуется.** Даже если все поля экземпляра 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 поля экземпляра без модификатора `transient`, 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 инициированием `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 ещё меньше. Для `StringList` сериализованная форма, применяемая по умолчанию, не обладает гибкостью и работает медленно. Однако она является *корректной* в том смысле, что в результате сериализации и десериализации экземпляра `StringList` получается точная копия исходного объекта, и все его инварианты сохраняются. Но для любого объекта, чьи инварианты привязаны к деталям реализации, это не так.

Рассмотрим, например, хэш-таблицу. 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 серьёзной ошибкой: сериализация и десериализация хэш-таблицы могут привести к созданию объекта, инварианты которого будут серьёзно 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 сериализованную форму, нужно пометить этим модификатором. К таковым относятся избыточные поля, чьи значения можно вычислить «первичным полям данных», например, 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 машинозависимую (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 поля экземпляра 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, `0` 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` (статья 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 окружить сериализацию объекта такой же синхронизацией, какую бы вы использовали для любого другому методу, читающего все состояние объекта**. Так, например, если у вас есть объект, поддерживающий многопоточность (статья 70), который обеспечивает потокобезопасность, 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 объявляйте `serialversionUID`. Тем самым вы исключите этот идентификатор из числа возможных причин несовместимости (статья 74). Это также даст некоторый выигрыш в производительности. Если `serialversionUID` не представлен, то при выполнении программы для его генерации потребуется выполнить трудоёмкие вычисления.

Для объявления `serialversionUID` 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 путём запуска для класса утилиту `serialver`. Однако можно взять число просто «из воздуха». Если вы меняете существующий класс, в котором нет поля `serialVersionUID`, и вы хотите, чтобы новая версия принимала сериализованные экземпляры старой, вам необходимо использовать значение, которое было автоматически сгенерировано старой версией класса — той, сериализованные экземпляры которой уже существуют.

> **Комментарий.** Современные IDE, такие как Eclipse, умеют сами добавлять в класс значение `serialVersionUID` по тому же алгоритму, по которому его вычисляет утилита `serialver`, так что вызывать её вручную обычно нет необходимости.

Подведём 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 проектирование

сериализованной формы для класса вы должны выделить не меньше времени, чем на проектирование его методов, видимых 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 неустранимое отрицательное влияние на сложность и производительность класса.

Статья 76. Включайте защитные проверки в метод readObject

В статье 39 представлен неизменяемый класс для интервалов времени, который содержит изменяемые закрытые поля даты. Чтобы сохранить свои инварианты и неизменяемость, этот класс создаёт защитную копию объектов `Date` в конструкторе и методах доступа. Приведём этот класс:

```
// Неизменяемый класс, который использует защитное 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.
    * @throws IllegalArgumentException, если start позже, чем end.
    * @throws NullPointerException, если start или end равны null.
    */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start + " after " + end);
    }
```

```
    public Date start() { return new Date(start.getTime()); )

    public Date end() { return new Date(end.getTime()); }

    ... // 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 объявление 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 ещё 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 защитные 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, грубо говоря, конструктор, который в качестве единственного входного параметра принимает поток байтов. В нормальных условиях этот поток байтов создаётся в результате сериализации нормально 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 объявление 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 класса 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 путём сериализации обычного экземпляра `Period` и ручного редактирования полученного потока байтов. 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` выбросит 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 злоумышленнику создать неправильный экземпляр класса `Period`. Однако здесь притаилась ещё 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 содержимое объектных ссылок, добавленных к потоку. Имея эти ссылки, злоумышленник получает доступ к объектам, на которые ссылаются закрытые поля `Date` объекта `Period`. Меняя эти экземпляры `Date`, он может менять и сам экземпляр `Period`. Следующий класс демонстрирует атаку такого рода:

```
public class MutablePeriod {
    // A period instance
    public final Period period;
```

```
    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;

    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Append rogue "previous object refs" for internal Date
             * fields in Period. For details, see "Java Object
             * Serialization Specification," Section 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4; // Ref # 4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            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 AssertionError(e);
        }
    }
```

```
    public static void main(String[] args) {
        MutablePeriod mp = new MutablePeriod();
        Period p = mp.period;
        Date pEnd = mp.end;

