Effective Java ™

Programming Language Guide

Joshua Bloch

ADDISON-WESLEY

Java ™

Эффективное программирование

Джошуа Блох

Издательство «Лори»

Благодарности

Я благодарю Патрика Чана (Patrick Chan) за то, что он посоветовал мне написать эту книгу и подбросил идею Лайзе Френдли (Lisa Friendly), главному редактору серии, а также Тима Линдхолма (Tim Lindholm), технического редактора серии, и Майка Хендриксона (Mike Hendrickson), исполнительного редактора издательства Addison- Wesley Professional. Спасибо Лайзе, Тиму и Майку за их поддержку при реализации 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 предоставили мне нечто значительное, о чем можно написать, а также многих разработчиков платформы Java, последователей 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 Benriett), Джо Дарси Ooe Darcy), Нил Гафтер (Neal Gafter), Айрис Гарсиа (Iris Garcia), Константин Кладко (Konstantin Kladko), Йена 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).

Выражаю благодарность моему руководителю Эндрю Беннетту (Andrew Bennett) и директору Ларри Абрахамсу (Larry Abrahams) за полную и страстную поддержку этого проекта. Спасибо Ричу Грину (Rich Green), вице-президенту компании Java Software, за создание условий, при которых разработчики имеют возможность творить и публиковать свои труды.

Мне чрезвычайно повезло с самой лучшей, какую только можно вообразить, группой рецензентов, и я выражаю 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 Ooe Bowbeer), Джиладу Браче (Gilad Bracha), Мэри Кампьон (Mary Campione), Джо Дарси Ooe Darcy), Дэвиду Экхардту (David Eckhardt), Джо Фьалли Ooe Fialli), Лайзе Френдли (Lisa Friendly), Джеймсу Гослингу (James Gosling), Питеру Хаггеру (Peter Haggar), Брайену KePl:лигану (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 (ВШ· Shannon), Питеру Стауту (Peter Stout), Филу Уодлеру (Phil Wadler), Давиду Холмсу (David Holmes) и двум анонимным рецензентам. Они внесли множество предложений, которые позволили существенно улучшить 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). Особая благодарность Дагу Ли (Doug Lea), озвучившему многие идеи этой книги. Даг неизменно щедро делился своим временем и знаниями.

Я благодарен Джули Дайникола (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 занятыми до предела, они всегда были дружелюбны и учтивы.

Я благодарю Гая Стила (Сиу Steele), написавшего предисловие. Его участие в этом проекте - большая честь для меня.

Наконец, спасибо 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, за написание предметного указателя и за то, что терпела меня, пока я корпел над этой книгой.

Содержание

7. Общие вопросы программирования

8. Исключения

9. Потоки

10. Сериализация

11. Литература

Предисловие

Если бы сослуживец сказал вам: "Моя супруга сегодня вечером готовит дома нечто необычное. Придешь?" (Spouse of me this night today manufactures the unusual meal in a home. You will join?), вам в голову, вероятно, пришли бы сразу три мысли: вас уже пригласили на обед; английский язык не является родным для вашего сослуживца; ну и прежде всего это слишком большое беспокойство.

Если вы сами когда-нибудь изучали второй язык, а затем пробовали пользоваться им за пределами аудитории, то вам известно, что есть три вещи, которые необходимо знать: каким образом структурирован язык (грамматика), как называется то, о чем вы хотите сказать (словарь), а также общепринятые и Эффективные варианты повседневной речи (лексические обороты). В аудитории обычно ограничиваются изучением лишь первых двух из этих вещей, и вы обнаруживаете, что окружающие постоянно давятся от смеха, выслушивая, как вы пытаетесь говорить понятно.

Практически так же обстоит дело с языком программирования. Вы должны понимать суть языка: является ли он алгоритмическим, функциональным, объектно-ориентированным. Вам нужно знать словарь языка: какие структуры данных, операции и возможности' предоставляют стандартные библиотеки. Кроме того, вам необходимо ознакомиться с общепринятыми и эффективными способами структурирования Кода. В книгах, посвященных языкам программирования, часто освещаются лишь первые два вопроса, приемы работы с языком, если и обсуждаются, то лишь кратко. Возможно, это происходит потому, что о первых двух вещах писать несколько проще. Грамматика и словарь - это свойства самого языка, тогда как способ его применения характеризует группу людей, пользующихся этим языком.

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

Есть и другой аспект. Программы, в отличие от произнесенных фраз и большинства изданных книг и журналов, имеют возможность меняться со временем. Недостаточно создать программный код, который эффективно работает и без труда может быть понят другими людьми. Нужно еще организовать этот код таким образом, чтобы его можно было легко модифицировать для не которой задачи *А* существует десяток вариантов написания программного кода. Из этих десяти семь оказываются неуклюжими, неэффективными или запутывающими читателя. Какой же из оставшихся трех вариантов будет представлять собой программный код, который потребуется в следующем году для новой версии программы, ,решающей задачу *А'?*

Существует много книг, по которым можно изучать грамматику языка программирования Java, в том числе книги *"The Java Programming Language"* авторов Arnold, Gosling и Holmes [ArnoldOO] и *"The Java Language Specification"* авторов Gosling, Joy, Bracha и вашего покорного слуги [JLS]. Немало книг посвящено библиотекам и прикладным интерфейсам, связанным с Java.

Эта книга посвящена третьей теме: общепринятым и эффективным приемам работы с языком Java. На протяжении нескольких лет Джошуа Блох (Joshua Block) трудился в компании Sun Microsystems, работая с языком программирования Java, занимаясь расширением и реализацией программного кода. Он изучил большое количество программ, написанных многими людьми, в том числе и мною. 8 настоящей книге он дает дельные советы о том, каким образом структурировать код, чтобы он работал хорошо, чтобы его могли понять другие люди, чтобы последующие модификации и усовершенствования доставляли меньше головной боли и чтобы ваши программы были приятными, элегантными и красивыми.

Гай А. Стил-младший (Сиу L. Steele Jr.) Берлингтон, шт. Массачусетс Апрель 2001

Предисловие автора

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

Эта книга является попыткой поделиться с вами моим опытом, чтобы вы смогли повторить мои успехи и избежать моих неудач. Оформление книги я позаимствовал из руководства Скотта Мейерса (Scott Meyers) *"Effective C++"*[Meyers98]; оно состоит из пятидесяти статей, каждая из которых посвящена одному конкретному правилу, направленному на улучшение программ и проектов. Я нашел такое оформление необычайно эффективным, и надеюсь, вы тоже его оцените.

Во многих случаях я иллюстрирую статьи реальными примерами из библиотек для платформы Java. Говоря, что нечто можно сделать лучше, я старался брать программный код, который писал сам, однако иногда я пользовался разработками коллег. Приношу мои искренние извинения, если, не желая того, обидел кого-либо. 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.

Эта книга предназначена не только для тех, кто занимается разработкой повторно используемых компонентов, тем не менее она неизбежно отражает мой опыт в написании таковых, накопленный за последние два десятилетия. Я привык думать в терминах прикладных интерфейсов (API) и предлагаю вам делать то же. Даже если вы не занимаетесь разработкой повторно используемых компонентов, применение этих терминов поможет вам повысить качество ваших программ. Более того, нередко случается писать многократно используемые компоненты, не подозревая об этом: вы создали 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 потратили на его разработку, когда писали программу в первый раз.

Мое особое внимание к разработке API может показаться несколько противоестественным для ярых приверженцев новых облегченных методик создания 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 API сослужит вам добрую службу в процессе последующей *перестройки* программы (refactoring). Основной задачей перестроения является усовершенствование структуры системы, а также исключение дублирующего программного кода. Этой цели невозможно достичь, если у 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.

Джошуа Блох

Купертино, шт. Калифорния

Апрель 2001

Глава 1

Введение

Эта книга писалась с той целью, чтобы помочь вам наиболее эффективно использовать язык программирования Java ™ и его основные библиотеки java.lang, java.util и java.io. В книге рассматриваются и другие библиотеки, но мы не касаемся графического интерфейса пользователя и специализированных API.

Книга состоит из пятидесяти семи статей, каждая из которых описывает одно правило. Здесь собран опыт самых лучших и опытных программистов. Статьи произвольно распределены по девяти главам, освещающим определенные аспекты проектирования программного обеспечения. Нет необходимости читать эту книгу от корки до корки: каждая статья в той или иной степени самостоятельна. Статьи имеют множество перекрестных ссылок, поэтому вы можете с легкостью построить по книге ваш собственный учебный курс.

Большинство статей сопровождается примерами программ. Главной особенностью этой книги является наличие в ней примеров программного кода, иллюстрирующих *многие шаблоны* (design pattern) и идиомы. Некоторые из них, такие как Singleton (статья 2), известны давно, другие появились недавно, например Finalizer Guardian (статья б) и Defensive readResolve (статья 57). Где это необходимо, шаблоны и идиомы имеют ссылки на основные работы в данной области [Camma95].

Многие статьи содержат при меры программ, иллюстрирующие приемы, которых следует избегать. Подобные примеры, иногда называемые *"антишаблонами",* четко обозначены комментарием *"// никогда не делайте так!"* В каждом таком случае в статье дается объяснение, почему пример плох, и предлагается альтернатива.

Эта книга не предназначена для начинающих: предполагается, что вы уже хорошо владеете языком программирования Java. В противном случае обратитесь к одному из множества прекрасных изданий для начинающих [ArnoldOO, CampioneOO]. Книга построена так, чтобы быть доступной для любого, кто работает с этим языком, тем не менее она дает пищу для размышлений даже опытным программистам.

В основе большинства правил этой книги лежит несколько фундаментальных принципов. Ясность и простота имеют первостепенное значение. Функционирование модуля не должно вызывать удивление у него 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 раньше, в идеале - уже на стадии компиляции.

Правила, изложенные в этой книге, не охватывают все сто процентов практики, но они описывают самые лучшие приемы программирования. Нет необходимости покорно следовать этим правилам, но и нарушать их нужно, лишь имея на то вескую причину. Как и для многих других дисциплин, изучение искусства программирования заключается сперва в освоении правил, а затем в изучении условий, когда они нарушаются.

Большая часть этой книги посвящена отнюдь не производительности программ.
Речь идет о написании понятных, прав ильных, полезных, надежных, гибких программ, которые удобно сопровождать. Если вы сможете сделать это, то добиться необходимой производительности будет несложно (статья 37). В некоторых статьях обсуждаются вопросы производительности, в ряде случаев приводятся показатели производительности. Эти данные, предваряемые выражением "на моей машине", следует рассматривать как приблизительные.

Для справки, моя машина - это старый компьютер домашней сборки с процессором 400 МГц Pentium® II и 128 Мбайт оперативной памяти под управлением Microsoft Windows NT® 4.0, на котором установлен Java 2 Standard Edition Software Development Kit (SDK) компании Sun. В состав этого SDK входит Java HotSpot ™ Client УМ компании Sun - финальная реализация виртуальной машины Java, предназначенной для клиентов.

При обсуждении особенностей языка программирования Java и его библиотек иногда необходимо ссылаться на конкретные версии для краткости в этой книге используются "рабочие", а не официальные номера версий. В таблице 1.1 показано соответствие между названиями версий и их рабочими номерами.

В некоторых статьях обсуждаются возможности, появившиеся в версии 1.4, однако в примерах программ, за редким 'исключением, я воздерживался от того, чтобы пользоваться ими. Эти примеры были проверены в версии 1.3. Большинство из них, если не все, без всякой переделки должны работать также с версией 1.2.

Официальное название версии	Рабочий номер версии
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v1.2	1.2
Java 2 Platform, Standard Edition, v1.3	1.3
Java 2 Platform, Standard Edition, v1.4	1.4

Примеры по возможности являются полными, однако предпочтение отдается не завершенности, а удобству чтения. В примерах широко используются классы пакеТ08 java.util и java. 10. Чтобы скомпилировать пример, вам потребуется добавить один или оба оператора import:

```
Import java.ut11.*;
Import java.io.*;
```

В примерах опущены детали. Полные версии всех примеров содержатся на web-сайте этой книги (http://java.sun.com/docs/books/effective). При желании любой из них можно скомпилировать и запустить.

Технические термины в этой книге большей частью используются в том виде, как они определены в *"The Java Language Specification, Second Edition"* [JLS]. Однако некоторые термины заслуживают отдельного упоминания. Язык Java поддерживает четыре группы типов: *интерфейсы* (interface), *классы* (class), *массивы* (array) и *простые типы* (primitive). Первые три группы называются *ссылочными типами* (reference type). Экземпляры классов и массивов '- это *объекты,* значения простых типов таковыми не являются. К *членам* класса (members) относятся его *поля* (fields), *методы* (methods), *классы-члены* (member classes) и *интерфейсы-члены* (mernber interfaces). Сигнатура метода (signature) состоит из его названия и типов, которые имеют его формальные параметры. Т ил значения, возвращаемого методом, в сигнатуру *не входит.*

Некоторые термины в этой книге используются в ином значении, чем в *"The Java . Language Speci/ication".* Так, *"наследование"* (inheritance) применяется как синоним *"образования подклассов"* (subclassing). Вместо использования для интерфейсов термина "наследование" в книге констатируется, что некий класс *реализует* (implement) интерфейс или что один интерфейс является *расширением* другого (extend) для описания' уровня доступа, который применяется, 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 встречается 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). Это позволяет избежать путаницы с одноименной конструкцией языка Java.) Программист, который пишет программу, применяющую некий API, называется *пользователем* (user) указанного API. Класс, в реализации которого используется некий API, называется *клиентом* (client) этого API.

Классы, интерфейсы, конструкторы, члены и сериализованные формы называются *элементами API* (API element). Внешний API образуется из элементов API, которые доступны за пределами пакета, где этот API был определен. Указанные элементы может использовать любой клиент, автор API берет на себя их поддержку. Неслучайно документацию именно к этим элементам генерирует утилита Javadoc при запуске в режиме по умолчанию. В общих чертах, внешний API пакета состоит из открытых (public) и защищенных (protected) членов, а также из конструкторов всех открытых классов и интерфейсов в пакете.

Глава 2

Создание и уничтожение объектов

В этой главе речь идет о создании и уничтожении объектов: как и когда создавать объекты, как убедиться в том, что' объекты уничтожались своевременно, а также как управлять операциями по очистке, которые должны предшествовать уничтожению объекта.

Рассмотрите возможность замены конструкторов статическими методами генерации.

Обычно для того, чтобы клиент мог получать экземпляр класса, ему предоставляется открытый (public) конструктор. Есть и другой, менее известный прием, который должен быть в арсенале любого программиста. Класс может иметь открытый *статический метод генерации* (static factory method), который является статическим методом, возвращающим экземпляр класса. Пример такого метода возьмем из класса Boolean (являющего оболочкой для простого типа boolean). Приведенный ниже статический метод генерации, который был добавлен в версию 1.4, преобразует значение boolean в ссылку на объект Boolean:

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

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

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

Класс может иметь только один конструктор с заданной сигнатурой. Известно, что программисты обходят данное 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. Т е, кто читает' программный код, в котором при меняются такие конструкторы, не смогут понять, что же он делает, если не будут сверяться с сопроводительной документацией на этот класс.

Поскольку статические методы генерации имеют имена, к ним не относится ограничение конструкторов, запрещающее иметь в классе более одного метода с заданной сигнатурой. Соответственно в ситуациях, когда очевидно, что в классе должно быть несколько конструкторов с одной и той же сигнатурой, следует рассмотреть возможность замены одного или нескольких конструкторов статическими методами генерации. Тщательно выбранные названия будут подчеркивать их различия.

Второе преимущество статических методов генерации заключается в том, что, в отличие от конструкторов, они не обязаны при каждом вызове создавать новый объект. Это позволяет использовать для неизменяемого класса (статья 13) предварительно созданный экземпляр либо кэшировать экземпляры класса по мере их создания, а затем раздавать их повторно, избегая создания ненужных дублирующих объектов. Подобный прием иллюстрирует метод Boolean. valueOf(boolean): он не создает объектов. Эта методика способна значительно повысить производительность программы, если часто возникает необходимость в создании одинаковых объектов, особенно в тех случаях, когда создание объектов требует больших затрат.

Способность статических методов генерации возвращать при повторных вызовах тот же самый объект можно использовать и для того, чтобы в любой момент времени; четко контролировать, какие экземпляры объекта еще существуют. На это есть две причины. Во-первых, это позволяет гарантировать, что некий класс является синглтоном (статья 2). Во-вторых, это дает возможность убедиться в том, что у неизменяемого класса не появилось двух одинаковых экземпляров: a.equals(b) тогда и только тогда, когда а==Ь. Если класс предоставляет' такую гарантию, его клиенты могут использовать оператор == вместо метода equals(Object), что приводит к существенному повышению производительности программы. Подобную оптимизацию реализует *шаблон перечисления типов,* описанный в статье 21,частично ее реализует также метод String.intern.

Третье преимущество статического метода генерации заключается в том, что, в отличие от конструктора, он может вернуть объект, который соответствует не только заявленному типу возвращаемого значения, но и любому его подтипу. Это предоставляет вам значительную гибкость в выборе класса для возвращаемого объекта. Например, интерфейс AP! может возвращать объект, не декларируя его класс как public. Сокрытие реализации классов может привести к созданию очень компактного API. Этот прием идеально подходит для конструкций, построенных на интерфейсах, где интерфейсы для статических методов генерации задают собственный тип возвращаемого значения.

Например, архитектура Collections Framework имеет двадцать полезных реализаций интерфейсов коллекции: неизменяемые коллекции, синхронизированные коллекции и т. д. С помощью статических методов генерации большинство этих реализаций сводится в единственный класс jaya.util.Collections, для которого невозможно создать экземпляр. Все классы, соответствующие возвращаемым объектами, не, являются открытыми.

API Collections Framework имеет гораздо меньшие размеры, чем это было бы, если бы в нем были представлены двадцать отдельных открытых классов для всех возможных реализаций. Сокращен не только 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 по этому классу. Более того, применение статического метода генерации дает клиенту право обращаться к возвращаемому объекту, используя его собственный интерфейс, а не интерфейс класса реализации, что обычно является хорошим приемом (статья 34).

Скрытым может быть не только класс объекта, возвращаемого открытым статическим методом генерации. Сам класс может меняться от вызова к вызову в зависимости от того, какие значения параметров передаются статическому методу генерации. 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.

В момент написания класса, содержащего статический метод генерации, класс, соответствующий возвращаемому объекту, может даже не существовать. Подобные гибкие статические методы генерации лежат в основе *систем* с *предоставлением услуг* (service provider framework), например] aya Cryptography Extension OCE). Система с предоставлением услуг - это такая система, в которой поставщик может создавать различные реализации интерфейса API, доступные пользователям этой системы. Чтобы сделать эти реализации доступными для применения, предусмотрен механизм *регистрации* (register). Клиенты могут пользоваться указанным API, не беспокоясь о том, с какой из его реализаций они имеют дело.

В Упомянутой системе]CE системный администратор регистрирует класс реализации, редактируя хорошо известный файл Properties: делает в нем запись, которая с13лзыветТ некий ключ-строку с именем соответствующего класса. Клиенты же используют статический метод генерации, который получает этот ключ в качестве параметра.

По схеме, восстановленной из файла Properties,статический метод генерации находит объект Class,а затем создает экземпляр соответствующего класса с помощью метода Class. newInstance.Этот прием демонстрируется в следующем фрагменте:

```
import java.util.*;
// Эскиз системы с предоставлением услуг
public abstract class Foo {
   // Ставит ключ типа String в соответствие объекту Class
   private static Map implementations = null;
// При первом вызове инициализирует карту соответствия
   private static synchronized void initMapIfNecessary() {
      if (implementations == null) {
         implementations = new HashMap();
          // Загружает названия классов и ключи из файла Properties,
          // транслирует названия в объекты Class, используя
         // Class. forName, и сохраняет их соответствие ключам
         // ...
      }
   }
    public static Foo getInstance(String key) {
      initMapIfNecessary();
      Class c = (Class) implementations.get(key);
      if (c == null)
         return new DefaultFoo();
   try {
         return (Foo) c.newInstance();
      } catch (Exception e) {
         return new DefaultFoo();
      }   }
   public static void main(String[] args) {
   System.out.println(getInstance("NonexistentFoo"));
    }  }
class DefaultFoo extends Foo {
}
```

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

Второй недостаток статических методов генерации состоит в том, что их трудно отличить от других статических методов. В документации API они не выделяются так, как это делается для конструкторов. Более того, статические методы генерации представляют собой отклонение от нормы. Поэтому иногда из документации к классу сложно понять, как создать экземпляр класса, в котором вместо конструкторов

клиенту предоставлены статические· методы генерации. Указанный недостаток может быть смягчен, если придерживаться стандартных соглашений, касающихся именования. Эти соглашения продолжают совершенствоваться, но два названия статических методов генерации стали уже общепринятыми:

- valueOf' возвращает экземпляр, который имеет то же значение, что и его параметры. Статические методы генерации с таким названием фактически являются операторами преобразования типов.

- 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.

Подведем итоги. И статические методы генерации, и открытые конструкторы имеют свою область применения. Имеет смысл изучить их достоинства и недостатки. Прежде чем создавать конструкторы, рассмотрите возможность использования статических методов генерации, поскольку последние часто оказываются лучше. Если вы проанализировали обе возможности и не нашли достаточных доводов в чью-либо пользу, вероятно, лучше всего создать конструктор, хотя бы потому, что этот подход является нормой.

Свойство синглтона обеспечивайте закрытым конструктором

Синглтон (singleton) - это класс, для которого экземпляр создается только один раз [Camma95, стр.127]. Синглтоны обычно представляют некоторые компоненты системы, которые действительно являются уникальными, например видеодисплей или файловая система.

Для реализации синглтонов используются два подхода. Оба они основаны на создании закрытого (private) конструктора и открытого (public) статического члена, который обеспечивает клиентам доступ к единственному экземпляру этого класса. В первом варианте открытый статический член является полем типа final:

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

    // ... // Остальное опущено

    public static void main(String[] args) {
        System.out.println(Elvis.INSTANCE);
    }
}
```

Закрытый конструктор вызывается только один раз для инициализации поля Elvis.INSTANCE. Отсутствие открытых или защищенных конструкторов га*рантирует* "вселенную с одним Elvis": после инициализации класса Elvis будет существовать ровно один экземпляр Elvis -не больше и не меньше. И клиент не может ничего с этим поделать.

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

```
// Синглтон со статическим методом генерации
import java.io.*;
public class Elvis {
   public static final Elvis INSTANCE = new Elvis();

   private Elvis() {
      // ...
   }
         // ... // Остальное опущено
}
```

Все вызовы статического метода Elvis. getInstanceвозвращают ссылку на один и тот же объект, и никакие другие экземпляры Elvisникогда не будут созданы.

Основное преимущество первого подхода заключается в том, что из декларации членов, составляющих класс, понятно, что этот класс является синглтоном: открытое статическое поле имеет тип fina!, а потому оно всегда будет содержать ссылку на один и тот же объект. Первый вариант, по сравнению со вторым, может также иметь некоторое преимущество в производительности. Однако хорошая реализация JVM должна свести это преимущество к минимуму благодаря встраиванию (in! ining) вызовов для статического метода генерации во втором варианте.

Основное преимущество второго подхода заключается в том, что он позволяет вам отказаться от решения сделать класс синглтоном, не меняя при этом его AP!. Статический метод генерации для синглтона возвращает единственный экземпляр этого класса, однако это можно легко изменить и возвращать, скажем, свой уникальный экземпляр для каждого, потока, обращающегося к этому методу.

Таким образом, если вы абсолютно уверены, что данный класс навсегда останется синглтоном, имеет смысл использовать первый вариант. Если же вы хотите отложить решение по этому вопросу, примените второй вариант.

Если требуется сделать класс синглтона сериализуемым (см. главу 10), недостаточно добавить к его декларации implements Serializable. Чтобы дать синглтону нужные гарантии, необходимо также создать метод readResolve(статья 57). В противном случае каждая десериализация сериализованного экземпляра будет приводить к созданию нового экземпляра, что в нашем примере станет причиной

обнаружения ложных Elvis. Во избежание этого добавьте в класс Elvis следующий метод readResolve:

```
// Метод readResolve, сохраняющий свойство синглтона
    private Object readResolve() throws ObjectStreamException {
        /*
         * Возвращает единственный истинный Elvis и позволяет
         * сборщику мусора разобраться с Elvis-самозванцем
        */
        return INSTANCE;
    }

    public static void main(String[] args) {
        System.out.println(Elvis.INSTANCE);
    }
```

Общая тема связывает эту статью со статьей 21, описывающей *шаблон для перечисления типов.* В обоих случаях используются закрытые конструкторы в сочетании с открытыми статическими членами, и это гарантирует, что для соответствующего класса после инициализации не будет создано никаких новых экземпляров. В настоящей статье для класса создается только один экземпляр, в статье 21 один экземпляр создается для каждого члена в перечислении. В следующей статье в этом направлении делается еще один шаг: отсутствие открытого конструктора является гарантией того, что для класса никогда не будет создано никаких экземпляров.

Отсутствие экземпляров обеспечивает закрытый конструкто

Время от времени приходится писать класс, который является всего лишь собранием статических методов и статических полей. Такие классы приобрели дурную репутацию, поскольку отдельные личности неправильно пользуются ими с целью написания процедурных программ с помощью объектно-ориентированных языков. Подобные классы требуют правильного применения. Их можно использ~ать для того, чтобы собирать вместе связанные друг с другом методы обработки простых значений или массивов, как это сделано в библиотеках java.lang.Math и java.util.Arrays, либо чтобы собирать вместе статические методы объектов, которые реализуют определенный интерфейс, как это сделано в j aya.util.Collections. Можно также собрать Методы в некоем окончательном (fina!) классе вместо того, чтобы заниматься расширением Класса.

Подобные *классы утилит* (uti!ity c!ass) разрабатываются не для того, чтобы СОздавать для них экземпляры - такой экземпляр был бы абсурдом. Однако если у Класса нет явных конструкторов, компилятор по умолчанию сам создает для него ОТКРытый конструктор (defau!t constructor), не имеющий параметров. Для пользователя этот Конструктор ничем не будет отличаться от любого другого. В опубликованных API нередко можно встретить классы, непреднамеренно наделенные способностью порождать экземпляры.

Попытки запретить классу создавать экземпляры, объявив его абстрактным, не работают. Такой класс может иметь подкласс, для которого можно' создавать экземпляры. Более того, это вводит пользователя в заблуждение, заставляя думать, что данный класс был разработан именно для наследования (статья 15). Существует, однако, простая идиома, гарантирующая отсутствие экземпляров. Конструктор по умолчанию создается только тогда, когда у класса нет явных конструкторов, и потому запретить создание экземпляров можно, поместив в класс единственный явный закрытый конструктор:

```
// Класс утилит, не имеющий экземпляров
publicclassUtilityClass{
// Подавляет появление конструктора по умолчанию, а заодно и
создание экземпляров класса
  privateUtilityClass() {
        // Этот конструктор никогда не будет вызван

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

Поскольку явный конструктор заявлен как закрытый (private), за пределами класса он будет недоступен. И если конструктор не вызывается в самом классе, это является гарантией того, что для класса никогда не будет создано никаких экземпляров. Эта идиома несколько алогична, так как конструктор создается здесь именно для того, чтобы им нельзя было пользоваться. Есть смысл поместить в текст про граммы комментарий, который описывает назначение данного конструктора.

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

Не создавайте дублирующих объектов

Вместо того чтобы создавать новый функционально эквивалентный объект всякий раз, когда в нем возникает необходимость, можно, как правило, еще раз использовать тот же объект. Применять что-либо снова - и изящнее, и быстрее. Если объект является *неизменяемым* (immutable), его всегда можно использовать повторно (статья 13).

Рассмотрим оператор, демонстрирующий, как делать не надо:

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

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

Исправленная версия выглядит просто:

```
String s = "No longer silly";
```

В этом варианте используется единственный экземпляр String вместо создания новых при каждом проходе. Более того, гарантируется, что этот объект будет повторно использоваться любым другим программным кодом, выполняемым на той же виртуальной машине, где содержится эта строка-константа [JLS,3.10.5]. Создания дублирующих объектов часто можно избежать, если неизменяемом классе, имеющем и конструкторы, и *статические методы генерации* (статья 1), предпочесть вторые первым. Например, статический метод генерации Boolean.val'ueOf(String) почти всегда предпочтительнее конструктора Boolean(String). При каждом вызове конструктор создает новый объект, тогда как от статического метода генерации этого не требуется. Вы можете повторно использовать не только неизменяемые объекты, но и изменяемые, если знаете, что последние уже не будут меняться. Рассмотрим более тонкий и более распространенный пример того, как не надо поступать, в том Числе с изменяемыми объектами, которые, будучи получены один раз, впоследствии остаются без Изменений:

```
import java.util.*;
public class Person {
   private final Date birthDate;

  // Прочиеполяопущены

  public Person(Date birthDate) {
      this.birthDate = birthDate;
   }

 // Никогда не делайте так

  public boolean isBabyBoomer() {
      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;
   }

   public static void main(String[] args) {
      Person p = new Person(new Date());

      long startTime = System.currentTimeMillis();
      for (int i=0; i<1000000; i++)
         p.isBabyBoomer();
      long endTime = System.currentTimeMillis();
      long time = endTime - startTime;
      System.out.println(time+" ms.");
   }
}
```

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

```
import java.util.*;
class Person {
   private final Date birthDate;
  public Person(Date birthDate) {
      this.birthDate = birthDate;
}
    /**
    * Даты начала и конца демографического взрыва
    */
   private static final Date BOOM_START;
   private static final Date BOOM_END;
  static {
      Calendar gmtCal =
         Calendar.getInstance(TimeZone.getTimeZone("GMT"));
      gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
      BOOM_START = gmtCal.getTime();
      gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
      BOOM_END = gmtCal.getTime();
   }
    public boolean isBabyBoomer() {
      return birthDate.compareTo(BOOM_START) >= 0 &&
           birthDate.compareTo(BOOM_END)   <  0;
   }
    public static void main(String[] args) {
      Person p = new Person(new Date());
      long startTime = System.currentTimeMillis();
      for (int i=0; i<1000000; i++)
         p.isBabyBoomer();
      long endTime = System.currentTimeMillis();
      long time = endTime - startTime;
      System.out.println(time+" ms.");
   }   }
```

В исправленной версии класса Person экземпляры Calendar, TimeZone и Date создаются только один раз в ходе инициализации, а не при каждом вызове метода isBabyBoomer. Если данный метод вызывается часто, это приводит к значительному выигрышу в производительности. На моей машине исходная версия программы тратит на миллион вызовов 36000 мс, улучшенная - 370 мс, т. е. она работает в сто раз быстрее. Причем повышается не только производительность программы, но и наглядность. Замена локальных переменных boomStart и boomEnd статическими полями типа final показывает, что эти даты рассматриваются как константы, и программный код становится более понятным. Для полной ясности заметим, что экономия от подобной оптимизации не всегда будет столь впечатляющей, просто здесь много ресурсов требует создание экземпляров Calendar.

Если метод isBabyBoomer вызываться не будет, инициализация полей BOOM_START и BOOM_END в улучшенной версии класса Person окажется напрасной. Ненужных действий можно избежать, использовав для этих полей *отложенную инициализацию* (lazily initializing) (статья 48), которая бы выполнялась при первом вызове метода

isBabyBoomer, однако делать это не рекомендуется. Как часто бывает в случаях с отложенной инициализацией, это усложнит реализацию и вряд ли приведет к заметному повышению производительности (статья 37).

Во всех примерах, приведенных в этой статье, было очевидным то, что рассматриваемые объекты можно использовать повторно, поскольку они неизменяемые. Однако в ряде ситуаций это не столь очевидно. Рассмотрим случай с адаптерами (adapter) [Сатта95, стр. 139], известными также как представления (view). Адаптер - это объект, который делегирован нижележащим объектом и который создает для него альтернативный интерфейс. Адаптер не имеет иных состояний, помимо состояния нижележащего объекта, поэтому для адаптера, представляющего данный объект, не нужно создавать более одного экземпляра.

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

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

И наоборот, отказ от создания объектов и поддержка собственного *пула объектов* (object pool) - плохая идея, если только объекты в этом пуле не будут крайне ресурсоемкими. Основной пример объекта, для которого оправданно создание пула, - соединение с базой данных (database connection). Затраты на установление такого соединения высоки, и потому лучше обеспечить многократное использование этого объекта. Однако в общем случае создание собственного пула объектов загромождает. Современные реализации JVM имеют хорошо оптимизированные сборщики мусора, которые при работе с небольшими объектами с легкостью превосходят подобные пулы объектов.

В противовес этой статье можно привести статью 24, посвященную резервному копированию (defensive copying). Если в настоящей статье говориться: ”Не создавайте новый объект, если вы обязаны исполнять имеющийся еще раз”, то статья 24 гласит: ” Не надо использовать имеющийся объект еще раз, если вы обязаны создать новый”. Заметим, что ущерб от повторного применения объекта, когда требуется резервное копирование, значительно превосходит ущерб от бесполезного создания дублирующего объекта. Отсутствие резервных копий там, где они необходимы, может привести к коварным ошибкам и дырам в системе безопасности, создание же ненужных объектов всего лишь влияет на стиль и производительность программы.

Уничтожайте устаревшие ссылки (на объекты)

При переходе с языка программирования с ручным управлением памятью, такого как C или *C++,* на язык с автоматической очисткой памяти (garbage-collect - "сбор!<а мусора") ваша работа как программиста существенно упрощается благодаря тому обстоятельству, что ваши объекты автоматически утилизируются, как только вы перестаете их использовать. Когда вы впервые сталкиваетесь с этой особенностью, то воспринимаете ее как волшебство. Легко может создаться впечатление, что вам больше не нужно думать об управлении' памятью, но это не совсем так.

Рассмотрим следующую реализацию простого стека:

```
import java.util.*;
//Можете ли вы заметить "утечку памяти"?
public class Stack {
   private Object[] elements;
   private int size = 0;
   public Stack(int initialCapacity) {
      this.elements = new Object[initialCapacity];
   }
         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;
   }
    /**
        * Убедимся в том, что в стеке есть место хотя бы еще
        * для одного элемента. Каждый раз, когда
        * нужно увеличить массив, удваиваем его емкость.
    */
  private void ensureCapacity() {
      if (elements.length == size) {
         Object[] oldElements = elements;
         elements = new Object[2 * elements.length + 1];
         System.arraycopy(oldElements, 0, elements, 0, size);
      }    }

   public static void main(String[] args) {
      Stack s = new Stack(0);
      for (int i=0; i<args.length; i++)
         s.push(args[i]);
      for (int i=0; i<args.length; i++)
         System.out.println(s.pop());
```

```
}    }
```


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

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

Утечка памяти в языках с автоматической сборкой мусора (или точнее, *непреднамеренное сохранение объектов* - unintentional object retention) весьма 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 недоступными сборщику мусора, а это может оказать большое влияние на производительность программы.

Проблемы такого типа решаются очень просто: как только ссылки устаревают, их нужно обнулять. В случае с нашим классом Stack ссылка становится устаревшей, как только ее объект выталкивается 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
    returп result;
    }
```

Обнуление устаревших ссылок дает и другое преимущество: если впоследствии кто-то по ошибке попытается разыменовать какую-либо из этих ссылок, программа незамедлительно завершится с диагностикой NullPointerExceptioп вместо того, чтобы спокойно выполнять неправильную работу. Всегда выгодно обнаруживать ошибки fIрограммирования настолько быстро, насколько это возможно. Когда программисты впервые сталкиваются с подобной проблемой, они начинают перестраховываться, обнуляя все ссылки на объекты, лишь только программа заканчивает работу с ними. Это не нужно, поскольку загромождает программу и снижает ее

производительность. Обнуление ссылок на объект должно быть не нормой, а исключением. Лучший способ избавиться от устаревшей ссылки - вновь использовать переменную, в которой она находилась, либо выйти из области видимости переменной. Это происходит естественным образом, если для каждой переменной вы задаете самую ограниченную область видимости (статья 29). Следует, заметить, что в современных реализациях JVM недостаточно просто выйти из блока, в котором определена переменная. Чтобы переменная пропала, необходимо выйти из соответствующего метода-контейнера.

Так когда же следует обнулять ссылку? Какая особенность класса Stack сделала его восприимчивым к утечке памяти? Класс Stack *управляет своей памятью. Пул хранения* состоит из массива элементов (причем его ячейками являются ссылки на объекты, а не сами объекты). Как указывалось выше, элементы в активной части массива считаются *занятыми,* в остальной - *свободными.* Сборщик мусора этого знать никак не может, и для него все ссылки на объекты, хранящиеся в массиве, в равной' степени действительны. Только программисту известно, что неактивная часть массива не нужна. Сообщить об этом сборщику мусора программист может, лишь вручную обнуляя элементы массива по мере того, как они переходят в не активную часть массива.

Вообще говоря, если какой-либо класс начинает управлять своей памятью, программист должен подумать об утечке памяти. Как только элемент массива освобождается, любые ссылки на объекты, имевшиеся в этом элементе, необходимо обнулять.

Другим распространенным источником утечки памяти являются КЭШи. Поместив однажды в кэш ссылку на некий объект, легко можно забыть о том, что она там есть, и держать ссылку в КЭШе еще долгое время после того, как она стала недействительной. Возможны два решения этой проблемы. Если вам посчастливилось создать кэш, в котором запись остается значимой ровно до тех пор, пока за пределами КЭШа остаются ссылки на ее ключ, представьте этот кэш как WeakHashMap, а когда записи устареют, они будут удалены автоматически. В общем случае время, на протяжении которого запись в КЭШе остается значимой. четко не оговаривается. Записи 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 (например, через API java.util. Timer), либо это может быть побочным эффектом от добавления в кэш новых записей. При реализации второго подхода применяется метод removeEldestEntry из класса java.util.LinkedHashMap, включенного в версию 1.4.

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

Остерегайтесь методов finalize

Методы finalizeнепредсказуемы, часто опасны и, как правило, не нужны. Их использование может привести к странному поведению программы, низкой производительности и проблемам с переносимостью. Метод finalizeимеет лишь несколько областей применения (см. ниже), а главное правило таково: следует избегать методов finalize.

Программистов, пишущих на *C++,* следует предостеречь в том, что нельзя рассматривать методы finalizекак аналог деструкторов в *C++.* В *C++* деструктор _ это обычный способ утилизации ресурсов, связанных с объектом, обязательное дополнение к конструктору. В языке программирования java,когда объект становится недоступен, очистку связанной с ним памяти осуществляет сборщик мусора. Со стороны же программиста никаких специальных действий не требуется. В C++ деструкторы используются для освобождения не только памяти, но и других ресурсов системы. В языке программирования javaдля этого обычно применяется блок try-finally.

Нет гарантии, что метод finalizebудут вызван немедленно [JLS, 12.6]С момента, когда объект становится недосТупен, и до момента выполнения метода finalizeможет пройти сколь угодно длительное время. Это означает, что с **помощью** метода **finalize нельзя выполнять никаких операций, критичных по времени.** Например, будет серьезной ошибкой ставить процедуру закрытия открытых файлов в зависимость от метода finalize,поскольку дескрипторы открытых файлов - ресурс ограниченный. Если из-за того, что jVMмедлит с запуском методов finalize,открытыми будут оставаться много файлов, программа может завершиться с ошибкой, поскольку ей не удастся открыть новые файлы.

Частота запуска методов finalizев первую очередь определяется алгоритмом сборки мусора, который существенно меняется от одной реализации jVMк другой. Точно так же может меняться и поведение программы, работа которой зависит от частоты вызова методов finalize.Вполне возможно, что программа будет превосходно работать с jVM,на которой проводится 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 ряде случаев можно спровоцировать произвольную задержку при удалении его экземпляров. Один мой коллега недавно отлаживал приложение СИ, которое было рассчитано на длительное функционирование, но таинственно умирало с ошибкой OutOfMemoryErro.Анализ показал, что в момент смерти приложение в очереди на удаление стояли тысячи графических объектов, ждавших лишь вызова метода finalize и утилизации. К несчастью, поток утилизации выполнялся с меньшим приоритетом, чем другой поток того же приложения, а потому удаление объектов не могло осуществиться в том же темпе, в каком они становились доступны для удаления. Спецификация языка Java не определяет, в каком из потоков

будут выполняться методы finalize. Поэтому нет иного универсального способа предотвратить проблемы такого рода, кроме как воздерживаться от использования методов finalize.

Спецификация языка Java не только не дает поручительства, что методы finalize будут вызваны быстро, она не гарантирует, что они вообще будут вызваны. Вполне возможно, что программа завершится, так и не вызвав метод finalize для некоторых объектов, ставших недоступными. Следовательно, обновление критического фиксируемого (persistent) состояния не должно зависеть от метода finalize. Например, ставить освобождение фиксируемой блокировки разделяемого ресурса, такого как база данных, в зависимость от метода finalize - верный способ привести всю вашу распределенную систему к сокрушительному краху.

Не соблазняйтесь методами System.gc и System.runFinalization. Они могут повысить вероятность запуска утилизации, но не гарантируют этого. Единственные методы, требующие гарантированного удаления,- это System.runFinalizersOnExit и его вредный близнец Runtime.runFinalizersOnExit. Эти методы некорректны и признаны устаревшими.

Стоит обратить внимание еще на один момент: если в ходе утилизации возникает необработанная исключительная ситуация (exception), она игнорируется, а утилизация этого объекта прекращается [JLS, 12.6].Необработанная исключительная ситуация может оставить объект в испорченном состоянии. И если другой поток попытается воспользоваться испорченным объектом, результат в определенной мере может быть непредсказуем. Обычно необработанная исключительная ситуация завершает поток и выдает распечатку стека, однако в методе finalize этого не происходит; он даже не выводит предупреждений.

Так чем же заменить метод finalize для класса, чьи объекты инкапсулируют ресурсы, требующие завершения, такие как файлы или потоки? Создайте метод для прямого завершения и потребуйте, чтобы клиенты класса вызывали этот метод для каждого экземпляра, ,когда он им больше не нужен. Стоит упомянуть об одной детали: экземпляр сам должен следить за тем, был ли он завершен. Метод прямо го завершения должен делать запись в некое м закрытом поле о том, что объект более не является действительным. Остальные методы класса должны проверять это поле и инициировать исключительную ситуацию IllegalStateException, если их вызывают после того, как данный объект был завершен.

Типичный пример метода прямого завершения - метод close в InputStream и OutputStream. Еще один пример - метод cancel из jaya.util.Timer, который нужным образом меняет состояние объекта, заставляя поток (thread), связанный с экземпляром Timer, аккуратно завершить свою работу. Среди примеров из пакета jaya.awt - Graphics.dispose и Window.dispose. На эти методы редко обращают внимание, что сказывается на производительности программы. То же самое касается метода Image. flush, который освобождает все ресурсы, связанные с экземпляром Image, но оставляет последний в таком состоянии, что его еще можно использовать, выделив вновь необходимые ресурсы.

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

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

  }finally {
    foo.terminate(); // Метод прямого Завершения
    }
```

Зачем же тогда вообще нужны методы finalize? У них есть два приемлемых применения. Первое - они выступают в роли "страховочной сетки" в том случае, если владелец объекта забывает вызвать метод прямо го завершения. Нет гарантии, что метод finalize будет вызван своевременно, однако в тех случаях (будем надеяться, редких), когда клиент не выполняет свою часть соглашения, т. е. не вызывает метод прямого завершения, критический ресурс лучше 'освободить поздно, чем никогда. Три класса, представленных как пример использования метода прямого завершения (InputStream, OutputStream и Titer), тоже имеют методы finalize, которые применяется в качестве страховочной сетки, если соответствующие методы завершения не были вызваны.

Другое приемлемое применение методов finalize связано с объектами, имеющими *"местных партнеров"* (native peers). Местный партнер - это местный объект (native object), к которому обычный объект обращается через машинно-зависимые методы. Поскольку местный партнер не является обычным объектом, сборщик мусора о нем не знает, и когда утилизируется обычный партнер, утилизировать местного партнера он не может. Метод finalize является приемлемым посредником для решения этой задачи при условии, что *местный партнер* не *содержит критических ресурсов.* Если же местный партнер содержит ресурсы, которые необходимо освободить немедленно, данный класс должен иметь метод прямого завершения. Этот метод завершения обязан делать все, что необходимо для освобождения соответствующего критического ресурса. Метод завершения может быть машинно-зависимым методом либо вызывать таковой.

Важно отметить, что здесь нет автоматического связывания методов finalize ("finalizer chaining"). Если в классе (за исключениемОЬject) есть метод finalize, но в подклассе он был переопределен, то метод finalize в подклассе должен вызывать. Метод finalize, из суперкласса. Вы должны завершить подкласс в блоке try, а затем в Соответствующем блоке finally вызвать метод finalize суперкласса. Тем самым

гарантируется, что метод finalize суперкласса будет вызван, даже если при завершении подкласса инициируется исключительная ситуация, и наоборот:

```
// Ручное связывание метода finalize
protected void finalize() throws Throwable {
        // Ликвидируем состояние подкласса

    finally {

    super.f1nal1ze();

    }

 }
```

Если разработчик подкласса переопределяет метод finalize суперкласса, но забывает вызвать его "вручную" (или не делает этого из вредности), метод finalize суперкласса так и не будет вызван. Защититься от такого беспечного или вредного подкласса можно ценой создания некоего дополнительного объекта для каждого объекта, подлежащего утилизации. Вместо того чтобы размещать метод finalize в классе, требующем утилизации, поместите его в анонимный класс (статья 18), единственным назначением которого будет утилизация соответствующего экземпляра. для каждого экземпляра контролируемого класса создается единственный экземпляр анонимного класса, называемый *хранителем утилизации* (fjnalizer guardian). Контролируемый экземпляр содержит в закрытом экземпляре поля единственную в системе ссылку на хранителя утилизации. Таким образом, хранитель утилизации становится доступен для удаления в момент утилизации контролируемого им экземпляра. Когда хранитель утилизируется, он выполняет процедуры, необходимые для ликвидации контролируемого им экземпляра, как если бы его метод finalize был методом контролируемого класса:

```
// Идиома хранителя утилизации finalizer Guardian)
public class Foo {
    // Единственная задача этого объекта - утилизировать
    // внешний объект Foo
    private final Object finalizerGuardian = new Object() protected void finalize() throws
        Throwable
            // Утилизирует внешний объект Foo


    } ;
        // Остальное опущено
}
```

Заметим, что у открытого класса Foo нет метода finalize (за исключением тривиального, унаследованного от класса Object), а потому не важно, был ли в методе f1nalize подкласса вызов метода super.finalize или нет. Возможность использования этой

методики следует рассмотреть для каждого открытого расширяемого класса, имеющего метод finalize.

Подведем итоги. Не применяйте методы finalize, кроме как в качестве страховочной сетки или для освобождения некритических местных ресурсов. В тех редких случаях, когда вы должны использовать метод finalize, не забывайте делать вызов super.finalize. И последнее: если вам необходимо связать метод finalize с открытым классом без модификатора final, подумайте о применении 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 объект

Хотя класс Object может иметь экземпляры, прежде всего он предназначен для расширения. Поскольку все его методы без модификатора fina! - equals, hashCode, toString, clone и finalize - служат для переопределения, для них есть *общие* со*лашения* (genera! contracts). Любой класс, в котором эти методы переопределяются, обязан подчиняться соответствующим соглашениям. В противном случае он будет препятствовать правильному функционированию других взаимодействующих с ним классов, работа которых зависит от выполнения указанных соглашений.

В этой главе рассказывается о том, как и когда следует переопределять методы класса Object, не имеющие модификатора fina!. Метод finalize в этой главе не рассматривается, речь о нем шла в статье 6. В этой главе обсуждается также метод Comparable.compareTo, который не принадлежит классу Object, однако имеет схожие свойства.

Переопределяя метод equals, соблюдайте общие соглашения

Переопределение метода equals кажется простой операцией, однако есть множество способов неправильного ее выполнения, 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 внутренне уникален. Это утверждение справедливо для таких классов как Thread, которые представляют не величины, а активные сущности. Реализации метода equals, предлагаемая классом Object, для этих классов работает совершенно правильно.

- Вас не интересует, предусмотрена ли в классе проверка "логического равенства". Например, в классе java.util.Random можно было бы переопределить метод equals с тем, чтобы проверять, будут ли два экземпляра Random генерировать одну и ту же последовательность случайных чисел, однако разработчики посчитали, что клиенты не должны знать о такой возможности и она им не понадобится. В таком случае тот вариант метода equals, который наследуется от класса Object, вполне приемлем.

- Метод equals уже переопределен в суперклассе, и функционал, унаследованный от суперкласса, вполне приемлем дли данного класса. Например, большинство реализаций интерфейса Set наследует реализацию метода equals от Класса AbstractSet, List наследует реализацию от AbsctractList, а Map - от AbstractMap.

- Класс является закрытым или доступен только в пределах пакета, и вы уверены, что его метод eQuals никогда не будет вызван. Сомнительно, что в такой ситуации метод eQuals следует переопределять, разве что на тот случай, если его однажды случайно вызовут:

```
Public Boolean equals (Object o) {
    Throw new UnsupportedOperationException ();
    }
```

Так когда же имеет смысл переопределять Object. equals? Тогда, когда для класса определено понятие логической эквивалентности (Jogica! equality), которая не совпадает с тождественностью объектов, а метод equals в суперклассе не был переопределен с тем, чтобы реализовать требуемый функционал. Обычно это случается с классами значении, такими как Integer или Date. Программист, сравнивающий ссылки на объекты значений с помощью метода equals, желает, скорее всего, 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 некоей схеме или элементов в некоем наборе, Имеющих необходимое и предсказуемое поведение.

Существует один вид классов значений, которым не нужно переопределение метода equals,- перечисление типов (статья 21). Поскольку для классов этого типа гарантируется, что каждому значению соответствует не больше одного объекта, метод equals из Object для этих классов будет равнозначен методу логического сравнения.

Переопределяя метод equals, вы должны твердо придерживаться принятых для него общих соглашений. Воспроизведем эти соглашения по тексту спецификации java,lang.Object:

Метод equals реализует *отношение эквивалентности:*

- *Рефлективность,* для любой ссылки на значение x выражение x.equals(x) должно возвращать true.
- *Симметричность,* для любых ссылок на значения x и y выражение x. equals(y) должно возвращать t rue тогда и только тогда, когда y.equals(x) возвращает true.
- *Транзитивность,* для любых ссылок на значения x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, то и выражение x. equals(z) должно возвращать true.
- *Непротиворечивость.* Для любых ссылок на значения x и y, если несколько раз вызвать x. equals(y), постоянно будет возвращаться значение true либо постоянно будет возвращаться значение false при условии, что никакая информация, используемая при сравнении объектов, не поменялась.
- Для любой ненулевой ссылки на значение x выражение x. equals(null) должно возвращать false.

Если у вас нет склонности к математике, все это может показаться ужасным, однако игнорировать это нельзя! Если вы нарушите условия, то рискуете получить программу, которая работает неустойчиво или заканчивается с ошибкой, а установить источник ошибок крайне сложно. Перефразируя Джона Донна (John Doппe), можно сказать: ни один класс - не остров. ("Нет человека, что был бы сам по себе, как остров ... " - Джон Донн, "Взывая на краю".- *Прим. пер.)* Экземпляры одного класса часто передаются другому классу. Работа многих классов, в том числе всех классов коллекции, зависит от того, соблюдают ли передаваемые им объекты соглашения для метода equals.

Теперь рассмотрим внимательнее соглашения для метода equals. На самом деле они не так уж сложны. Как только вы их поймете, придерживаться их будет совсем не Трудно.

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

Симметрия. Второе требование гласит, что любые два объекта должны сходиться во мнении, равны ли они между собой. В отличие от предыдущего, представить

непреднамеренное нарушение этого требования несложно. Например, рассмотрим следующий класс:

```
/**
* Строка без учета регистра. Регистр исходной строки сохраняется
* методом toString, однако, при сравнениях игнорируется.
*/

public final class CaseInsensitiveString {
   private String s;

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

// Ошибка нарушение симметрии!
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;
   }

         /* Одностороннее взаимодействие


   // Fixed
   public boolean equals(Object o) {
      return o instanceof CaseInsensitiveString &&
         ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
   }

*/

   // ...  // Остальное опущено


  static void main(String[] args) {
      CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
      String s = "polish";

      System.out.println(cis.equals(s));
      System.out.println(s.equals(cis));
   }
}
```

Исходя 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.

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

Как и предполагалось, выражение cis,equals(s) возвращает true. Проблема заключается в том, что хотя метод equals в классе CaseInsensitiveString знает о существовании обычных строк, метод equals в классе String не догадывается о строках, нечувствительных к регистру. Поэтому выражение s.equals(cis) возвращает false, явно нарушая симметрию. Предположим, вы помещаете в коллекцию строку, нечувствительную к регистру:

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


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

Для устранения этой проблемы удалите из метода equals попытку взаимодействия с классом String. Сделав это, вы сможете перестроить метод так, чтобы он содержал один оператор возврата:

```
Public Boolean equals (Object o) {
    Return o instanceof CaseInsensitiveString&&
    ((CaseInsensitiveString) o), s.equalsIgnoreCase(s);
```

Транзитивность. Третье требование в соглашениях для метода equals гласит: 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. Допустим, что программист создает подкласс, придающий своему суперклассу новый *аспект* иными словами, подкласс привносит некую информацию, окаэыва10~УЮ влияние на процедуру сравнения. Начнем с простого неизменяемого класса, соответствующего точке в двухмерном пространстве:

```
public class Point {

    private final int x;

    private final int y;

    public Point(int x, int y) {

    this.x = x;

    this.y = y;

}

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 elass ColorPoint  extends Point
```

```
private 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 при сравнении с помощью методов equals информация о цвете будет игнорироваться. Хотя такое решение и не нарушает общих соглашений для метода equals, очевидно, что оно неприемлемо. Допустим, вы пишите метод equals, который возвращает значение t rue, только если его аргументом является цветная точка, имеющая то же положение и тот же цвет:

```
// Ошибка нарушение симметрии
publie boolean equals (Objeet o) {
if (1(0 instaneeof ColorPoint))
return false;
      ColorPoint cp = (ColorPoint) o;
      return super.equals(o) &&cp.color == color;
      }
```

Проблема этого метода заключается в том, ~то 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 пояснения создадим одну обычную точку и одну цветную:

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

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!
public boolean equals (Object o) {
      If (!(o instanceof Point))
            return false;
      // Если о - обычный Point, выполнить сравнение
      // без проверки цвета
      if (! (o instanceof ColorPoint))
      return o.equals(this);
      // Если о - ColorPoint, выполнить полное сравнение
      ColorPoint ep = (ColorPoint) o;
return super.equals(o) && cp.color == color;
}
```

Такой подход обеспечивает симметрию, но за счет транзитивности:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);

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

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

Так где же решение? Оказывается, это фундаментальная проблема эквивалентных отношений в объектно-ориентированных языках. Не существует способа расширить класс, порождающий экземпляры, и добавить к нему новый аспект, сохранив при этом соглашения для метода equals. Зато есть изящный обходной путь. Следуйте совету из статьи 14: "Предпочитайте композиции наследование". Вместо того чтобы заставлять ColorPoint расширять класс Point, поместите в ColorPoint закрытое поле Point и открытый метод представления (статья 4), который в том же месте, где находится цветная точка, показывает обычную точку:

```
// Добавляет новый аспект, не нарушая соглашений для equals
public class ColorPoint {
      private Point point;
      private Color color;

      public ColorPoint(int x, int y, Color color) {

      point = new Point(x, y);
            this.color = color;


      /**
       * Для данной цветной точки возвращает точку-представление
       */
      public Point asPoint()
            return point;


      public boolean equals(Object o) {


      if (!(o instanceof ColorPoint» return false;
            ColorPoint cp = (ColorPoint)o;
            return cp.point.equals(point) &&cp.color.equals(color);


           }


           // Остальное опущено

}
```

В библиотеках для платформы Java содержатся классы, которые являются подклассами для класса, создающего экземпляры, и при этом придают ему новый аспект. Например, java.sql.Timestamp является подклассом класса java.util,Date и добавляет поле для наносекунд. Реализация метода equals в Timestamp нарушает правило симметрии, и это может привести к странному поведению программы, если объекты Timestamp и Date использовать в одной коллекции или смешивать 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, но если вы сделаете это, устранение возникших в результате ошибок может быть непростым. Класс Timestamp не является правильным, и подражать ему не надо.

Заметим, что вы *можете* добавить аспект в подкласс *абстрактного* класса, не нарушая при этом соглашений для метода equals. Это важно для тех разновидностей иерархии классов, которые вы получите, следуя совету из статьи 20: "Заменяйте объединение иерархией классов". Например, вы можете иметь простой абстрактный класс Shape, а также подклассы Ci,rcle, добавляющий поле радиуса, и Rectangle, добавляющий поля *длины* и *ширины.* Только что продемонстрированные проблемы не будут возникать до тех пор, пока нет возможности создавать экземпляры суперкласса.

Непротиворечивость. Четвертое требование в соглашениях для метода equals гласит: если два объекта равны, они должны быть равны все время, пока один из них (или оба) не будет изменен. Это не столько настоящее требование, сколько напоминание о том, что изменяемые объекты в разное время могут быть равны разным объектам, а неизменяемые объекты - не могут. Когда вы пишите класс, подумайте, не следует ли его сделать неизменяемым (статья 13). 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.

Отличие от null (non-nullity). Последнее требование гласит, что все объекты должны отличаться от нуля (null). Трудно себе представить, что в ответ на вызов o.equals(null) будет случайно возвращено значение true, однако вполне вероятно случайное инициирование исключительной ситуации NullPointerException. Общие соглашения этого не допускают. Во многих классах методы equals имеют защиту в виде явной про верки аргумента на null:

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

Такая проверка не является обязательной. Проверяя 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 для проверки того, что аргумент имеет правильный тип:

```
public boolean equals(Object o){
 if (I(o instanceof MyType))
 return false;
}
```

Если бы эта проверка типа отсутствовала, а метод equals получил бы аргумент неправильного типа, то он бы инициировал исключительную ситуацию ClassCastException, что нарушает соглашения для метода equals. Однако здесь есть оператор instanseof, и если его первый операнд равен null, то независимо от типа второго операнда он возвратит false (JLS,15.19.2]. Поэтому при передаче null проверка типа вернет false, и, следовательно, нет необходимости делать отдельную проверку для null: Собрав все это вместе, получаем рецепт для создания высококачественного метода equals:

1. Используйте оператор == для проверки, является ли аргумент ссылкой на указанный объект, Если является, возвращайте true . Это всего лишь способ повьшгения производительности программы, которая будет низкой, если процедура сравнения оказывается трудоемкой.
2. Используйте оператор instanceof для проверки, имеет ли аргумент правильный тип. Если не имеет, возвращайте false. Обычно
прав ильный тип - это тип того класса, которому принадлежит данный метод. В некоторых случаях это может быть какой-либо интерфейс, реализуемый данным классом. Если класс реализует интерфейс, который уточняет соглашения для метода equals, то в качестве типа указывайте этот интерфейс, что позволит выполнять сравнение классов, реализующих интерфейс. Подобным свойством обладают интерфейсы коллекций Set, List, Map и Map. Entry.
3. Приводите аргумент к правильному типу. Поскольку эта операция следует за проверкой i.nstanceof, она гарантированно будет выполнена.
4. Пройдитесь по всем "значимым" полям класса и убедитесь в том, что значение 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 - false.
Если на шаге 2 тип был определен как интерфейс, вы должны получать доступ к значимым полям аргумента, используя методы самого интерфейса. Если же тип аргумента определен как класс, то в зависимости от условий вам, возможно, удастся получить прямой доступ к полям аргумента. Для простых полей, за исключением типов float и double, для сравнения применяйте оператор ==. Для полей со ссылкой на объекты рекурсивно

вызывайте метод equals. Для поля float преобразуйте его значение в int с помощью метода Float. floatToIntBi ts,
а затем сравнивайте полученные значения, используя оператор -- для полей double преобразуйте их значения в long с помощью метода Double. doubleToLongBi ts, а затем сравнивайте полученные значения long, используя оператор ==.(Особая процедура обработки полей float и double нужна потому, что существуют особые значения Float. NaN, -0.Of, а также аналогичные значения для типа double. См. документацию по Float. equals.) При работе с полями массивов применяйте перечисленные правила для каждого элемента отдельно.
Некоторые поля, предназначенные для ссылки на объекты, вполне оправданно могут иметь значение null. Чтобы не допустить возникновения исключительной ситуации NullPointerException,
для сравнения подобных полей используйте следующую идиому:

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

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

(field == o. field 11 (field != null && field. equals(o, field)))

для некоторых классов, например для представленного выше CaseInsensitiveString, сравнение полей оказывается гораздо сложнее, чем простая проверка равенства. Так ли это, должно быть понятно из спецификации на соответствующий класс. Если так, то, возможно, потребуется придать каждому объекту некую *каноническую форму.*
В результате метод equals сможет выполнять' простое и точное сравнение канонических форм вместо того, чтобы пользоваться более трудоемким и неточным вариантом сравнения. Описанный прием более подходит для *неизменяемых* классов (статья 13), поскольку, когда объект меняется, приходится приводить его каноническую форму в соответствие последним изменениям.

На производительность метода equals может оказывать влияние очередность сравнения полей. для достижения наилучшей производительности нужно в первую очередь сравнивать те поля, которые будут различаться с большей вероятностью, либо те, которые сравнивать проще. В идеале оба эти качества должны совпадать.
Не следует сравнивать поля, не являющиеся частью логического состояния объекта, например поля Object, используемые для синхронизации операций. Нет необходимости сравнивать *избыточные поля,* значение которых можно вычислить, основываясь на "значащих полях" объекта, однако сравнение этих полей может, повысить производительность метода equals. Если значение

избыточного поля равнозначно суммарному описанию объекта в целом, то сравнение подобных полей позволит сэкономить на сравнении действительных данных, если будет выявлено расхождение.

5. Закончив написание собственного метода equals, задайте себе вопрос: является ли он симметричным, транзитивным и непротиворечивым? (Оставшиеся два свойства обычно получаются сами собой.) Если ответ отрицательный, разберитесь, почему не удалось реализовать эти свойства, и подправьте метод соответствующим образом.

В качестве конкретного примера метода equals, который был выстроен по приведенному выше рецепту, можно посмотреть PhoneNumber.equals из статьи 8. Несколько заключительных предостережений:

- Переопределяя метод equals, всегда переопределяйте метод hashCode (статья 8).
- Не старайтесь быть слишком умным. Если вы проверяете лишь равенство полей, соблюдать условия соглашений для метода equals совсем не трудно. Если же в поисках равенства вы излишне агрессивны, можно легко нарваться на неприятности. Так, использование синонимов в каком бы то ни было обличии обычно оказывается плохим решением. Например, класс File не должен пытаться считать равными символьные связи (в системе UNIX), относящиеся к одному и тому же файлу. К счастью, он этого и не делает.
- Не надо писать метод equals, использующий ненадежные ресурсы. Если вы делаете это, то соблюсти требование непротиворечивости будет крайне трудно. Например, метод equals в классе java.net.URL использует IP-адреса хостов, соответствующих сравниваемым адресам URL. Процедура преобразования имени хоста в IP-адрес может потребовать выхода в компьютерную сеть, и нет гарантии, что это всегда будет давать один и тот же результат. Это может привести к нарушению соглашений для метода equals, сравнивающего адреса URL, и на практике уже, создавало проблемы. (К сожалению, описанную схему сравнения уже нельзя поменять из-за требований обратной совместимости.) За некоторыми исключениями, 'методы equals обязаны выполнять детерминированные операции с объектами, находящимися в памяти.
- Декларируя метод equals, не нужно указывать вместо Object другие типы объектов. Нередко программисты пишут метод equals следующим образом, а потом часами ломают голову над тем, почему он не работает правильно:

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

Проблема заключается в том, что этот метод не переопределяет (override) метод Object. equals, чей аргумент имеет тип Object, а перегружает его (overload) (статья 26). Подобный "строго типизированный" метод equals можно создать в дополнение к обычному методу equals, однако поскольку оба метода возвращают один и тот результат, нет никакой причины делать это. При определенных условиях это может дать минимальный выигрыш в производительности, но не оправдывает дополнительного усложнения программы (статья 37).

Переопределяя метод equals, всегда переопределяйте hashCod

Распространенным источником ошибок является отсутствие переопределения метода hashCode. Вы должны переопределять метод hashCode в каждом классе, где переопределен метод equals. Невыполнение этого условия приведет к нарушению общих соглашений для метода Object.hashCode, а это не позволит вашему классу правильно работать в сочетании с любыми коллекциями, построенными на использовании хэш-таблиц, в том числе с HashMap, HashSet и HashTable.

Приведем текст соглашений, представленных в спецификации jaya lang.Object:

- 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 не равны друг другу, 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 является второе условие: равные объекты должны иметь одиноко - вый хаш·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 имеющих между собой ничего общего. Поэтому метод hashCode скорее всего возвратит для этих объектов два случайных числа. а не одинаковых, как того требует соглашение.

В качестве примера рассмотрим следующий упрощенный класс PhoneNumber, в котором метод equals построен по рецепту из статьи 7:

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

Предположим, что вы попытались использовать этот класс с HashMap:

```
Map m = new HashMap();
m. put (new PhbneNumber( 408, 867, 5309), "Jenny");
```

Вы вправе ожидать, что m.get(new PhoneNumber(408, 867, 5309)) возвратит строку "Jenny", однако он выдает null. Заметим, что здесь задействованы два экземпляра класса PhoneNumber: один используется для вставки в таблицу HashMap, а другой,

равный ему экземпляр,- для поиска. Отсутствие в классе PhoneNumber переопределенного метода hashCode приводит к тому, что двум равным экземплярам соответствует разный хэш-код, т. е. имеем нарушение соглашений для этого метода. Как следствие, метод get ищет указанный телефонный номер в другом сегменте хэш-таблицы, а не там, где была сделана запись с помощью метода put. Разрешить эту проблему можно, поместив в класс PhoneNumber правильный метод hashCode.

Как же должен выглядеть метод hashCode? Написать действующий, но не слишком хороший метод нетрудно. Например, следующий метод всегда приемлем, но пользоваться им не надо никогда:

```
// Самая плохая из допуотимых хэш-функций – никогда
// непользуйтеоно!
public int hashCode() { return 42; }
```

Данный метод приемлем, поскольку для равных объектов он гарантирует возврат одного и того же хэш-кода. Плохо то, что он гарантирует получение одного и того же хэш-кода для *любого* объекта. Соответственно, любой объект будет привязан к одному и тому же сегменту хэш-таблиицы, а сами хэш-таблиицы вырождаются в связные списки, для программ, время работы которых с ростом хэш-таблии должно увеличиваться линейно, имеет место квадратичная зависимость. Для больших хэш-таблии это равносильно переходу от работоспособного к неработоспособному варианту.

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

1. Присвойте переменной result (тип int) некоторое ненулевое число, скажем, 17.
2. Для каждого значимого поля f в вашем объекте (т. е. поля, значение которого принимается в расчет методом equals), выполните следующее:
   a. Вычислите для поля хэш-код c (тип int):
      1. Если поле имеет тип boolean, вычислите (f ? O : 1).
      2. Если поле имеет тип byte, char, short или int, вычислите (int)f.
      3. Если поле имеет тип long, вычислите (int)(f - (f >>>32)).
      4. Если поле имеет тип float, вычислите Float. floatToIntBits(f).
      5. Если - тип double, вычислите Double. doubleToLongBits(f), а затем преобразуйте полученное значение, как указано в п. 2.a.3.

6. 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 для него метод hashCode. Если значение поля равно null, возвращайте 0 (можно любую другую константу, но традиционно используется 0).

7. 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 объедините полученные значения так, как описано в п. 2.Ь.

b. Объедините хэш-код с, вычисленный на этапе а, с текущим значением поля resul t следующим образом:

```
result = 37*result + c;
```

3. Верните значение resul t.

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

Из процедуры получения хэш-кода можно исключить *избыточные поля.* Иными словами, можно исключить любое поле, чье значение можно вычислить, исходя из значений полей, задействованных в рассматриваемой процедуре. Вы *обязаны* исключать из процедуры все поля, которые не используются в ходе проверки равенства. Иначе может быть нарушено второе правило в соглашениях для hashCode.

На этапе 1 используется ненулевое начальное значение. Благодаря этому не будут игнорироваться те обрабатываемые в первую очередь поля, у которых значение хэш-кода, полученное на этапе 2.а, оказалось нулевым. Если же на этапе 1 в качестве начального значения использовать нуль, то ни одно из этих обрабатываемых в первую очередь полей не сможет повлиять на общее значение хэш-кода, что способно привести к увеличению числа коллизий. Число 17 выбрано произвольно.

Умножение на шаге 2.Ь создает зависимость значения хэш-кода от очередности обработки полей, а это обеспечивает гораздо лучшую хэш-функцию в случае, когда в классе много одинаковых полей. Например, если из хэш-функции для класса String, построенной по этому рецепту, исключить умножение, то все анаграммы (слова, полученные от некоего исходного слова путем перестановки букв) будут иметь один и тот же хэш-код. Множитель 37 выбран потому, что является простым нечетным числом. Если бы это было четное число и при умножении произошло переполнение,

информация была бы потеряна, поскольку умножение числа на 2 равнозначно его , арифметическому сдвигу. Хотя преимущества от применения простых чисел не столь очевидны, именно их принято использовать для этой цели.

Используем описанный рецепт для класса PhoneNumber. В нем есть три значимых поля, все имеют тип short. Прямое применение рецепта дает следующую хэш-функцию

```
public int hashCode() {
int result = 17;
      result = 37*result + areaCode;
      result = 37*result + exchange;
      result = 37*result + extension;                    \
     return result;
     }
```

Поскольку этот метод возвращает результат простого детерминированного вычисления, исходными данными для которого являются три значащих поля в экземпляре PhoneNumber, очевидно, что равные экземпляры PhoneNumber будут иметь равный хэш-код. Фактически этот метод является абсолютно правильной реализацией hashCode для класса PhoneNumber наряду с методами из библиотек Java версии 1.4. Он прост, довольно быстр и правильно разносит неравные телефонные номера по разным сегментам хэш-таблицы.

Если класс является неизменным и при этом важны затраты на вычисление хэш-кода, вы можете сохранять хэш-код в самом 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, вам следует вычислять соответствующий хэш-код уже в момент создания соответствующего экземпляра. С другой стороны, вы можете выбрать *инициализацию , отложенную* до первого обращения к методу hashCode (статья 48). Хотя достоинства подобного режима для нашего класса PhoneNumbers не очевидны, покажем, как это делается:

```
// Отложенная инициализация, кэшируемый hashCode
private volatile int hashCode = 0;          //(см. статью 48)

public int hashCode() {
     if (hashCode == o) {
     int result = 17;
          result = 37*result + areaCode;
          result = 37*result + exchange;
          result = 37*result + extension;
          hashCode = result;
     }
     return hashCode;
     }
```

Рецепт, изложенный в этой статье, позволяет создавать довольно хорошие хэш-функции, однако он не соответствует ни современным хэш-функциям, ни хэш-функциям из библиотек для платформы Java, которые реализованы в версии 1.4. Разработка подобных хэш-функций является предметом активных исследований, которые лучше оставить математикам и ученым, работающим в области теории вычислительных машин. Возможно, в последней версии платформы Java для библиотечных классов будут представлены современные хэш-функции, а также методы-утилиты, которые позволят рядовым программистам самостоятельно создавать такие хэш-функции. Пока же описанные в этой статье приемы применяются для большинства приложений.

Повышение производительности не стоит того, чтобы при вычислении хэш-кода игнорировать значимые части объекта. Хотя хэш-функция способна работать быстрее, ее качество может ухудшиться до такой степени, что обработка хэш-таБлицы будет производиться слишком медленно. В частности, не исключено, что фэш-функция столкнется с большим количеством экземпляров, которые существенно разнятся как раз в тех частях, которые вы решили игнорировать. В этом случае хэш-функция сопоставит всем этим экземплярам всего лишь несколько значений хэш-кода. Соответственно, коллекция, основанная на хэш-функциях, будет вызывать падение производительности в квадратичной зависимости от числа элементов. Это не просто теоретическая проблема. Хэш-функция класса String, реализованная во всех версиях платформы Java до номера 1,2, проверялась самое большее для строк с 16 символами и равномерным распределением пробелов по всей строке, начиная с первого символа. для больших коллекций иерархических имен, таких как URL, эта хэшфую(ция демонстрировала то патологическое поведение, о котором здесь говорилось.

У многих классов в библиотеках для платформы Java, таких как Str1ng, Integer и Date, конкретное значение, возвращаемое методом hashCode, определяется как функция от значения экземпляра. Вообще говоря, это не слишком хорошая идея, поскольку она серьезно ограничивает возможности по улучшению хэш-функций в будущих версиях. Если бы вы оставили детали реализации фэш-йункции не конкретизированными и в ней обнаружился бы изъян, вы бы могли исправить эту хэш-функцию в следующей версии, не опасаясь утратить совместимость с теми клиентами, работа которых зависит от того, какое конкретное значение возвращает хэш-функция.

Всегда переопределяйте метод toStrIn

В классе java.lang.Object предусмотрена реализация метода toString, однако возвращаемая им строка, как правило, совсем не та, которую желает видеть пользователь вашего класса. Она состоит из названия класса, за которым следуют символ "коммерческого at" (@) и его хэш-код в виде беззнакового шестнадцатеричного числа, например "PhoneNumber@163Ь91". Общее соглашение для метода toString: возвращаемая строка должна быть "лаконичным, но информативным, легко читаемым

представлением объекта". Хотя можно поспорить, является ли лаконичной и легко читаемой строка "PhoneNumber@163b91", она не столь информативна, как, например, такая ,строка: "(408) 867 -5309'~. Далее в соглашении для метода toSt ring говорится:
"Рекомендуется во всех подклассах переопределять этот метод". Хороший совет, ничего не скажешь.

Эти соглашения не столь строги, как соглашения для методов equals и hashCode (статьи 7 и 8), однако, качественно реализовав метод toString, вы сделаете свой класс более приятным в использовании. Метод toSt ring вызывается автоматически, когда ваш объект передается методу println, оператору сцепления строк (+) или assert (в версии 1.4). Если вы создали хороший метод toString, 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, но сообщения не станут понятней, если не сделать этого. Преимущества от реализации удачного метода toString передаются не только экземплярам этого класса, но и объектам, которые содержат ссылки на эти экземпляры, особенно это касается коллекций. Что бы вы хотели увидеть: "{Jenny=PhoneNumber@163b91}" или же "{Jenny=(408) 867-5309}"?

Будучи переопределен, метод toString должен передавать *всю* полезную информацию, которая содержится в объекте, как это было показано в примере с телефонными номерами. Однако это не годится для больших объектов или объектов, состояние которых трудно представить в виде строки. В подобных случаях метод toString возвращает такое резюме, как "Manhattan white pages (1487536 listings)" или "Thread [main, 5, main]". В идеале полученная строка не должна требовать разъяснений. (Последний пример с Th read не отвечает этому требованию.)

При реализации метода toString вы должны принять одно важное решение: будете ли вы описывать в документации формат возвращаемого значения. Это желательно делать для *классов-значении* (value class), таких как телефонные номера и таблицы. Задав определенный формат, вы получите то преимущество, что он будет стандартным, однозначным и удобным для чтения представлением соответствующего объекта. Это представление можно использовать для ввода, вывода, а также для создания удобных для прочтения записей в фиксируемых объектах, например в документах XML. При задании определенного формата, как правило, полезно бывает создать соответствующий конструктор объектов типа String (или статический метод генерации, см. статью 1), что позволит программистам с легкостью осуществлять преобразование между объектом и его строковым представлением. Такой подход используется в 'библиотеках платформы Java для многих классов-значений, включая BigInteger, BigDecimal и большинство примитивных классов-оболочек (wrapper).

Неудобство от конкретизации формата значения, возвращаемого методом toString, заключается в том, что если ваш класс используется широко, то, задав формат, вы оказываетесь привязаны к нему навсегда. Другие программисты будут писать код, который анализирует данное представление, генерирует и использует его при записи

объектов в базу данных (persistent data). Если в будущих версиях вы поменяете формат представления, они будут очень недовольны, поскольку вы разрушите созданные ими код и данные. Отказавшись от спецификации формата, вы сохраняете возможность внесения в него новой информации н его совершенствования в последующих
версиях.

Будете вы объявлять формат или нет, вы должны четко обозначить ваши
намерения. Если вы описываете формат, то обязаны сделать это пунктуально. В качестве примера представим метод toString, который должен сопровождать класс PhoneNumber (статья 8):

```
/**
 * Возвращает представление данного телефонного номера в виде строки.
 * Строка состоит из четырнадцати символов, имеющих формат
 * "(XXX) YYY-ZZZZ" , где XXX - код зоны, УУУ -номер АТС,
 * ZZZZ - номер абонента в АТС. (Каждая прописная буква представляет * одну
 десятичную цифру.)
 *
 * Если какая-либо из трех частей телефонного номера мала и
 * не заполняет свое поле, последнее дополняется ведущими нулями. *
 Например, если значение номера абонента в АТС равно 123, то
 * последними четырьмя символами в строковом представлении будут "0123".
 *
 * Заметим, что закрывающую скобку, следующую за кодом зоны, и первую * цифру
 номера АТС разделяет один пробел.
 */
public String toString() {
    return "(" + toPaddedString(areaCode, 3) + ") " +
     toPaddedString(exchange, 3) + "-" +
    toPaddedString(extension, 4);
/**
 * Преобразует значение типа int в строку указанной длины, дополненную
 * ведущими нулями. Предполагается. что i >= 0 ,
 * 1 <= length <= 10, a Integer.toString(i) <= length.
 */
private static String toPaddedString(int i, int  length)
    String s = Integer.toString(i);
    return ZEROS[length - s.length()] + s;
    }
private static String[] ZEROS =
    { "", "0", "00", "000", "0000",
     "00000". "000000", "0000000". "00000000", "OOOOOOOOO"};
```

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

```
/**
 * Возвращает краткое описание зелья. Точные детали представления
 * не конкретизированы и могут меняться, однако следующее представление . *
может рассматриваться в качестве типичного:
 *
 * "[Зелье #9: тип=любовный, аромат=скипидар, вид=тушь]"
 */
public String toString () { ... }
```

Прочитав этот комментарий, программисты, разрабатывающие объекты, сохраняемые в базе данных, или программный код, который зависит от особенностей формата, уже не смогут винить никого, кроме самих себя, если формат однажды поменяется.

Вне зависимости от того, описываете вы формат или нет, всегда полезно предоставлять альтернативный программный доступ ко всей информации, которая содержится в значении, возвращаемом методом toString. Например, класс PhoneNumber должен включать в себя методы доступа к коду зоны, номеру А те и номеру абонента в А те. В противном случае программистам, которым нужна эта информация, *придется* делать разбор данной строки. Помимо того, что вы снижаете производительность приложения и заставляете программистов выполнять ненужную работу, это чревато ошибками и приводит к созданию ненадежной системы, которая перестает работать, как только вы меняете формат. Не предоставив альтернативных методов доступа, вы де-факто превращаете формат строки в элемент API, даже если в документации и указали, что он может быть изменен.

Соблюдайте осторожность при переопределении метода Clo

Интерфейс Cloneable проектировался в качестве *дополнительного интерфейса* (mixin) (статья 16), позволяющего объектам объявлять о том, что они могут быть клонированы. К сожалению, он не может использоваться для этой цели. Его основной недостаток - отсутствие метода clone; в самом же классе Object метод clone является закрытым. Вы не можете, не обращаясь к *механизму отражения свойств* (reflection) (статья 35), вызывать для объекта метод clone лишь на том основании, что он реализует интерфейс Cloneable. Даже отражение может завершиться неудачей, поскольку нет гарантии, что у данного объекта есть доступный метод clone. Несмотря на этот и другие. недочеты, данный механизм используется настолько широко, что Имеет смысл с ним разобраться. В этой статье рассказывается о том, каким образом создать хороший метод clone, обсуждается, когда имеет смысл это делать, а также кратко описываются альтернативные подходы.

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

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

Общие соглашения для метода clone довольно свободны. Они описаны в спецификации класса java.lang.Object:

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

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

и

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

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

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

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

Такое соглашение создает множество проблем. Условие "никакие конструкторы не вызываются" является слишком строгим. Правильно работающий метод clone может воспользоваться конструкторами для создания внутренних объектов клона. Если же класс является окончательным (final), метод clone может просто вернуть объект, СОЗДанный конструктором.