        // Let's turn back the clock
        pEnd.setYear(78);
        System.out.println(p);

        // Bring back the 60s!
        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` создаётся с неповреждёнными 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 защитного копирования. **При десериализации объекта крайне важно создать защитные копии для всех полей, содержащих ссылки на объекты, которые не должны попасть в распоряжение клиентов.** Поэтому каждый сериализуемый неизменяемый класс, содержащий закрытые изменяемые компоненты, должен в своём методе `readObject` создавать защитные копии для этих компонентов. Следующий метод `readObject` достаточен для того, чтобы объект `Period` оставался неизменяемым и сохранялись его инварианты:

```
// Метод readObject с защитным копированием и проверкой правильности
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    // Резервное копирование наших изменяемых компонентов
    start = new Date(start.getTime());
    end = new Date(end.getTime());

    // Проверка инвариантов
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

Заметим, что защитное копирование осуществляется перед проверкой корректности и что для защитного копирования не используется метод `clone` из класса `Date`. Указанные особенности реализации необходимы для защиты объекта `Period` (статья 39). Заметим также, что выполнить защитное копирование для полей `final` невозможно. Следовательно, для того чтобы можно было воспользоваться методом `readObject`, мы должны убрать модификатор `final` с полей `start` и `end`. Это огорчает, но приходится выбирать из двух зол меньшее. Разместив в классе новый метод `readObject` и удалив модификатор `final` из полей `start` и `end`, мы обнаруживаем, что класс `MutablePeriod` перестал работать. Приведённая 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
```

Комментарий. Как уже говорилось в комментарии к статье 39, в данном случае корень проблемы лежит в изменяемости класса `Date`. В новых программах, начиная с Java 8, вместо него лучше использовать неизменяемые классы из пакета `java.time`, например, `Instant`.

В версии 1.4 к классу `ObjectOutputStream` были добавлены методы `writeUnshared` и `readUnshared` с целью воспрепятствовать атакам, использующим захват ссылок на объекты, без накладных расходов на защитное копирование [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 защитное копирование, но они не обеспечивают должных гарантий.

Есть простой тест, показывающий, приемлем ли метод `readObject`, предлагаемый по умолчанию. Согласны ли вы добавить в класс открытый конструктор, который в качестве параметров принимает значения полей вашего объекта, записываемых в сериализованную форму, а затем заносит эти значения в соответствующие поля без какой-либо проверки? Если вы не можете ответить на этот вопрос утвердительно, вам нужно явным образом реализовать метод `readObject`, который должен выполнять все необходимые проверки параметров и создавать все защитные копии, как это требуется от конструктора.

Между конструкторами и методами `readObject` существует ещё 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 переопределён, то он будет вызван прежде, чем будет десериализовано состояние соответствующего подкласса. Скорее всего, это приведёт к сбою программы.

Подведём 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 формами. Приведём в кратком изложении рекомендации по написанию «пуленепробиваемого» метода `readObject`:

- В классах, где есть поля, которые хранят ссылки на объект и при этом должны оставаться закрытыми, для каждого объекта, который должен быть помещён в такое поле, необходимо создавать защитную копию. В эту категорию попадают также изменяемые компоненты неизменяемых классов.
- Проверьте выполнение инвариантов и в случае их невыполнения выбрасывайте исключение `InvalidObjectException`. Проверка должна производиться после создания всех защитных копий.

- Если после десериализации необходимо проверить целый граф объектов, следует использовать интерфейс `ObjectInputValidation`. [JavaSE6, Serialization]
- Ни прямо, ни косвенно не вызывайте переопределяемые методы из метода `readObject`.

Статья 77. Для контроля экземпляров предпочитайте перечислимые типы 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 перестаёт быть синглтоном, если в его объявление были добавлены слова `implements Serializable`. Не имеет значения, использует ли этот класс сериализованную форму, предлагаемую по умолчанию, или специальную форму (статья 73), предоставляется ли пользователю в этом классе явный метод `readObject` или нет (статья 76). Любой метод `readObject`, явный или используемый по умолчанию, возвращает новый экземпляр класса, а не тот экземпляр, который был создан в момент инициализации класса.