Условие, что x.clone().getClaSs() должно быть тождественно x.getClass(), является слишком слабым. Как правило, программисты полагают, что если они расширяют класс и вызывают в полученном подклассе метод super. clone, то получаемый в результате объект будет экземпляром этого подкласса. Реализовать такую схему суперкласс может *только одним способом* - вернуть объект, полученный в результате вызова метода super.clone. Если метод clone возвращает объект, созданный конструктором, это будет экземпляр другого класса. Поэтому, если в расширяемом

классе вы переопределяете метод clone, то возвращаемый объект вы должны получать вызовом super. clone. Если все суперклассы данного класса выполняют это условие, рекурсивный вызов метода supe r. clone в конечном счете приведет к вызову метода clone из класса Object и к созданию экземпляра именно того класса, который нужен. Этот механизм отдаленно напоминает автоматическое сцепление конструкторов, за исключением того, что оно не является принудительным.

В версии 1.3 интерфейс Cloneable не раскрывает, какие обязанности берет на себя класс, реализуя этот интерфейс. В спецификации не говорится ничего, помимо того, что реализация данного интерфейса влияет на реализацию метода clone в классе Object. На практике же требования сводятся к тому, что в классе, реализующем интерфейс Cloneable, должен быть представлен правильно работающий открытый метод clone. Вообще же, выполнить это условие невозможно, если только все суперклассы этого класса не будут иметь правильную реализацию метода clone, открытую или защищенную.

Предположим, что вы хотите реализовать интерфейс Cloneable с помощью класса, чьи суперклассы имеют правильно работающие методы clone. В зависимости от природы этого класса объект, который вы получите после вызова super.clone(), может быть, а может и не быть похож на тот, что вы будете иметь в 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 требуется. Такой вариант демонстрирует, например, класс PhoneNumbe r из статьи 8. Все, что здесь нужно,- это обеспечить в классе Object открытый доступ к защищенному методу clone:

```
public Object clone() {
try {
        return super,clone();
        catch(CloneNotSupportedException e) {
        throw new Error("Assertion failure");       // Этого не может быть
        }
   }
```

Однако если ваш объект содержит поля, имеющие ссылки на изменяемые объекты, такая реализация метода clone может иметь катастрофические последствия. Рассмотрим класс Stack' из статьи 5:

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    public Stack(int initialCapacity) {
    this.elements = new Object[initialCapacity];
    }
```

```
public void push(Object e) {

ensureCapacity();

elements[size ++] = e;

}

public Object pop() {

if (size == O)
        throw new EmptyStackException();
        Object result = elements[- size];
    elements[size] = null;        // Убираем устаревшую ссылку
     return result;
      }



// Убедимся в том, что в стеке есть место хотя бы
 // еще для одного элемента
private void ensureCapacity() {
     if (elements.length == size) {
     Object oldElements[] = elements;
          elements = new Object[2 * elements.length + 1];
          System.arraycopy(oldElements, 0, elements, O, size);
          }
     }
 }
```

Предположим, что вы хотите сделать этот класс клонируемым. Если его метод clone просто вернет результат вызова super.clone(), полученные 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 выдает бессмысленные результаты либо инициирует исключительную ситуацию NullPointerException.

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

```
public Object clone() throws CloneNotSupportedException {

Stack result = (Stack) super.clone();

result.elements = (Object[]) elements.clone();

return result;

}
```

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

Не всегда бывает достаточно рекурсивного вызова метода clone. Предположим, что вы пишите метод clone для хэш-таблицы, состоящей из набора сегментов (buckets), каждый из которых содержит либо ссылку на пер выи элемент в связном списке, имеющем несколько пар ключ/значение, либо null, если сегмент пуст. для лучшей производительности в этом классе вместо java.util.LinkedList используется собственный упрощенный связный список:

```
public class HashTable implements Cloneable{
private'Entry[] buckets = ... ;
    private static class Entry {
        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, что вы рекурсивно клонируете массив buckets, как это делалось для класса Stack:

```
// Ошибка: объекты будут иметь общее внутреннее состояние
public Object clone() throws CloneNotSupportedException {
HashTable result = (HashTable) super.clone();
    result.buckets = (Entry[]) buckets.clone();
    return result;
    }
```

Хотя клон и имеет собственный набор сегментов, последний ссылается на те же Связные списки, что и исходный набор, а это может привести к непредсказуемому поведению и клона, и оригинала. для устранения этой проблемы вам придется отдельна

копировать связный список для каждого сегмента. Представим один из распространенных приемов:

```
public class HashTable implements Cloneable {
private Entry[] buckets = ... .;
    private static class Entry {
        Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
             this.next = next;
        }
        // Рекурсивно копирует связный список. начинающийся
        // с указанной записи
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());


    public Object clone() throws CloneNotSupportedException


     HashTable result = (HashTable) super.clone();


    result.buckets = new Entry[buckets.length];
        for (int i = 0; i < buckets.lenght; i ++)
            if (buckets[i] != null)
             result.buckets[i] = (Entry)
            buckets[i]:deepCopy();
        return result;
        }
        // Остальное опущено
        }
```

Закрытый класс HashTable. Entry был привнесен для реализации метода "глубокого копирования" (deep copy). Метод clone в классе HashTable размещает в памяти новый массив buckets нужного размера, а затем в цикле просматривает исходный набор buckets, выполняя глубокое копирование каждого непустого сегмента. Чтобы скопировать связный список, начинающийся с указанной записи, метод глубокого копирования (deepCopy) из класса Ent гу рекурсивно вызывает самого себя. Этот прием выг/\ядит изящно и прекрасно работает для не слишком длинных сегментов, однако он не совсем подходит для клонирования связных списков, поскольку для каждого элемента в списке он делает в стеке новую запись. И если список buckets

окажется большим, может возникнуть переполнение стека. Во избежание этого можно заменить в методе deepCopy рекурсию итерацией:

```
// Копирование в цикле связного списка.
// начинающегося с указанной записи
Entry deepCopy() {
    Entry result = new Entry(key, value, next);
    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);
    return result;
    }
```

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

Как и конструктор, метод clone не должен вызывать каких-либо переопределяемых методов создаваемого клона (статья 15). Если метод clone вызывает переопределенный метод, этот метод будет выполняться до того, как подкласс, в котором он был определен, установит для клона нужно"е состояние. Это может привести к разрушению и клона, и самого оригинала. Поэтому метод put (key, val ue) должен быть либо не переопределяемым (final), либо закрытым. (Если это закрытый метод, то, по-видимому, он является вспомогательным (helper method) для другого, открытого и переопределяемого метода.)

Метод clone в классе Object декларируется как способный инициировать исключительную ситуацию CloneNotSupportedException, однако в пере определенных методах clone эта декларация может быть опущена. Метод clone в окончательном классе не должен иметь такой декларации, поскольку работать с методами, не инициирующими обрабатываемых исключений, приятнее, чем с теми, которые их инициируют (статья 41). Если же метод clone пере определяется в расширяемом классе, особенно в классе, предназначенном для наследования (статья 15), новый метод clone должен иметь декларацию для исключительной ситуации CloneNotSupportedException. Это дает возможность изящно отказаться в подклассе от клонирования, реализовав следующий метод clone:

```
// Метод клонирования, гарантирующий невозможность
// клонирования экземпляров
public final Object clone() throws CloneNotSupportedException{
throw new CloneNotSupportedException();
}
```

Следовать указанному совету необязательно, поскольку если для переопределяемого метода clone не было заявлено, что он может инициировать CloneNotSupportedException, новый метод clone в подклассе, не подлежащем клонированию, всегда может инициировать необрабатываемое исключение, например UnsupportedOperationException. Однако установившаяся практика говорит, что в этих условиях прав ильным будет исключение CloneNotSupportedExceptrion.

Подведем итоги. Все классы, реализующие интерфейс Cloneable, должны переопределять метод clone как открытый. Этот метод должен сначала вызвать метод supe г. 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 внутренние копии можно получить рекурсивным вызовом метода clone, такой подход не всегда является самым лучшим. Если класс содержит только поля простого типа и ссылки на неизменяемые объекты, 'ГО, по-видимому, нет полей, нуждающихся в восстановлении. Из этого правила есть исключения. Например, поле, предоставляющее серийный номер или иной уникальный идентификатор, а также поле, показывающее время создания объекта, нуждаются в восстановлении, даже если они имеют простой тип или являются неизменяемыми.

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

Изящный подход к копированию объектов - создание конструктора копий.
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 метод генерации:

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

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

Более того, конструктор копий (или статический метод генерации) может иметь аргумент, тип которого соответствует интерфейсу, реализуемому этим классом. Например, все реализации коллекций общего назначения, по соглашению, имеют конструктор копий с аргументом типа Collection или Map. Конструкторы копий, использующие интерфейсы, позволяют клиенту выбирать для копии вариант реализации вместо того, чтобы принуждать его принимать реализацию исходного класса. Допустим, что у вас есть объект LinkedList 1 и вы хотите скопировать его как экземпляр ArrayList. Метод c!one не предоставляет такой возможности, хотя это легко делается с помощью конструктора копий new ArrayList(l).

Рассмотрев все проблемы, связанные с интерфейсом Cloneable, можно с уверенностью сказать, что остальные интерфейсы не должны становиться его расширением, а классы, предназначенные для наследования (статья 15), не должны его реализовывать. Из-за множества недостатков этого интерфейса некоторые высококвалифицированные программисты предпочитают никогда не переопределять метод clone и никогда им не пользоваться за исключением, быть может, случая простого копирования массивов. Учтите. что, если в классе, предназначенном для наследования, вы не создадите, по меньшей мере, правильно работающий *защищенный* метод clone, реализация интерфейса Cloneable в подклассах станет невозможной.

Подумайте над реализацией интерфейса Comparable

В отличие от других обсуждавшихся в этой главе методов, метод compareTo в классе Object не декларируется. Пожалуй, это единственный такой метод в интерфейсе java.lang.ComparaЬle. 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 ComparaЬle, класс показывает, что его экземпляры обладают *естественным свойством упорядочения (natura!*ordering). Сортировка массива объектов, реализующих интерфейс ComparaЬle, выполняется просто:

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

Для объектов ComparaЬle так же просто выполняется поиск, вычисляются предельные значения и обеспечивается поддержка автоматически сортируемых коллекций. Например, следующая программа, использующая тот факт, что класс String реализует I1нтерфейс ComparaЬle, печатает в алфавитном порядке список аргументов, указанных в командной строке, удаляя при этом дубликаты:

```
public class WordList {
    public static void main(String[] args) {
    Set 5 = new TreeSet();
    s,addAll(Arrays.asList(args));
     System.out.println(s);
    }
}
```

Реализуя интерфейс СотрагаЬle, вы разрешаете вашему классу взаимодействовать со всем обширным набором общих алгоритмов и реализаций коллекций, которые связаны с этим интерфейсом. Приложив немного усилий, вы получаете огромные возможности. Практически все классы значений в библиотеках платформы Java реализуют интерфейс СотрагаЬle. И если вы пишите класс значений с очевидным свойством естественного упорядочения - алфавитным, числовым либо хронологическим,- вы должны хорошо продумать реализацию этого интерфейса. В этой статье рассказывается о том, как к этому приступить.

Общее соглашение для метода compareTo имеет тот же характер, что и соглашение Для метода equals. Приведем его текст по спецификации интерфейса СотрагаЬle:

> Выполняет сравнение текущего и указанного объекта и определяет их очередность. Возвращает отрицательное целое число, нуль или положительное целое число в зависимости от того, меньше ли текущий объект, равен или больше указанного объекта. Если тип указанного объекта не позволяет сравнивать его с текущим объектом, инициируется исключительная ситуация ClassCastException.
>
> В следующем описании запись *sgn(выражение)* обозначает математическую функцию *signum,* которая, по определению, возвращает -1, О 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 х и у. (Это подразумевает, что выражение х. сотрагеТо(у) должно инициировать исключительную ситуацию тогда и только тогда, когда у. сотрагеТо(х) инициирует исключение.)
> - Разработчик должен также гарантировать транзитивность отношения (x.сотрагеТо(у»О & & y.compareTo(z»O) подразумевает x.compareTo(z»O.
> - Наконец, разработчик должен гарантировать, что из тождества х. compareTo(y) == О вытекает тождество sgn(x.compareTo(z)) == sgn(y.compareTo(z)) для всех z.
> - Настоятельно рекомендуется выполнять условие (х. сотрагеТо(у) = =О) = =(х. equalS(y)). Вообще говоря, для любого класса, который реализует интерфейс СотрагаЬle, но нарушает это условие, сей факт должен быть четко оговорен (в сопроводительной документации). Рекомендуется использовать следующую формулировку: "Примечание: данный класс имеет естественное упорядочение, не согласующееся с условием равенства".

Как и соглашения для метода equals (статья 7), соглашения для compareTo не так сложны, как это кажется. Для одного класса любое разумное отношение упорядочения будет соответствовать соглашениям для compareTo. Для сравнения разных классов метод compareTo, в отличие от метода equals, использоваться не должен: если сравниваются две ссылки на объекты различных классов, можно инициировать исключительную ситуацию ClassCastException. Метод compareTo обычно так и делает. И хотя представленное соглашение не исключает сравнения между классами, в библиотеках для платформы Java, в частности в версии 1.4, нет классов, которые поддерживали бы такую возможность.

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

Рассмотрим условия соглашения для сотрагеТо. Первое условие гласит, что если вы измените порядок сравнения двух ссылок на объекты, про изойдет 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: если первый объект больше второго, а второй объект больше третьего, то первый объект должен быть больше третьего. Последнее условие: объекты, сравнение которых дает равенство, при сравнении с любым третьим объектом должны показывать одинаковый результат.

Из этих трех условий следует, что проверка равенства, осуществляемая с помощью метода compareTo, должна подчиняться тем же самым ограничениям, которые продиктованы соглашениями для метода equals: рефлективность, симметрия, транзитивность и отличие от null. Следовательно, здесь справедливо то же самое предупреждение: невозможно расширить порождающий экземпляры класс, вводя новый аспект и не нарушая при этом соглашения для метода сотрагеТо (статья 7). Возможен обходной Путь. Если вы хотите добавить важное свойство к классу, реализующему интерфейс СотрагаЬle, не расширяйте его, а напишите новый независимый класс, в котором для исходного класса выделено отдельное поле. Затем добавьте метод представления, возвращающий значение этого поля. Это позволит вам реализовать во втором классе любой метод сотрагеТо, который вам нравится. При необходимости клиент может рассматривать экземпляр второго класса как экземпляр первого класса.

Последний пункт соглашений для compareTo, являющийся скорее предположением, чем настоящим условием, постулирует, что проверка равенства, осуществляемая с помощью метода compareTo, обычно должна давать те же самые результаты, что и метод equals. Если это условие выполняется, считается, что упорядочение, задаваемое методом compareTo, *согласуется* с *проверкой равенства* (consistent with equals). Если же оно нарушается, то упорядочение называется *несогласующимся* с *проверкой равенства* (inconsistent with equals).

Класс, чей метод compareTo устанавливает порядок, несогласующийся с условием равенства, будет работоспособен, однако отсортированные коллекции, содержащие элементы этого класса, могут не соответствовать общим соглашениям для соответствующих интерфейсов коллекций (Collection, Set или Map). Дело в том, что общие соглашения для этих интерфейсов определяются в терминах метода equals, тогда как в отсортированных коллекциях ИСПОЛl>зуется проверка равенства, которая реализуется методом compareTo, а не equals.Если это произойдет, катастрофы не будет, но иногда это следует учитывать.

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"),этот набор будет содержать два элемента, поскольку два добавленных в него экземпляра класса BigDecimalне 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 BigDecimalоказываются равны при их сравнении с помощью метода comapreTo. (См. документацию на BigDecimal.)

Процедура написания метода compareTo похожа на процедуру для метода equals, но есть несколько ключевых различий. Перед преобразованием типа нет необходимости проверять тип аргумента. Если аргумент имеет неправильный тип, метод compareTo *обязан* инициировать исключительную ситуацию ClassCastExceptionЕсли аргумент имеет значение null, метод compareT o *должен* инициировать исключительную ситуацию NullPointerExceptionT o же самое вы получите, если приведете аргумент к правильному типу, а затем попытаетесь обратиться к его членам.

Сравнение полей само по себе является упорядочивающим сравнением, а не сравнением с проверкой равенства. Сравнение полей, имеющих ссылки на объекты, осуществляйте путем рекурсивного вызова метода сотрareTo. Если поле не реализует интерфейс Сотра rable или вам необходимо не стандартное упорядочение, вы можете использовать явную реализацию интерфейса ComparatorЛибо напишите ваш собственный метод, либо воспользуйтесь уже имеющимся, как это было в случае с методом сотра reTo в классе CaseInsensitiveString(статья 7):

```
public int compareTo(Object o) {
    CaseInsensitiveString cis = (CaseInsensitiveString)o;
    return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
```

Поля простого типа нужно сравнивать с помощью операторов < и >, массивы применяя эти инструкции для каждого элемента. Если у класса есть несколько значимых полей, порядок их сравнения критически важен. Вы должны начать с самого значимого поля и затем следовать в порядке убывания значимости. Если сравнение дает что-либо помимо нуля (означающего равенство), все, что вам нужно сделать,-

возвратить этот результат. Если самые значимые поля равны, продолжайте сравнивать следующие по значимости поля и т. д. Если все поля равны, равны и объекты, поэтому возвращайте нуль. Такой прием демонстрирует метод compareTo для класса PhoneNumbeиз статьи 8:

```
public int compareTo(Object o) {
PhoneNumber pn = (PhoneNumber)o;
    // Сравниваем коды зон
    if (areaCode < pn.areaCode)
     return -1;
    if (areaCode > pn.areaCode)
    return 1;
    // Коды зон равны, сравниваем номера АТС
     if (exchange < pn.exchange)
          return -1;
    if (exchange > pn.exchange)
    return 1;
    // Коды зон и номера АТС равны, сравниваем номера абонентов
     if (extension < pn.extension)
         return -1;
    if (extension > pn.extension)
     return 1;
    return o; // Все поля равны
    }
```

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

```
public int compareTo(Object o) {
PhoneNumber pn = (PhoneNumber)o;
    // Сравниваем коды зон
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff 1= O)
         return areaCodeDiff;
    // Коды зон равны, сравниваем номера АТС
    int exchangeDiff = exchange - pn.exchange;
    if (exchangeDiff != 0)
         return exchangeDiff;
    // Коды зон и номера АТС равны, сравниваем номера абонентов
    return extension - pn.extension;
    }
```

Такая уловка работает прекрасно, но применять ее следует крайне осторожно.
Не пользуйтесь ею, если у вас нет уверенности в том, что рассматриваемое поле не может иметь отрицательное значение или, что бывает чаще, разность между наименьшим и наибольшим возможными значениями поля меньше или равна значению INTEGER. MAX_VALUE (2^31 - 1). Причина, по которой этот прием не работает, обычно заключается в том, что 32-битовое целое число со знаком является недостаточно большим, чтобы показать разность двух 32-битовых целых чисел с произвольным знаком. Если **i** – большое положительное целое число, а **j** – большое отрицательное целое число, то при вычислении разности (i-j) произойдет переполнение и будет возвращено отрицательное значение. Следовательно, полученный нами метод compareTo работать не будет: для некоторых аргументов будет выдаваться 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 бывает крайне трудно, поскольку не· правильный метод compareTo работает правильно со многими входными значениями.

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 этой главе даются рекомендации, которые помогут вам наилучшим образом при менять эти элементы, чтобы ваши классы и интерфейсы были удобными, надежными и гибкими.

Сводите к минимуму доступность классов и членов

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

Сокрытие информации важно по многим причинам, большинство из которых Связано с тем обстоятельством, что этот механизм эффективно *изолирует* друг от друга модули, составляющие систему, позволяя разрабатывать, тестировать, оптимизировать, использовать, исследовать и обновлять их по отдельности. Благодаря этому ускоряется разработка системы, так как различные модули могут создаваться параллельно. Кроме того, уменьшаются расходы на сопровождение приложения, поскольку каждый модуль можно быстро изучить и отладить, минимально рискуя навредить остальным модулям. Само по себе сокрытие информации не может обеспечить

хорошей производительности, но оно создает условия для эффективного управления производительностью. Когда разработка системы завершена и процедура ее профилирования показала, работа каких модулей вызывает падение производительности (статья 37), можно заняться их оптимизацией, не нарушая функционирования 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 декларации (pr1vate, 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. Другими словами, вы должны использовать самый низший из возможных уровней доступа, который еще допускает правильное функционирование создаваемой программы.

Для классов и интерфейсов верхнего уровня (не являющихся вложенными) существуют лишь два возможных уровня доступа: *доступный только в пределах пакета* (package-private) и *открытый* (public). Если вы объявляете класс или интерфейс верхнего уровня с модификатором public, он будет открытым, в противном случае он будет доступен только в пределах пакета. Если класс или интерфейс верхнего уровня можно сделать доступным только в пакете, так и нужно поступать. 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 клиентам. Если же вы делаете класс или интерфейс открытым, на вас возлагается обязанность всегда поддерживать его с целью сохранения совместимости.

Если класс или" интерфейс верхнего уровня, доступный лишь в пределах пакета, используется только в одном классе, вы должны рассмотреть возможность превращения его в закрытый класс (или интерфейс), который будет вложен именно в тот класс, где он используется (статья 18). Тем самым вы еще более уменьшите его доступность. Однако это уже не так важно, как сделать необоснованно открытый класс доступным только в пределах пакета, поскольку класс, доступный лишь в пакете, уже является частью реализации этого пакета, а не его внешнего API.

Для членов класса (полей, методов, вложенных классов и вложенных интерфейсов) существуют 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 -были указаны модификаторы доступа.

- Защищенный (protected) - член доступен для подклассов того класса, "где этот член был объявлен (с небольшими ограничениями [jLS, 6.6.2]); доступ к члену можно получить из любого класса в пакете, где этот член был объявлен.

- Открытый (public) - член доступен отовсюду.

После того как для вашего класса 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 какому-то члену, вы можете убрать модификатор private и сделать этот член доступным в пределах всего пакета. Если вы обнаружите, что таких членов слишком много, еще раз проверьте модель вашей системы и попытайтесь найти другой вариант разбиения на классы, при котором они были бы лучше изолированы друг от друга. Как было сказано, и закрытый член, и член, доступный только в пределах пакета, являются частью реализации класса и обычно не оказывают воздействия на его внешний API. Однако они могут "просочиться" во внешний API, если класс реализует интерфейс Serializable (статьи 54 и 55).

Если уровень доступа для члена открытого класса меняется с доступного в пакете на защищенный, уровень доступности данного члена резко возрастает. Для этого класса защищенный член является частью внешнего API, а потому ему навсегда должна быть обеспечена поддержка. Более того, наличие защищенного члена в классе, передаваемом за пределы пакета, представляет собой открытую передачу деталей реализации (статья 15). Потребность в использовании защищенных членов должна возникать сравнительно редко.

Существует одно правило, ограничивающее ваши возможности по уменьшению доступности методов. Если какой-либо метод переопределяет метод суп ер класса, то методу в подклассе не разрешается иметь более низкий уровень доступа, чем был у метода в суперклассе [JLS, 8.4.6.3]. Это необходимо для гарантии того, что экземпляр подкласса можно будет использовать повсюду, где можно было использовать экземпляр суперкласса. Если вы нарушите это правило, то когда попытаетесь скомпилировать этот подкласс, компилятор сгенерирует сообщение об ошибке. Частный случай правила: если класс реализует некий интерфейс, то все методы класса, представленные в этом интерфейсе, должны быть объявлены как открытые (public). Это объясняется тем, что в интерфейсе все методы неявно подразумеваются открытыми.

Открытые поля (в отличие от открытых методов) в открытых классах должны быть редким явлением (если вообще должны ПОЯВЛЯТЬСЯ). Если поле не имеет модификатора final или имеет модификатор и ссылается на изменяемый объект, то, делая его открытым, вы упускаете возможность наложения ограничений на значения,

которые могут быть записаны в этом поле. Вы также лишаетесь возможности предпринимать какие-либо действия в ответ на изменение этого поля. Отсюда простой вывод: классы с открытыми изменяемыми полями небезопасны в системе с несколькими потоками (not thread-safe). Даже если поле имеет модификатор final и не ссылается на изменяемый объект, объявляя его открытым, вы отказываетесь от возможности гибкого перехода на новое представление внутренних данных, в котором это поле будет отсутствовать.

Из правила, запрещающего открытым классам иметь открытые поля, есть одно исключение. С помощью полей public static final классы могут предоставлять вовне константы. Согласно договоренности, названия таких полей состоят из прописных букв, слова в названии разделяются символом подчеркивания (статья 38). Крайне важно, чтобы эти поля содержали либо простые значения, либо ссылки на неизменяемые объекты (статья 13). Поле с модификатором final, содержащее ссылку на изменяемый объект, обладает всеми недостатками поля без модификатора final: хотя саму ссылку изменить нельзя, объект, на который она указывает, может быть изменен - с роковыми последствиями.

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

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

Открытый массив следует заменить закрытым массивом и открытым неизменяемым списком:

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

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

```
private static final Type[] PRIVATE_VALUES = { ... };

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

Подведем итоги. Всегда следует снижать уровень доступа, насколько это возможно. Тщательно разработав наименьший открытый API, вы должны не дать возможности 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.

Предпочитайте постоянство

Неизменяемый класс - это такой класс, экземпляры которого нельзя поменять. Вся информация, содержащаяся в любом его экземпляре, записывается в момент его создания и остается неизменной в течение всего времени существования этого объекта. В библиотеках для платформы]aya имеется целый ряд неизменяемых классов; в том числе String, простые классы-оболочки, Biglnteger и BigDecimal. На это есть много веских причин: по сравнению с изменяемыми классами, их проще проектировать, разрабатывать и и<;пользовать. Они менее подвержены ошибкам и более надежны.

Делая класс неизменяемым, выполняйте следующие пять правил:

1. Не создавайте каких-либо методов, которые модифицируют представленный объект (эти методы называются *мутаторами* (mutator)).
2. Убедитесь в том, что ни один метод класса не может быть переопределен. Это предотвратит потерю свойства неизменяемости данного класса в небрежном или умышленно плохо написанном подклассе. Защита методов от переопределения обычно осуществляется путем объявления класса в качестве окончательного, однако есть и другие способы (см. ниже).
3. Сделайте все поля окончательными (final). Это ясно выразит ваши намерения, причем в некоторой степени их будет поддерживать сама система. Это может понадобиться и для обеспечения правильного поведения программы в том случае, когда ссылка на вновь созданный экземпляр передается из одного потока в другой без выполнения синхронизации [Pugh01a] (как результат ведущихся работ по ис~равлению *модели памяти* в]aya).
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 12).
5. Убедитесь в монопольном доступе ко всем изменяемым компонентам. Если в вашем классе есть какие-либо поля, содержащие ссылки на изменяемые объекты, удостоверьтесь в том, что клиенты этого класса не смогут получить ссылок на эти объекты. Никогда не инициализируйте такое поле ссылкой на объект, полученной от клиента, метод доступа не должен возвращать хранящейся в этом поле ссылки на объект. При использовании конструкторов, методов доступа к полям и методов readObject (статья 56) создавайте *резервные копии* (defensive copies) (статья 24).

В при мерах из предыдущих статей многие классы были неизменяемыми. Так, класс PhoneNumber (статья 8) имеет метод доступа для каждого атрибута, но не имеет соответствующего мутатора. ГПредставим более сложный пример:

```
public final class Complex {
private final float re;
private final float im;

    public Complex(float re, float im) {

        this. re = re;
         this.im = im;
  }
  // Методы доступа без соответствующих мутаторо

   public float realPart() { return re; }
   public float 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) {
  float 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);
       }
  public boolean equals(Object o) {
   if (o == this)
            return true;
       if (!(o instanceof Complex))
       return false;

c = (Complex)o;
return (Float.floatToIntBits(re) ==Float.floatToIntBits(c.re)) && (Float.floatToIntBits(im) ==
Float.floatToIntBits(c.im));
}

// Чтобы понять,
// почему используется
// метод floatToIntBits
// см. статью 7.

            public int hashCode() {
                 int result = 17 + Float.floatToIntBits(re);
                 result = 37*result + Float.floatToIntBits(im);
                 return result;
                 }
            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 на то, 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 (functiona! approach), поскольку рассматриваемые методы возвращают результат применения некоей функции к своему операнду, не изменяя при этом сам операнд. Альтернативой является более распространенный *процедурный* подход (procedura! approach), при котором метод выполняет для своего операнда некую процедуру, которая меняет его состояние.

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

Неизменяемые объекты по своей сути безопасны при работе с потоками (thread-safe): им не нужна синхронизация. Они не могут быть разрушены только из-за того, что одновременно к ним обращается несколько потоков. Несомненно, это самый простой способ добиться безопасности при работе с потоками. Действительно, ни один поток никогда не сможет обнаружить какого-либо воздействия со стороны другого потока через неизменяемый объект. По этой причине неизменяемые объекты можно свободно использовать для совместного доступа. Неизменяемые классы должны задействовать это преимущество, заставляя клиентов везде, где возможно, применять уже существующие экземпляры. Один из простых приемов, позволяющих достичь этого: для часто используемых значений создавать константы типа publiC static final. Например, в классе Compex можно представить следующие константы:

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

Можно сделать еще один шаг в этом направлении. В неизменяемом классе MO~HO предусмотреть статические методы генерации, которые кэшируют часто запрашиваемые экземпляры вместо того, чтобы при каждом запросе создавать новые экземпляры, дублирующие уже имеющиеся. Подобные статические методы генерации есть в классах 8iglnteger и 8oo1ean. Применение статических методов генерации заставляет клиентов совместно использовать уже имеющиеся экземпляры, а не создавать новые. Это снижает расход памяти и сокращает работу по ее освобождению.

Благодаря тому, что неизменяемые объекты можно свободно предоставлять для 'совместного доступа, не требуется создавать для них *резервные копии* (defensive copies) (статья 24). В действительности вам вообще не нужно делать никаких копий, поскольку они всегда будут идентичны оригиналу. Соответственно, для неизменяемого класса не надо, да и не следует создавать метод clone и *конструктор копии* (copy constructor) (статья 10). Когда платформа Java только появилась, еще не было четкого понимания этого обстоятельства, и потому класс String имеет конструктор копий. Лучше им не пользоваться (статья 4).

Можно совместно использовать не только неизменяемый объект, но и его содержимое. Например, класс 8iglnteger применяет внутреннее представление знак/модуль (sign/magnitude). Знак числа задается полем типа int, его модуль массивом int. Метод инвертирования negate создает новый экземпляр 8iglnteger с тем же модулем и с противоположным знаком. При этом нет необходимости копировать массив, поскольку вновь созданный экземпляр 8iglnteger имеет внутри ссылку на тот же самый массив, что и исходный экземпляр.

Неизменяемые объекты образуют крупные строительные блоки для остальных объектов, как изменяемых, так и неизменяемых. Гораздо легче обеспечивать поддержку инвариантов сложного объекта, если известно, что составляющие его объекты не будут менять его "снизу". Частный случай данного принципа: неизменяемый объект формирует большую схему соответствия между ключами и набором элементов.

При этом вас не должно беспокоить то, что значения, однажды записанные в эту схему или набор, вдруг поменяются, и это приведет к разрушению инвариантов схемы или набора.

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

```
BigInteger moby = ...;

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

Метод flip8it создает новый экземпляр класса 8igInteger длиной также в миллион битов, который отличается от своего оригинала только одним битом. Этой операции требуются время и место, пропорциональные размеру экземпляра 8igInteger. Противоположный подход использует java.util.BitSet. Как и BigInteger, BitSet представляет последовательность битов произвольной длины, однако, в отличие от BigInteger, BitSet является изменяемым классом. В классе BitSet предусмотрен метод, позволяющий в экземпляре, содержащем миллионы битов, менять значение отдельного бита в течение фиксированного времени.

Проблема производительности усугубляется, 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 применяется для ускорения многошаговых операций, таких как возведение в степень по модулю. По всем перечисленным выше причинам использовать изменяемый класс-компаньон гораздо сложнее. Однако делать этого вам, к счастью, не надо. Разработчики класса BigInteger уже выполнили за вас всю тяжелую работу.

Описанный прием будет работать превосходно, если вам удастся точно предсказать, какие именно сложные многошаговые операции с вашим неизменяемым классом будут нужны клиентам. Если сделать это невозможно, самый лучший вариант создание *открытого* изменяемого класса-компаньона. В библиотеках для платформы Java такой подход демонстрирует класс String, для которого изменяемым классом Компаньоном является StringBuffer. В силу ряда причин BitSet вряд ли 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 любое переопределение своих методов.

Помимо возможности сделать класс окончательным (final), есть еще два способа решения этой проблемы. Первый способ: вместо того, чтобы делать окончательным сам класс, сделать окончательными все его методы, пометив как final. Единственное преимущество данного подхода заключается в том, что он позволяет программистам расширять класс, добавляя к нему новые методы, выстроенные поверх старых. По эффективности это то же самое, что вводить ,новые методы как статические в отдельном вспомогательном классе, не создающем экземпляров (статья 3), а потому использовать такой подход не рекомендуется.

Второй прием заключается в том, чтобы сделать все конструкторы неизменяемого класса закрытыми либо доступными только в пакете и вместо открытых конструкторов использовать открытые *статические методы генерации* (статья 1). Для пояснения представим, как бы выглядел класс Complex, если бы применялся такой подход:

```
// Неизменяемый класс со статическими методами генерации
// вместо конструкторов
publiC class Complex {
    private final float re;
    private final float im;

    private Complex(float re, float im) {
     this. re = re;
     this.im = im;

    public static Complex valueOf(float re, float im) {

     return new Cотplex(re, im);}

    // Остальное не изменилось

    }
```

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

Как показано в статье 1, статические методы генерации объектов имеют множество преимуществ по сравнению с конструкторами. Предположим, что вы хотите создать механизм генерации комплексного числа, отталкиваясь от его полярных координат. Использовать здесь конструкторы плохо, поскольку окажется, что собственный конструктор класса Complex будет иметь ту же самую сигнатуру, которую мы только что применяли: Complex(float, float).

Со статическими методами генерации все проще - достаточно добавить второй статический метод генерации с таким названием, которое четко обозначит его функцию:

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

Когда писались классы BigInteger и BigDecimal, не было согласия в том, что неизменяемые классы должны быть фактически окончательными. Поэтому любой метод этих классов можно переопределить. К сожалению, исправить что-либо впоследствии уже было нельзя, не потеряв при этом совместимость версий снизу вверх. Поэтому, если вы пишите класс, безопасность которого зависит от неизменяемости аргумента с типом BigInteger или BigDecimal, полученного от ненадежного клиента, вы должны выполнить проверку и убедиться в том, что этот аргумент действительно является "настоящим" классом BigInteger или BigDecimal, а не экземпляром какого-либо ненадежного подкласса. Если имеет место последнее, необходимо создать резервную копию этого экземпляра, поскольку придется исходить из того, что он может оказаться изменяемым (статья 24):

```
public void foo(BigInteger b) {
    if (b.getClass() != BigInteger.class)
        b = new BigInteger(b.toByteArray());
        }
```

Список правил для неизменяемых классов, представленный в начале статьи, гласит, что ни один метод не может модифицировать объект и все поля должны иметь модификатор final. Эти правила несколько строже, чем необходимо, и их можно ослабить с целью повышения производительности программы. Действительно, ни один метод не может произвести такое 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 трудоемких вычислений. Если в дальнейшем потребуется произвести те же самые вычисления, будет возвращено ранее сохраненное значение, ненужные вычисления выполняться не будут. Такая уловка работает надежно именно благодаря неизменяемости объекта: неизменность его состояния является гарантией того, что если вычисления выполнять заново, то они приведут опять к тому же результату.

Например, метод hashCode из класса PhoneNumbeT (статья 8) вычисляет хэш-код. Получив код в первый раз, метод сохраняет его на тот случай, если потребуется вычислять его снова. Такая методика, представляющая собой классический пример *отложенной инициализации* (статья 48), используется также и в классе String. Никакой синхронизации здесь не требуется, поскольку никаких проблем с повторным вычислением хэша не возникает.

Приведем общую идиому для кэширующей функции с отложенной инициализацией для неизменяемого объекта:

```
// Кэширующая функция с отложенной инициализацией
// для неизменяемого объекта
.private volatile Foo cachedFooVal = UNLIKELY_FOO_VALUE;

publlic Foo foo() {
    Foo result = cachedFooVal;
    if (result == UNLIKELY_FOO_VALUE)
    result = cachedFooVal = fooValue();
    return result;
    }

// Закрытая вспомогательная функция, вычисляющая

// значение нашего объекта foo
private Foo fooVal() { ... }
```

Следует добавить одно предостережение, касающееся сериализуемости объектов. Если вы решили, что ваш неизменяемый класс должен реализовывать интерфейс Serializable, но при этом у него есть одно или несколько полей, которые ссылаются на изменяемые объекты, то вы обязаны предоставить явный метод readObject или readResolve, даже если для этого класса можно использовать сериализуемую форму, предоставляемую по умолчанию. Метод readObject, применяемый по умолчанию, позволил бы пользователю создать изменяемый экземпляр вашего во всех остальных ситуациях неизменяемого класса. Эта тема детально раскрывается в статье 56.

Подведем итоги. Не стоит для каждого метода get писать метод set. Классы должны оставаться неизменяемыми, если нет веской причины делать их изменяемыми. Неизменяемые классы имеют массу преимуществ, единственный же их недостаток - возможные проблемы с производительностью при определенных условиях. Небольшие объекты значений, такие как PhoneNumber и Complex, всегда следует делать неизменяемыми. (В библиотеках для платформы Java есть несколько классов например java.util.Date и java.awt.Point, которые должны быть неизменяемыми, но таковыми не являются.) Вместе с тем вам следует серьезно подумать, прежде чем делать неизменяемыми более крупные объекты значений, такие как 5tring и BigInteger. Создавать для вашего неизменяемого класса открытый изменяемый класс-компаньон следует, *только если* вы уверены в том, что это необходимо для получения приемлемой производительности (статья 37) ..

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

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

Перечисленные правила иллюстрирует класс TimerTask. Он· является изменяемым, однако пространство его состояний намеренно оставлено небольшим. Вы создаете экземпляр, задаете порядок его выполнения и, возможно, отменяете это решение. Как только задача, контролируемая таймером, запускается на исполнение или отменяется, 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 продемонстрировать свойство неизменяемости. Он не обладает достоинствами промышленной реализации класса комплексных чисел. для умножения и деления комплексных чисел он использует обычные формулы, для которых нет правильного округления и которые имеют скудную семантику для комплексных значений NaN и бесконечности [Kahan91, Smith62, Thomas94].

Предпочитайте компоновку наследованию

Наследование (inheritance) - это мощный способ обеспечения многократного использования кода, но не всегда лучший инструмент для работы. При неправильном применении наследование приводит к появлению ненадежных программ. Наследование можно безопасно использовать внутри пакета, где реализация и подкласса, и суперкласса находится под контролем одних и тех же программистов. Столь же безопасно пользоваться наследованием, когда расширяемые классы специально созданы и документированы для последующего расширения (статья 15). Однако наследование обыкновенных неабстрактных классов за пределами пакета сопряжено с риском. Напомним, что в этой книге слово "наследование" (inheritance) применяется для обозначения *наследования реализации* (implementation inheritance), когда один класс расширяет Другой. Проблемы, обсуждаемые в этой статье, не касаются *наследование интерфейса* (interface inheritance), когда класс реализует интерфейс или же один интерфейс расширяет другой.

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

Предположим, что у нас есть программа, использующая класс HashSet. для повышения производительности нам необходимо запрашивать у HashSet, сколько элементов было добавлено с момента его создания (не путать с его текущим размером, который при удалении .элемента уменьшается). Чтобы обеспечить такую возможность, мы пишем вариант класса HashSet, который содержит счетчик количества попыток добавления элемента и предоставляет метод доступа к этому счетчику. В классе HashSet есть два метода, с помощью которых можно добавлять элементы: add и addAl1. Переопределим оба метода:

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

    public InstrumentedHashSet(int initCap, float loadFactor) {

    super(initCap, loadFactor);

    }

    public boolean add(Object o) {

    addCount ++;
        return super.add(o);
        }

    public boolean addAll(Collection c) {

    addCount += c.size();
        return super.addAll(c);
        }

   publlic 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 s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[] {"Snap", "Crackle", "Pop"}));
```

Мы предполагаем, что метод getAddCount должен возвратить число 3, но он возвращает 6. Что же

не так? Внутри класса HashSet метод addAll реализован прверх 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. addAl1 была вызвана реализация addAll в классе HashSet. В свою очередь, это влечет вызов метода add, переопределенного в классе InstrumentedHashSet,- по одному разу для каждого элемента. Каждый из этих трех вызовов добавляет к значению addCount еще единицу, так что в итоге общий прирост составляет шесть: добавление каждого элемента с помощью метода addAl1 .засчитывается дважды.