Метод `readResolve` позволяет нам вернуть другой экземпляр класса вместо того, который был создан методом `readObject` [Serialization, 3.7]. Если класс десериализуемого объекта имеет метод `readResolve` с соответствующим объявлением, то по завершении десериализации этот метод будет вызван для вновь созданного объекта. Клиенту вместо ссылки на вновь созданный объект передаётся ссылка, возвращаемая этим методом. В большинстве случаев, когда используется этот механизм, ссылка на новый объект не сохраняется, а сам объект фактически является «мертворождённым» и немедленно становится объектом для сборки мусора.

Если класс `Elvis` реализует интерфейс `Serializable`, то для того, чтобы обеспечить свойство синглтона, достаточно будет создать следующий метод `readResolve`:

```
// Метод readResolve для контроля экземпляров - можно сделать лучше!
private Object readResolve() {
    // Возвращает единственного истинного Элвиса и даёт возможность
    // сборщику мусора избавиться от Элвиса-самозванца
    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 экземпляров, все поля со ссылками на объекты *должны* быть объявлены как `transient`.** В противном случае целеустремлённый злоумышленник может украсть ссылку на десериализованный объект до того, как для него будет вызван метод `readResolve`, используя приём, который частично похож на атаку `MutablePeriod` из статьи 76.

Эта атака немного сложна, но основная идея проста. Если синглтон содержит поле со ссылкой на объект без модификатора `transient`, содержимое этого поля будет десериализовано до того, как будет запущен метод синглтона `readResolve`. Это позволяет создать такую сериализованную форму, которая сможет «украсть» ссылку на исходный десериализуемый синглтон во время десериализации объекта, на который указывает ссылка.

Вот как это работает. Сначала пишется класс-вор, у которого есть и метод `readResolve`, и поле экземпляра, ссылающееся на сериализованный синглтон, в котором «прячется» класс-вор. В потоке сериализации происходит замена поля синглтона, у которого отсутствует модификатор `transient`, экземпляром класса-вора. Теперь у вас появилась цикличность: синглтон ссылается на экземпляр класса-вора, а тот ссылается на синглтон. Поскольку синглтон содержит экземпляр класс-вора, то при десериализации синглтона первым делом выполняется метод `readResolve` класса-вора — а в этот момент поле его экземпляра все ещё ссылается на частично десериализованный синглтон.

Метод `readResolve` класса-вора 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 выбросила бы `ClassCastException` при попытке системы сериализации сохранить украденную ссылку в это поле. Для конкретности рассмотрим следующий приём взломанного синглтона:

```
// Взломанный синглтон - у него есть поле со ссылкой на объект,
// не объявленное как transient!
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() throws ObjectStreamException {
        return INSTANCE;
    }
}
```

А вот класс-вор, созданный как описано выше:

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private Object readResolve() {
        // Save a reference to the "unresolved" Elvis instance
        impersonator = payload;

        // Return an object of correct type for favorites field
```

```
        return new String[] { "A Fool Such as I" };
    }

    private static final long serialVersionUID = 0;
}
```

И наконец, уродливая программа, которая десериализует поток байт, созданный вручную, и тем самым получает два разных экземпляра дефектного синглтона. Метод десериализации здесь опущен, так как он идентичен такому же методу в статье 76:

```
public class ElvisImpersonator {
    // Этот байтовый поток не мог быть получен
    // 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, 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) {
        // Initializes ElvisStealer.impersonator and returns
        // the real Elvis (which is 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 `favoriteSongs` как `transient`, но лучшим решением будет сделать `Elvis` единственным элементом перечислимого типа (статья 3). Исторически метод `readResolve` использовался для всех сериализуемых классов с контролем экземпляров, но начиная с версии 1.5, для контроля экземпляров есть лучшее решение. поддерживать контроль экземпляров в сериализуемом классе. Как нам продемонстрировала атака `ElvisStealer`, приём с `readResolve` хрупок и требует огромной осторожности.

Если вместо этого вы реализуете сериализуемый класс с контролем экземпляров 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. С вашей стороны для этого не потребуется никаких дополнительных усилий. Вот как будет выглядеть пример с классом `Elvis`, реализованным в виде перечислимого типа:

```
// 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));
    }
```

```
}
```

Тем не менее использование метода `readResolve` для контроля над экземплярами всё ещё имеет свои области применения. Если вам нужно написать сериализуемый класс с контролем экземпляров, экземпляры которого не известны на момент компиляции, вы не сможете представить класс как перечислимый тип.

Уровень доступа к методу `readResolve` очень важен. Если вы поместите метод `readResolve` в класс, объявленный как `final`, он должен быть закрытым. Если вы поместите метод `readResolve` в класс, доступный для наследования, вам нужно внимательно отнестись к его уровню доступа. Если он закрытый, то его действие не будет распространяться на подклассы. Если он закрытый только в рамках пакета, его действие будет распространяться только на подклассы внутри того же пакета. Если он защищённый или открытый, то его действие будет распространяться на все подклассы, не переопределяющие его. Если метод `readResolve` защищён или является открытым и подклассы его не переопределяют, то десериализация и экземпляра подкласса произведёт экземпляр суперкласса, что, скорее всего, приведёт к исключению `ClassCastException`.

Подведём итоги. Насколько возможно, используйте перечислимые типы, чтобы обеспечить инварианты контроля экземпляров. Если это невозможно и вам нужен класс, который был бы сериализуемым и при этом осуществлял контроль экземпляров, то вам нужен метод `readResolve`, чтобы убедиться, что все поля экземпляров класса либо имеют примитивный тип, либо объявлены как `transient`.

Статья 78. Рассмотрите возможность использования прокси-классов сериализации вместо сериализованных экземпляров

Как уже говорилось в статье 74 и обсуждалось на протяжении всей главы, применение интерфейса `Serializable` увеличивает вероятность ошибок и проблем с безопасностью, так как он создаёт экземпляры с помощью внеязыкового (extralinguistic) механизма вместо обычных конструкторов. Есть тем не менее приём, который существенно

снижает подобный риск. Этот приём известен как *шаблон прокси-класса сериализации* (serialization proxy pattern).

Шаблон прокси-класса сериализации довольно прямолинеен. Во-первых, внутри сериализуемого класса создайте закрытый статический вложенный класс, который в точности представляет собой логическое состояние экземпляра внешнего класса. Этот вложенный класс, известный как *прокси-класс сериализации* (serialization proxy), должен иметь единственный конструктор, типом параметра которого является внешний класс. Этот конструктор просто копирует данные из своего аргумента: ему не требуется проверка целостности или защитное копирование. По построению сериализованная форма прокси-класса сериализации по умолчанию является идеальной сериализованной формой для внешнего класса. И внешний класс, и его прокси-класс сериализации должны быть объявлены как `implements Serializable`.

Например, рассмотрим неизменяемый класс `Period`, написанный в статье 39 и сериализованный в статье 76. Вот прокси-класс сериализации для этого класса. Класс `Period` настолько прост, что его прокси-класс сериализации содержит в точности те же самые поля, что и основной класс:

```
// Прокси-класс 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 внешнему классу. Этот метод можно дословно скопировать в любой класс с прокси-классом 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 внешнего класса. Другими словами, метод `writeReplace` преобразует экземпляр внешнего класса в экземпляр прокси-класса сериализации до выполнения самой сериализации.

При использовании метода `writeReplace` система сериализации никогда не выдаст сериализованный экземпляр внешнего 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 внешнему классу:

```
// Метод readObject для шаблона прокси-класса сериализации
private void readObject(ObjectInputStream stream)
        throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

Наконец, добавьте в класс `SerializationProxy` метод `readResolve`, который возвратит логически эквивалентный экземпляр внешнего класса. Присутствие этого метода приводит к тому, что система сериализации преобразует экземпляр прокси-класса сериализации обратно в экземпляр внешнего класса при десериализации.

Этот метод `readResolve` создаёт экземпляр внешнего класса, используя только открытый API, и в этом и заключается вся красота шаблона. Он практически устраняет внеязыковую природу сериализации, потому что десериализованный экземпляр создаётся с использованием тех же самых конструкторов и статических фабричных методов, что и любой другой экземпляр. Это освобождает вас от необходимости отдельно проверять, что десериализованные экземпляры подчиняются инвариантам класса. Если статические фабричные методы или конструкторы устанавливают эти инварианты, а методы экземпляра сохраняют их, то у вас есть гарантия, что сериализация будет также их поддерживать.

Вот метод `readResolve` для класса `Period.SerializationProxy`.