Мы могли бы "исправить" Подкласс, отказавшись от переопределения метода addAl1. Полученный класс будет работать, но правильность его работы зависит от того обстоятельства, что метод addAll в классе HashSet реализуется поверх метода add. Такое "использование самого себя" является деталью реализации, и нет гарантии, что она будет сохранена во всех реализациях платформы Java, не поменяется при переходе от одной версии к другой. Следовательно, полученный класс InstrumentedHashSet может быть ненадежен.

Ненамного лучшим решением будет пере определение addAl1 в качестве метода, который в цикле просматривает представленный набор и для каждого элемента один раз в'ызывает метод add. Это может гарантировать правильный результат независимо от того, реализован ли метод addAll в классе HashSet поверх метода add, поскольку реализация addAl1 в классе HashSet больше не применяется. Однако и такой прием не решает всех проблем. Он подразумевает повторную реализацию методов суперкласса, которые могут приводить, а могут не приводить к использованию классом самого себя. Этот вариант сложен, трудоемок и подвержен ошибкам. К тому же это не всегда возможно, поскольку некоторые методы нельзя реализовать, не имея доступа к закрытым полям, которые недоступны для подкласса.

Еще одна причина ненадежности подклассов связана с тем, что в новых версиях суперкласс может обзавестись новыми методами. Предположим, безопасность программы зависит от того, что все элементы, помещенные в некоторую коллекцию, должны соответствовать некоему утверждению. Выполнение этого условия можно гарантировать, создав для этой коллекции подкласс, пере определив в нем все методы, добавляющие элемент, таким образом, чтобы перед добавлением элемента проверялось его соответствие рассматриваемому утверждению. Такая схема работает замечательно до тех пор, пока в следующей версии суперкласса не появится новый метод, Который также может добавлять элемент в коллекцию. Как только это произойдет, Станет возможным добавление "незаконных" элементов' в экземпляр подкласса простым вызовом нового метода, который не был пере определен 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 воздержаться 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 с другим типом возвращаемого значения, то ваш подкласс перестанет компилироваться [ILS,8.4.6.3]. Если же вы создали в подклассе метод с точно такой же сигнатурой, как и у нового метода в суперклассе, то переопределите последний и опять столкнетесь с обеими описанными выше проблемами. Более того, вряд ли ваш метод будет отвечать требованиям, предъявляемым к новому методу в суперклассе, так как, когда вы писали этот метод в подклассе, они еще не были сформулированы.

К счастью, можно 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 *передачей вызова* (forwarding), а соответствующие методы нового класса носят название *методов переадресации* (forwarding method). 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:

```
// Класс-оболочка: вместо наследования используется композиция

public class InstrumentedSet implements Set {
      private final Set s;
      private int addCount = 0;
      public InstrumentedSet(Set s) {
            this.s = s; }


      public boolean add(Object o){


       addCount ++;
            return s. add(o); }


      public boolean addAll(Collections c) {


      addCount += c.size();
            return s.addAll(c); }


    public int getAddCount(){


    return addCount; }

     // Методы переадресации
         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 iterator() { return s.iterator(); }
          public boolean remove(Object o) { return s. remoye(o); }
          public boolean containsAll(Collection c) { return s.containsAll(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 Object[] toArray(Object[] a) { return s.toArray(a); }
          public boolean equals(Object o) { return s.equals(o); }
          public 1nt hashCode() { return s.hashCode(); }
         public Str1ng toStr1ng() { return s.toStr1ng(); }
```

Создание класса Inst rumentedSet стало возможным благодаря наличию интерфейса Set, в котором собраны функции 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 выполнения измерений. 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 предоставленным ему конструктором. Например:

```
Set s1 = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

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

```
static void f(Set s) {
    InstrumentedSet sInst = new InstrumentedSet(s);
        // Внутри этого метода вместо s используем sInst
        }
```

Класс InstrumentedSet называется *классом-оболочкои* (wrapper), поскольку Каждый экземпляр InstrumentedSet является оболочкой для другого экземпляра Set. Он также известен как шаблон *Decorator* (декоратор) [Camma95, стр. 175], класс InstrumentedSet "украшает" Set, добавляя ему новые функции. Иногда сочетание композиции и переадресации ошибочно называют *делегированием* (delegation). Однако формально назвать это делегированием нельзя, если только объект-оболочка не передает себя "обернутому" объекту [Camma95, стр.20].

Недостатков у классов-оболочек немного. Первый связан с тем, что классы-оболочки не приспособлены для использования в *схемах* с *обратным вызовом* (callback framework), где один объект передает другому объекту ссылку на самого себя для последующего вызова (callback - обратный вызов). Поскольку обернутый объект не знает о своей оболочке, он передает ссылку на самого себя (this), и, как следствие, обратные вызовы минуют оболочку. Это называется *проблемои 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 оказывает существенного влияния. Писать методы переадресации несколько утомительно, однако это частично компенсируется тем, что вам нужно создавать лишь один конструктор.

Наследование уместно только в тех случаях, когда подкласс действительно является подтипом (subtype) суперкласса. Иными словами, класс *B* должен расширять класс *A* только тогда, когда между двумя этими классами существует отношение типа "является". Если вы хотите сделать класс *B* расширением класса *A,* задайте себе вопрос: "Действительно ли каждый *B* является *A?"* Если вы не можете с уверенностью ответить на этот вопрос утвердительно, то *B* не должен расширять *A.* Если же ответ отрицательный, часто это оказывается, что *B* должен иметь закрытый от всех экземпляр *A* и предоставлять при этом меньший по объему и более простой API: *A* не является необходимой частью *B,* это лишь 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. 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 безо всякой необходимости раскрываете детали реализации. Получающийся при этом API привязывает вас к первоначальной реализации, навсегда ограничивая производительность вашего класса. Более серьезно то, что: демонстрируя внутренние элементы класса, вы позволяете клиенту обращаться к ним напрямую. Самое меньшее это 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): старый метод учитывает значения по умолчанию, тогда как второй метод, унаследованный от класса Hashtable, этого не делает. И самое серьезное: напрямую модифицируя. суперкласс, клиент получает возможность разрушать инварианты подкласса. В случае с классом Properties разработчики рассчитывали, что в качестве ключей и значений можно будет применять только строки, однако прямой доступ к базовому классу H.ashtable позволяет обходить это условие. Как только указанный инвариант нарушается, пользоваться другими элементами API для класса Properties (методами load и store) становится невозможно. Когда эта проблема была обнаружена, исправлять что-либо было слишком поздно, поскольку появились клиенты, работа которых Зависит от возможности применения ключей и значений, не являющихся строками.

Последняя группа вопросов, которые вы должны рассмотреть, прежде чем *р*ешиться использовать наследование вместо композиции: есть ли в АР! того Класса, который вы намереваетесь расширять, какие-либо изъяны? если есть, то не волнует ли вас то обстоятельство, что эти изъяны перейдут в АР! вашего класса? Наследование копиру~т любые дефекты в АР! суперкласса, тогда как композиция ПОзволяет разработать новый АР!, который скрывает эти недостатки.

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

Проектируйте и документируйте наследование либо запрещайте его.

Статья предупреждает вас об опасностях создания подклассов для "чужого" класса, наследование которого не предполагалось и не было документировано. Что же означает "класс, спроектированный и документированный для наследования"?

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

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

```
publlic 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 присутствовал 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 iterato r коллекции возвращает объект итератора. Заметим, что если у итератора не реализован метод гетоуе, то данная реализация инициирует исключительную ситуацию Unsuppo rtedOpe rationException.

Приведенное описание не оставляет сомнений в том, что переопределение метода iterator повлияет на работу метода гетоуе. Более того, в ней точно указано, каким образом работа экземпляра Iterator, возвращаемого методом iterator, будет влиять ii работу метода гетоуе. Сравните это с ситуацией, рассмотренной в статье 14, когда программист, создающий подкласс для HashSet, просто не мог знать, повлияет 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 может потребоваться создание механизма для диагностирования своей собственной внутренней деятельности в виде правильно выбранных защищенных методов или, в редких случаях, защищенных полей. Например, рассмотрим метод removeRange из класса j ava. util. Abst ractList:

protected void removeRange(int fromIndex, int toIndex)

Удаляет из указанного 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 ArrayListна (toIndex - fromIndex)элементов. (Если toIndex== fromIndex,процедура ни на что не влияет.)

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

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

Параметры:

FromIndex	индекс первого удаляемого элемента индекс
toIndex	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 полностью переписывать весь механизм subList -задача не из легких!

Как же решить; какие из защищенных методов и полей мощно раскрывать при построении класса, предназначенного для наследования? К сожалению, чудодейственного рецепта здесь не существует. Лучшее, что можно сделать,- это выбрать самую приемлемую гипотезу и проверить ее на практике, написав несколько подклассов. Вы должны предоставить клиентам минимально возможное число защищенных методов и полей, поскольку каждый из них связан с деталями реализации. С другой стороны, их количество не должно быть слишком малым, поскольку отсутствие защищенного метода может сделать класс практически негодным для наследования.

Готовя к наследованию класс, который, по-видимому, получит широкое распространение, учтите, что вы навсегда задаете схему использования классом самого себя, а также реализацию, неявно представленную защищенными методами и полями. Такие обязательства могут усложнять или даже делать невозможным дальнейшее улучшение Производительности и функциональных возможностей в будущих версиях класса.

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

Есть лишь несколько ограничений, которым обязан соответствовать класс, чтобы его наследование стало возможным. Конструкторы класса не должны вызывать переопределяемые методы, непосредственно или опосредованно. Нарушение этого правила может привести к аварийному завершению программы. Конструктор суперкласса выполняется прежде конструктора подкласса, а потому переопределяющий метод в подклассе будет вызываться перед запуском конструктора этого подкласса. И если переопределенный метод зависит от инициализации, которую осуществляет конструктор подкласса, то этот метод будет работать совсем не так, как ожидалось. для пояснения приведем пример класса, нарушающеГ9 это правило:

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

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

```
final class Sub extends Super {

private final Date date;
        // Пустое поле final заполняется конструктором

    Sub() {
        date = new Date();
    }
    // Переопределяет метод Super.m, используемый конструктором
    Super() publiC void m() {
        System.out.println(date); }


    public static void main(String[] args) {


     Sub s = new Sub();
        s.m();
        }
  }
```

Предполагается, что эта про грамма напечатает текущую дату дважды, однако в первый раз она выводит null, поскольку метод т вызывается конструктором Super() прежде, чем конструктор Sub() получает возможность инициализировать поле даты. Отметим, что данная программа видит поле final в двух разных состояниях.

Реализация интерфейсов Cloneable и Se rializable при проектировании наследования создает особые трудности. Вообще говоря, реализовывать какой-либо из этих интерфейсов в классах, предназначенных для наследования, не очень хорошо уже потому, что они создают большие сложности для программистов, расширяющих этот класс. Есть, однако, специальные приемы, которые можно использовать с тем, чтобы обеспечить передачу реализации этих интерфейсов в подкласс, а не заставлять его реализовывать их заново. Эти приемы описаны в статьях 10 и 54.

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

И, наконец, если вы решили реализовать интерфейс Serializable в классе, предназначенном для наследования, а у этого класса есть метод readResolve или wri teReplace, то вы должны делать этот метод не закрытым, а защищенным. Если эти методы будут закрытыми, то подклассы будут молча игнорировать их. Это еще один случай, когда для обеспечения наследования детали реализации класса становятся частью его API.

Таким образом, проектирование класса для наследования накладывает на него существенные ограничения. В ряде ситуаций это необходимо делать, например, когда речь идет об абстрактных классах, содержащих "скелетную реализацию" интерфейса (статья 16). В других ситуациях этого делать нельзя, например, в случае с неизменяемыми классами (статья 13).

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

Наилучшим решением этой проблемы является запрет на создание под·классов для тех классов, которые не были специально разработаны и не имеют требуемого описания для безопасного выполнения данной операции. Запретить создание подклассов можно двумя способами. Более простой заключается в объявлении класса как окончательного (final). Другой подход состоит в том, чтобы сделать все Конструкторы класса закрытыми или доступными лишь в пределах пакета, а вместо них создать открытые статические методы генерации. Такая альтернатива, дающая возможность гибко использовать класс внутри подкласса, обсуждалась в статье 13. Приемлем любой из указанных подходов.

Возможно, этот совет несколько сомнителен, поскольку так много программистов выросло с привычкой создавать для обычного неабстрактного класса подклассы лишь для того, чтобы добавить 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 14, создает превосходную альтернативу наследованию, используемому всего лишь для изменения функциональности.

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

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

Предпочитайте интерфейсы абстрактным классам.

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

Имеющийся класс несложно подогнать под реализацию нового интерфейса.

Все, что для этого нужно,- добавить в класс необходимые методы, если их еще нет, и внести в декларацию класса пункт о реализации. Например, когда платформа Java была дополнена интерфейсом Comparable, многие существовавшие классы были перестроены под его реализацию. С другой стороны, уже имеющиеся классы, вообще говоря, нельзя перестраивать для расширения нового абстрактного класса. Если вы хотите, чтобы два класса расширяли один и тот же абстрактный класс, вам придется поднять этот абстрактный класс в иерархии типов настолько высоко, чтобы прародитель обоих этих классов стал его Подклассом. К сожалению, это вызывает значительное нарушение иерархии типов, заставляя всех потомков общего предка расширять новый абстрактный класс независимо от того, целесообразно это или нет.

Интерфейсы идеально подходят для создания дополнений (mixin). Помимо своего "первоначального типа", класс может реализовать некий дополнительный тип (mixin), объявив о том, что в нем реализован дополнительный функционал. Например, Comparable является дополнительным интерфейсом, который дает классу возможность декларировать, что его экземпляры упорядочены по отношению к другим, сравнимым с ними объектам. Такой интерфейс называется mixin, поскольку позволяет к первоначальным функциям некоего типа примешивать (mixed in) дополнительные функциональные возможности. Абстрактные классы нельзя использовать для создания дополнений по той же причине, по которой их невозможно встроить в уже имеющиеся классы: класс не может иметь более одного родителя, и в иерархии классов нет подходящего места, куда можно поместить mixin.

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

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

В жизни некоторые певцы являются и 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. В действительности мы можем определить третий интерфейс, который расширяет оба Интерфейса 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 в степени n сочетаний, которые, возможно, придется поддерживать. Это называется комбинаторным взрывом (combinatorial explosion). Раздутые иерархии классов могут 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, описанную в статье 14. Если же для определения типов вы при меняете абстрактный класс, то вы не оставляете программисту, желающему добавить новые функциональные возможности, иного выбора, кроме как использовать наследование. Получаемые в результате классы будут не такими мощными и не такими надежными, как классы-оболочки.

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

По соглашению, скелетные реализации носят названия вида Abstract/nterface, где interface - это имя реализуемого ими интерфейса. Например, в архитектуре Collections Framework представлены скелетные реализации для всех основных интерфейсов коллекций: AbstractCollection, AbstractSet, AbstractList и AbstractMap.

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

```
// Адаптер интерфейса List для массива целых чисел (int)
static List intArrayAsList(final int[] a)
    if (a = =null)
        throw new NullPointerException();
    return new AbstractList() {
        public Object get(int i) {
            return new Integer(a[i]); }

        public int size() {

        return a.length; }

        public Object set(int i, Object o) {

        int oldVal = a[i];
            a[i] = «Integer)o).intValue();
            return new Integer(oldVal);
            }
    };
}
```

Если принять во внимание все, что делает реализация интерфейса List, то этот пример демонстрирует всю мощь скелетных реализаций. Кстати, пример является *адаптером* (Adapter) [Сатта95, стр. 139], который позволяет представить массив int в виде списка экземпляров Integer. Из-за всех этих преобразований из значений int в экземпляры Integer и обратно производительность метода не очень высока. Отметим, что здесь приведен лишь статический метод генерации, сам же класс является недоступным *анонимным IUIaccom* (статья 18), спрятанным внутри статического метода генерации.

Достоинство скелетных реализаций заключается в том, что они оказывают помощь в реализации абстрактного класса, не налагая при этом строгих ограничений, как это имело бы место, если бы для определения типов использовались абстрактные классы. для большинства программистов, реализующих интерфейс, расширение скелетной реализации - это очевидный, хотя и необязательный выбор. Если имеющийся класс нельзя заставить расширять скелетную реализацию, он всегда может реализовать представленный интерфейс сам. Более того, скелетная реализация помогает в решении стоящей перед разработчиком задачи. Класс, который реализует данный интерфейс, может переадресовывать вызов метода, указанного в интерфейсе, содержащемуся внутри его экземпляру закрытого класса, расширяющего скелетную реализацию. Такой прием, известный как *искусственное множественное наследование* (simulated multiple inheritance), тесно связан с идиомой класса-оболочки (статья 14). 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 при в едем скелетную реализацию интерфейса Map. Entry. В том виде, как это показано здесь, Класс не включен в библиотеки для платформы Java, хотя, вероятно, это следовало бы сделать.

```
// Скелетная реализация
public abstract class AbstractMapEntry implements Map. Entry {
// Примитивы
    public abstract Object getKey();
    public abstract Object getValue();
```

```
// Элементы в изменяемых схемах должны переопределять этот метод
publiC Object setValue(Object value) {
      throw new UnsupportedOperationException();
}
// Реализует основные соглашения для метода Map.Entry.equals
public boolean equals(Object o) {
      if (o == this)
      return true;
      if (!(o instanceof Map,Entry))
      return false;
      Map.Entry arg = (Map. Entry)o;
      return eq(getKey(), arg.getKey()) && eq(getValue(),
            arg.getValue());
private static boolean eq(Object 01, Object 02) {
return (01 == null ? 02 == null : 01.equals(02));
}
// Реализует основные соглашения для метода Map. Entry.hashCode
 public int hashCode() {
      return
            (getKey() == null ? 0 : getKey().hashCode())
            (getValue() == null ? 0 : getValue().hashCode());
            }
  }
```

Поскольку скелетная реализация предназначена для наследования, вы должны выполнять все указания по разработке и документированию, представленные в статье 15. для краткости в предыдущем примере опущены комментарии к документации, однако качественное документирование для скелетных реализаций абсолютно необходимо.

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 прием не работает.

Вообще говоря, в открытый интерфейс невозможно добавить какой-либо метод, не разрушив все имеющиеся программы, которые используют этот интерфейс. В классе, ранее реализовавшем этот интерфейс, новый метод не будет представлен, и, как следствие, класс компилироваться не будет. Ущерб можно несколько уменьшить, если

новый метод добавить одновременно и· в скелетную реализацию, и в интерфейс, однако по-настоящему это не решит проблемы. Любая реализация интерфейса, не наследующая скелетную реализацию, все равно работать не будет.

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

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

Используйте интерфейсы только для определения типов

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

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

```
// Шаблон интерфейса констант - не использоваты!

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 класса не представляет никакого интереса. На практике это может даже сбить их с толку. Хуже того, это является неким обязательством: если в будущих версиях класс поменяется так, что ему уже не будет нужды использовать данные константы, он все равно должен будет реализовывать этот интерфейс для обеспечения совместимости на уровне двоичных кодов (binary compatibility). Если же интерфейс констант реализует неокончательный класс; константами из этого интерфейса будет засорено пространство имен всех его подклассов.

В библиотеках для платформы Java есть несколько интерфейсов с константами, например jaya.io.ObjectSt reamConstants. Подобные интерфейсы нужно воспринимать как отклонение от нормы, и подражать им не следует.

Для передачи констант существует несколько разумных способов. Если константы сильно связаны с имеющимся классом или интерфейсом, вы должны добавить их непосредственно в этот класс или интерфейс. Например, все классы-оболочки в библиотеках платформы Java, связанные с числами, такие как Integer и Float, предоставляют константы MIN_VALUE и MAX_VALUE. Если же константы лучше рассматривать как члены перечисления, то передавать их нужно с помощью класса перечисления (статья 21). В остальных случаях вы должны передавать константы с помощью вспомогательного класса (utility c!ass), не имеющего экземпляров (статья 3). Представим вариант вспомогательного класса для предыдущего примера PhysicalConstants:

```
// Вспомогательный класс для констант

public class PhysicalConstants {
      private PhysicalConstants() { }
            // Предотвращает появление экэемпляра

      public static final double AVOGADROS_NUMBER  =6.02214199e23;

      public static final doubleBOLTZMANN_CONSTANT  =1.3806503e-23;

      public static final double ELECTRON_MASS  =9.10938188e-31;
}
```

Хотя представление PhysicalConstants в виде вспомогательного класса требует, чтобы клиенты связывали названия констант с именем класса, это не большая цена за получение осмысленного API. Возможно, что со временем язык Java позволит

осуществлять импорт статических полей. Пока же вы можете сократить размер программы, поместив часто используемые константы в локальные переменные или закрытые статические поля, например:

private static final double PI = Math. PI;

Таким образом, интерфейсы нужно использовать только для определения типов. Их не 'следует применять для передачи констант.

Предпочитайте статические классы-члены нестатичес

Класс называется вложенным (nested), если он определен внутри другого класса. Вложенный класс должен создаваться только для того, чтобы обслуживать окружающий его класс. Если вложенный класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня. Существуют четыре категории вложенных классов: статический класс-член (static member class), нестатический класс-член (nonstatic member class), анонимный класс (anonymous class) и локальный класс (local class). За исключением первого, остальные категории классов называются внутренними (inner class). В этой статье рассказывается о том, когда и какую категорию вложенного класса нужно использовать и почему.

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

В одном из распространенных вариантов статический класс-член используется как открытый вспомогательный класс, который пригоден для применения, только когда есть внешний класс. Например, рассмотрим перечисление, описывающее операции, которые может выполнять калькулятор (статья 21). Класс Operation должен быть открытым статическим классом-членом класса Calculator. Клиенты класса Calculator могут ссылаться на операции, выполняемые калькулятором, используя такие имена, как Calculator.Operation.PLUS или Calculator.Operation.MINUS. Этот вариант приводится ниже.

С точки зрения синтаксиса, единственное различие между статическими и нестатическими классами-членами заключается в том, что в декларации статических Классов-членов присутствует модификатор static. Несмотря на свою синтаксическую схожесть, эти две категории вложенных классов совершенно разные. Каждый экземпляр нестатического члена-класса неявным образом связан с содержащим его

экземпляром класса-контеинера (enclosing instance). Из метода в экземпляре нестатического класса-члена можно вызывать методы содержащего его экземпляра, либо, используя специальную конструкцию this [JLS 15.8.4], можно получить ссылку на включающий экземпляр. Если экземпляр вложенного класса может существовать в отрыве от экземпляра внешнего класса, то вложенный класс не может быть нестатическим классом-членом: нельзя создать экземпляр нестатического класса-члена, не создав включающего его экземпляра.

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

Нестатические классы-члены часто используются для определения адаптера (Adapter) [Сатта95, стр. 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 представлении / (оллекции (collection view), возвращаемых методами keySet, entrySet и values интерфейса Map. Аналогично, в реализациях интерфейсов коллекций, таких как Set и List, нестатические классы-члены обычно используются для создания итераторов:

```
// Типичный вариант использования нестатического класса-члена

public class MySet extends AbstractSet {
        // Основная часть класса опущена

    publiC Iterator iterator() {
        return new MyIterator(); }
        private class MyIterator implements Iterator {
        }
}
```

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

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

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

Анонимные классы в языке программирования Java не похожи ни на какие другие.
Анонимный класс не имеет имени. Он не является членом содержащего его класса. Вместо того чтобы быть декларированным с остальными членами класса, он одновременно декларируется и порождает экземпляр в момент использования. Анонимный класс можно поместить в любом месте программы, где разрешается применять выражения. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический класс-член: в нестатическом контексте появляется окружающий его экземпляр.

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

Анонимный класс обычно служит для создания объекта функции (function object), такого как экземпляр класса Comparator. Например, при вызове следующего метода строки в массиве, будут отсортированы по их длине:

```
// Типичный пример использования анонимного класса

Arrays.sort(args, new Comparator() {
      public int compare(Object o1, Object o2) {
            return ((String)o1).length() – ((String)o2).length();

} );
```

Другой распространенный случай использования анонимного класса - создание *объекта процесса* (process object), такого как экземпляры классов Thread, Runnable или TiterTask. Третий вариант: в статическом методе генерации (см. метод intArrayAsList в статье 16). Четвертый вариант: инициализация открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс (см. класс Operation в статье 21). Если, как было рекомендовано ранее, класс Operation будет статическим членом класса Calculator, то отдельные константы класса Operation окажутся дважды вложенными классами:

```
// Типичный пример использования открытого
// статического класса-члена
public class Calculator {
      public static abstract class Operation{
       private final String name;

            Operation(String пате) { this.name = name; }

            public String toString() { return this.name; }

            // Выполняет арифметическую операцию, представленную

            // данной константой
            abstract double eval(double x, double y);

           // Дважды вложенные анонимные классы
          public static final Operation PLUS = new Operation("+"){
          double eval(double x, double y) { return x + y; }
          };
          public static final Operation MINUS = new Operation("-") {
          double eval(double x, double y)  {return x - y; }
          };
           public static final Operation TIMES = new Operation("*"){
           double eval(double x, double y)  { , return x * y; }
            };
          public static final Operation DIVIDE = new Operat1on("/"){
          double eval(double x, double y) { return x / y; }

             };
    }
    // Возвращает результат указанной операции
    public double calculate(double x, Operation op, double y) {
    return op.eval(x, y);
           }
}
```

Локальные классы, вероятно, относятся к наиболее редко используемой из четырех категорий вложенных классов. Локальный класс можно декларировать везде, где разрешается декларировать локальную переменную, и он подчиняется тем же самым правилам видимости. Локальный класс имеет несколько признаков, объединяющих его с каждой из трех других категорий вложенных классов. Как и классы-члены, локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр тогда и только тогда, когда применяются в нестатическом контексте. Как и анонимные классы, они должны быть достаточно короткими, чтобы не мешать удобству чтения метода или инициализатора, в котором они содержатся.

Подведем итоги. Существуют четыре категории вложенных классов, каждая из которых занимает свое место. Если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах метода, используйте класс-член. Если каждому экземпляру класса-члена необходима ссылка на включающий его экземпляр, делайте его нестатическим, в остальных случаях он должен быть статическим. Предположим, что класс находится внутри метода. Если вам нужно создавать экземпляры этого класса только в одном месте программы и уже есть тип, который характеризует это класс, сделайте его анонимным классом. В противном случае это должен быть локальный класс.

Глава 5

Замена конструкций на С языке

Между языками программирования C и Java много общего, однако в Java отсутствует несколько конструкций языка с. В большинстве случаев совершенно очевидно, почему та или иная конструкция языка C опущена и как без нее обойтись. В этой главе предлагается замена для ряда конструкций языка C, альтернатива которым не столь очевидна.

Общая идея, объединяющая статьи этой главы, заключается в том, что все непринятые конструкции скорее были ориентированы на данные, назвать их объектно-ориентированными нельзя. Язык программирования Java обеспечивает мощную систему типизации, и потому предлагаемая замена в полной мере использует преимущества этой системы, добиваясь более высокого качества абстракции, чем имели соответствующие конструкции C.

Даже если вы решите пропустить эту главу, все же прочтите статью 21, в которой обсуждается шаблон перечисления, который заменяет конструкцию enum из языка C. Во время написания этой книги данный шаблон был мало известен, однако он имеет ряд преимуществ перед другими, широко используемыми сегодня методами.

Заменяйте структуру классом

Конструкция struct языка C не была принята в языке программирования Java потому, что класс выполняет все то же самое, что может делать структура, и даже более того. Структура группирует несколько полей данных в один общий объект, тогда как класс связывает с полученным объектом операции, а также позволяет скрывать поля данных от пользователей объекта. Иными словами, класс может инкапсулировать

encapsulate) свои данные в объекте, доступ к которому осуществляется только через его методы. Тем самым у разработчика появляется возможность менять внутреннее представление объекта (статья 12).

После первого знакомства с языком Java некоторые программисты, ранее пользовавшиеся языком C, приходят к заключению, что в некоторых случаях класс слишком тяжеловесен, чтобы заменить структуру, однако это не так. Вырожденный класс, состоящий исключительно из полей данных, примерно равнозначен структуре из языка C:

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

Поскольку доступ к таким классам осуществляется через поле данных, они лишены преимуществ инкапсуляции. Вы не можете поменять структуру такого класса, не изменив его API. Вы не можете использовать каких-либо инвариантов. Вы 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:

```
// Класс с инкапсулированной структурой
class Point {
    private float x;
    private float y;

    public Point(float x, float y) {

        this. x = x;
        this.y = y;
        }
    public float getX() { return x; }
    public float getY() { return y; }

    public void setX(float x) { this.x = x; }

    public void setY(float y) { this.y = y; }

    }
```

В отношении открытых классов борцы за чистоту языка программирования совершенно правы: если класс доступен за пределами пакета, то предусмотрительный программист создает соответствующие методы доступа, оставляя возможность изменения внутреннего представления этого класса. Если открытый класс показал клиенту свои поля данных, то всякая возможность менять это представление может быть потеряна, поскольку программный код клиентов открытого класса может оказаться где угодно.

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

Несколько классов в библиотеках для платформы Java нарушают совет, касающийся запрещения непосредственного доступа к полям открытого класса. В частности, это классы Point и Dimension из пакета java.awt. Не следует подражать этим классам, лучше рассматривать их как предупреждение. В статье 37 показано, как раскрытие внутреннего содержания класса Dimension привело к серьезным проблемам с производительностью, которые нельзя было разрешить, не затрагивая клиентов.

Замеияйте объединиеиие иерархией классов

В языке С конструкция union чаще всего служит для построения структур, в которых можно хранить более одного типа данных. Обычно такая структура содержит по крайней мере два поля: объединение (union) и тег (tag). Тег - это обыкновенное поле, которое используется для указания, какие из возможных типов можно хранить в объединении. Чаще всего тег представлен перечислением (unum) какого-либо типа. Структуру, которая содержит объединение и тег, иногда называют явным объединением (discriminated union).

в приведенном ниже примере на языке С тип shape_t - это явное объединение, которое можно использовать для представления как прямоугольника, так и круга. Функция area получает указатель на структуру shape_t и возвращает площадь фигуры либо -1. О, если структура недействительна:

```
/* Явное объединение */
#include "math.h"
typedef enum { RECTANGLE, CIRCLE } shapeType_t;
typedef struct {
   double length;
   double width; }
   rectangleDimensions_t;
```

```
typedef struct {
double radius;
} circleDimensions_t;

typedef struct {

shapeType_t tag;

union {
            rectangleDimensions_t rectangle;
            circleDimensions_t circle;
      } dimensions;
   }shape_t;

double area(shape_t *shape){

 switch(shape->tag) {
            case RECTANGLE: {
                  double length = shape->dimensions. rectangle.length;
                  double width = shape->dimensions. rectangle.width;
                  return length * width;
            }
            case CIRCLE: {
                  double r = shape->dimensions.circle.radius;
                   return M_PI * (r*r); }
  default: return -1.0;
/* Неверный тег */

            }
}
```

Создатели языка программирования Java решили исключить конструкцию union, поскольку имеется лучший механизм определения типа данных, который можно использовать для представления объектов разных типов: создание подклассов. Явное объединение в действительности является лишь бледным подобием иерархии классов.

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

Далее, для каждого типа, который может быть представлен объединением, определите неабстрактный подкласс корневого класса. В примере такими типами являются круг и прямоугольник. В каждый подкласс поместите те поля данных, которые характерны для соответствующего типа. Так, радиус является характеристикой круга,

а длина и ширина описывают прямоугольник. Кроме того, в каждый подкласс поместите соответствующую реализацию для всех абстрактных методов в корневом классе. Представим иерархию классов для нашего примера явного объединения:

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

      Rectangle(double length, double width) {

            this.length = length;
            this.width = width;
            }
      double a rea() { return length * width; }
      }
```

По сравнению с явным объединением, иерархия классов имеет множество преимуществ. Главное из них заключается в том, что иерархия типов обеспечивает их безопасность. В данном примере каждый экземпляр класса Shape является либо правильным экземпляром Circle, либо прав ильным экземпляром Rectangle. Поскольку язык С не устанавливает связь между тегом и объединением, возможно создание структуры shape_t, в которой содержится мусор. Если в теге указано, что shape_t соответствует прямоугольнику, а в объединении описывается круг, все пропало. И даже если явное объединение инициализировано правильно, оно может быть передано не той функции, которая соответствует значению тега.

Второе преимущество иерархии классов заключается в простоте и ясности программного кода. Явное объединение загромождено шаблонами: декларация типа перечисления, декларация поля тега, поле переключения тега, обработка непредвиденных значений тега и т. Д. Программный код объединения неудобно читать даже из-за того, что операции для различных типов перемешаны, а не разделены по типу.

Третьим преимуществом иерархии классов является простота ее расширяемости, даже если над ней работают независимо несколько групп программистов. для расширения иерархии классов достаточно добавить в нее новый подкласс. Если вы забыли переопределить один из абстрактных методов суперкласса, компилятор напомнит об этом недвусмысленным образом. для расширения явного объединения в С потребуется

доступ к его исходному коду. Придется добавить новое значение в представленный тип перечисления, а также новую альтернативу в оператор switch для каждой операции в объединении. И наконец, нужно снова компилировать программу. И если для какого-либо метода вы забыли указать новую альтернативу, это не будет обнаружено до тех пор, пока программа не будет запущена на исполнение, и лишь при условии, что вы тщательно проверяете неидентифицированные значения тега и генерируете соответствующее сообщение об ошибке.

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

```
class Square extends Rectangle {
Square (double side) {
super(side, side);
}
      double side() {
            return length; // Возвращает длину или, что то же самое, ширину
            }
 }
```

Иерархия классов, представленная в этом примере, не является единственно возможной для явного объединения. Данная иерархия содержит несколько конструкторских решений, заслуживающих особого упоминания. для классов в иерархии, за исключением класса Square, доступ к полям обеспечивается непосредственно, а не через методы доступа. Это делается для краткости, и было бы ошибкой, если бы классы были открытыми (статья 19). Указанные классы являются ·неизменяемыми, что не всегда возможно, но это обычно хорошее решение (статья 13).

Поскольку в языке программирования Java нет конструкции union, вы можете решить, что реализация явного объединения не представляет опасности. Однако и здесь можно получить программный код, имеющий те же самые недостатки. Как бы вы ни хотели написать класс с явным полем тега, подумайте, не стоит ли этот тег исключить, а сам класс заменить иерархией классов.

Другой вариант использования конструкции union из языка С не связан с явными объединениями и касается внутреннего представления элемента данных, что нарушает систему типизации. Например, следующий фрагмент программы на языке С печатает машинно-зависимое шестнадцатеричное представление поля типа float:

```
union {
   float f;
   int bits;
   }sleaze;
```

```
sleaze.f = 6.699e-41;
        /* Помещает данные в одно из полей объединения ... */
          printf("%x\n", sleaze.bits); /* ... и читает их из другого */
```

Это непортируемое решение можно использовать, особенно в системном программировании, но его нельзя реализовать в языке программирования Java. Фактически это противоречит самому духу языка, который гарантирует контроль типов и готов на все, чтобы изолировать программистов от машинно-зависимого внутреннего представления программ.

В пакете java.lang есть методы преобразования чисел с плавающей точкой в двоичную форму, однако в целях переносимости эти методы определены в соответствии со строго регламентированным двоичным представлением. Следующий фрагмент кода, который почти равнозначен предыдущему фрагменту на языке С, гарантирует, что программа будет всегда печатать один и тот же результат независимо от того, где она запущена:

```
Systет.out.println(
    Integer.toHexString(Float.floatTolntBits(6. 69ge-41f)));
```

Заменяйте конструкцию enum классом

Конструкция enum языка С отсутствует в языке программирования Java. Обычно эта конструкция определяет тип, соответствующий перечислению (enumerated type), т. е. тип, допустимыми значениями которого являются константы из фиксированного набора. К сожалению, конструкция enum не очень хорошо определяет перечисления. Она лишь задает набор именованных целочисленных констант, не обеспечивая ни безопасности типов, ни минимального удобства использования. Так, в С допустимы не только записи:

```
typedef enum { FUJI, PIPPIN, GRANNY_SMITH } apple_t; /* Сорта яблок*/
typedef enum { NAVEL, TEMPLE, BLOOD } оrапge_t; /* Сорта апельсинов*/
оrапge_t myFavorite = PIPPIN; /* Путает яблоки и апельсины */
```

но и этот кошмар:

```
orange_t x = (FUJI - PIPPIN)/TEMPLE;          /* Яблочный соус какой-то! */
```

Конструкция enum не образует пространства имен для генерируемых ею констант. Поэтому следующая декларация, в которой одно из названий используется вновь, вступает в конфликт с предыдущей декларацией orange_t:

```
typedef enum { BLOOD, SWEAT, TEARS } fluid_t;  /* Это не группа, а жидкости */
```

Типы, декларируемые конструкцией enum, ненадежны. Если не позаботиться о сохранении значений всех имеющихся констант, можно получить непредсказуемые изменения в работе системы, если добавить в перечисление новые константы, но не выполнить повторную компиляцию клиентов. Разные группы разработчиков не могут добавлять в перечисление новые константы независимо друг от друга, поскольку новые константы перечисления скорее всего будут конфликтовать между собой. Конструкция enum не предоставляет простых способов перевода констант перечисления в печатные строки или подсчета количества констант в каком-либо типе.

К сожалению, в языке программирования Java большое распространение имеет шаблон перечислений, который, как показано ниже, перенимает указанные недостатки конструкции enum из языка C:

```
// Шаблон перечисления·целых чисел (int enum)

 // весьма сомнительныйl
public class PlayingCard {
    public static final int SUIT_CLUBS        =0;
    public static final int SUIT_DIAMONDS   =1’
    public static final int SUIT_HEARTS      =2’
    public static final int SUIT_SPADES      =3;
```

Вы можете встретить другой вариант этого шаблона, когда вместо констант int применяются константы String. Его тоже не следует использовать. Хотя для своих констант он передает печатные строки, это может привести к проблемам с производительностью, поскольку связано с операцией сравнения строк. Более того, неискушенные пользователи могут получить программный код клиентов, жестко при вязанный к строковым константам, вместо того, чтобы использовать названия соответствующих полей. И если в жестко заданные строковые константы вкрадутся опечатки, это не будет выявлено на этапе компиляции и повлечет сбои при выполнении программы.

К счастью, язык программирования Java предлагает альтернативное решение, которое лишено недостатков распространенных шаблонов для int и String и имеет множество преимуществ. Это шаблон, называемый перечислением типов (typesafe enum). К сожалению, он пока мало известен. Основная идея шаблона проста: определите класс, представляющий отдельный элемент перечисления, но не создавайте для него никаких открытых конструкторов. Вместо них создайте поля public static final, по одному для каждой константы перечисления. Покажем, как этот шаблон выглядит в простейшем случае:

```
// Шаблон typesafe enum
public class Suit {
    private final String name;
    private Suit(String name) { this.name = name; }
```

```
public String toString() { return name;}
public static final Suit CLUBS= new Suit("clubs"); // трефы
publiC static final Suit DIAMONDS = new Suit("diamonds"); // бубны
public static final Suit HEARTS= new Suit("hearts"); // черви
public static final Suit SPADES= new Suit("spades"); // пики
}
```

Поскольку клиенты не могут создавать экземпляров данного класса или расширять его, не будет никаких объектов этого типа, за исключением тех, что доступны в полях public static final. И хотя класс даже не декларирован как final, расширять его невозможно: конструктор подкласса должен вызывать конструктор суперкласса, а такой конструктор ему недоступен.

Как и подразумевает название, шаблон typesafe enum обеспечивает безопасность типов уже на стадии компиляции. Декларируя метод с параметром типа Suit, вы имеете гарантию того, что любая переданная вам ссылка на объект, отличная от null, представляет одну из четырех правильных карточных мастей. Любая попытка передать объект неправильного типа будет обнаружена на стадии компиляции так же, как и любая попытка присвоить результат вычислений, соответствующий одному перечислению, переменной, которая соответствует другому типу перечисления: Различные классы перечислений с одинаковыми названиями констант могут мирно сосуществовать благодаря тому, что каждый класс имеет собственное пространство имен.

Добавлять константы в класс перечисления можно, не компилируя вновь всех его клиентов, поскольку открытые статические поля со ссылкой на объекты, в которых содержатся константы перечисления, образуют уровень изоляции между клиентом и классом перечисления. Сами константы никогда не компилируются в классе клиента, как это было в случае с более распространенным шаблоном, использующим int или String.