```
private Object readResolve() {
    return new Period(start, end); // Uses public constructor
}
```

Как и подход с использованием защитного копирования (статья 76), подход с использованием прокси-класса сериализации препятствует сложным атакам с использованием созданных вручную байтовым потоков и атакам кражи внутреннего поля (статья 76). В отличие от двух предыдущих подходов этот подход позволяет полям класса `Period` быть объявленными как `final`, что требуется для того, чтобы класс `Period` действительно был 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 скомпрометированы при атаках сериализации; вам также не нужно явно выполнять проверки правильности данных в процессе десериализации.

Есть ещё одна сторона, в которой шаблон прокси-класса сериализации мощнее защитного копирования. Шаблон прокси-класса сериализации позволяет сериализованному и десериализованному экземплярам принадлежать к разным классам. Вы, возможно, подумаете, что на практике эта возможность бесполезна, но это не так.

Рассмотрим `EnumSet` (статья 32). У этого класса нет открытого конструктора, только статические фабричные методы. С точки зрения клиента они возвращают экземпляры `EnumSet`, но на самом деле она возвращают экземпляры одного из двух подклассов в зависимости от размера перечислимого типа (статья 1). Если у перечислимого типа 64 или менее элементов, то статические фабричные методы возвращают `RegularEnumSet`; в противном случае они возвращают `JumboEnumSet`. Теперь посмотрим, что происходит, если вы сериализуете объект `EnumSet`, в котором перечислимый тип содержит 60 элементов, затем добавите к нему ещё пять элементов и десериализуете этот объект. При сериализации это был экземпляр `RegularEnumSet`, но лучше бы ему быть экземпляром `JumboEnumSet` при десериализации. На самом деле происходит именно это, потому что `EnumSet` использует шаблон прокси-класса сериализации. Если вам интересно, вот как выглядит прокси-класс сериализации класса `EnumSet`. Он действительно настолько прост:

```
// Прокси-класс сериализации EnumSet
private static class SerializationProxy <E extends Enum<E>>
        implements Serializable {
    // Тип элементов данного объекта EnumSet
    private final Class<E> elementType;

    // Элементы, содержащиеся в данном объекте EnumSet
    private final Enum[] 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 прокси-класса сериализации есть два ограничения. Он несовместим с классами, расширяемыми своими клиентами (статья 17). Он также несовместим с некоторыми классами, диаграммы объектов которых содержат циклические зависисости: если вы попытаетесь вызвать метод такого объекта из метода `readResolve` прокси-класса сериализации, то получите исключение `ClassCastException`, так как у вас ещё нет объекта, а только экземпляр его прокси-класса сериализации.

Наконец, за дополнительные возможности и безопасность шаблона прокси-класса сериализации приходится платить. Но моей машине сериализовывать и десериализовывать экземпляры `Period` с помощью прокси-класса сериализации на 14% более затратно, чем с помощью защитного копирования.

Подведём итоги. Рассмотрите шаблон прокси-класса сериализации каждый раз, когда вам приходится писать методы `readObject` или `writeObject` на классе, который нерасширяем для своих клиентов. Этот шаблон, вероятно, является простейшим путём сериализации объектов с нетривиальными инвариантами.

Комментарий. Для самостоятельного изучения можно порекомендовать разобраться с красивым вариантом этого шаблона, используемым в пакете `java.time` в Java 8. Вместо того, чтобы создавать по одному прокси-классу сериализации на каждый открытый класс, библиотека обходится только одним прокси-классом `java.time.Ser` на весь пакет (и ещё двумя аналогичными классами, по одному на каждый подпакет). Класс десериализуемого экземпляра задаётся в виде числового поля-признака, а саму логику сериализации и десериализации этот разделяемый прокси-класс отдаёт на откуп каждому из обслуживаемых им открытых классов с помощью методов `readExternal` и `writeExternal`, доступных только в пределах пакета.

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 TM 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.

9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition]

[Posix] 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

[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

[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/

[Wulf72] Wulf, W. *A Case Against the GOTO*. In *Proceedings of the 25th ACM National Conference 2* (1972): 791–797.