Поскольку перечисления типов - это вполне самостоятельные классы, допустимо переопределение метода toString таким образом, чтобы можно было преобразовывать значения в печатные строки. При желании вы сможете сделать еще один шаг в этом направлении и с помощью стандартных средств связать класс перечисления с региональными настройками. Заметим, что названия строк используются лишь в методе toString, при проверке равенства они не применяются, поскольку реализация метода equals, унаследованная от класса Object, выполняет сравнение, проверяя равенство ссылок.

В общем случае вы можете усилить класс перечислений любым методом, который покажется вам подходящим. Например, наш класс Sui t мог бы выиграть от добавления метода, который возвращает цвет или изображение масти. Класс может начать свою жизнь как простое перечисление и со временем развиться в абстракцию с полным набором свойств.

Поскольку в класс перечисления разрешается добавлять любые методы, его можно заставить реализовать любой интерфейс. Предположим, вы хотите, чтобы класс Suit реализовывал интерфейс Comparable, и клиенты могли сортировать сдачу

карт в бридже по мастям. Представим слегка видоизмененный первоначальный шаблон, который поддерживает такой трюк. Статическая переменная nextOrdinal используется для того, чтобы каждому экземпляру класса в момент его создания назначать порядковый номер. Эти номера применяются в методе compareTo для упорядочения экземпляров.

```
// Перечисление, использующее порядковые номера
public class Suit implements Comaprable {
        private final String name;

        // Порядковый номер следующей масти

        private static int nextOrdinal = 0;

        // Назначение порядкового номера данной масти

         private final int ordinal = nextOrdinal++;
         private Suit(String name) { this.name = name; }
public String toString()  {return name; }

public int compareTo(Object o) {
return ordinal – ((Suit)O).ordinal;

public String toString() { return name;}

public static final Suit CLUBS= new Suit("clubs"); // трефы

publiC static final Suit DIAMONDS = new Suit("diamonds"); // бубны

public static final Suit HEARTS= new Suit("hearts"); // черви

public static final Suit SPADES= new Suit("spades"); // пики

}
```

Поскольку константы перечисления являются объектами, вы можете помещать их в коллекции. Допустим, вы хотите, чтобы класс Suit передавал неизменяемый перечень мастей в обычном порядке. Достаточно добавить в этот класс следующие две декларации полей:

```
private static final Suit[] PRIVATE_VALUES =
      { CLUBS, DIAMONDS, HEARTS, SPADES };
public static final List VALUES =
     Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

В отличие от простейшего шаблона typesafe enum, классы представленной формы, использующей порядковые номера, можно сделать сериализуемыми (serializable) (см. главу 10), при ложи в для этого минимум усилий. Недостаточно добавить в декларацию класса слова implements Serializab'le, нужно еще предоставить метод readResolve (статья 57):

```
private Object readResolve() throws ObjectStreamExceptin {
return PRIVATE_VALUES[ordinal];        // Канонизация
}
```

Этот метод, автоматически вызываемый системой сериализации, предупреждает появление в результате десериализации дублирующих констант. Это гарантирует, что каждая константа в перечислении будет представлена одним единственным объектом, и, следовательно, не нужно переопределять метод Obj ect. equals. Без этого условия метод Obj ect. equals давал бы отрицательный результат, сравнивая две равные, но неидентичные константы перечисления. Заметим, что метод readResolve ссылается на массив PRIVATE_VALUES, а потому вы должны декларировать этот массив, даже если решили не предоставлять клиентам список VALUES. Заметим также, что поле пате в методе readResolve не используется, а потому его можно и даже нужно исключить из сериализованной формы.

Полученный в результате класс не вполне надежен: конструкторы любых вновь добавленных значений должны ставиться после уже имевшихся с тем, чтобы ранее сериализованные экземпляры класса при десериализации не поменяли свое значение. Это происходит потому, что в сериализованной форме (статья 55) для константы перечисления хранится лишь ее порядковый номер. И если у константы перечисления поменяется порядковый номер, то константа, сериализованная с прежним номером, при десериализации получит новое значение.

Каждой константе, которая используется только в пределах того пакета, где содержится класс перечисления, может быть сопоставлена одна или несколько ахем поведения программы. Такие схемы лучше всего реализовать в классе как методы, доступные только в пределах пакета. В результате за каждой константой перечисления выстраивается скрытый набор схем поведения, что дает возможность пакету, содержащему перечисление, правильно реагировать на получение соответствующей константы.

Если класс перечисления имеет методы, работа которых кардинально меняется при замене одной константы класса на другую, то вы обязаны для каждой такой константы использовать отдельный закрытый класс либо анонимный внутренний класс. Это позволяет каждой константе иметь собственную реализацию всех таких методов и обеспечивает автоматический вызов нужной реализации. Альтернативный подход заключается в том, чтобы представить каждый такой метод как дерево ветвлений, функционирование которого зависит от того, для какой константы был вызван метод. Подобный подход уродлив, подвержен ошибкам и вряд ли способен обеспечить производительность, сравнимую с автоматической диспетчеризацией методов в виртуальной машине Java.

Рассмотрим класс перечисления, иллюстрирующий оба приема, о которых говорилось выше. Класс Operation описывает работу калькулятора с четырьмя основными функциями. Все, что вы можете делать с константой типа Operation за пределами пакета, в котором этот класс определен, - это вызывать методы класса Object: toString, hashCode, equals и т. д. Внутри же пакета вы можете выполнить арифметическую операцию, соответствующую константе. По-видимому, в пакете будет представлен некий объект высокого уровня, соответствующий калькулятору, который будет предоставлять клиенту один или несколько методов, получающих в качестве параметра константу типа Operation. Заметим, что сам Operation является абстрактным

классом, содержащим один единственный абстрактный метод eval, который доступен только в пределах пакета и выполняет соответствующую арифметическую операцию. для каждой константы определен анонимный внутренний класс, так что для каждой константы можно определить собственную версию метода eval:

```
// Перечисление типов, схемы поведения которого
 // закреплены за константами
publiC abstract class Operation {
    private final String name;
    Operation(String name) { this.name = пате; }
    public String toString() { return this.name; }
    //Выполняет арифметическую операцию,
    // представленную указанной константой
    abstract double eval(double x, double y);
    public static final Operation PLUS = new Operation("+") {
    double eval(double x, double y) { return x + y; }
    } ;
    public static final Operation MINUS = new Operation("-") {
    double eval(double x, double y) { return x - y; }
    } ;
    public static final Operation TIMES : new Operation("*") {
    double eval(double x, double y) { return x * y; }
    } ;
    public static final Operation DIVIDED_BY = new Operation("/") {
    double eval(double x, double y) { return x / y; }
    } ;
```

Вообще говоря, по производительности перечисления типов сравнимы с перечислениями целых констант. Два разных экземпляра для класса перечисления typesafe enum никогда не смогут представить клиенту одно и то же значение, а потому для проВерки логического равенства используется быстрая проверка тождественности ссылок. Клиенты класса перечисления typesafe enum могут использовать оператор == вместо метода equals. Гарантируется, что результаты будут те же самые, а сам оператор ==, возможно, будет работать быстрее.

Если класс перечисления используется широко, его следует сделать классом верхНего уровня. Если работа с перечислением связана с определенным классом верхнего уровня, перечисление следует сделать статическим классом-членом указанного класса верхнего уровня (статья 18). Например, класс java. math. BigDecimal содержит набор

констант перечисления типа int, соответствующих различным способам округления дробной части числа (rounding modes). Эти способы округления образуют полезную абстракцию, которая в сущности не связана с классом BigDecimal. Реализовать их лучше в самостоятельном классе j ava. math. RoundingMode. В результате любой программист, которому понадобились различные режимы округления, мог бы воспользовался этой абстракцией, что привело бы к лучшей согласованности между различными
API.

Базовый шаблон перечисления типов, проиллюстрированный выше в реализациях класса Suit, зафиксирован: пользователи не могут дополнять перечисление новыми элементами, поскольку у класса нет конструкторов, которые были бы доступны пользователю. Фактически это делает данный класс окончательным независимо от того, декларирован ли он с модификатором final или нет. Как правило, это именно то, чего вы ждете от класса, но иногда вам необходимо сделать класс перечисления расширяемым. Например, это может потребоваться, когда вы используете класс перечисления для представления различных форматов кодирования изображений, но хотите, чтобы третьи лица могли добавлять в него поддержку новых форматов.

Для того чтобы сделать класс перечисления расширяемым, создайте защищенный конструктор. Тогда другие разработчики смогут расширять этот класс и дополнять новыми константами свои подклассы. Вам не нужно беспокоиться о конфликтах между константами перечисления, как в случае применения шаблона int епит. Расширяемый вариант шаблона typesafe епит использует преимущества собственного пространства имен пакета с тем, чтобы для расширяемого перечисления создать "магически управляемое" пространство имен. Различные группы разработчиков могут расширять это перечисление, ничего не зная друг о друге,- конфликта между их расширениями не возникнет.

Простое добавление элемента в расширяемое перечисление еще не является гарантией того, что этот элемент будет иметь полную поддержку: методы, принимающие элемент перечисления, должны учитывать возможность получения элементов, неизвестных программисту. Если целесообразность использования сложного дерева ветвлений для фиксированного перечисления сомнительна, то для расширяемых перечислений это вообще смертельно, поскольку каждый раз, когда программист расширяет тип, соответствующий перечислению, ветвление само собой нарастать не будет.

Один из способов решения этой проблемы заключается в том, чтобы снабдить класс перечисления всеми теми методами, которые необходимыми для описания схем, соответствующих константам этого класса. Метод, которым клиенты класса не пользуются, следует делать защищенным, однако должна оставаться возможность его переопределения в подклассах. Если такой метод не имеет приемлемой реализации, предлагаемой по умолчанию, его следует декларировать не только как защищенный, но и как абстрактный.

В расширяемом классе перечисления полезно пере определить методы equals и hashCode новыми окончательными методами, которые обращаются к методам из класса Object. Тем самым гарантируется, что ни в одном подклассе эти методы не

будут случайно переопределены, благодаря чему все равные объекты типа перечисления будут к тому же идентичны (a.equals(b) возвращает true тогда и только тогда, когда а == Ь).

```
// Методы, защищенные от переопределения
public final boolean equals(Object that) {
return super.equals(that);
}
public final int hashCode() {
return super.hashCode();
}
```

Заметим, что представленный вариант расширения не совместим с вариантом, обеспечивающим сравнение: если вы попытаетесь объединить их, то схема упорядочения элементов подкласса будет зависеть от очередности инициализации подклассов, а она может меняться от программы к программе, от запуска к запуску.

Приведенный расширяемый вариант шаблона перечисления совместим с вариантом, обеспечивающим сериализацию, однако объединение этих вариантов требует некоторой осторожности. Каждый подкласс должен назначить собственные порядковые номера и использовать свой собственный метод readResolve. В сущности, каждый класс отвечает за сериализацию и десериализацию своих собственных экземпляров. Для пояснения представим вариант класса Ope ration, который был исправлен таким образом, чтобы быть и расширяемым, и сериализуемым:

```
// Сериализуемый и расширяемый класс перечисления
public abstract class Operation implements Serializable {
 private final transient String name;
      protected Operation(String name) { this.name = name; }

      public static Operation PLUS = new Operation("+") {
            protected double eval(double x, double y) { return x+y; }
      } ;
      public static Operation MINUS = new Operation("-") {
       protected double eval(double x, double y) { return x-y; }
      } ;
      public static Operation TIMES = new Operation("*") {
      protected dQuble eval(double x, double y) { return x*y; }
      } ;
      public static Operation DIVIDE = new Operation("/") {
       protected double eval(double x, double y) { return x/y; }
      } ;
```

```
// Выполнение арифметической операции,

// представленной данной константой
protected abstract double eval(double x, double y);

public String toString() { return this.name; }
// Препятствует переопределению в подклассах
 // метода Object.equals
public final boolean equals(Object that) {
return super.equals(that);
}

public final int hashCode() {

      return super.hashCode();
}
// Следующие четыре декларации необходимы для сериализации
private static int nextOrdinal = o;
private final int ordinal = nextOrdinal++;
private static final Operation[] VALUES =
      { PLUS. MINUS, TIMES, DIVIDE};
Object readResolve() throws ObjectStreamException {
      return VALUES[ordinal];      // Канонизация
                  }
      }
```

Представим подкласс класса Operation, в который добавлены операции логарифма и экспоненты. Причем этот подкласс может находиться за пределами того пакета, где содержится исправленный класс Operation. Данный подкласс может быть открытым и сам может быть расширяемым. Возможно мирное сосуществование различных подклассов, написанных независимо друг от друга.

```
// Подкласс расширяемого сериализуемого перечисления
abstract class ExtendedOperation extends Operation {

  ExtendedOperation(String name) { super(name); }
    public static Operation LOG = new ExtendedOperation("log") {
    protected double eval(double x, double y) {
              return Math.log(y) / Math.log(x);

    } ;
    public static Operation EXP = new ExtendedOperation("exp") {
    protected double eval(double x, double y) {
              return Math.pow(x, y);
              }

    } ;
```

```
// Следующие четыре декларации необходимы для сериализации

 private static int nextOrdinal = 0;
private final int ordinal = nextOrdinal++;
private static final Operation[] VALUES = { LOG, EXP };
Object readResolve() throws ObjectStreamException {
      return VALUES[ordinal];      // Канонизация
      }
}
```

Заметим, что в представленных классах методы readResolve показаны как доступные в пределах пакета, а не закрытые. Это необходимо потому, что экземпляры классов Operation и ExtendedOperation фактически являются экземплярами анонимных подклассов, а потому закрытые методы readReasolve были бы бесполезны (статья 57).

Шаблон typesafe епит, по сравнению с шаблоном int enum, имеет несколько недостатков. По-видимому, единственным серьезным его недостатком является то, что он не так удобен для объединения констант перечисления в наборы. В перечислениях целых чисел для констант традиционно выбираются значения в виде различных неотрицательных степеней числа два, сам же набор представляется как побитовое OR соответствующих констант:

```
// Вариант шаблона int впит с битовыми флажками
      public static final int SUIT_CLUBS         =0;
      public static final int SUIT_DIAMONDS   =1'
      public static final int SUIT_HEARTS       =2'
      public static final int SUIT_SPADES       =3;
      public static final int SUIT_BLACK ; SUIT_CLUBS I SUIT_SPADES;
```

Набор констант перечисления, представленный таким образом, является кратким и чрезвычайно быстрым. Для набора констант перечисления typesafe enum вы можете использовать универсальную реализацию набора, заимствованную из Collections Framework, однако такое решение не является ни кратким, ни быстрым:

```
Set blackSuits ; new HashSet();
blackSuits.add(Suit.CLUBS);
blackSuits.add(Suit.SPADES);
```

Хотя наборы констант перечисления typesafe eum вряд ли можно сделать такими же компактными и быстрыми, как наборы констант перечисления int enum, указанное неравенство можно уменьшить путем специальной реализации набора Set, которая обслуживает элементы только определенного типа, а для самого набора использует Внутреннее представление в виде двоичного вектора. Такой набор лучше реализовывать в том же пакете, где описывается тип его элементов. Это позволяет через поля или методы, доступные только в пределах пакета, получать доступ к битовому значению, которое соответствует внутреннему представлению каждой константы в перечислении. Имеет смысл создать открытые конструкторы, которые в качестве параметров

принимают короткие последовательности элементов, что делает возможным применение идиом следующего типа:

```
hand.discard( new SuitSet( Suit.CLUBS, Suit.SPADES));
```

Небольшой недостаток констант перечисления typesafe enums, по сравнению с перечислением int enum, заключается в том, что для них нельзя использовать оператор switch, поскольку они не являются целочисленными. Вместо этого вы применяете оператор if, например, следующим образом:

```
if (suit == Suit.CLUBS) {
}else if (suit == Suit.DIAMONDS) {
}else if (suit == Suit.HEARTS) {
}else if (suit == Suit.SPADES) {
}else {
throw new NullPointerException("Null Suit");      //suit = = null
}
```

Оператор if работает не так быстро, как оператор switch, однако эта разница вряд ли будет существеной. Более того, при работе с константами перечисления typesafe enum потребность в большом ветвлении должна возникать редко, поскольку они подчиняются автоматической диспетчеризации методов, осуществляемой JVM, как показано в примере с Operation.

Еще один недостаток перечислений связан с потерей места и времени при загрузке классов перечислений и создании объектов для констант перечисления. Если отбросить такие стесненные в ресурсах устройства, как сотовые телефоны и тостеры, эта проблема на практике вряд ли будет заметна.

Подведем итоги. Преимущества перечислений typesafe enum перед перечислениями int eenum огромны, и ни один из недостатков не кажется непреодолимым, за исключением случая, когда перечисления применяются прежде всего как элемент набора либо в среде, серьезно ограниченной в ресурсах. Таким образом, когда обстоятельства требуют введения перечисления, на ум сразу же должен приходить шаблон typesafe enum. API, использующие перечисления typesafe enum, гораздо удобнее для программиста, чем API, ориентированные на перечисления int enum. Единственная причина, по которой шаблон typesafe enum не применяется более интенсивно в интерфейсах API для платформы Java, заключается в том, что в то время, когда писались многие из этих API, данный шаблон еще не был известен. Наконец, стоит повторить еще раз, что потребность в перечислениях любого вида должна возникать сравнительно редко, поскольку большинство этих типов после создания подклассов стали устаревшими (статья 20).

Указатель на функцию заменяйте кпассом и интерфейсом

Язык С поддерживает указатели на функции (function pointer), что позволяет программе хранить и передавать возможность вызова конкретной функции. Указатели на функции обычно применяются для того, чтобы разрешить клиенту, вызвавшему функцию, уточнить схему ее работы, для этого он передает ей указатель на вторую функцию. Иногда это называют обратным вызовом (callback). Например, функция qsort из стандартной библиотеки С получает указатель на функцию-компаратор (comparator), которую затем использует для сравнения элементов, подлежащих сортировке. Функция-компаратор принимает два параметра, каждый из которых является указателем на некий элемент. Она возвращает отрицательное целое число, если элемент, на который указывает первый параметр, оказался меньше элемента, на который указывает второй параметр, нуль, если элементы равны между собой, и положительное целое число, если первый элемент больше второго. Передавая указатель на различные функции-компараторы, клиент может получать различный порядок сортировки. Как демонстрирует шаблон Strategy [Сатта95, стр. 315], функция-компаратор представляет алгоритм сортировки элементов.

В языке Java указатели на функции отсутствуют, поскольку те же самые возможности можно получить с помощью ссылок на объекты. Вызывая в объекте некий метод, действие обычно производят над самим этим объектом. Между тем можно построить объект, чьи методы выполняют действия над другими объектами, непосредственно предоставляемыми этим методам. Экземпляр класса, который предоставляет клиенту ровно один 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 s1.1ength() - s2.1ength();
                }
    }
```

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

Как часто бывает с классами конкретных методик сравнения, класс StringLengthComparator не имеет состояния: у него нет полей, а потому все его экземпляры функционально эквивалентны друг другу. Таким образом, во избежание расходов на создание ненужных объектов можно сделать этот класс синглтоном (статьи 4 и 2):

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

Для того чтобы передать методу экземпляр класса StringLengthComparator, нам необходим соответствующий тип параметра. Использовать непосредственно тип StringLengthComparator нехорошо, поскольку это лишит клиентов возможности выбирать какие-либо другие алгоритмы сравнения. Вместо этого следует определить интерфейс Comparator и переделать класс StringLengthComparator таким образом, чтобы он реализовывал этот интерфейс. Другими словами, необходимо определить интерфейс методики сравнения (strategy interface), который должен соответствовать классу конкретной стратегии:

```
// Интерфейс методики сравнения
public interface Comparator {
    public int compare(Object 01, Object 02);
    }
```

Оказывается, что представленное определение интерфейса Comparator есть в пакете java.util. Никакого волшебства в этом нет, вы могли точно так же определить его сами. Так, для того чтобы можно было сравнивать не только строки, но и другие объекты, метод compare в интерфейсе принимает параметры типа Object, а не String. Следовательно, приведенный выше класс Str1ngLengthComparator необходимо слегка изменить, чтобы реализовать интерфейс Comparator. Перед вызовом метода length параметры типа Object нужно привести к типу String.

Классы конкретных методик сравнения часто создаются с помощью анонимных классов (статья 18). Так, следующий оператор сортирует массив строк по их длине:

```
Arrays.sort(stringArray, new Comparator() {
    public int compare(Object 01, Object 02) {
    String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length(); }

});
```

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

```
// Предоставление конкретной методики сравнения

class Host {
        // Основная часть класса опущена
    private static plass StrLenCmp
            implements Comparator, Serializable {
        public int compare(Object o1, Object o2) {
            String s1 = (String)o1;
            String s2 = (String)o2;
            return s1.1ength() - s2.length(); }

    }
    // Возвращаемый компаратор является сериализуемым
     public static final Comparator
         STRING_LENGTH_COMPARATOR = new StrLenCmp(); }
```

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

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

Глава 6

Методы

В данной главе рассматривается несколько аспектов проектирования методов: как обрабатывать параметры и возвращаемые методом значения, как строить сигнатуры методов и как документировать методы. Значительная часть материала относится как к методам, так и к конструкторам. Особое внимание уделяется удобству, устойчивости и гибкости программ.

Проверяйте достоверность параметров

Большинство методов и конструкторов имеет ограничения на то, какие значения могут быть переданы с параметрами. Например, нередко указывается, что индексы должны быть неотрицательными, а ссылки на объекты отличны от null. Вы обязаны четко документировать все эти ограничения и начинать метод с их проверки. Это частный случай более общего принципа: стараться выявлять ошибки как можно скорее после того, как они произойдут. В противном случае обнаружение ошибки станет менее вероятным, а определение источника ошибки - более трудоемким.

Если методу передано неверное значение параметра, но перед началом обработки он проверяет полученные параметры, то вызов этого метода быстро и аккуратно завершится с инициированием соответствующего исключения. Если же метод не проверяет своих параметров, может произойти несколько событий. Метод может завершиться посередине обработки, инициировав непонятное исключение. Хуже, если метод завершится 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 инициироваться, когда значения параметров нарушают ограничения, используйте тег @throws генератора документации Javadoc (статья 44). Как правило, это будет исключение IllegalArgumentException, IndexOutOfBoundsException или NullPointe rException (статья 42). После того, как вы документировали ограничения для параметров метода и исключения, которые будут инициироваться в случае нарушения ограничений, установить эти ограничения для метода не составит труда. Приведем типичный пример:

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

Если метод не предоставляется в распоряжение пользователей, то вы как автор пакета контролируете все условия, при которых этот метод вызывается, а потому можете и обязаны убедиться в том, что ему будут передаваться только правильные значения параметра. Методы, не являющиеся открытыми, должны проверять свои параметры, используя утверждения (assertion), а не с помощью обычных проверок. В случае применения платформы Java, поддерживающей утверждения (версия 1.4 и выше), вы должны пользоваться конструкцией assert, в противном случае следует применять собственный механизм проверки утверждений.

Особенно важно проверять правильность параметров, которые не используются методом, а откладываются для обработки в дальнейшем. Например, рассмотрим статический метод генерации из статьи 16, который получает массив целых чисел и возвращает представление этого массива в виде экземпляра List. Если клиент метода передаст значение null, метод инициирует исключение NullPointerException, поскольку содержит явную проверку. Если бы проверка отсутствовала, метод возвращал бы ссылку на вновь Сформированный экземпляр List, который будет инициировать исключение NullPointerException, как только клиент попытается им воспользоваться. К сожалению, к тому моменту определить происхождение экземпляра List будет уже трудно, что может значительно усложнить задачу отладки.

Частным случаем принципа, требующего проверки параметров, которые должны быть сохранены для использования в дальнейшем, являются конструкторы. Для конструкторов очень важно проверять правильность параметров, чтобы не допустить создания объектов, нарушающих инварианты соответствующего класса.

Существуют исключения из правила, обязывающего перед выполнением вычислений проверять параметры метода. Важное значение имеет ситуация, когда явная проверка 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 списке должны быть взаимно сравнимы. В ходе его сортировки каждый объект в нем будет сравниваться с каким-либо другим объектом из того же списка. Если объекты не будут взаимно сравнимы, в результате одного из таких сравнений будет инициировано исключение ClassCastException, а это именно то, что должен делать в таком случае метод sort. Таким образом, нет смысла выполнять упреждающую проверку взаимной сравнимости элементов в списке. Заметим, однако, что неразборчивое использование данного подхода может привести к потере такого качества, как атомарность сбоя (статья 46).

Иногда в ходе обработки неявно осуществляется требуемая проверка некоторых параметров, однако когда проверка фиксирует ошибку, инициируется совсем не то исключение. Другими словами, исключение, которое инициируется в ходе обработки и связано с обнаружением неверного значения параметра, не соответствует тому исключению, которое, согласно вашему описанию, должно инициироваться этим методом. В таких случаях для преобразования внутреннего исключения в требуемое вы должны использовать ~диому трансляции исключении, описанную в статье 43.

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 накладывается на параметры, тем лучше, при условии, что метод для каждого полученного значения параметра может сделать что-то разумное. Часто, однако, некоторые ограничения обусловлены реализацией абстракций.

Подведем итоги. 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.

При необходимости создавайте резервные копии

Одно из особенностей, благодаря которой работа с языком программирования Java доставляет такое удовольствие, является его *безопасность.* Это означает, что в отсутствие машинно-зависимых методов (native method) он неуязвим по отношению

к переполнению буферов и массивов, к неконтролируемым указателям, а таКже к 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 будете изолированы от других классов, если не приложите со своей стороны не которого усилия. Вы должны писать программы с защитой, исходя из предположения, что клиенты вашего класса будут предпринимать все возможное для того, чтобы разрушить его инварианты. Это действительно так, когда кто-то пытается взломать систему безопасности. Однако скорее всего вашему классу придется иметь дело с непредвиденным 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 неизменного периода времени:

```
// Неправильный класс "неизменяемого"
public final class Period {
      private final Date start;
      private final Date end;
      /**
       * @рагат start - начало периода.
       * @рагат end - конец периода; не должен предшествовать началу.
       * @throws IllegalArgumentException. если start позже, чем end.
       * @throws"NullPointerException, если start или end равен null.
       */
   public Period(Date start, Date end) {
   if (start.compareTo(end) > 0)
                throw new IllegalArgumentException(start + " after " + end);
this.start = start;
this.end = end;
}
             public Date start() {
             return start;   }
```

```
public Date end() {

return end;   }

        // Остальное опущено

        }
```

На первый взгляд может показаться, что это неизменяемый класс, который успешно выполняет условие, заключающееся в том, что началу периода не предшествует его же конец. Однако, воспользовавшись изменяемостью объекта Date, можно с легкостью нарушить этот инвариант:

```
// Атака на содержимое экземпляра Period
 Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);      // Изменяет содержимое объекта p!
```

Для того чтобы защитить содержимое экземпляра Period от нападений такого типа, для каждого изменяемого параметра конструктор должен создавать резервную копию (defensive copy) и использовать именно эти копии, а не оригинал, как составные части экземпляра Period:

```
// Исправленный конструктор:
// создает резервные копии для параметров
public Period(Date start, Date end) {
      this.start = new Date(start.getTime());
      this.end = new Date(end.getTime());
      if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start +" after"+ end);  }
```

С новым конструктором описанная ранее атака уже не может воздействовать на экземпляр Period. Заметим, что резервные копии создаются до проверки правильности параметров (статья 23), 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.

Заметим также, что для создания резервных копий мы не пользовались методом clone из класса Date. Поскольку Date не является окончательным классом, нет гарантии, что метод clone возвратит объект именно класса java.util.Date - он может вернуть экземпляр ненадежного подкласса, созданного специально для нанесения ущерба. Например, такой подкласс может записывать в закрытый статический список

ссылку на экземпляр в момент создания последнего, а затем предоставить злоумышленнику доступ к этому списку. В результате злоумышленник получит полный контроль над всеми этими экземплярами. Чтобы предотвратить атаки такого рода, не используйте метод clone для создания резервной копии параметра, который имеет тип, позволяющий ненадежным партнерам создавать подклассы.

Обновленный конструктор успешно защищает от вышеописанной атаки, ,однако все равно остается возможность модификации экземпляра Period, поскольку его методы предоставляют доступ к его внутренним частям, которые можно поменять:

```
// Вторая атака на содержимое экземпляра Period
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Изменяет внутренние данные p!
```

для защиты от второй атаки модифицируйте методы доступа таким образом, чтобы возвращались

резервные копии изменяемых внутренних полей.

```
// Исправленные методы доступа:
// создаются резервные копии внутрених полей
public Date start() {
      return (Date) start.clone(); }

public Date end() {
      return (Date) end.clone();  }
```

Получив новый конструктор и новые методы доступа, класс Period стал действительно неизменяемым. Теперь некомпетентному программисту или злоумышленнику не удастся нарушить инвариант, гласящий, что начало периода предшествует его концу. Это так, поскольку, за исключением самого класса Period, никакой другой класс не имеет возможности получить доступ хоть к какому-нибудь изменяемому полю экземпляра Period. Указанные поля действительно инкапсулированы в этом объекте.

Заметим, что новые методы доступа, в отличие от нового конструктора, для создания резервных копий используют метод clone. Такое решение приемлемо (хотя и необязательно), поскольку мы точно знаем, что внутренние объекты Date в классе Period относятся к классу java.util.Date, а не какому-то потенциально ненадежному подклассу.

Резервное копирование параметров производится не только для неизменяемых классов. Всякий раз, когда вы пишете метод или конструктор, который помещает во внутреннюю структуру объект, созданный клиентом, задумайтесь, не является ли этот объект потенциально изменяемым. Если да, проанализируйте, будет ли ваш класс устойчив к изменениям в объекте, когда он получит доступ к этой структуре данных. Если ответ отрицательный, вы должны создать резервную копию объекта и поместить ее в структуру данных вместо оригинала. Например, если ссылку на объект,

предоставленный клиентом, вы предполагаете использовать как элемент во внутреннем экземпляре Set или как ключ во внутреннем экземпляре Map, следует учитывать, что инварианты этого набора или схемы могут быть нарушены, если после добавления в них объект вдруг поменяется.

То же самое справедливо и в отношении резервного копирования внутренних компонентов перед возвращением их клиенту. Вы должны дважды подумать, является ли ваш класс изменяемым или нет, прежде чем передавать клиенту ссылку на внутренний компонент, который можно изменить. Возможно, вам все же следует возвращать резервную копию. Крайне важно также помнить о том, что массивы ненулевой длины всегда являются изменяемыми. Поэтому для внутреннего массива вы всегда должны делать резервную копию, прежде чем возвращать его клиенту. Как альтернатива, вы можете возвращать пользователю неизменяемое представление этого массива. Оба приема показаны в статье 12.

Таким образом, урок, который можно извлечь из всего сказанного, заключается в том, что в качестве составных частей объектов вы должны по возможности использовать неизменяемые объекты, чтобы не пришлось беспокоиться о резервном копировании (статья 13). В случае же с примером Period стоит отметить, что опытные программисты для внутреннего представления времени часто применяют не ссылку на объект Date, а простой тип long, возвращаемый методом Date, getTime(). И поступают они так в первую очередь потому, что Date является изменяемым.

Не всегда можно создать резервную копию изменяемого параметра перед его включением в объект. Существуют такие методы и конструкторы, чей вызов означает явную передачу объекта, на который указывает параметр-ссылка. Вызывая такой метод, клиент дает обещание, что он не будет напрямую менять этот объект. Для метода или конструктора, предполагающего, что ему будет полностью передано управление изменяемым объектом, предоставленным клиентом, это обстоятельство должно быть четко оговорено в документации.

Классы, где содержатся методы или конструкторы, вызов которых означает передачу управления объектом, не способны защитить себя от злоумышленника. Такие классы можно использовать только тогда, когда есть взаимное доверие между классом и его клиентами или же когда нарушение инвариантов класса не способно нанести ущерба никому, кроме самого клиента. Последнюю ситуацию иллюстрирует шаблон класса-оболочки (статья 14). При определенном характере класса-оболочки клиент может разрушить инварианты этого класса, используя прямой доступ к объекту уже после того, как он попал в оболочку, однако обычно это не наносит вреда никому, кроме самого клиента.

Тщательно проектируйте сигнатуру метода

В этой статье приводятся советы по проектированию API, не удостоившиеся собственной статьи. Собранные вместе, они помогут сделать ваш API не dтоль подверженным ошибкам, более удобным и простым в изучении.

Тщательно выбирайте названия методов. Названия всегда должны соответствовать стандартным соглашениям по именованию (статья 38). Вашей главной целью должен быть выбор таких имен, которые будут понятны и согласуются с остальными названиями в том же пакете. Второй целью должен быть выбор имен, отвечающих более 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 существует консенсус. Бесценным источником является "The Java Developers Almanac" Патрика Чана (Patrick Chan) [ChanOO], содержащий декларации всех без исключения методов в библиотеках платформы Java, индексированные в алфавитном порядке, если, к примеру, вы сомневаетесь, назвать ли метод remove или delete, то, бегло про смотрев указатель в этой книге, поймете, что, без всяких сомнений, выбор remove лучше: есть сотни методов, чьи названия начинаются со слова remove, и лишь жалкая горстка имен, которые начинаются со слова delete.

Не заходите слишком далеко в погоне за удобством своих методов. Каждый метод должен выполнять собственную часть работы. Избыток методов делает класс сложным для изучения, использования, описания, тестирования и сопровождения. В отношении интерфейсов это верно вдвойне: большое количество методов усложняет жизнь и разработчикам, и пользовате,лям. для каждого действия, поддерживаемого вашим типом, создайте полнофункциональный метод. Сокращенный вариант операции рассматривайте лищь в том случае, если он будет использоваться часто. Если есть сомнения, забудьте об этом варианте.

Избегайте длинного перечня параметров. Правило таково, что на практике три параметра нужно рассматривать как максимум, и чем параметров меньше, тем лучше. Большинство программистов не способны помнить более длинные списки параметров. Если целый ряд методов превышает этот предел, вашим API невозможно будет пользоваться, не обращаясь беспрестанно к его описанию. Особенно вредны длинные последовательности параметров одного и того же типа. И это не только потому, что ваш пользователь не сможет запомнить порядок их следования. Если он по ошибке поменяет их местами, его программа все равно будет компилироваться и работать. Только вот делать она будет совсем не то, что хотел ее автор.

для сокращения слишком длинных списков параметров можно использовать два приема. Первый заключается в разбиении метода на несколько методов, каждому ИЗ которых нужно лишь какое-то подмножество его параметров. Если делать это неаккуратно, может получиться слишком много методов, однако этот же прием помогает сократить количество методов путем увеличения их ортогональности. Например, рассмотрим интерфейс java.util.List. У него нет методов для поиска индекса первого и последнего элемента в подсписке, каждому из них потребовалось бы по три параметра. Вместо этого он предлагает метод subList, который принимает два параметра И возвращает представление подсписка. Для получения желаемого результата можно объединить метод subList с методами indexOf и lastIndexOf, принимающими по одному параметру. Более того, метод subList можно сочетать с любыми другими Методами экземпляра List, чтобы выполнять самые разные операции для подсписков. Полученный API имеет высокое соотношение мощности и размера.

Второй прием сокращения чрезмерно длинных перечней параметров заключается в создании вспомогательных классов, обеспечивающих агрегирование параметров. Обычно эти вспомогательные классы являются статическими классами-членами (статья 18). Данный прием рекомендуется использовать, когда становится понятно, что часто возникающая последовательность параметров на самом деле представляет некую отдельную сущность. Предположим, что вы пишите класс, реализующий карточную игру, и выясняется, что постоянно передается последовательность из двух параметров: достоинство карты и ее масть. И ваш API и содержимое вашего класса, вероятно, выиграют, если для представления карты вы создадите вспомогательный класс и каждую такую последовательность параметров замените одним параметром, соответствующим этому вспомогательному классу.

Выбирая тип параметра, отдавайте предпочтение интерфейсу, а не классу. Если для декларации параметра имеется подходящий интерфейс, всегда используйте его, а не класс, который реализует этот интерфейс. Например, нет причин писать метод, принимающий параметр типа Hashtable, лучше использовать Map. Это позволит вам передавать этому методу Hashtable, HashMap, TreeMap, подмножество TreeMap, и вообще любую, пока еще не написанную реализацию интерфейса Map. Применяя же вместо интерфейса класс, вы навязываете вашему клиенту конкретную реализацию и вынуждаете выполнять ненужное и потенциально трудоемкое копирование в том случае, если входные данные будут представлены в какой-либо иной форме.

Объекты-функции (статья 22) применяйте с осторожностью. Некоторые языки, в частности Smalltalk и различные диалекты Lisp, поощряют стиль программирования, изобилующий объектами, представляющими функции, которые можно применять к другим объектам. Программисты, имеющие опыт работы с такими языками, могут поддаться соблазну и использовать тот же стиль для Java, но это не слишком хороший выбор. Проще всего создавать объект-функцию с помощью анонимного класса (статья 18), однако даже это приводит к некоторому загромождению синтаксиса и ограничивает мощность и производительность в сравнении со встроенными управляющими конструкциями (inline control construct). Более того, стиль программирования, когда вы постоянно создаете объекты-функции и передаете их из одного метода в другой, расходится с господствующей тенденцией, а потому, если вы будете придерживаться этого стиля, другим программистам будет трудно разобраться в вашем коде. Это не означает, что объекты-функции не имеют права на использование. Напротив, они важны для многих мощных шаблонов, таких как Strategy [Camma95, стр. 315] и Visitor [Camma95, стр. 331]. Точнее говоря, объекты-функции следует применять только при наличии веских причин.

Перегружая методы, соблюдайте осторожность

Приведем пример попытки классифицировать коллекции по признаку - Набор, список или другой вид коллекций,- предпринятой из лучших побуждений:

```
// Ошибка: неверное использование перезагрузки!
public class CollectionClassifier {
      public static String classify(Set s) {
       return "Set";  }

      public static String classify(List 1) {

      return "List"; }

      public static String classify(Collection c) {

      return "Unknown Collection"; }

      public static void main(String[] args) {
            Collection[] tests = new Collection[] {
                  new HashSet(),              // Набор
                  new ArrayList(),            // Список
                  new HashMap().values()      // Не набор и не список
            } ;
            for (int i = o; i < tests.length; i++)
                  System.out.println(classify(tests[i]));
            }
      }
```

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 перезагружается (overload), и выбор варианта перезагрузки осуществляется на стадии компиляции. Для всех трех проходов цикла параметр на стадии компиляции имеет один и тот же тип Collection. И хотя во время выполнения программы при каждом проходе используется другой тип, это уже не влияет на выбор варианта перезагрузки. Поскольку во время компиляции параметр имел тип Collection, может применяться только третий вариант перезагрузки: classify(Collection). И именно этот перезагруженный метод вызывается при каждом проходе цикла.

Поведение этой программы такое странное потому, что выбор перезагруженных методов является статическим, тогда как выбор переопределенных методов динамическим. Правильный вариант переопределенного метода выбирается при выполнении программы, исходя из того, какой тип в этот момент имеет объект, для которого был вызван метод. Напомним, что переопределение (overrid) метода осуществляется тогда, когда подкласс имеет декларацию метода с точно такой же сигнатурой, что и у декларации метода предка. Если в подклассе метод был переопределен и затем данный метод был вызван для экземпляра этого подкласса, то выполняться

будет уже переопределенный метод независимо от того, какой тип экземпляр подкласса имел на стадии компиляции. Для пояснения рассмотрим маленькую программу:

```
class A{
    String name() { return "A"; }

class B extends A {
    String name() { return "B"; }

class C extends A {
    String name() { return "C"; }

public class Overriding {
    public static void main(String[] args) {
        A[] tests = new A[] {new A(), new B(), new C() };
        for (int i = 0; i < tests.length; i++)
            System.out.print(tests[i].name());
        }
  }
```

Метод пате декларируется в классе Д и пере определяется в классах В и С. Как и ожидалось, эта программа печатает "ABC", хотя на стадии компиляции при каждом проходе в цикле экземпляр имеет тип Д. Тип объекта на стадии компиляции не влияет на то, какой из методов будет исполняться, когда поступит запрос на вызов переопределенного метода: всегда выполняется "самый точный" переопределяющий метод. Сравните это с перезагрузкой, когда тип объекта на стадии выполнения уже не влияет на то, какой вариант перезагрузки будет использоваться: выбор осуществляется на стадии компиляции и всецело основывается на том, какой тип имеют параметры на стадии компиляции.

В примере с ColleetionClassi fier программа должна была определять тип параметра, автоматически переключаясь на соответствующий перезагруженный метод на основании того, какой тип имеет параметр на стадии выполнения. Именно это делает метод name в примере "ABC". Перезагрузка метода не имеет такой возможности. Исправить программу можно, заменив все три варианта перезагрузки метода elassify единым методом, который выполняет явную проверку instaneeOf:

```
public static String classify(Collection c) {
return (c instanceof Set ? "Set" :
            (c instancepf List ? "List" : "Unknown Collection"));  }
```

Поскольку переопределение является нормой, а перезагрузка - исключением, именно переопределение задает, что люди ожидают увидеть при вызове метода. Как показал пример CollcetionClassifier, перезагрузка может не оправдать эти ожидания. Не следует писать код, поведение которого не очевидно для среднего программиста. Особенно это касается интерфейсов API. Если рядовой пользователь API не знает, какой из перезагруженных методов будет вызван для указанного набора параметров, то работа с таким API, вероятно, будет сопровождаться ошибками. Причем ошибки эти проявятся скорее всего только на этапе выполнения в виде некорректного поведения программы, и многие программисты не смогут их диагностировать. Поэтому необходимо избегать запутанных вариантов перезагрузки.

Стоит обсудить, что же именно сбивает людей с толку при использовании перезагрузки. Безопасная, умеренная политика предписывает никогда не предоставлять два варианта перезагрузки с одним и тем же числом параметров. Если вы придерживаетесь этого ограничения, у программистов никогда не возникнет сомнений по поводу того, какой именно вариант перезагрузки соответствует тому или иному набору параметров. Это ограничение не слишком обременительно, поскольку вместо того чтобы использовать перезагрузку, вы всегда можете дать методам различные названия.

Например, рассмотрим класс ObjectOutputStream. Он содержит варианты методов write для каждого простого типа и нескольких ссылочных типов. Вместо того чтобы перезагружать метод write, они применяют такие сигнатуры, как writeBoolean(boolean), writeInt(int) и writeLong(long). Дополнительное преимущество такой схемы именования по сравнению с перезагрузкой заключается в том, что можно создать методы read с соответствующими названиями, например readBoolean(), readInt() и readLong(). И действительно, в классе ObjectInputStream есть методы чтения с такими названиями.

В случае с конструкторами у вас нет возможности использовать различные названия, несколько конструкторов в классе всегда подлежат перезагрузке. Правда, в отдельных ситуациях вы можете вместо конструктора предоставлять статический метод генерации (статья 1), но это не всегда возможно. Однако, с другой стороны, при применении конструкторов вам не нужно беспокоиться о взаимосвязи между перезагрузкой и переопределением, так как конструкторы нельзя переопределять. Поскольку вам, вероятно, придется предоставлять несколько конструкторов с одним и тем же количеством параметров, полезно знать, в каких случаях это безопасно.

Предоставление нескольких перезагруженных методов с одним и тем же количеством параметров вряд ли запутает программистов, если всегда понятно, какой вариант перезагрузки соответствует заданному набору реальных параметров. Это как раз тот случай, когда у каждой пары вариантов перезагрузки есть хотя бы один формальный параметр с совершенно непохожим типом. Два типа считаются совершенно непохожими, если экземпляр одного из этих типов невозможно привести к другому типу. В этих условиях выбор варианта перезагрузки для данного набора реальных параметров полностью диктуется тем, какой тип имеют параметры в момент выполнения программы, и никак не связан с их типом на стадии компиляции. Следовательно, исчезает главный источник путаницы.

Например, класс ArrayList имеет конструктор, принимающий параметр int, и конструктор, принимающий параметр типа Collection. Трудно представить себе условия, когда возникнет путаница с вызовом двух этих конструкторов, поскольку простой тип и ссылочный тип совершенно непохожи. Аналогично, у класса BigInteger есть конструктор, принимающий массив типа byte, и конструктор, принимающий String. Это также не создает путаницы. Типы массивов и классы совершенно непохожи, за исключением Object. Совершенно непохожи также типы массивов и интерфейсы (за исключением Serializable и Cloneable). Наконец, в версии 1.4 класс Throwable имеет конструктор, принимающий параметр String, и конструктор, принимающий параметр Throwable. Классы String и Throwable не родственные, иначе говоря, ни один из этих классов не является потомком другого. Ни один объект, не может быть экземпляром двух неродственных классов, а потому неродственные классы совершенно непохожи.

Можно привести еще несколько примеров, когда для двух типов невозможно выполнить преобразование ни в ту, ни в другую сторону []LS, 5.1.7]. Однако в сложных случаях среднему программисту трудно определить, который из вариантов перезагрузки, если таковой имеется, применим к набору реальных параметров. Спецификация, определяющая, какой из вариантов перезагрузки должен использоваться, довольно сложна, и все ее тонкости понимают лишь немногие из программистов [JLS, 15.12.1-3] ..

Иногда, подгоняя существующие классы под реализацию новых интерфейсов, вам приходится нарушать вышеприведенные рекомендации. Например, многие типы значений в библиотеках для платформы Java до появления интерфейса Comparable имели методы compareTo с типизацией (self-typed). Представим декларацию исходного метода compareTo с типизацией для класса String:

```
public int compareTo(String s);
```

С появлением интерфейса Comparable все эти классы были перестроены под реализацию данного интерфейса, содержащую новый, более общий вариант метода compareTo со следующей декларацией:

```
public int compareTo( Object o);
```

Полученный таким образом вариант перезагрузки явно нарушает изложенные выше требования, но он не наносит ущерба, поскольку оба перезагруженных метода, будучи вызваны с одинаковыми параметрами, неизменно выполняют одно и то же. Программист может не знать, какой из вариантов перезагрузки будет задействован, но это не имеет значения, пока оба метода возвращают один и тот же результат. Стандартный прием, обеспечивающий опиеанную схему перезагрузки, заключается в том, чтобы ставить более общий вариант перезагрузки перед более частным:

```
public int compareTo(Object o) {
return compareTo((String) o); }
```

Аналогичная идиома иногда используется и для методов equals:

```
public boolean equals(Object o) {
      return o instanceof String &&equals((String)o);  }
```

Эта идиома безопасна и может повысить производительность, если на стадии компиляции тип параметра будет соответствовать параметру в более частном варианте перезагрузки (статья 37).

Хотя библиотеки для платформы Java в основном следуют приведенным здесь советам, все же можно найти несколько мест, где они нарушаются. Например, класс String передает два перезагруженных статических метода генерации valueOf(char[]) и valueOf(Object), которые, получив ссылку на один и тот же объект, выполняют совершенно разную работу. Этому нет четкого объяснения, и относиться к данным методам следует как к аномалии, способной вызвать настоящую неразбериху.

Подведем итоги. То, что вы можете осуществлять перезагрузку методов, еще не означает, что вы должны это делать. Обычно лучше воздерживаться от перезагрузки методов, которые имеют несколько сигнатур с одинаковым количеством параметров. Но иногда, особенно при наличии вызова конструкторов, невозможно следовать этому совету. Тогда постарайтесь избежать ситуации, при которой благодаря приведению типов один и то же набор параметров может использоваться разными вариантами перезагрузки. Если такой ситуации избежать нельзя, например, из-за того, что вы переделываете уже имеющийся класс под реализацию нового интерфейса, удостоверьтесь в том, что все варианты перезагрузки, получая одни и те же параметры, будут вести себя одинаковым образом. Если же вы этого не сделаете, программисты не смогут эффективно использовать перезагруженный метод или конструктор и не смогут понять, почему он не работает.

Возвращайте массив нулевой длины, а не null

Нередко встречаются методы, имеющие следующий вид:

```
private List cheesesInStock = ... ;

/**
 * @return массив, содержащий все сыры, имеющиеся в магазине,
 * или null, если сыров для продажи нет.
 */
public Cheese[] getCheeses() {
      if (cheesesInStock.size() == 0)
      return null;
      }
```

Нет причин рассматривать как особый случай ситуацию, когда в продаже нет сыра. Это требует от клиента написания дополнительного кода для обработки возвращаемого методом значения null, например:

```
Cheese[] cheeses = shop.getCheeses();
if (cheeses ! = null &&
Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
 System.out.println("Jolly good, just the thing.");
```

вместо простого:

```
if  (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
     System.out.println("Jolly good, just the thing.");
```

Такого рода многоречивость необходима почти при каждом вызове метода, который ВМ~СТО массива нулевой длины возвращает null. Это чревато ошибками, так как разработчик клиента мог и не написать специальный код для обработки результата null. Ошибка может оставаться незамеченной годами, поскольку подобные методы, как правило, возвращают один или несколько объектов. Следует еще упомянуть 0 том, что возврат null вместо массива приводит к усложнению самого метода, возвращающего массив.

Иногда можно услышать возражения, что возврат значения null предпочтительнее возврата массива нулевой длины потому, что это позволяет избежать расходов на размещение массива в памяти. Этот аргумент несостоятелен по двум причинам. Во-первых, на этом уровне нет смысла беспокоиться о производительности, если только профилирование программы не покажет, что именно этот метод является основной причиной падения производительности (статья 37). 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 13). На самом деле, именно это и происходит, когда вы применяете стандартную идиому для выгрузки элементов из коллекции в массив с контролем типа:

```
private List cheesesInStock = ... ,

private fiпal static Cheese[] NULL_CHEESE_ARRAY = new Cheese[0];

/**
 * @returп массие, содержащий все сыры, имеющиеся в магазине
 */
public Cheese[] getCheeses() {
     returп (Cheese[] cheesesInStock.toArray(NULL_CHEESE_ARRAУ);  }
```

В этой идиоме константа в виде массива нулевой длины передается методу toArray для того, чтобы показать, какой тип он должен возвратить. Обычно метод toArray выделяет место в памяти для возвращаемого массива, однако если коллекция пуста,

она размещается во входном массиве, а спецификация Collection,toArray(OЬject[] дает гарантию, что если входной' массив будет достаточно вместителен, чтобы содержать коллекцию, возвращен будет именно он. Поэтому представленная идиома никогда не будет сама размещать в памяти массив нулевой длины, а в качестве такового использует "константу с указанием типа".

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

Для всех открытых элементов API doc - комментарии

Если API будет использоваться, его нужно описывать. Обычно документация к API пишется вручную, и поддержание соответствия между документацией и программным кодом - весьма неприятная работа. Среда программирования Java облегчает эту задачу с помощью утилиты, называемой */avadoc.* Она автоматически генерирует документацию к API, отталкиваясь от исходного текста программы, дополненного специальным образом оформленными *комментариями к документации* (documentation comment), которые чаще называют *doc - комментариями* (doc comment). Утилита Javadoc предлагает простой, эффективный способ документирования API и используется повсеместно.

Если вы еще не знакомы с соглашениями для doc - комментариев, то обязаны их изучить. Эти соглашения не являются частью языка программирования Java, но де-факто они образуют свой API, который обязан знать каждый программист. Соглашения сформулированы в *"The Javadoc Tool Home Page"* [Javadoc-b].

Чтобы должным образом документировать API, следует предварять doc - комментарием каждую предоставляемую пользователям декларацию класса, интерфейса, конструктора, метода и поля. Единственное исключение обсуждается в конце статьи. Если doc - комментарий отсутствует, самое лучшее, что может сделать Javadoc,- это воспроизвести декларацию 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, вы должны написать doc - комментарии даже для тех классов, интерфейсов, конструкторов, методов и полей, которые не предоставляются пользователям.

Doc - комментарий для метода должен лаконично описывать соглашения между методом и его клиентами. Соглашение должно оговаривать, *что* делает данный метод, а не *как* он это делает. Исключение составляют лишь методы в классах, предназначенных для наследования (статья 15). В doc - комментарии необходимо перечислить все *предусловия* (precondition), т. е. утверждения, которые должны

быть истинными для того, чтобы клиент мог вызвать этот метод, и *постусловия* (postcondition), т. е. утверждения, которые будут истинными после успешного завершения вызова. Обычно предусловия неявно описываются тегами @throws для необработанных исключений. Каждое необработанное исключение соответствует нарушению некоего предусловия. Предусловия также могут быть указаны вместе с параметрами, которых они касаются, в соответствующих тегах @param.

Помимо пред- и постусловий для методов должны быть также документированы любые *побочные эффекты.* Побочный эффект (side effect) - это поддающееся наблюдению изменение состояния системы, которое является неявным условием для достижения постусловия. Например, если метод запускает. фоновый поток, это должно быть отражено в документации. Наконец, комментарии к документации должны описывать *безопасность класса при работе* с *потоками* (thread safety) , которая обсуждается в статье 52.

В целях полного описания соглашений doc - комментарий для метода должен включать в себя: тег @pakam для каждого параметра, тег @return, если только метод не возвращает тип void, и тег @throws для каждого исключения, инициируемого этим методом, как обработанного, так инеобработанного (статья 44). По соглашению, текст, который следует за тегом @param или @return, представляет собой именную конструкцию *(noun* phrase - термин грамматики английского языка). Описывающую значение данного параметра или возвращаемое значение. Текст, следующий за тегом @throws, должен состоять из слова if и именной конструкции, описывающей условия, при которых инициируется данное исключение. Иногда вместо именных конструкций используются арифметические выражения. Все эти соглашения иллюстрирует следующий краткий doc - комментарий из интерфейса List:

```
/**
 * Возвращает элемент, который занимает заданную позицию
 * в указанном списке.
 * @param index - индекс элемента, который нужно возвратить; индекс должен
 *                быть неотрицательным, и его значение должно быть
 *                меньше размера списка.
 * @return       элемент, занимающий в списке указанную позицию.
 * @throws       IndexOutOfBoundsException, если индекс лежит вне диапазона.
 *               (<tt> index &lt; () ||index &gt; = this.size()</tt>)
 */
Object 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. Чаще

всего применяются следующие теги: <p> для разделения параграфов, <code> и <tt> для представления фрагментов кода, <рге> для более длинных фрагментов кода.

Теги <code> и <tt> в значительной степени эквивалентны. Тег <code> используется чаще и,

согласно спецификации HTML 4.01, является более предпочтительным, поскольку <tt> - это *элемент стилевого оформления шрифта.* (Пользоваться элементами стилевого оформления шрифтов здесь не рекомендуется, предпочтение отдается каскадным таблицам стилей [HTML401]). Некоторые программисты все же предпочитают тег <tt>, поскольку он короче и не столь агрессивен.

Не забывайте, что для генерации метасимволов HTML, таких как знак "меньше" «),знак "больше" (» и амперсанд (&), необходимы escape - последовательности. Чтобы получить знак "меньше", используйте escape - последовательность "< ", знак "больше" - последовательность "> ", знак амперсанда - последовательность "& ". В предыдущем doc - комментарии escape -последовательность применена в теге @throws.

Наконец, отметим появление в doc - комментарии слова "this". По соглашению, слово "this" всегда ссылается на тот объект, которому принадлежит вызываемый метод, соответствующий данному doc - комментарию.

Первым предложением любого doc - комментария является *общее описание* (summary description) того элемента, к которому этот комментарий относится. Общее описание должно позиционироваться как описывающее функции соответствующей сущности. Во избежание путаницы никакие два члена или конструктора в одном классе или интерфейсе не должны иметь одинакового общего описания. Особое внимание обращайте на перезагруженные методы, описание которых часто хочется начать с одного и того же предложения.

Внимательно следите за тем, чтобы в первом предложении doc - комментария не было точки. В противном случае общее описание будет завершено прежде времени. Например, комментарий к документации, который начинается с фразы "A college degree, such as B. S., M. S., or Ph. D. ",приведет к появлению в общем описании фразы "A college degree, *such* as B." Во избежание подобных проблем лучше не использовать в общем описании сокращений и десятичных дробей. Для получения символа точки нужно заменить его *числовым представлением (numeric* encoding) ". ". Хотя это работает, исходный текст про граммы приобретает не слишком красивый вид:

```
/**
* A college degree, such as B&#46;S&#46;, M&#46;S&#46; or
* Ph&#46;D.
*/
public class Degree { ... }
```

Тезис, что первым *предложением* в doc - комментарии является общее описание, отчасти вводит в заблуждение. По соглашению, оно редко бывает законченным предложением. Общее описание методов и конструкторов должно представлять собой

глагольную конструкцию (verb phrase - термин грамматики английского языка), которая описывает операцию, осуществляемую этим методом. Например:

- ArrayList(int initialCapacity) - создает пустой список с заданной начальной емкостью.

- Collection. size() - возвращает количество элементов в указанной коллекции.

Общее описание классов, интерфейсов и полей должно быть именной конструкцией, описывающей сущность, которую представляет экземпляр этого класса, интерфейса или само поле. Например:

- TimerTask - задача, которая может быть спланирована классом Titer для однократного или повторяющегося исполнения.
- Math.PI - значение типа double, наиболее близкое к числу (отношение длины окружности к ее диаметру).

В этой статье рассмотрены лишь основные соглашения, касающиеся doc - комментариев, существует целый ряд других соглашений. Имеется несколько руководств по стилю написания doc - комментариев [Javadoc-a, VermeulenOO]. Есть также утилиты, проверяющие соблюдение этих правил [Qoclint].

Начиная с версии 1.2.2, утилита Javadoc имеет возможность "автоматически использовать вновь" и "наследовать" комментарии к методам. Если метод не имеет doc - комментария, Javadoc находит среди приемлемых наиболее близкий dOc-комментарий, отдавая при это предпочтение интерфейсам, а не суперклассам. Подробности алгоритма поиска комментариев приводятся в *"The Javadoc Manual".*

Это означает, 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 нее есть одно ограничение. Наследование doc – комментариев слишком бескомпромиссно: наследующий метод никак не может поменять унаследованный doc - комментарий. Между тем метод нередко уточняет соглашения, унаследованные от интерфейс и в этом случае он действительно нуждается в собственном doc - комментарии.

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, например *weblint*[Weblint], доступны в Интернете.

Следует привести еще одно предостережение, связанное с комментариями к документации. Комментарии должны сопровождать все элементы внешнего API, но этого не всегда достаточно. для сложных API, состоящих из множества взаимосвязанных

классов, комментарии к документации часто требуется дополнять внешним документом, описывающим общую архитектуру данного API. Если такой документ существует, то комментарии к документации в соответствующем классе или пакете должны ссылаться на него.

Подведем итоги. Комментарии к документации - самый лучший, самый Эффективный способ документирования API. Написание комментариев нужно считать обязательным для всех элементов внешнего API. Выберите стиль, который не противоречит стандартным соглашениям. Помните, что в

комментариях к документации можно использовать любой код HTML, причем метасимволы HTML необходимо маскировать escape - последовательностями.

Глава 7

Общие вопросы программирования

Данная глава посвящена обсуждению основных элементов языка Java. В ней рассматриваются интерпретация локальных переменных, использование библиотек и различных типов данных, а также две выходящие за рамки языка возможности: *отражение* (reflection) и *машинно-зависимые методы* (native method). Наконец, обсуждаются оптимизация и соглашения по именованию.

Сводите к минимуму область видимости локальных переменн

Эта статья по своей сути схожа со статьей 12 "Сводите к минимуму доступность классов и членов". Сужая область видимости локальных переменных, вы повышаете удобство чтения и сопровождения вашего кода, сокращаете вероятность возникновения ошибок.

Язык программирования указывает, что локальные переменные должны декларироваться в начале блока. И программисты продолжают придерживаться этого порядка, хотя от него уже нужно отказываться. Напомним, что язык программирования Java позволяет объявлять переменную в любом месте, где может стоять оператор.

Самый сильный прием сужения области видимости локальной перемен заключается в декларировании ее в том месте, где она впервые используется. Декларация переменной до ее использования только засоряет программу: появляется еще одна строка, отвлекающая читателя, который пытается разобраться в том, что делает программа. К тому моменту, когда переменная применяется, читатель может уже не помнить ни ее тип, ни начальное значение. Если программа совершенствуется и переменная больше не нужна, легко забыть убрать ее декларацию, если та находится далеко от места первого использования переменной.

К расширению области видимости локальной переменной приводит не только слишком ранее, но и слишком позднее ее декларирование. Область видимости локальной переменной начинается в том месте, где она декларируется, и заканчивается с завершением блока, содержавшего эту декларацию. Если переменная декларирована за пределами блока, где она используется, то она остается видимой и после того, как программа выйдет из этого блока. Если переменная случайно была использована до или после области, в которой она должна была применяться, последствия могут быть катастрофическими.

Почти каждая декларация локальной переменной должна содержать инициализатор. Если у вас недостаточно информации для правильной инициализации переменной, вы должны отложить декларацию до той поры, пока она не появится. Исключение из этого правила связано с

использованием операторов try / catch. Если для инициализации переменной применяется метод, инициирующий появление обрабатываемого исключения, то инициализация переменной должна осуществляться внутри блока try. Если переменная должна использоваться за пределами блока try, декларировать ее следует перед блоком try, там, где она еще не может быть "правильно инициализирована" (статья 35).

Цикл предоставляет уникальную возможность для сужения области видимости переменных. Цикл for позволяет объявлять *переменные цикла* (loop variable), ограничивая их видимость ровно той областью, где они нужны. (Эта область состоит из собственно тела цикла, а также из предшествующих ему полей инициализации, проверки и обновления.) Следовательно, если после завершения цикла значения его переменных не нужны, предпочтение следует отдавать циклам for, а не while.

Представим, например, предпочтительную идиому для организации цикла по некоей коллекции:

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
doSomething(i.next());  }
```

для пояснения, почему данный цикл for предпочтительнее более очевидного цикла while, рассмотрим следующий фрагмент кода, в котором содержатся два цикла while и одна ошибка:

```
Iterator i = c.iterator();
 while (i.hasNext()) {
doSomething(i.next());  }
Iterator i2 = c2.1terator();
        while (i.hasNext()) {       // Ошибка!
doSomethingElse(i2.next());  }
```


Второй цикл содержит ошибку копирования фрагмента программы: инициализируется новая переменная цикла i2, но используется старая i, которая, к сожалению, остается в поле видимости. Полученный код компилируется без замечаний и выполняется без инициирования исключительных ситуаций, только вот делает не то, что нужно. Вместо того чтобы организовывать итерацию по c2, второй цикл завершается немедленно, создавая ложное впечатление, что коллекция c2 пуста. И поскольку программа ничего об этой ошибке не сообщает, та может оставаться незамеченной долгое время.

Если бы аналогичная ошибка копирования была допущена при применении цикла for, полученный код не был бы даже скомпилирован. для той области, где располагается второй цикл, переменная первого цикла уже была бы за пределами видимости:

```
for (Iterator i = c. iterator(); i. hasNext(); ) {
doSomething(i.next()); }
// Ошибка компиляции - символ i не может быть идентифицирован
```

```
for (Iterator i2 = c2.iterator(); i.hasNext();; ) {
doSomething(i2.next()); }
```

Более того, если вы пользуетесь идиомой цикла for, уменьшается вероятность того, что вы допустите ошибку копирования, поскольку нет причин использовать в двух этих циклах различные названия переменных. Эти циклы абсолютно независимы, а потому нет никакого вреда от повторного применения названия для переменной цикла. На самом деле это даже стильно.

Идиома цикла for имеет еще одно преимущество перед идиомой цикла while, хотя и не столь существенное. Идиома цикла for короче на одну строку, что помогает при редактировании уместить содержащий ее метод в окне фиксированного размера и повышает удобство чтения.

Приведем еще одну идиом)\ цикла для про смотра списка, которая минимизирует область видимости локальных переменных:

```
// Высокопроизводительная идиома для просмотра списков
// с произвольным доступом
for (int i = 0, n = list.size(); i < n; i++) {
doSomething(list.get(i)); }
```

Эта идиома полезна для реализаций интерфейсов List с произвольным доступом, таких как ArrayList и Vector, поскольку для таких списков она, скорее всего, работает быстрее, чем приведенная выше "предпочтительная идиома". По поводу этой идиомы важно заметить, что в ней используются *две* переменные цикла: **i** и n, и обе имеют

абсолютно правильную область видимости. Вторая переменная важна для производительности идиомы. Без нее метод size пришлось бы вызывать при каждом Проходе цикла, что отрицательно сказалось бы на производительности. Если вы уверены, что список действительно предоставляет произвольный доступ, пользуйтесь этой идиомой. В противном случае производительность цикла будет падать в квадратичной зависимости от размера списка.

Похожие идиомы есть и для других задач с циклами, например:

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
doSomething(i) ; }
```

И в этой идиоме применяются две переменные цикла. Вторая из них - n - служит для исключения' ненужных вычислений при повторных проходах цикла. Как правило, этой идиомой вы должны пользоваться в тех случаях, когда условие цикла содержит вызов некоего метода, но этот метод при каждом проходе гарантированно возвращает один и тот же результат.

Последний прием, позволяющий уменьшить область видимости локальных переменных,

заключается в создании небольших, четко **позиционированных** методов. Если в пределах одного и того же метода вы сочетаете две операции, то локальные переменные, относящиеся к одной из них, могут попасть в 'область видимости другой. Во избежание этого разделите метод на два, по одному методу для каждой операции.

Изучите библиотеки и пользуйтесь ими

Предположим, что нужно генерировать случайные целые числа в диапазоне от нуля до некоторой верхней границы. Столкнувшись с такой распространенной задачей, многие программисты написали бы небольшой метод примерно следующего содержания:

```
static Random rnd = new Random();

// Неправильно, хотя встречается часто

static int random(int n) {
      return Math.abs(rnd.nextlnt()) %n;  }
```

Неплохой метод, но он несовершенен: у него есть три недостатка. Первый состоит в том, что если n - это небольшая степень числа два, то последовательность генерируемых случайных чисел через очень короткий период начнет повторяться. Второй заключается в том, что если n не является степенью числа два', то в среднем некоторые Числа будут получаться гораздо чаще других. Если n большое, указанный недостаток

может проявляться довольно четко. Графически это 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 ббб ббб. 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, потерпеть полны фиаско, выдавая результат, выходящий за пределы указанного диапазона. ~то происходит потому, что метод пытается преобразовать значение, возвращенное мет дом rnd.nextlnt (), в неотрицательное целое число, используя метод Math.abs. Если nextlnt() вернул Integer.MIN_VALUE,

то Math.abs также возвратит Intege r. MIN_ VALUE. Затем, если n не является степенью числа два, оператор остатка (%) вернет отрицательное число. Это почти наверняка вызовет сбой в вашей программе, и воспроизвести обстоятельства этого сбоя будет трудно.

Чтобы написать такой вариант метода random, в котором были бы исправлены все эти три недостатка, необходимо изучить генераторы линейных конгруэнтных псевдослучайных чисел, теорию чисе.л и арифметику дополнения до двух. К счастью, делать это вам не нужно, все это уже сделано 'для вас. Необходимый метод называется Random. nextlnt(int), он был добавлен в пакет java. util стандартной библиотеки в версии 1.2.

Нет нужды вдаваться в подробности, каким образом метод nextlnt(int) выполняет свою работу (хотя любопытные личности могут изучить документацию или исходный текст метода). Старший инженер с подготовкой в области алгоритмов провел много времени за разработкой, реализацией и тестированием этого метода, а затем показал метод экспертам в данной области с тем, чтобы убедиться в его правильности. После этого библиотека прошла стадию предварительного тестирования и была опубликована, тысячи программистов широко пользуются ею в течение нескольких лет. До сих пор ошибок в указанном методе найдено не было. Но если какой-либо дефект обнаружится, он будет исправлен в следующей же версии. Обращаясь к стандартной библиотеке, вы используете знания написавших ее экспертов, а также опыт тех, кто работал с нею до вас.

Второе преимущество от применения библиотек заключается в том, что вам не нужно терять время на решение специальных задач, имеющих лишь косвенное отношение к вашей работе. Как и большинство программистов, вы должны тратить время на разработку своего приложения, а не на подготовку его фундамента.

Третье преимущество от использования стандартных библиотек заключается в том, что их производительность имеет тенденцию повышаться со временем, причем без каких-либо усилий с вашей стороны. Множество людей пользуется библиотеками, они применяются в стандартных промышленных тестах, поэтому организация, которая осуществляет поддержку этих библиотек, заинтересована в том, чтобы заставить их работать быстрее. Например, стандартная библиотека арифметических операций с многократно увеличенной точностью java.math была переписана в версии 1.3, что привело к впечатляющему росту ее производительности.

Со временем библиотеки приобретают новые функциональные возможности.

Если в каком-либо классе библиотеки не хватает важной функции, сообщество разработчиков даст знать об этом недостатке. Платформа Java всегда развивалась при серьезной поддержке со стороны сообщества разработчиков. Прежде этот процесс был неформальным, сейчас же существует официальное движение, называемое Java *Community* Process OCP). Так или иначе, библиотеки пополняются недостающими функциями.

Последнее преимущество от применения стандартных библиотек заключается в том, что ваш код соответствует господствующим в данный момент тенденциям. Такой код намного легче читать и сопровождать, его могут использовать множество разработчиков.

Учитывая 'все эти преимущества, логичным казалось бы применение библиотек, а не частных разработок, однако многие программисты этого не делают. Но почему? Может быть, потому, что они не знают о возможностях имеющихся библиотек. С каждой следующей версией в библиотеки включается множество новых функций, и стоит быть в курсе этих новшеств. Вы можете внимательно изучать соответствующую документацию в режиме online либо прочесть о новых библиотеках в самых разных книгах [J2SE-APIs, ChanOO, Flanagan99, Chan98]. Библиотеки слишком объемны, чтобы просматривать всю документацию, однако каждый программист должен хорошо знать java.lang, java. util и в меньшей степени java.io. Остальные библиотеки изучаются по мере необходимости.

Обзор всех возможностей библиотек выходит за рамки данной статьи, однако некоторые из них заслуживают особого упоминания. В версии 1.2 в пакет java.util была добавлена архитектура *Collections Framework*.Она должна входить в основной набор инструментов каждого программиста. Collections F ramework - унифицированная архитектура, предназначенная для представления и управления коллекциями и позволяющая манипулировать коллекциями независимо от деталей представления. Она сокращает объемы работ по программированию и в то же время повышает производительность. Эта архитектура позволяет достичь унифицированности несвязанных API, упрощает проектирование и освоение новых API, способствует повторному использованию программного обеспечения.

Указанная архитектура базируется на шести интерфейсах коллекций (Collection, Set, List, Map, SortedSet и SortedMap). Она включает в себя реализацию этих интерфейсов и алгоритмы работы с ними. Наследуемые от коллекций классы Vector и Hashtable были перестроены под эту архитектуру, и потому, чтобы воспользоваться преимуществами Collections Framework, вам не придется отказываться от этих классов.

Collections Framework существенно уменьшает объем программного кода, необходимого для решения многих скучных задач. Например, у вас есть вектор строк, и вы хотите отсортировать его в алфавитном порядке. Эту работу выполняет всего одна строка:

```
Collections.sort(v);
```

Если нужно сделать то же самое, но игнорируя регистр, воспользуйтесь конструкцией:

```
Collections.sort(v, String.CASE_INSENSITIVE_ORDER);
```

Предположим, вы хотите напечатать все элементы массива. Многие программисты в таких случаях пользуются циклом for, но в этом нет необходимости, если применить следующую идиому:

```
System.out.println(Arrays.asList(a));
```

Наконец, предположим, что вам необходимо узнать все ключи, для' которых два экземпляра класса Hashtable - h1 и h2 - имеют одинаковые значения. До появления архитектуры Collections Framework это потребовало бы большого количества программного кода, сейчас же решение занимает всего три строки:

```
Map tmp = new HashMap(h1);
```

```
tmp.entrySet().retainAll(h2.entrySet());
Set result = tmp.keySet();
```

Приведенные примеры затрагивают лишь немногие из возможностей, которые предлагает Collections Framework. Если вы хотите узнать больше, обратитесь к документации на web-узле компании Sun [Collections] либо прочтите учебник [Bloch99].

Среди библиотек от третьих компаний упоминания заслуживает пакет util, concurrent Дага Ли (Doug Lea) [Lea01], в котором представлены утилиты высокого уровня, упрощающие программирование параллельных потоков.

В версии 1.4 в библиотеке появилось много дополнений. Самыми примечательными являются следующие:

- java.util.regex - полноценная функция для регулярных выражений в духе Perl.
- java.util.prefs - функция для сохранения пользовательских предпочтений и сведений о конфигурации программы.
- java.nio - высокопроизводительная функция ввода-вывода, обеспечивающая *масштабируемый ввод-вывод* (scalable 1/0), похожий на вызов poll в Unix, и *ввод-вывод, отображаемый в памяти* (memory-mapped 1/0), похожий на вызов nmap в Unix.

- java.util.LinkedHashSet, LinkedHashMap, IdentityHashMap новые реализации коллекций.

Иногда функция, заложенная в библиотеке, не отвечает вашим потребностям. И чем специфичнее ваши запросы, тем это вероятнее. Если вы изучили возможности, предлагаемые библиотеками в некоей области, и обнаружили, что они не соответствуют вашим потребностям, используйте альтернативную реализацию. В функциональности, предоставляемой любым конечным набором библиотек, всегда найдутся пробелы. И если необходимая вам функция отсутствует, у вас нет иного выбора, как реализовать ее самостоятельно.

Подведем итоги. Не изобретайте колесо. Если вам нужно сделать нечто, что кажется вполне обычным, в библиотеках уже может быть класс, который делает это. Вообще говоря, программный код в библиотеке наверняка окажется лучше кода, который вы напишете сами, а со временем он может стать еще лучше. Мы не ставим под сомнение ваши способности как программиста, однако библиотечному коду уделяется гораздо больше внимания, чем может позволить себе средний разработчик при реализации тех же самых функций.

Если требуются точные ответы, избегайте использования типов float

Типы float и double в первую очередь предназначены для научных и инженерных расчетов. Они реализуют *бинарную арифметику* с *плавающей точкой* (binary f!oating-point arithmetic), которая была тщательно выстроена с тем, чтобы быстро получать правильное приближение для широкого диапазона значений. Однако эти типы не дают точного результата, и в ряде случаев их нельзя использовать. **Типы** float и double не подходят для денежных расчетов, поскольку с их помощью

невозможно представить число 0.1 (или любую другую отрицательную степень числа десять).

Например, у вас в кармане лежит $1.03, и вы тратите 42 цента. Сколько денег у вас осталось? Приведем фрагмент наивной программы, которая пытается ответить на этот вопрос:

```
System.out.println(1.03 - .42);
```

Как ни печально, программа выводит 0.6100000000000001. И это не единственный случай. Предположим, что у вас в кармане есть доллар, и вы покупаете девять прокладок для крана по десять центов за каждую. Какую сдачу вы получите?

```
System.out.println(1.00 - 9*.10);
```

Если верить этому фрагменту программы, то вы получите $0.09999999999999995. 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 так далее вплоть до доллара. Вы покупаете по одной конфете каждого вида, начиная с той, что стоит 10 центов, и так далее, пока у вас еще есть возможность взять следующую конфету. Сколько конфет вы купите и сколько получите сдачи? Решим эту задачу следующим образом:

```
// Ошибка: использование плавающей точки для денежных расчетов!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0 ;
    for (double price = .10; funds >= p   rice; price += .10) {
    funds -= price;
        itemsBought++; }
    System.out.printl n(itemsBought  + "items bought.");
    System.out.println("Change: $" + funds);   }
```

Запустив программу, вы выясните, что можете позволить себе .три конфеты и у вас останется еще $0.39999999999999999. Но это неправильный ответ! Правильный путь решения задачи заключается в применении для денежных расчетов типов BigDecima l, int или long. Представ м простое преобразование предыдущей программы, которое позволяет использовать тип BigDecima l вместо double:

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal( ".10”);
    int itemsBought = 0 ;
    BigDecima l funds = new BigDecimal("1.00");
```

```
for (BigDecimal price = TEN_CENTS;
     funds.compareTo(price) >= 0  ;
     price = price.add(TEN_CENTS))   {
     itemsBought++;
     funds = fu nds.subtract(price);   }

System.out.print1n(itemsBought   + " items bought.");
System.out.print1n("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 BigDecima l имеет два недостатка: он не столь удобен и медленнее, чем простой арифметический тип. Последнее можно считать несущественным, если вы решаете единственную маленькую задачу, а вот неудобство может раздражать.

Вместо BigDecima l можно использовать int или long (в зависимости от обрабатываемых величин) и самостоятельно отслеживать положение десятичной точки. В нашем при мере расчеты лучше производить не в долларах, а в центах. Продемонстрируем этот подход:

```
public static void main(String[] args) {
 int itemsBought = 0 ;
     int funds = 100;
      for (int price = 10; funds >= price; price += 10) {
      itemsBought++;
           funds - = price;
      }
     System.out.println(itemsBought + " items bought.");
     System.out.println("Money left over: "    + funds  + "cents"); }
```

Подведем итоги. Для вычислений, требующих точного результата, не используйте типы float и double. Если вы хотите, чтобы система сама отслеживала положение десятичной точки, и вас не пугают неудобства, связанные с отказом от простого типа, используйте BigDecima l. Применение этого класса имеет еще то преимущество, что он дает вам полный контроль над округлением: для любой операции, завершающейся округлением, предоставляется на выбор восемь режимов округления. Это пригодится, если вы будете выполнять экономические расчеты с жестко заданным алгоритмом округления. Если для вас важна производительность, вас не пугает необходимость самостоятельно отслеживать положение десятичной точки, а обрабатываемые значения не слишком велики, используйте тип int или long. Если значения содержат не более девяти десятичных цифр, можете применить тип int. Если в значении не больше восемнадцати десятичных цифр, используйте тип long. Если же в значении более восемнадцати цифр, вам придется работать с BigDecima l.

Не используйте строкам, где более уместен иной тип

Тип String создавался для того, чтобы представлять текст, и делает он это прекрасно. Поскольку строки широко распространены и имеют хорошую поддержку в языке Java, возникает естественное желание использовать строки для решения тех задач, для которых они не предназначались. В этой статье обсуждается несколько операций, которые не следует проделывать со строками.

Строки - плохая замена другим типам значений. Когда данные попадают в программу из файла, сети или с клавиатуры, они часто имеют вид строки. Естественным является стремление оставить их в том же виде, однако это оправданно лишь тогда, когда данные по своей сути являются текстом. Если получены числовые данные,

При конкатенации строк опасайтесь потери произво

Оператор конкатенации строк (+) - удобный способ объединения нескольких строк в одну. Он превосходно справляется с генерацией отдельной строки для вывода и с созданием строкового представления для небольшого объекта с фиксированным размером, но не допускает масштабирования. Время, которое необходимо оператору конкатенации для последовательного объединения *n* строк, пропорционально квадрату числа n. К сожалению, это следствие того факта, что строки являются неизменяемыми (статья 13). При объединении двух строк копируется содержимое обеих строк.

Например, рассмотрим метод, который создает строковое представление для выписываемого счета, последовательно объединяя строки для каждого пункта в счете:

```
// Неуместное объединение строк - плохая производительность

public String statement() {

String s = " " ;

for (int i =0;i < numItems(); i++)

        s += lineForItem(i);        // Объединение строк

return s;
```

Если количество пунктов велико, этот метод работает очень медленно. Чтобы добиться приемлемой производительности, создаваемое представление счета должно храниться в классе StringBuffer, а не String:

```
public String statement() {
```

```
StringBuffer s= new StringBuffer(numItems()*LINE_WIDTH);

for (int i= 0;i < numItems(); i++)

s.append(lineForItem(i));

return s.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 девяносто раз быстрее первого. Поскольку первый метод демонстрирует квадратичную зависимость от количества пунктов, а второй - линейную, разница в производительности при большем количестве пунктов становится еще более разительной. Заметим, что второй метод начинается с предварительного размещения в памяти объекта StringBuffer,достаточно крупного, чтобы в нем поместился результат вычислений. Даже если отказаться от этого и создать StringBuffer,имеющий размер по умолчанию, он будет работать в сорок пять раз быстрее, чем первый метод.

Мораль проста: не пользуйтесь оператором конкатенации для объединения большого Числа строк, если производительность имеет важное значение. Лучше применять метод append из Класса Str1ngBufferВ качестве альтернативы можно использовать массив символов или обрабатывать строки по одной, не объединяя их.

Для ссылки на объект используйте его интерфейс

В статье 25 дается совет: в качестве типа параметра указывать интерфейс, а не класс. В более общей формулировке: ссылаясь на объект, вы должны отдавать предпочтение не классу, а интерфейсу. Если есть подходящие типы интерфейсов, то параметры, возвращаемые значения, переменные и поля следует декларировать, указывая интерфейс. Единственный случай, когда вам нужно ссылаться на класс объекта,- при его создании. Для пояснения рассмотрим случай с классом Vector,который является реализацией интерфейса List. Возьмите за правило писать так

```
// Хорошо: указывается тип интерфейса.
List subscribers= new Vector();
```

а не так:

```
// Плохо: в качестве типа указан класс!
Vector subscribers= new Vector();
```

Если вы выработаете привычку указывать в качестве типа интерфейс, ваша про грамма будет более гибкой. Когда вы решите поменять реализацию, все, что для этого потребуется,- изменить название класса в конструкторе (или использовать другой статической метод генерации). Например, первую из представленных деклараций можно переписать следующим образом:

```
List subscribers= new ArrayList();
```

И весь окружающий код сможет продолжить работу. Код, окружающий эту декларацию, ничего не знал о прежнем типе, который реализовывал интерфейс. Поэтому он не должен заметить подмену декларации.

Однако нельзя упускать из виду следующее: если первоначальная реализация интерфейса выполняла некие особые функции, не предусмотренные общими соглашениями для этого интерфейса, и программный код зависел от этих функций, крайне важно, чтобы новая реализация интерфейса обеспечивала те же функции. Например, если программный код, окружавший первую декларацию, зависел от того обстоятельства, что Vectorсинхронизирован по отношению к потокам, то замена класса Vectorна ArrayListбудет некорректной.

Есть лишь несколько сложных приложений, которым необходим механизм отражения. В их число входят визуализаторы классов, инспекторы объектов, анализаторы программного кода и интерпретирующие встроенные системы. Отражение можно также использовать в системах вызова удаленных процедур (RPC system) с целью снижения потребности в компиляторах запросов. Если у вас есть сомнения, подпадает ли ваше приложение в одну из этих категорий, вероятнее всего, оно к ним не относится.

В ы можете без больших затрат использовать многие преимущества механизма отражения, если будете применять его в усеченном виде. Во многих программах, которым нужен класс, отсутствовавший на момент компиляции, для ссылки на него можно использовать соответствующий интерфейс или суперкласс (статья 34). Если это то, что вам нужно, вы можете сначала опосредованно создать экземпляры, а затем обращаться к ним обычным образом. используя их интерфейс или суперкласс. Если соответствующий конструктор, как часто бывает, не имеет параметров, вам даже не нужно обращаться к пакету java.lang.ref1ect - требуемые функции предоставит метод C1ass. newlnstance.

В качестве примера приведем программу, которая создает экземпляр интерфейса Set, чей класс задан первым аргументом командной строки. Остальные аргументы командной строки программа вставляет в полученный набор и затем распечатывает его. При выводе аргументов программа уничтожает дубликаты (первый аргумент игнорируется). Порядок печати аргументов зависит от того, какой класс указан в первом аргументе. Если вы указываете "java.util.HashSet", аргументы выводятся в произвольном порядке, если - "java.uti1.TreeSet", они печатаются в алфавитном порядке, поскольку элементы в наборе TreeSet сортируются.

```
// Опосредованное создание экземпляра с доступом
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.");          // Класс не найден
      System. exi t (1) ;  }


        // Создает экземпляр класса

        Sets = null;
        try {
          s = (Set) cl.newlnstance();
          catch(IllegalAccessExceptlon e) {
          System.err.println("Class not accessible. ");        // Доступ к классу невозможен
```

```
System.exit(1);
```



```
} catch(InstantiationException e) {
System.err.println("Class not instantiable.");  // Класс не позволяет создать экземпляр
System.exit(1); }

// Проверяетнабор

s.addAll(Arrays.asList(args).subList(1, args.length-1));

 System.out.println(s); }
```

Хотя эта программа является всего лишь игрушкой, она показывает мощный прием. Этот код легко превратить в универсальную программу про верки наборов, которая проверяет правильность указанной реализации Set, активно воздействуя на один или несколько экземпляров и выясняя, выполняют ли они соглашения для интерфейса Set. Точно так же его можно превратить в универсальный инструмент для анализа производительности. Методика, которую демонстрирует эта программа, в действительности достаточна для создания полноценной *инфраструктуры* с *предоставлением услуг* (service provider framework) (статья 1). В большинстве случаев описанный прием это все, что нужно знать об отражении классов.

Этот пример иллюстрирует два недостатка системы отражения. Во-первых, во время его выполнения могут возникнуть три ошибки, которые, если бы не использовался механизм отражения, были бы обнаружены еще на стадии компиляции. Во-вторых, Для генерации экземпляра класса по его имени потребовалось Двадцать строк кода, тогда как вызов конструктора уложился бы ровно в одну строку. Эти недостатки, однако, касаются лишь той части программы, которая создает объект как экземпляр класса. Как только экземпляр создан, он неотличим от любого другого экземпляра Set. Благодаря этому значительная часть кода в реальной программе не поменяется от локального применения механизма отражения.

Приемлемым, хотя и редким вариантом использования отражения является разрушение зависимости класса от других классов, методов и полей, которые в момент выполнения могут отсутствовать, Это может пригодиться при написании пакета, который должен работать с различными версиями какого-либо другого пакета. Прием заключается в том, чтобы компилировать наш пакет для работы в минимальном окружении (обычно это поддержка самой старой версии), а доступ ко всем новым классам и методам осуществлять через механизм отражения. Чтобы это работало, необходимо Предпринимать правильные действия, когда в ходе выполнения программы обнаружится, что тот или иной новый класс или метод, к которому вы пытаетесь получить доступ, в данный момент отсутствует. Эти действия могут заключаться в применении каких-либо альтернативных средств, позволяющих достичь той же цели, или же в использовании усеченного функционала.

Подведем итоги. Отражение - это мощный инструмент, который необходим для решения определенных сложных задач системного программирования. Однако у него

есть много недостатков. Если вы пишите программу, которая должна работать с классами, неизвестными на момент компиляции, то вы должны по возможности использовать механизм отражения только для создания экземпляров отсутствовавших классов, а для доступа к полученным объектам следует применять некий интерфейс или суперкласс, который известен уже на стадии компиляции.

Соблюдайте осторожность при использовании машинно-зависимы

Интерфейс Java Native Interface (JNI) дает возможность приложениям на языке Java делать вызов" *машинно-зависимых" методов* (native method), т. е. специальных методов, написанных на *машинно-зависимом языке программирования,* таком как C или *C++.* Перед тем как вернуть управление языку Java, машинно-зависимые методы могут выполнить любые вычисления, используя машинно-зависимый язык.

Исторически машинно-зависимые методы имели три основные области применения. Они предоставляли доступ к механизмам, соответствующим конкретной платформе, таким как реестр и блокировка файла. Они обеспечивали доступ к унаследованным библиотекам кода, которые, в свою очередь, могли дать доступ к унаследованным данным. Наконец, машинно-зависимые методы использовались для того, чтобы писать на машинно-зависимом языке те части приложений, которые критичны для быстродействия, тем самым повышая общую производительность.

Применение машинно-зависимых 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 **j**ava.util.prefs выполняет функции реестра. Оправданно также использование машинно-зависимых методов для доступа к унаследованному коду, однако есть более хорошие способы доступа к унаследованному коду. Например, интерфейс JDBC (Java DataBase Connectivity) обеспечивает доступ к унаследованным базам данных.

В версии **1.3** использование машинно-зависимых методов для повышения производительности редко оправдывает себя. В предыдущих версиях это часто было необходимо, однако сейчас созданы более быстрые реализации JVM. И теперь для большинства задач можно получить сравнимую производительность, не прибегая к машинно-зависимым методам. Например, когда в версию 1.1 был включен пакет java.math.BigInteger был реализован поверх быстрой библиотеки арифметических операций с многократно увеличенной точностью, написанной на языке C. В то время это было необходимо для получения приемлемой производительности. В версии 1.3 класс BigInteger полностью переписан на языке Java и тщательно отрегулирован. Для большинства операций и размеров операндов новая версия оказывается быстрее пер во начальной во всех реализациях JVM 1.3 компании *Sun.*

Применение машинно-зависимых методов имеет серьезные недостатки. Поскольку машинно-зависимые методы *небезопасны* (статья 24), использующие их приложения теряют устойчивость к ошибкам, связанным с памятью. Для каждой новой платформы машинно-зависимый программный код необходимо компилировать заново, может потребоваться даже его изменение. С переходом на машинно-зависимый код и с возвратом в Java связаны высокие накладные расходы, а потому, если машинно-зависимый методы выполняют лишь небольшую работу, их применение может *снизить* производительность приложения. Наконец, машинно-зависимые методы сложно писать и трудно читать.

Подведем итоги. Хорошо подумайте, прежде чем использовать машинно-зависимые методы. Если их и можно применять для повышения производительности, то крайне редко. Если вам необходимо использовать машинно-зависимые методы для доступа к низкоуровневым ресурсам или унаследованным библиотекам, пишите как можно меньше машинно-зависимого кода и тщательно его тестируйте. Единственная ошибка в машинно-зависимом коде может полностью разрушить все ваше приложение.

Соблюдайте осторожность при оптимизации

Есть три афоризма, посвященных оптимизации, которые обязан знать каждый.
Возможно, они пострадали от слишком частого цитирования, однако при ведем их на тот случай, если вы с ними не знакомы:

> Во имя эффективности (без обязательности ее достижения) делается больше вычислительных ошибок, чем по каким-либо иным причинам, включая непроходимую тупость.
>
> *- Уильям Вульф (William A. Wulf) [Wulf72]*

> Мы *обязаны* забывать о мелких усовершенствованиях, скажем, на 97 % рабочего времени: опрометчивая оптимизация - корень всех зол.
>
> *- Дональд Кнут (Donald E. Knuth) [Knuth74]*

> Что касается оптимизации, то мы следуем двум правилам:
> *Правило 1.* Не делайте этого.
> *Правило 2 (только для экспертов).* Пока не делайте этого - т. е. нет пока у вас абсолютно четкого, но неоптимизированного решения.
>
> *- М. А. Джексон (M. A. Jackson) {fackson75]*

Эти афоризмы на два десятилетия опередили язык программирования Java. В них отражена сущая правда об оптимизации: легче причинить вред, чем благо, особенно если вы взялись за оптимизацию преждевременно. В процессе оптимизации вы можете получить программный код, который не будет ни быстрым, ни правильным, и его уже так легко не исправить.

Не жертвуйте здравыми архитектурными принципами во имя производительности. 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 12).

Это не означает, что вы должны игнорировать вопрос производительности до тех пор, пока ваша программа не будет завершена. Проблемы реализации могут быть решены путем последующей оптимизации, однако глубинные архитектурные пороки, которые ограничивают производительность, практически невозможно устранить, не пере писав заново систему. Изменение задним числом фундаментальных положений вашего проекта может породить систему с уродливой структурой, которую сложно сопровождать и совершенствовать. Поэтому вы должны думать о производительности уже в процессе разработки приложения.

Старайтесь избегать конструкторских решений, ограничивающих производительность. Труднее всего менять те компоненты, которые определяют взаимодействие модулей с окружающим миром. Главными среди таких компонентов являются API, протоколы физического уровня и форматы записываемых данных. Мало того, что эти компоненты впоследствии сложно или невозможно менять, любой из них способен существенно ограничить производительность, которую можно получить от системы.

Изучите влияние на производительность тех проектных решений, которые заложены в ваш API. Создание изменяемого открытого типа может потребовать создания множества ненужных резервных копий (СТАТЬ$/ 24). 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: 14) и последний пример: указав в API не тип интерфейса, а тип реализующего его класса, так оказываетесь привязаны к определенной реализации этого интерфейса, даже несмотря на то, ЧТО в будущем, возможно, будут написаны еще более быстрые его реализации (статья 34).

Влияние архитектуры API на производительность велико. Рассмотрим метод `getSize` из класса `java.wt.Component`. То, что этот критичный для производительности метод возвращает экземпляр `Dimension`, а также то, что экземпляры `D1mension` являются изменяемыми, приходит к тому, что любая реализация этого метода при каждом вызове создает новый экземпляр `D1mension`. И ХОТ$/, начиная с версии 1.3, создание небольших объектов обходится относительно дешево, бесполезное создание миллионов объектов может нанести производительности приложения реальный ущерб.

В данном случае имеется несколько альтернатив. В идеале класс `Dimension` должен стать неизменяемым (статья 13). Либо метод `getSize` можно заменить двумя методами, возвращающими отдельные простые компоненты объекта `Dimension`.

И действительно, с целью повышения производительности в версии 1.2 два таких метода были добавлены в интерфейс класса `Component`. Однако уже существовавший к тому времени клиентский код продолжает пользоваться методом `getSize`, и его производительность по-прежнему страдает от первоначально принятых проектных решений для API.

Хорошая схема API, как правило, сочетается с хорошей производительностью.
Не стоит искажать API ради улучшения производительности. Проблемы с производительностью, которые заставили вас переделать API, могут исчезнуть с появлением новой платформы или других базовых программ, а вот искаженный API и связанная с ним головная боль останутся с вами навсегда.

После того как вы тщательно спроектировали программу и выстроили четкую, краткую и хорошо структурированную ее реализацию, можно подумать об оптимизации, если, конечно, вы еще не удовлетворены производительность программы. Напомним два правила Джексона: "не делайте этого" и "не делайте этого пока (для экспертов)". Он мог бы добавить еще одно: измеряйте производительность до и после попытки ее оптимизации.

Возможно, вы будете удивлены, но нередко попытки оптимизации не оказывают поддающегося измерению влияния на производительность, иногда они даже ухудшают ее. Основная причина заключается в том, что сложно догадаться, где программа теряет время. Та часть программы, которую вы считаете медленной, может оказаться ни при чем, и вы зря потратите время, пытаясь ее оптимизировать. Общее правило гласит, что 80% времени программы теряют на 20% своего кода.

Средства профилирования помогут вам определить, где именно следует сосредоточить усилия по оптимизации. Подобные инструменты предоставляют вам информацию о ходе выполнения программы, например: сколько примерно времени требуется каждому методу, сколько раз он был вызван. Это укажет вам объект для настройки, а также может предупредить вас о необходимости замены самого алгоритма. Если в вашей программе скрыт алгоритм с квадратичной (или еще худшей) зависимостью, никакие настройки эту проблему не решат. Следовательно, вам придется заменить алгоритм более эффективным. Чем больше в системе программного кода, тем большее значение имеет работа с профилировщиком. Это все равно, что искать иголку в стоге сена: чем больше стог, тем больше пользы от металлоискателя. Java 2 SDK поставляется с простым профилировщиком, несколько инструментов посложнее можно купить отдельно.

Задача определения эффекта оптимизации для платформы Java стоит острее, чем для традиционных платформ, по той причине, что язык программирования Java не имеет четкой *модели производительности* (performance model). Нет четкого определения относительной стоимости различных базовых операций. "Семантический разрыв" между тем, что пишет программист, и тем, что выполняется центральным процессором, здесь гораздо значительнее, чем у традиционных компилируемых языков, и это сильно усложняет надежное предсказание того, как будет влиять на производительность какая-либо оптимизация. Существует множество мифов о производительности, которые на поверку оказываются полуправдой, а то и совершенной ложью.

Помимо того, что модель производительности плохо определена, она меняется от одной реализации JVM к другой и даже от версии к версии. Если вы будете запускать свою программу в разных реализациях JVM, про следите эффект от вашей оптимизации для каждой из этих реализаций. Иногда в отношении производительности вам придется идти на компромисс между различными реализациями JVM.

Подведем итоги. Не старайтесь писать быстрые программы - лучше пишите хорошие, тогда у вас появится и скорость. Проектируя системы, обязательно думайте о производительности, особенно если вы работаете над API, протоколами нижних уровней и форматами записываемых данных. Закончив построение системы, измерьте ее производительность. Если скорость приемлема, ваша работа завершена. Если нет, локализуйте источник проблем с помощью профилировщика и оптимизируйте соответствующие части системы. Первым шагом должно быть исследование выбранных алгоритмов: никакая низкоуровневая оптимизация не компенсирует плохой выбор алгоритма. При необходимости повторите эту процедуру, измеряя производительность после каждого изменения, пока не будет получен приемлемый результат.

При выборе имен придерживайтесь общепринятых соглашений

Платформа Java обладает хорошо устоявшимся набором соглашений, касающихся *выбора имен* (naming convention). Многие из них приведены в *"The Java Language Specification"*[JLS, 6.8]. Соглашения об именовании делятся на две ка-г:егории: типографские и грамматические.

Типографских соглашений, касающихся выбора имен для пакетов, классов, интерфейсов и полей, очень мало. Никогда не нарушайте их, не имея на то веской причины. API, не соблюдающий эти соглашения, будет трудно использовать. Если соглашения нарушены в реализации, ее будет сложно сопровождать. В обоих случаях нарушение соглашений может запутывать и раздражать других программистов, работающих с этим кодом, а также способствовать появлению ложных допущений, приводящих к ошибкам.

Названия пакетов должны представлять собой иерархию, отдельные части которой отделены друг от друга точкой. Эти части должны состоять из строчных букв и изредка цифр. Название любого пакета, который будет использоваться за пределами организации, обязано начинаться с доменного имени вашей организации в Интернете, которому предшествуют домены верхнего уровня, например edu. cmu, com.sun, gov.nsa.Исключение из этого правила составляют стандартные библиотеки, а также необязательные пакеты, чьи названия начинаются со слов **j**ava и **j**avax. Пользователи не должны создавать пакетов с именами, начинающимися с **j**ava или **j**avax. Детальное описание правил, касающихся преобразования названий доменов Интернета в префиксы названий пакетов, можно найти в *"The Java Language Specification"*[JLS, 7.7].

Вторая половина в названии пакета должна состоять из одной или нескольких частей, описывающих этот пакет. Части должны быть короткими, обычно не длиннее восьми символов. Поощряются выразительные сокращения, например util вместо utilities.Допустимы акронимы, например awt.Такие части, как правило, должны состоять из одного единственного слова или сокращения.

Многие пакеты имеют имена, в которых, помимо названия домена в Интернете, присутствует только одно слово. Большее количество частей в имени пакета нужно лишь для больших систем, чей размер настолько велик, что требует создания неформальной иерархии. Например, в пакете javax.swing представлена сложная иерархия пакетов с такими названиями, как javax.swing.plaf.metal. Подобные пакеты часто называют подпакетами, однако это относится исключительно к области соглашений, поскольку для иерархии пакетов нет лингвистической поддержки.

Названия классов и интерфейсов состоят из одного или нескольких слов, причем в каждом слове первая буква должна быть заглавной, например Timer или TimerTask. Необходимо избегать аббревиатур, за исключением акронимов и нескольких общепринятых сокращений, таких как max и min. Нет полного единодушия по поводу того, должны ли акронимы полностью писаться прописными буквами или же заглавной у них должна быть только первая буква. Хотя чаще в верхнем регистре пишется все название, есть один сильный аргумент в пользу того, чтобы заглавной была только первая буква. В последнем случае всегда ясно, где кончается одно слово и начинается другое, даже если рядом стоят несколько акронимов. Какое название класса вы предпочли бы увидеть: HTTPURL или HttpUrl ?

Названия методов и полей подчиняются тем же самым типографским соглашениям, за исключением того, что первый символ в названии всегда должен быть строчным, например remove, ensureCapacity. Если первым словом в названии метода или поля оказывается акроним, он весь пишется строчными буквами.

Единственное исключение из предыдущего правила касается "полей-констант" (constant field), названия которых должны состоять из одного или нескольких слов, написанных заглавными буквами и отделенных друг от друга символом подчеркивания, например VALUES или NEGATIVE_INFINITY. Поле-константа - это поле static final, значение которого не меняется. Если поле static final имеет простой тип или неизменяемый ссылочный тип (статья 13), то это поле-константа. Даже если тип поля относится к потенциально изменяемым, это все равно может быть поле-константа при условии, что объект, на который оно ссылается, является неизменяемым. Например, перечисление типов может представить пространство своих перечислимых констант в виде неизменяемой константы типа List (статья 21). Заметим, что поля-константы *единственное место*, где допустимо использование символа подчеркивания.

Названия локальных переменных подчиняются тем же типографским соглашениям, что и названия членов классов, за исключением того, что в них можно использовать аббревиатуры, отдельные символы, а также короткие последовательности символов, смысл которых зависит от того контекста, где эти локальные переменные находятся. Например: i, xref, houseNumber. Примеры типографских соглашений приведены в таблице 7.1.

По сравнению с типографскими, грамматические соглашения, касающиеся именования, более гибкие и спорные. Для пакетов практически нет грамматических соглашений. для именования классов обычно используются существительные или именные конструкции, например Timer и BufferedWriter. Интерфейсы именуются так же,

как классы, например Collect1on и Comparator, либо применяются названия с окончаниями, образованными от прилагательных "-able" и "-ible", например Runnable и Accessible.

Примеры типографских соглашений

Тип идентификатора	Примеры
Пакет	Com.sun.medialib, com.sun.jdi.event
Класс или интерфейс	Timer, TimerTask, KeyFactorySpi, HttpServlet
Метод или поле	Remove, ensureCapacity, getCrc
Поле-константа	VALUES, NEGATIVE_INFINITY
Локальная переменная	I, xref, houseNumber

Для методов, выполняющих какое-либо действие, в качестве названия используются глаголы или глагольные конструкции, например append и drawlmage. для методов, возвращающих булево значение, обычно применяются названия, в которых сначала идет слово "is", а потом существительное, именная конструкция' или любое слово (фраза), играющее роль прилагательного, например isDigit, isProbablePrime, isEmpty, isEnabled, isRunning.

Для именования методов, не связанных с булевыми операциями, а также методов, возвращающих атрибут объекта, для которого они были вызваны, обычно используется существительное, именная конструкция либо глагольная конструкция, начинающаяся с глагола "get", например size, hashCode, getTime. Отдельные пользователи требуют, чтобы применялась лишь третья группа (начинающаяся с "get"), но для подобных претензий нет никаких оснований. Первые две формы обычно делают текст программы более удобным для чтения, например:

```
if (car.speed() > 2* SPEED_LIMIT)
generateAudibleAlert("Watch out for cops!");
```

Форма, начинающаяся с "get", -обязательна, если метод принадлежит к классу *Bean* [JavaBeans]Ее можно также рекомендовать, если в будущем вы собираетесь превратить свой класс в Bean. Наконец, серьезные основания для использования данной формы имеются в том случае, если в классе уже есть метод, присваивающий этому же атрибуту новое значение. При этом указанные методы следует назвать *getAttribute* и *setAttribute.*

Несколько названий методов заслуживают особого упоминания. Методы, которые преобразуют тип объекта и возвращают независимый объект другого типа, часто называются *toType,* например toString, toArray. Методы, которые возвращают *представление* (статья 4), имеющее иной тип, чем сам объект, обычно называются *asType,* например asList. Методы, возвращающие простой тип с тем же значением,

что и у объекта, в котором они были вызваны, называются typeValue, например intValue. для статических методов генерации широко используются названия valueOf и getInstance (статья 1).

Грамматические соглашения для названий полей формализованы в меньшей степени и не играют такой большой роли, как в случае с классами, интерфейсами и методами, поскольку хорошо спроектированный API, если и предоставляет какое либо поля, то немного. Поля типа boolean обычно именуются так же, как логические методы доступа, но префикс "is" у них опускается, например 1п1 tialized, composite. Поля других типов, как правило, именуются с помощью существительного или именной конструкции, например height, digits, bodyStyle. Грамматические соглашения для локальных переменных аналогичны соглашениям для полей, только их соблюдение еще менее обязательно.

Подведем итоги. Изучите стандартные соглашения по именованию и доведите их использование до автоматизма. Типографские соглашения просты и практически однозначны; грамматические соглашения более сложные и свободные. Как сказано в *"The Java Language Specification"* [JLS, 6.8], не нужно рабски следовать этим соглашениям, если длительная практика их применения диктует иное решение. Пользуйтесь здравым смыслом.

Глава 8

Исключения

Если исключения (exception) используются наилучшим образом, они способствуют написанию понятных, надежных и легко сопровождаемых программ. При неправильном применении результат может быть прямо противоположным. В этой главе даются рекомендации по эффективному использованию исключений.

Используйте исключения лишь в исключительных ситуаци

Однажды, если вам не повезет, вы сделаете ошибку в программе, например, такую:

```
// Неправильное использование исключений. Никогда так не делайте!
  try {
        int i = 0;
        while(true)
              a[i++]. f();
              } 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 инициированием исключительной ситуации ArrayIndexOutOfBoundException, ее перехватом и последующим игнорированием. Предполагается, что это эквивалентно стандартной идиоме цикла по массиву, которую узнает любой программист java:

```
for (int i = 0 ;i < a.length; i++)
a[i].f();
```

Но почему же кто-то выбрал идиому, использующую исключения, вместо другой, испытанной и правильной? Это вводящая в заблуждение попытка улучшить производительность, которая исходит из ложного умозаключения, что, поскольку виртуальная машина проверяет границы при всех обращениях к массиву, обычная проверка на завершение цикла (i < a.length) избыточна и ее следует устранить. В этом рассуждении неверны три момента:

- Так как исключения создавались для применения в исключительных условиях, лишь очень немногие реализации JVM пытаются их оптимизировать (если таковые есть вообще). Обычно создание, инициирование и перехват исключения дорого обходится системе

- Размещение кода внутри блока try-catch препятствует выполнению определенных процедур оптимизации, которые в противном случае могли бы быть исполнены в современных реализациях jVM.

- Стандартная идиома цикла по массиву вовсе не обязательно приводит к выполнению избыточных проверок, в процессе оптимизации некоторые современные реализации jVM отбрасывают их.

Практически во всех современных реализациях jVM идиома, использующая исключения, работает гораздо медленнее стандартной идиомы. На моей машине идиома, использующая исключения, выполняет цикл от 0 до 99 в семьдесят раз медленней стандартной.

Идиома цикла, использующая исключения, снижает производительность и делает непонятным программный код. Кроме того, нет гарантий, что она будет работать. При появлении непредусмотренной разработчиком ошибки указанная идиома может без предупреждений завершить работу, маскировав ошибку, и тем самым значительно усложнить процесс отладки. Предположим, вычисления в теле цикла содержат ошибку, которая приводит к выходу за границы при доступе к какому-то совсем другому массиву. Если бы применялась правильная идиома цикла, эта ошибка породила бы необработанное исключение, которое вызвало бы немедленное завершение потока с соответствующим сообщением об ошибке. В случае же порочной, использующей исключения идиомы цикла исключение, вызванное ошибкой, будет перехвачено и неправильно интерпретировано как обычное завершение цикла.

Мораль проста: исключения, как и подразумевает их название, должны применяться лишь для исключительных ситуаций, при обычной обработке использовать их не следует никогда. Вообще говоря, вы всегда должны предпочитать стандартные, легко распознаваемые идиомы идиомам с ухищрениями, предлагающим лучшую производительность. Даже если имеет место реальный выигрыш в производительности, он может быть поглощен неуклонным совершенствованием реализаций jVM. А вот коварные ошибки и сложность поддержки, вызываемые 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 определенных непредсказуемых условий, то в этом же классе, как правило, должен присутствовать отдельный метод, проверяющий состояние (state-testing), который показывает, можно ли вызывать первый метод. Например, класс Iterator имеет зависящий от состояния метод next, который возвращает элемент для следующего прохода цикла, а также соответствующий метод проверки состояния hasNext. Это позволяет применять для просмотра коллекции в цикле следующую стандартную идиому:

```
for (Iterator i = collection. iterator(); i. hasNext(); ) {
 Foo foo= (Foo) i.next();  }
```

Если бы в классе Iteratorне было бы метода hasNext,клиент был бы вынужден использовать следующую конструкцию:

```
// Не пользуйтесь этой отвратительной идиомой
// для просмотра коллекции в цикле!
try {
      Iterator i= collection.iterator();
       while(true) {
            Foo foo = (Foo) i.next();
      }
  } catch (NoSuchElementException e) {  }
```

Этот пример так же плох, как и пример с просмотром массива в цикле, приведенный в начале статьи. Идиома, использующая исключения, отличается от стандартной идиомы не только многословностью и запутанностью, но также худшей производительностью и способностью скрывать ошибки, возникающие в других, несвязанных с нею частях системы.

В качестве альтернативы отдельному методу про верки состояния можно использовать особый зависящий от состояния метод: он будет возвращать особое значение, например null,при вызове для объекта, имеющего неподходящее состояние. Для класса Iteratorэтот прием не годится, поскольку null является допустимым значением для метода next.

Приведем некоторые рекомендации, которые помогут вам сделать выбор между методом проверки состояния и особым возвращаемым значением. Если к объекту возможен одновременный доступ без внешней синхронизации или если смена его СОСТОЛШИЙ инициируется извне, может потребоваться прием с особым возвращаемым

значением, поскольку состояние объекта может поменяться в период между вызовом метода, который проверяет состояние, и вызовом соответствующего метода, который зависит от состояния объекта. Особое возвращаемое значение может потребоваться для повышения производительности, когда метод проверки состояния, может при необходимости дублировать работу метода, зависящего от состояния объекта. Однако при прочих равных условиях метод проверки состояния предпочтительнее особого возвращаемого значения. При его использовании легче читать текст программы, а также проще обнаруживать и исправлять неправильное построение программы.

Применяйте обрабатываемые исключения для восстановления, для программных ошибок используйте исключения времени выпол

В языке программирования Java предусмотрены три типа объектов Throwable: *обрабатываемые исключения* (checked exception), *исключения времени выполнения (run-time* exception) и *ошибки* (error). Программисты обычно путают, при каких условиях следует использовать каждый из этих типов. Решение не всегда очевидно, но есть несколько общих правил, в значительной мере упрощающих выбор.

Основное правило при выборе между обрабатываемым и необрабатываемым исключениями гласит: используйте обрабатываемые исключения для тех условий, когда есть основания полагать, что инициатор вызова способен их обработать. Генерируя обрабатываемое исключение, вы принуждаете инициатора вызова обрабатывать его в операторе catch или передавать дальше. Каждое обрабатываемое исключение, которое, согласно декларации, инициирует некий метод, является, таким образом, серьезным предупреждением для пользователя API о том, что при вызове данного метода могут возникнуть соответствующие условия.

Предоставляя пользователю API обрабатываемое исключение, разработчик API передает ему право осуществлять обработку соответствующего условия. Пользователь может пренебречь этим правом, перехватив исключение и проигнорировав его. Однако, как правило, это оказывается плохим решением (статья 47).

Есть два типа необрабатываемых объектов Throwable: исключения времени выполнения и ошибки. Поведение у них одинаковое: ни тот, ни другой не нужно и, вообще говоря, нельзя перехватывать. Если программа инициирует необрабатываемое исключение или ошибку, то, как правило, это означает, что восстановление невозможно и дальнейшее выполнение программы принесет больше вреда, чем пользы. Если программа не перехватывает такой объект, его появление вызовет остановку текущего потока команд с соответствующим сообщением об ошибке.

Используйте исключения времени выполнении для индикации программных ошибок. Подавляющее большинство исключений времени выполнения сообщает о нарушении предусловий (precondition violation).Нарушение предусловия означает

лишь то, что клиент API не смог выполнить соглашения, заявленные в спецификации к этому API. Например, в соглашениях для доступа к массиву оговаривается, что индекс массива должен попадать в интервал от нуля до "длина массива минус один". Исключение ArrayIndexOutOfBounds указывает, что это предусловие было нарушено.

Хотя в спецификации языка Javaэто не оговорено, существует строго соблюдаемое соглашение о том, что ошибки зарезервированы в JVM для того, чтобы фиксировать дефицит ресурсов, нарушение инвариантов и другие условия, делающие невозможным дальнейшее выполнение программы [Chan98, HorstmanOO]Поскольку эти соглашения признаны практически повсеместно, лучше для Error вообще не создавать новых подклассов. Все реализуемые вами необрабатываемые исключения должны прямо или косвенно наследовать класс RuntimeException.

Для исключительной ситуации можно определить класс, который не наследует классов Exception, RuntimeException и Error. В спецификации· языка Javaтакие классы напрямую не оговариваются, однако неявно подразумевается, что они будут вести себя так же, как обычные обрабатываемые исключения (которые являются подклассами класса Exception, но не RuntimeException). Когда же вы должны использовать этот класс? Если одним словом, то никогда. Не имея никаких преимуществ перед обычным обрабатываемым исключением, он будет запутывать пользователей вашего API.

Подведем итоги. для ситуаций, когда можно обработать ошибку и продолжить исполнение, используйте обрабатываемые исключения, для программных ошибок применяйте исключения времени выполнения. Разумеется, ситуация не всегда однозначна, как белое и черное. Рассмотрим случай с исчерпанием ресурсов, которое может быть вызвано программной ошибкой, например, размещением в памяти неоправданно большого массива, или настоящим дефицитом ресурсов. Если исчерпание ресурсов вызвано временным дефицитом или временным увеличением спроса, эти условия вполне могут быть изменены. Именно разработчик API принимает решение, возможно ли восстановление работоспособности программы в конкретном случае исчерпания ресурсов. Если вы считаете, что работоспособность можно восстановить, используйте обрабатываемое исключение. В противном случае применяйте исключение времени выполнения. Если неясно, возможно ли восстановление, то по причинам, описанным в статье 41,лучше остановиться на необрабатываемом исключении.

Разработчики API часто забывают, что исключения - это вполне законченные объекты, для КОТОРЫХ можно определять любые методы. Основное назначение таких методов - создание кода, который увязывал бы исключение с дополнительной информацией об условии, вызвавшем появление данной исключительной ситуации. Если таких методов нет, программистам придется разбираться со строковым представлением этого исключения, выуживая из него дополнительную информацию. Эта крайне плохая практика. Классы редко указывают какие-либо детали в своем строковом представлении, само строковое представление может меняться от реализации к реализации, от версии к версии. Следовательно, программный код, который анализирует строковое представление исключения, скорее всего окажется непереносимым и ненадежным.

Поскольку обрабатываемые исключения обычно указывают на ситуации, когда возможно продолжение выполнения, для такого типа исключений важно создать методы, которые предоставляли бы клиенту информацию, помогающую возобновить работу. Предположим, что обрабатываемое исключение инициируется при неудачной попытке позвонить с платного телефона из-за того, что клиент не предоставил достаточной суммы денег. Для этого исключения должен быть реализован метод доступа, который запрашивает недостающую сумму с тем, чтобы можно было сообщить о ней пользователю телефонного аппарата.

Избегайте ненужных обрабатываемых исключений

Обрабатываемые исключения -' замечательная особенность языка программирования Java. В отличие от возвращаемых кодов, они *заставляют* программиста отслеживать условия возникновения исключений, что значительно повышает надежность приложения. Это означает, что злоупотребление обрабатываемыми исключениями может сделать API менее удобным для использования. Если метод инициирует одно или несколько обрабатываемых исключений, то в программном коде, из которого этот метод был вызван, должна присутствовать обработка этих исключений в виде одного или нескольких блоков catch, либо должно быть декларировано, что этот код сам инициирует исключения и передает их дальше. В любом случае перед программистом стоит нелегкая задача.

Такое решение оправданно, если даже при надлежащем применении интерфейса API невозможно предотвратить возникновение условий для исключительной ситуации, *однако* программист, пользующийся данным API, столкнувшись с этим исключением, мог бы предпринять какие-либо полезные действия. Если не выполняются оба этих условия, лучше пользоваться необрабатываемым исключением. Роль лакмусовой бумажки в данном случае играет вопрос: как программист будет обрабатывать исключение? Является ли это решение лучшим:

```
} catch(TheCheckedException e) {
   throw new Error("Assertion error");
        // Условие не выполнено. Этого не должно быть никогда!
        }
```

А что скажете об этом:

```
} catch(TheCheckedException e) {
e.printStackTrace();                    //Ладно, закончили работу.
System. exit( 1);  }
```

Если программист, применяющий API, не может сделать ничего лучшего, то больше подходит необрабатываемое исключение. Примером исключения, не выдерживающего подобной проверки, является CloneNotSupportedException. Оно инициируется

методом Object. clone, который должен использоваться лишь для объектов, реализующих интерфейс Cloneable (статья 10). Блок catch практически всегда соответствует невыполнению утверждения. Так что обрабатываемое исключение не дает программисту преимуществ, но требует от последнего дополнительных усилий и усложняет программу.

Дополнительные действия со стороны программиста, связанные с обработкой обрабатываемого исключения, значительно увеличиваются, если это *единственное* исключение, инициируемое данным методом. Если есть другие исключения, метод будет стоять в блоке try, так что для этого исключения понадобится всего лишь еще один блок catch. Если же метод инициирует только одно обрабатываемое исключение, оно будет требовать, чтобы вызов соответствующего метода был помещен в блок try. В таких условиях имеет смысл подумать: не существует ли какого-либо способа избежать обрабатываемого исключения.

Один из приемов, позволяющих превратить обрабатываемое исключение в необрабатываемое, состоит в раз биении метода, инициирующего исключение, на два метода, первый из которых будет возвращать *булево значение,* указывающее, будет ли инициироваться исключение. Таким образом, в результате преобразования API последовательность вызова

```
// Вызов с обрабатываемым исключением
try {
      obj.action(args);
      catch(TheCheckedException e) {
      // Обработать исключительную ситуацию
      }
```

принимает следующий вид:

```
// Вызов с использованием метода проверки состояния
// и необрабатываемого исключения
if (obj.actionPermitted(args)) {
      obj.action(args); }
      else{ }
      // Обработать исключительную ситуацию
```

Такое преобразование можно использовать не всегда, Если же оно допустимо, это может сделать работу с AP I более удобной. Хотя второй вариант последовательности вызова выглядит не лучше первого, полученный AP I имеет большую гибкость, В ситуации, когда программист знает, что вызов будет успешным, или согласен на завершение потока в случае неудачного вызова, преобразованный AP I позволяет использовать следующую упрощенную последовательность вызова:

```
obj,aotion(args):
```

Если вы предполагаете, что применение упрощенной последовательности вызова будет нормой, то описанное преобразование API! приемлемо. API, полученный в результате этого преобразования, в сущности, тот же самый, что и API! с методом "проверки состояния" (статья 39). Следовательно, к нему относятся те же самые предупреждения: если к объекту одновременно и без внешней синхронизации могут иметь доступ сразу несколько потоков или этот объект может менять свое состояние по команде извне, указанное преобразование использовать не рекомендуется. Это связано с тем, что в промежутке между вызовом actionPermitted и вызовом action состояние объекта может успеть поменяться. Если метод actionPermitted при необходимости и мог бы дублировать работу метода action, то от преобразования, вероятно, стоит отказаться по соображениям производительности.

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 когда и обычно этого добиваются. Общее правило, гласящее, что повторно используемый код - это хорошо, относится и к технологии исключении. В библиотеках для платформы Java реализован основной набор необрабатываемых исключений, перекрывающий большую часть потребностей в исключениях для API. В этой статье обсуждаются наиболее часто применяемые исключения.

Повторное использование уже имеющихся исключений имеет несколько преимуществ. Главное то, что они упрощают освоение и применение вашего API, поскольку соответствуют установленным соглашениям, с которыми программисты уже знакомы. С этим же связано второе преимущество, которое заключается в том, что программы, использующие ваш API, легче читать, поскольку там нет незнакомых, сбивающих с толку исключений. Наконец, чем меньше классов исключений, тем меньше требуется места в памяти и времени на их загрузку.

Чаще всего используется исключение IllegalArgumentException. Обычно оно инициируется, когда вызываемому методу передается аргумент с неправильным значением. Например, IllegalArgumentException может инициироваться в случае, если для параметра, указывающего количество повторов для некоей процедуры, передано Отрицательное значение.

Другое часто используемое исключение - IllegalStateException. Оно обычно инициируется, если в соответствии с состоянием объекта вызов метода является неправомерным. Например, это исключение может инициироваться, 'Когда делается попытка использовать некий объект до его инициализации надлежащим образом.

Вряд ли можно утверждать, что все неправильные вызовы методов сводятся к неправ ильным аргументам или неправильному состоянию, поскольку для определенных типов неправильных аргументов и состояний стандартно используются совсем другие

исключения. Если при вызове какому-либо параметру было передано null, тогда как значения null для него запрещены, то в этом случае в соответствии с соглашениями должно инициироваться исключение NullPointerException, а не IllegalArgumentException. Точно так же, если параметру, который соответствует индексу не коей последовательности, при вызове было передано значение, выходящее за границы допустимого диапазона, инициироваться должно исключение IndexOutOfBoundsException, а не IllegalArgumentException.

Еще одно универсальное исключение, о котором необходимо знать: ConcurrentModificationException. ()но должно инициироваться, когда объект, предназначавшийся для работы в одном потоке или с внешней синхронизацией, обнаруживает, что его изменяют (или изменили) из параллельного потока.

Последнее универсальное исключение, заслуживающее упоминания,- UnsupportedOperationException. ()но инициируется, если объект не имеет поддержки производимой операции. По сравнению с другими исключениями, обсуждавшимися в этой статье, UnsupportedOperationException применяется довольно редко, поскольку большинство объектов обеспечивает поддержку всех реализуемых ими методов. Это исключение используется при такой реализации интерфейса, когда отсутствует поддержка одной или нескольких заявленных в нем дополнительных функций. Например, реализация интерфейса List, имеющая только функцию добавления элементов, будет инициировать это исключение, если кто-то попытается удалить элемент.

В таблице 8.1 собраны самые распространенные из повторно используемых исключений.

Часто используемые исключения

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 имеет поддержки указанного метода

Помимо перечисленных исключений, при определенных обстоятельствах могут применяться и другие исключения. Например, при реализации таких арифметических объектов, как комплексные числа и матрицы, уместно пользоваться исключениями ArithmeticException и NumberFormatException. Если исключение отвечает вашим

потребностям - пользуйтесь им, но только чтобы условия, при которых вы будете его инициировать, не вступали в противоречие с документацией к этому исключению. Выбирая исключение, следует исходить из его семантики, а не только из названия. Кроме того, если вы хотите дополнить имеющееся исключение информацией об отказе (статья 45), не стесняйтесь создавать для него подклассы.

И наконец, учитывайте, что выбор исключения - не всегда точная наука, поскольку "поводы для использования", приведенные в таблице 8.1, не являются взаимоисключающими. Рассмотрим, например, объект, соответствующий колоде карт. Предположим, что для него есть метод, осуществляющий выдачу карт из колоды, причем в качестве аргумента ему передается количество требуемых карт. Допустим, что при вызове с этим параметром было передано значение, превышающее количество карт, оставшихся в колоде. Эту ситуацию можно толковать как IllegalArgumentException (значение параметра "размер сдачи" слишком велико) либо как IllegalStateException (объект "колода" содержит слишком мало карт для обработки запроса). В данном случае, по-видимому, следует использовать IllegalArgumentException, но непреложных правил здесь не существует.

Инициируйте исключения, соответствующие абстракции

Если метод инициирует исключение, не имеющее видимой связи с решаемой задачей, это сбивает с толку. Часто это происходит, когда метод передает исключение, инициированное абстракцией нижнего уровня. Это не только приводит в замешательство, но и засоряет интерфейс верхнего уровня деталями реализации. Если в следующей версии реализация верхнего уровня поменяется, то также могут поменяться и инициируемые им исключения, в результате чего могут перестать работать имеющиеся клиентские программы.

Во избежание этой проблемы верхние уровни приложения должны перехватывать исключения нижних уровней и, в свою очередь, инициировать исключения, которые можно объяснить в терминах абстракции верхнего уровня. Описываемая идиома, которую мы называем трансляцией исключении. (exception translation), выглядит следующим образом:

```
// Трансляция исключения
try {
    // Использование абстракции нижнего уровня
    // для выполнения наших указаний
  } catch(LowerLevelException e) {
    throw new HigherLevelException( ... );  }
```

Приведем конкретный пример трансляции исключения, взятый из класса AbstractSequentialList, который представляет собой *скелетную реализацию* (статья 16) интерфейса List. В этом примере трансляция исключения продиктована спецификацией метода get в интерфейсе List:

```
/**
 * Возвращает элемент, находящийся в указанной позиции
 * в заданном списке.
 * @throws IndexOutOfBoundsException, если индекс находится
 * за пределами диапазона (index < 0II index >=size()).
 */
public Object get(int index) {
      ListIterator i = listIterator(index);
      try {
            return i. next();
         catch (NoSuchElementException e) {
            throw new IndexOutOfBoundsException("Index: " + index); }
            }
```

В тех случаях, когда исключение нижнего уровня может быть полезно при анализе ситуации, вызвавшей исключение, лучше использовать особый вид трансляции исключений, называемый *сцеплением исключении* (exception chaining). При этом исключение нижнего уровня передается с исключением верхнего уровня; в последнем создается открытый метод доступа, позволяющий извлечь исключение нижнего уровня:

```
// Сцепление исключений
try {
      // Использование абстракции нижнего уровня
      // для выполнения наших указаний

   } catch(LowerLevelException e) {
      throw new H1gherLevelException(e);  }
```

в версии 1.4 сцепление исключений поддерживается классом Throwable. В этой и последующих версиях можно использовать преимущества такой поддержки, связав ваш конструктор исключения верхнего уровня с конструктором Throwable(Throwable):

```
// Сцепление исключений в версии 1.4
HigherLevelException(Throwable t) {
      super(t);   }
```

Если же вы используете более раннюю версию, ваше исключение должно само сохранять исключение нижнего уровня и предоставлять соответствующий метод доступа:

```
// Сцепление исключений в версии, предшествующей 1.4
private Throwable cause;
HigherLevelException(Throwable t) {
cause = t; }

public Throwable getCause() {

return cause;  }
```

Дав методу доступа название getCause и применив указанную декларацию, вы получите гарантию того, что ваше исключение будет взаимодействовать с механизмом сцепления исключений для данной платформы так, как если бы вы использовали исключение в версии 1.4. Это дает то преимущество, что в исключение верхнего уровня стандартным образом будет интегрирована трассировка стека для исключения нижнего уровня. Кроме того, это позволяет задействовать стандартные средства отладки для доступа к исключению нижнего уровня.

Хотя трансляция исключений лучше, чем бессмысленная передача исключений с нижних уровней, злоупотреблять *ею* не следует. Самый хороши" способ обработки исключений нижнего уровня - полностью исключить их возможность. Для этого перед выбором метода нижнего уровня необходимо убедиться в том, что он будет выполнен успешно, иногда добиться этого можно путем явной проверки аргументов метода верхнего уровня перед их передачей на нижний уровень.

Если предупредить появление исключений на нижних уровнях невозможно, то лучшее решение состоит в том, чтобы верхний уровень молча обрабатывал эти исключения, изолируя клиента от проблем нижнего уровня. В таких условиях чаще всего достаточно протоколировать исключения, используя какой-либо механизм регистрации, например java,util,logging, появившийся в версии 1.4. Это дает возможность администратору исследовать возникшую проблему и в то же время изолирует от нее программный код клиента и конечного пользователя.

В ситуациях, когда невозможно предотвратить возникновение исключений на нижних уровнях или изолировать от них верхние уровни, как правило, должен применяться механизм трансляции исключений. Непосредственную передачу исключений с нижележащего уровня на верхний следует разрешать только тогда, когда, исходя из описания метода на нижнем уровне, можно дать гарантию, что все инициируемые им исключения будут приемлемы для абстракции верхнего уровня.

Для каждого метода документируйте все инициируемые исключе

Описание инициируемых методом исключений составляет важную часть документации, которая необходима для правильного применения метода. Поэтому крайне важно, чтобы вы уделили время тщательному описанию всех исключений, инициируемых каждым методом.

Обрабатываемые исключения всегда декларируйте по отдельности с помощью тега @throws (Javadoc), четко описывайте условия, при которых каждое из них инициируется. Не пытайтесь сократить описание, объявляя о том, что метод инициирует некий суперкласс исключений, вместо того, чтобы декларировать несколько классов возможных исключений. Например, никогда не объявляйте, что метод инициирует исключение Exceptionили, что еще хуже, исключение ThrowableПомимо того, что такая формулировка не дает программисту никакой информации о том, какие исключения могут быть инициированы данным методом, она значительно затрудняет работу с методом, поскольку надежно перекрывает любое другое исключение, которое может быть инициировано в этом же месте.

Хотя язык Java не требует, чтобы программисты декларировали необрабатываемые исключения, которые могут быть инициированы данным методом, имеет смысл документировать их столь же тщательно, как и обрабатываемые исключения. Необрабатываемые исключения обычно представляют ошибки программирования (статья 40), ознакомление программиста со всеми этими ошибками может помочь ему избежать их. Хорошо составленный перечень необрабатываемых исключений, которые может инициировать метод, фактически описывает *предусловия* для его успешного выполнения. Важно, чтобы в документации к каждому методу были описаны его предусловия, а описание необрабатываемых исключений как раз и является наилучшим способом выполнения этого требования.

Особенно важно, чтобы для методов интерфейса были описаны необрабатываемые исключения, которые могут быть ими инициированы. Т акая документация 'является частью *основных соглашении* для интерфейса и обеспечивает единообразное поведение различных его реализаций.

Для описания каждого необрабатываемого исключения, которое может быть инициировано методом, используйте тег @throws(Javadoc), однако не нужно с помощью ключевого слова throws включать необрабатываемые исключения в декларацию метода. Программист, пользующийся вашим API, должен знать, какие из исключений обрабатываются, а какие - нет, поскольку в первом и втором случаях на него возлагается различная ответственность. Наличие описания, соответствующего тегу @throws, и отсутствие заголовка к методу, соответствующего декларации throws, создает мощный визуальный сигнал, помогающий программисту отличить обрабатываемые исключения от необрабатываемых.

Следует отметить, что документирование всех необрабатываемых исключений, которые могут быть инициированы каждым методом,- это идеал, который не всегда достижим в реальности. Когда производится пересмотр класса и предоставляемый пользователю метод меняется так, что начинает инициировать новые необрабатываемые исключения,' это не является нарушением совместимости ни на уровне исходных текстов, ни на уровне байт-кода. Предположим, некий класс вызывает метод из другого класса, написанного независимо. Авторы первого класса могут тщательно документировать все необрабатываемые исключения, инициируемые каждым методом. Однако если второй класс был изменен так, что теперь он инициирует дополнительные необрабатываемые исключения, первый класс (не претерпевший изменений) тоже будет передавать эти новые необрабатываемые исключения, хотя он их и не декларировал.

Если одно и то же исключение по одной и то же причине инициируется несколькими методами, его описание можно поместить в общий комментарий к документации для всего класса, а не описывать его отдельно для каждого метода. Примером такого рода является исключение NullPointerException. Прекрасно было бы в комментарии к классу сказать "все методы этого класса инициируют исключение NullPointerException, если с каким-либо параметром была передана нулевая ссылка на объект" или другие слова с тем же смыслом.

Вописание исключения добавляйте информацию

Если выполнение программы завершается аварийно из-за необработанного исключения, система автоматически распечатывает трассировку стека для этого исключения. Трассировка стека содержит *строковое представление* данного исключения, результат вызова его метода toString. Обычно это представление состоит из названия класса исключения и *описания исключения* (detail message). Часто это единственная информация, с которой приходится иметь дело программистам или специалистам по наладке, исследующим сбой программы. И если воспроизвести этот сбой нелегко, то получить какую-либо еще информацию будет трудно или даже вообще невозможно. Поэтому крайне важно, чтобы метод toString в классе исключения возвращал как можно больше информации о причинах отказа. Иными словами, строковое представление исключения должно зафиксировать отказ для последующего анализа.

Для фиксации сбоя строковое представление исключения должно содержать значения всех параметров и полей, "способствовавших появлению этого исключения". Например, описание исключения IndexOutOfBounds должно содержать нижнюю границу, верхнюю границу и действительный индекс, который .не уложился в эти границы. Такая информация говорит об отказе очень многое. Любое из трех значений или все они вместе могут быть неправильными. Представленный индекс может оказаться на единицу меньше нижней границы или быть равен верхней границе ("ошибка

границы" - fencepost error) либо может иметь несуразное значение, как слишком маленькое, так и слишком большое. Нижняя граница может быть больше верхней (серьезная ошибка нарушения внутреннего инварианта). Каждая из этих ситуаций указывает на свою проблему, и если программист знает, какого рода ошибку следует искать, это в огромной степени облегчает диагностику.

Хотя добавление в строковое представление исключения всех относящихся к делу "достоверных данных" является критическим, обычно нет надобности в том, чтобы оно было пространным. 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 важней его вразумительности.

Один из приемов, гарантирующих, что строковое представление исключения будет содержать информацию, достаточную для Описания сбоя, состоит в том, чтобы эта информация запрашивалась в конструкторах исключения, а в строке описания. Само же описание исключения можно затем генерировать автоматически для представления этой информации. Например, вместо конструктора String исключение IndexOutOfBounds могло бы иметь следующий конструктор:

```
/**
* Конструируем IndexOutOfBoundsException
* @param lowerBound – самое меньшее из разрешенных значений индекса
 * @param upperBound – самое большее из разрешенных значений индекса плюс один
* @param index  – действительное значение индекса
*/
Public IndexOutOfBoundsExoeption(int lowerBound, int index) {
// Генерируем описание исключения,
// фиксирующее обстоятельства отказа
super( "Lower bound: " + lowerBound +
“,Upper bound: " + uppe rBound +
        “,Index: "          + index);
}
```

К сожалению, хотя ее очень рекомендуют, эта идиома в библиотеках для платформы Java используется не слишком интенсивно. С ее помощью программист, инициирующий исключение, может с легкостью зафиксировать обстоятельства сбоя: Вместо того чтобы заставлять каждого пользующегося классом генерировать свое строковое представление, в этой идиоме собран фактически весь код, необходимый для того, чтобы 'качественное строковое представление генерировал сам класс исключения.

Как отмечалось в статье 40, возможно, имеет смысл, чтобы исключение предоставляло методы доступа к информации об обстоятельствах сбоя (в представленном выше примере это lowerBound, upperBound и Index). Наличие таких методов доступа для обрабатываемых исключений еще важнее, чем для необрабатываемых, поскольку информация об обстоятельствах сбоя может быть полезна для восстановления работоспособности программы. Программный доступ к деталям необрабатываемого исключения редко интересует программистов (хотя это и не исключено). Однако, согласно общему принципу (статья 9), такие методы доступа имеет смысл создавать .даже для необрабатываемых исключений.

Добивайтесь атомарности методов по отношению к сбоям

После того как объект инициирует исключение, обычно необходимо, Ч1'обы он оставался во вполне определенном, пригодном для дальнейшей обработки состоянии, даже несмотря на то, что сбой произошел непосредственно в процессе ВЫI1Dлнения -операции. Особенно это касается обрабатываемых исключений, когда предполагается, что клиент будет восстанавливать работоспособность программы. Вообще говоря, вызов метода, завершившийся сбоем, должен оставлять обрабатываемый объект в том же состоянии, в каком тот был перед вызовом. Метод, обладающий таким свойством, называют *атомарным по отношению к сбою* (failure atomic).

Добиться такого эффекта можно несколькими способами'. Простейший способ заключается в создании неизменяемых объектов (статья 13). Если объект неизменяемый, получение атомарности не требует усилий. Если операция заканчивается сбоем, это может помешать созданию нового объекта, но никогда не оставит уже имеющийся объект в неопределенном состоянии, поскольку состояние каждого неизменяемого объекта согласуется в момент его создания и после этого уже не меняется.

для методов, работающих с изменяемыми объектами, атомарность по отношению к сбою чаще всего достигается путем проверки правильности параметров перед выполнением операции (статья 23). Благодаря этому, любое исключение будет инициироваться до того, как начнется модификация объекта. В качестве примера рассмотрим метод Staok.pop из статьи 5:

```
public Objeot pop() {
if (size == 0)
        throw new EmptyStaokExoeption();
```

```
Object result = elements[-size];
        elements[size] = null;    // Убираем устаревшую ссылку
return result;
```

Если убрать начальную проверку размера, метод все равно будет инициировать исключение при попытке получить элемент из пустого стека. Однако при этом он будет оставлять поле size в неопределенном (отрицательном) состоянии. А это приведет к тому, что сбоем будет завершаться вызов любого метода в этом объекте. Кроме того, само исключение, инициируемое методом pop, не будет соответствовать текущему уровню абстракции (статья 43).

Другой прием, который тесно связан с предыдущим и позволяет добиться атомарности по отношению к сбоям, заключается в упорядочении вычислений таким образом, чтобы все фрагменты кода, способные повлечь сбой, предшествовали первому фрагменту, который модифицирует объект. Такой прием является естественным расширением предыдущего в случаях, когда невозможно произвести проверку аргументов, не выполнив хотя бы части вычислений. Например, рассмотрим случай с классом TreeMap, элементы которого сортируются по не коему правилу. Для того чтобы в экземпляр TreeMap можно было добавить элемент, последний должен иметь такой тип, который допускал бы сравнение с помощью процедур, обеспечивающих упорядочение TreeMap. Попытка добавить элемент неправильного типа, естественно, закончится сбоем (и исключением ClassCastException), который произойдет в процессе поиска этого элемента в дереве, но до того, как в этом дереве что-либо будет изменено.

Третий, редко встречающийся прием, заключается в написании специального *кода восстановления* (recovery code), который перехватывает сбой, возникающий в ходе выполнения операции, и заставляет объект вернуться в то состояние, в котором он находился в момент, предшествующей началу операции. Этот прием используется главным образом для структур, записываемых в базу данных.

Наконец, последний прием, позволяющий добиться атомарности метода, заключается в том, чтобы выполнять операцию на временной копии объекта, и как только операция будет завершена, замещать содержимое объекта содержимым его временной копии. Такой прием подходит для случая, когда вычисления могут быть выполнены намного быстрее, если поместить данные во временную структуру. Например, метод Collections.sort перед выполнением сортировки загружает полученный список в некий массив с тем, чтобы облегчить доступ к элементам во время внутреннего цикла сортировки. Это сделано для повышения производительности, однако имеет и другое дополнительное преимущество - гарантию того, что предоставленный методу список останется нетронутым, если процедура сортировки завершится сбоем.

К сожалению, не всегда можно достичь атомарности по отношению к отказам. Например, если два потока одновременно, без должной синхронизации пытаются модифицировать некий объект, последний может остаться в неопределенном состоянии. А потому после перехвата исключения ConcurrentModificationException нельзя

полагаться на то, что объект все еще при годен к использованию. Ошибки (в отличие от исключений), как правило, невосстановимы, и потому методам не нужно даже пытаться сохранять атомарность в случае появления ошибки.

Даже там, где можно получить атомарность по отношению к сбоям, она не всегда желательна. Для некоторых операций она существенно увеличивает затраты ресурсов и Сложность вычислений. Вместе с тем очень часто это свойство достигается без особого Труда, если хорошо разобраться с проблемой. Как правило, любое исключение, добавленное в спецификацию метода, должно оставлять объект в том состоянии, в котором он находился до вызова метода. В случае нарушения этого правила в документации API Должно быть четко указано, в каком состоянии будет оставлен объект. К сожалению, множество имеющейся документации к API не стремится достичь этого идеала.

Не игнорируйте исключений

Этот совет кажется очевидным, но он нарушается настолько часто, что заслужи вает повторения. Когда разработчики API декларируют, что некий метод инициирует исключение, этим они пытаются что-то вам сkiзать. Не игнорируйте это! Игнориро вать исключения легко: необходимо всего лишь Окружить вызов метода оператором try с пустым блоком catch:

```
// Пустой блок catch игнорирует исключение - крайне
// подозрительный код!
try {
} catch (SomeException e) { }
```

Пустой блок catch лишает исключение смысла, который состоит в том, чтобы вы обрабатывали исключительную ситуацию. Игнорировать исключение это все равно, что игнорировать пожарную тревогу: выключить сирену, чтобы больше ни у кого не было возможности узнать, есть ли здесь настоящий пожар. Либо вам удастся Всех обмануть, либо результаты окажутся катастрофическими. Когда бы вы ни увиде ли пустой блок catch, в вашей голове должна включаться сирена. Блок catch обязан содержать, по крайней мере, комментарий, объясняющий, почему данное исклю чение следует игнорировать.

Ситуацию, когда игнорирование исключений может оказаться целесообразным, Иллюстрирует такой пример как визуализация изображений в мультипликации. Если экран обновляется через равные промежутки времени, то, возможно, лучший способ Справиться с временным сбоем - игнорировать его и подождать следующего обновления экрана.

Представленная в этой статье рекомендация в равной степени относится как к обрабатываемым, так и к необрабатываемым исключениям. Вне зависимости от того, представляет ли исключение предсказуемое условие или программную ошибку, если оно игнорируется и используется пустой блок catch, то в результате программа, столкнувшись с ошибкой, будет работать дальше, никак на нее не реагируя. Затем в любой произвольный момент времени программа может завершиться с ошибкой, и программный код, где это произойдет, не будет иметь никакого отношения к действительному источнику проблемы. Должным образом обработав исключение, вы можете избежать отказа. Даже простая передача необрабатываемого исключения вовне вызовет, по крайней мере, быстрый останов программы, при котором будет сохранена информация, полезная при устранении сбоя.

Глава 9

Потоки

Потоки позволяют выполнять одновременно несколько операций в пределах одной программы. Многопоточное программирование сложнее однопоточного, так что совет из статьи 30 здесь особенно актуален: если существует библиотечный класс, который может оградить вас от многопоточного программирования низкого уровня, во что бы то ни стало воспользуйтесь им. Одним из примеров таких классов является java.util.Titer. Второй при мер - пакет util.concurrent Дага Ли (Doug Lea) [Lea01], содержащий целый набор утилит высокого уровня для управления потоками. Но,несмотря на наличие библиотек, вам все равно время от времени приходится писать или поддерживать программный код для многопоточной обработки. В этой главе содержатся советы, которые помогут вам создавать понятные, правильные и хорошо документированные программы для работы с потоками.

Синхронизируйте доступ потоков к совместно используемым изменяе

Использование ключевого слова synchronized дает гарантию, что в данный момент времени некий оператор или блок будет выполняться только в одном потоке. Многие программисты рассматривают синхронизацию лишь как средство блокировки потоков, которое не позволяет одному потоку наблюдать объект в промежуточном состоянии, пока тот модифицируется другим потоком. С этой точки зрения, объект создается с согласованным состоянием (статья 13), а затем блокируется методами, имеющими к нему доступ. Эти методы следят за состоянием объекта и (дополнительно) могут вызывать для него *переход состояния* (state transition), переводя объект из одного согласованного состояния в другое. Правильное выполнение синхронизации гарантирует, что ни один метод никогда не сможет наблюдать этот объект в промежуточном состоянии.

Такая точка зрения верна, но не отражает всей картины. Синхронизация не только запрещает потоку наблюдать объект в промежуточном состоянии, она также дает гарантию, что объект будет переходить из одного согласованного состояния в другое в результате выполнения четкой последовательности шагов. Каждый поток, попадая в синхронизированный метод или блок, видит результаты выполнения всех предыдущих переходов под управлением того же самого кода блокировки. После того как поток покинет синхронизированную область, любой поток, попадающий в область; синхронизированную с помощью той же блокировки, увидит результат перехода в новое состояние, осуществленного предыдущим потоком (если переход имел место).

Язык Java гарантирует, что чтение и запись отдельной переменной, если это не переменная типа long или double, являются атомарными операциями. Иными словами, чтение переменной (кроме long и double) будет возвращать значение, которое было записано в эту переменную одним из потоков, даже если несколько потоков без какой-либо синхронизации одновременно записывают новые значения в эту переменную.

Возможно, вы слышали, что для повышения производительности при чтении и записи атомарных данных нужно избегать синхронизации. Это неправильный совет с опасными последствиями. Хотя свойство атомарности гарантирует, что при чтении атомарных данных поток не увидит случайного значения, нет гарантии, что значение, записанное одним потоком, будет увидено другим: синхронизация необходима как для блокирования потоков, так и для надежного взаимодействия между ними. Это является следствием сугубо технического аспекта языка программирования Java, который называется *моделью памяти* (тетогу model) [JLS, 17]. Вероятно, в ближайшей версии модель памяти будет существенно пересмотрена [Pugh01a], однако описанная особенность скорее всего не поменяется.

Отсутствие синхронизации для доступа к совместно используемой переменной может иметь серьезные последствия, даже если переменная имеет свойство атомарности как при чтении, так и при записи. Рассмотрим следующую функцию генерации серийного номера:

```
// Ошибка: требуется синхронизация private static int
nextSerialNumber = 0;

public static int generateSerialNumber() return nextSerialNumber+
+;
```

Эта функция должна гарантировать, что при каждом вызове метода generateSerialNumber будет возвращаться другой серийный номер до тех пор, пока не будет произведено вызова. Для защиты инвариантов данного генератора серийных номеров синхронизация не нужна, поскольку таковых у него нет. Состояние генератора содержит лишь одно атомарно записываемое поле (nextSerialNumber), для которого допустимы любые значения. Тем не менее без синхронизации этот метод не работает. Оператор приращения (++) осуществляет чтение и запись в поле nextSerialNumber, а потому атомарным не является. Чтение и запись - независимые операции, которые

выполняются последовательно. Несколько параллельных потоков могут наблюдать в поле nextSerialNumber одно и то же значение и возвращать один и тот же серийный номер.

Еще более удивительный случай: один поток может несколько раз вызвать метод generateSerialNumber и получить последовательность серийных номеров от 0 до n. После этого другой поток может вызвать метод generateSerialNumber и получить серийный номер, равный нулю. Без синхронизации второй поток может не *увидеть* ни одного из изменений, произведенных первым потоком. Это следствие применения вышеупомянутой модели памяти.

Исправление метода generateSerialNumber сводится к простому добавлению в его декларацию слова synchronized. Тем самым гарантируется, что различные вызовы не будут смешиваться, и каждый новый вызов будет видеть результат обработки всех предыдущих обращений. Чтобы сделать этот метод "железобетонным", возможно, имеет смысл заменить int на long или инициировать какое-либо исключение, если nextSerialNumber будет близко к переполнению.

Рассмотрим процедуру остановки потока. Платформа Java предлагает методы принудительной остановки потока, но они являются устаревшими и по своей сути *небезопасны:* работа с ними может привести к разрушению объектов. Для остановки потока рекомендуется использовать прием, заключающийся в том, что в классе потока создается некое опрашиваемое поле, которому можно присвоить новое значение, указывающее на то, что этот поток должен остановить себя сам. Обычно такое поле имеет тип Ьoolean или является ссылкой на объект. Поскольку чтение и запись этого поля атомарный, у некоторых программистов появляется соблазн предоставить к нему доступ без синхронизации. Нередко можно увидеть программный код такого рода:

```
// Ошибка: требуется синхронизация
public class StoppableThread extends Thread {
private boolean stopRequested = false;
    public void run() {
    boolean done = false;
        while (!stopRequested && !done) {
                // Здесь выполняется необходимая обработка
                }
    }
    public void requestStop() {
     stopRequested = true;  }
    }
```

Проблема приведенного кода заключается в том, что в отсутствие синхронизации нет гарантии (если ее вообще можно дать), что поток, подлежащий остановке, "увидит", что другой поток поменял значение stopRequested. В результате метод requestStop

может оказаться абсолютно неэффективным. И хотя вы вряд ли действительно столкнетесь со странным поведением программы, пока не запустите ее в многопроцессорной, системе, гарантировать ее правильную работу нельзя. Разрешить эту проблему можно, непосредственно синхронизировав любой доступ к полю stopRequested:

```
// Правильно синхронизированное совместное завершение потока
public class StoppableThread extends Thread {
    private boolean stopRequested = false;
    public void гип() {
    boolean done = false;
        while (!stopRequested() &&!done) {
                // Здесьвыполняетсянеобходимаяобработка}
                }
            public synchronized void requestStop() {
            stopRequested = true;  }

    private synchronized boolean stopRequested() {

    return stopRequested; }

    }
```

Заметим, что выполнение каждого из синхронизированных методов является атомарным: синхронизация используется исключительно для обеспечения взаимодействия потоков, а не для блокировки. Очевидно, что исправленный программный код работает, а расходы на синхронизацию при каждом прохождении цикла вряд ли можно заметить. Однако есть корректная альтернатива, которая не столь многословна, и ее ПРОИЗВ6Дительность чуть выше. Синхронизацию можно опустить, если объявить stopRequested с модификатором volatile (асинхронно-изменяемый). Этот модификатор гарантирует, что любой поток, который будет читать это поле, увидит самое последнее записанное значение.

Наказание за отсутствие в предыдущем примере синхронизации доступа к полю stopRequested оказывается сравнительно небольшим: результат вызова метода requestStop может проявиться через неопределенно долгое время. Наказание за отсутствие синхронизации доступа к изменяемым, совместно используемым данным может быть более суровым. Рассмотрим *идиому двойной проверки* (double-check) отложенной инициализации:

```
// Двойная проверка отложенной инициализации – неправильная!
private static Foo foo = null;
```

```
public static Foo getFoo() {
    if (foo == null) {
    synchronized (Foo.class)
    if (foo == null)
                    foo = new Foo(); }
                    }
                    return foo; }
```

Идея, на которой построена эта идиома, заключается в том, чтобы избежать затрат на синхронизацию доступа к уже инициализированному полю foo. Синхронизация используется здесь только для того, чтобы не позволить сразу нескольким потокам инициализировать данное поле. Идиома дает гарантию, что поле будет инициализировано не более одного раза и что все потоки, вызывающие метод getFoo, будут получать правильную ссылку на объект. К сожалению, это не гарантирует, что ссылка на объект будет работать правильно. Если поток прочел ссылку на объект без синхронизации, а затем вызывает в этом объекте какой-либо метод, может оказаться, что метод обнаружит свой объект в частично инициализированном состоянии, а это приведет к катастрофическому сбою программы.

То, что поток может видеть объект с отложенным созданием в частично инициализированном состоянии, кажется диким. Объект был полностью собран прежде, чем его ссылка была "опубликована" в поле (foo), откуда ее получат остальные потоки. Однако в отсутствие синхронизации чтение "опубликованной" ссылки на объект еще не дает гарантии, что соответствующий поток увидит все те данные, которые были записаны в память перед публикацией ссылки на объект. В частности, нет гарантии того, что поток, читающий опубликованную ссылку на объект, увидит самые последние значения данных, составляющих внутреннюю структуру этого объекта. Вообще говоря, идиома двойной проверки не работоспособна, хотя она и может действовать, если переменная, совместно используемая разными потоками, содержит простое значение, а не ссылку на объект [Pugh01b].

Решить эту проблему можно несколькими способами. Простейший из них - полностью отказаться от отложенной инициализации:

```
// Нормальная статическая инициализация (неотложенная)
 private static final Foo foo = new Foo();
public static Foo getFoo()  {
     return foo; }
```

Этот вариант, безусловно, работает, и метод getFoo оказывается настолько быстр, насколько это возможно. Здесь нет ни синхронизации, ни каких-либо еще вычислений. Как говорилось В статье 37, вы должны писать простые, понятные, правильные

программы, оставляя оптимизацию на последний момент. Приступать к оптимизации следует только тогда, когда измерения покажут, что она необходима. Поэтому отказ от отложенной инициализации, как правило, оказывается наилучшим решением. Если вы отказались от отложенной инициализации, измерили расход ресурсов и обнаружили, что он чрезмерно высок, то необходимо должным образом синхронизировать метод для выполнения отложенной инициализации:

```
// Правильно синхронизированная отложенная инициализация
private static Foo foo = null;

public static synchronized Foo getFoo() {

if (foo == null)
        foo = new Foo();
        return foo; }
```

Этот метод работает, но при каждом вызове теряется время на синхронизацию. для современных реализаций jVM эти потери сравнительно невелики. Однако если, измеряя производительность вашей системы, вы обнаружили, что не можете себе позволить ни обычную инициализацию, ни синхронизацию каждого доступа, есть еще один вариант. Идиому *класса, 8ыполняющеzо инициализацию по запросу* (initializeon-demand holder class), лучше применять в том случае, когда инициализация статического поля, занимающая много ресурсов, может и не потребоваться, однако если уж поле понадобилось, оно используется очень интенсивно. Указанная идиома представлена ниже:

```
//Идиома класса, выполняющего, инициализацию по запросу
private static class FooHolder {
    static final Foo foo = new Foo(); }
public static Foo getFoo() { return FooHolder. foo; }
```

Преимуществом этой идиомы является гарантия того, что класс не будет инициализироваться до той поры, пока он не потребуется [jLS, 12.4.1]. При первом вызове метод getFoo читает поле FooHolder. foo, заставляя класс FooHolder выполнить инициализацию. Красота идиомы заключается в том, что метод getFoo не синхронизирован и всего лишь предоставляет доступ к полю foo, так что отложенная инициализация практически не увеличивает издержек доступа. Единственным недостатком этой идиомы является то, что она не работает с экземплярами полей, а только со статическими полями класса.

Подведем итоги. Когда несколько потоков совместно работают с изменяемыми данными, каждый поток, который читает или записывает эти данные, должен пользоваться блокировкой. Пусть гарантии, связанные с атомарностью чтения и записи, не удерживают вас от выполнения правильной синхронизации. Без

синхронизации невозможно гарантировать, что изменения в объекте, сделанные одним потоком, будут 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 чрезвычайно чувствительны к деталям реализации jVM и к особенностям компьютера.

При некоторых условиях использование модификатора volatile представляет собой реальную альтернативу обычной синхронизации, однако это пока новаторский прием. Более того, границы его применимости станут известны лишь по завершении ведущихся ныне работ над моделью памяти.

Избегайте избыточной синхронизации

Статья 48 предупреждает об опасностях недостаточной синхронизации. Данная статья посвящена обратной проблеме. В зависимости от ситуации избыточная синхронизация может приводить к снижению производительности приложения, взаимной блокировке потоков или даже к непредсказуемому поведению программы.

для исключения возможности взаимной блокировки (deadlock) никогда не передавайте управление клиенту, если находитесь в синхронизированном методе или блоке. Иными словами, из области синхронизации не следует вызывать открытые или защищенные методы, которые предназначены для переопределения. (Такие методы обычно являются абстрактными, но иногда по умолчанию могут иметь определенную реализацию.) С точки зрения класса, содержащего синхронизированную область, такой метод является *чужим.* У класса нет сведений о том, что этот метод делает, нет над ним контроля. Клиент может реализовать этот метод таким образом, чтобы он создавал другой поток, выполняющий обратный вызов этого же класса. Затем вновь созданный поток может попытаться получить доступ к области, блокированной первым потоком, что приведет к блокировке нового потока. И если метод, создавший новый поток, ждет его завершения, возникает взаимная блокировка.

для пояснения рассмотрим класс, в котором реализована *очередь заданий* (work queue). Этот класс позволяет клиентам ставить задания в очередь на асинхронную обработку. Метод enqueue может вызываться столь часто, сколь это необходимо. Конструктор класса запускает фоновый поток, который удаляет из очереди записи в том порядке, в котором они были сделаны, и обрабатывает их, используя метод processItem. Если очередь заданий больше не нужна, клиент вызывает метод stop, чтобы заставить поток изящно остановиться после завершения всех заданий, находящихся в обработке.

```
public abstract class WorkQueue {
    private final List queue = new LinkedList();
    private boolean stopped = false;
```

```
    protected WorkQueue() { new WorkerThread().start(); }
    public final void enqueue(Object workItem) {
        synchronized (queue) {
            queue.add(workIt~m);
            queue.notify(); }
    }

    public final void stop() {
        synchronized (queue) {
        stopped = true;
        queue. notify(); }
    }

    protected abstract void processItem(Object workItem)
        throws InterruptedException;

    // Ошибка: вызов чужого метода из синхронизированного блока!

    private class WorkerThread extends Thread {
        public void run() {
            while (true) {    //Главный цикл
                synchronized (queue)  {
                    try {
                        while (queue.isEmpty() && $topped)
                            queue. wait();
                    }catch  (InterruptedException e) {
                        return; }
                    if (stopped)
                        return;
                    Object workItem = queue. remove(0);
                    try {
                        processItem(workItem): // Блокировка!
                    } catch (InterruptedException e) {
                        return;  }
                    }
                }
            }
        }
    }
}
```

Чтобы воспользоваться этим классом, вы должны создать для него подкласс с тем, чтобы предоставить реализацию абстрактного метода processItem. Например, следующий подкласс выводит на печать задания из очереди, не более одной записи в секунду и независимо от того, с какой скоростью в очереди появляются новые задания:

```
class DisplayQueue extends WorkQueue {
protected void processItem(Object workItem)
 throws InterruptedException {
System,out.println(workItem);
Thread.sleep(1000);  }
}
```

Поскольку класс WorkQueue вызывает абстрактный метод processItem из синхронизированного блока, для него может наступить взаимная блокировка. Действительно, работа со следующим подклассом по описанной выше схеме приведет к взаимной блокировке:

```
class DeadlockQueue extends WorkQueue {
   protected void processItem(final Object workItem)
    throws InterruptedException {
        // Создаем новый поток, который возвращает workItem в очередь
         Thread child = new Thread() {
              public void run() { enQueue(workItem); }
              };
     child. start();
     child. join();    //Взаимная блокировка!
     }
  }
```

Этот пример приведен потому, что здесь нет причин, которые заставили бы метод processItem создавать фоновый поток, однако описываемая проблема абсолютно реальна. Вызовы из синхронизированного блока методов, реализуемых за пределами класса, приводили к множеству взаимных блокировок в таких реальных системах, как программные пакеты с графическим интерфейсом пользователя. К счастью, эта проблема легко устранима. Вынесите вызов метода за пределы синхронизированного блока, как показано в следующем примере:

```
// Чужой метод за пределами синхронизированного блока
// "открытый вызов"
private class WorkerThread extends Thread {
public void run() {
          while (true) {          // Главный цикл
               Object workItem = null;
```

```
    synchronized (queue) {
        try {
                while (queue.isEmpty() && !stopped)
                        queue.wait();
        } catch (InterruptedException e) {
                return ; }
        if (stopped)
                return;
        workItem= queue. remove(0);
    }
    try {
                processItem(workItem); // Блокировки нет
    } catch (InterruptedExcepti~n
                 return; }
                }
        }
}
```

Чужой метод, который вызывается за пределами синхронизированной области, называется *открытым вызовом* (open call) [LeaOO,2.4.1.3]. Открытые вызовы не только предотвращают взаимную блокировку, но и значительно увеличивают распараллеливание вычислений. Если бы чужой метод вызывался из блокированной области и выполнялся сколь угодно долго, то все это время остальные потоки без всякой на то необходимости получали бы отказ в доступе к совместно используемому объекту.

Правило таково, что в пределах синхронизированной области нужно выполнять как можно меньше работы: заблокируйте ресурс, проверьте совместно используемые данные, преобразуйте их при необходимости и разблокируйте ресурс. Если требуется выполнение операций, отнимающих много времени, найдите способ вынести их за пределы синхронизированной области.

Вызов чужого метода из синхронизированной области может привести к более серьезным сбоям, чем просто взаимная блокировка, если он происходит в тот момент, когда инварианты, защищаемые синхронизацией, временно недействительны. (В первом примере с очередью заданий этого случиться не может, поскольку при вызове метода processItem очередь находится в непротиворечивом состоянии.) Возникновение таких сбоев не связано с созданием в чужом методе новых потоков. Это происходит, когда чужой метод делает обратный вызов пока что некорректного класса. И поскольку блокировка в языке программирования Java является *рекурсивной,* подобные обратные вызовы не приведут к взаимной блокировке, как это было бы, если бы вызовы производились из другого потока. Поток, из которого делается вызов, уже заблокировал область, и потому этот поток успешно пройдет через блокировку

во второй раз, несмотря на то, что над данными, защищенными блокировкой, в этот момент будет выполняться принципиально иная операция. Последствия такого сбоя могут быть катастрофическими - блокировка фактически не справил ась со своей работой. Рекурсивные блокировки упрощают построение многопоточных объектно-ориентированных программ, однако они могут превратить нарушение живучести в нарушение безопасности.

Мы обсудили проблемы параллельности потоков, теперь обратимся к производительности. Хотя с момента появления платформы Java расходы на синхронизацию резко сократились, полностью они не исчезнут никогда. И если часто выполняемую операцию синхронизировать без всякой необходимости, это может существенно сказаться на производительности приложения. Например, рассмотрим классы StringBufferи BufferedInputStreamЭти классы имеют поддержку много поточности (статья 52), однако почти всегда их использует только один поток, а потому осуществляемая ими блокировка, как правило, оказывается избыточной. Они содержат методы, выполняющие тонкую обработку на уровне отдельных символов или байтов, а потому не только склонны выполнять ненужную работу по блокированию потоков, но имеют тенденцию использовать множество таких блокировок. Это может привести к значительному снижению производительности. В одной из статей сообщалось почти о 20% потерь для каждого реального приложения [Heydon99]. Вряд ли вы столкнетесь со столь существенным падением производительности, обусловленным излишней синхронизацией, однако 5-100/0 потерь вполне возможны.

Можно утверждать, что все это относится к разряду "маленьких усовершенствований", о которых, как говорил Кнут, нам следует забыть (статья 37). Однако если вы пишите абстракцию низкого уровня, которая в большинстве случаев будет работать с одним единственным потоком или как составная часть более крупного синхронизированного объекта, то следует подумать над тем, чтобы отказаться от внутренней синхронизации этого класса. Независимо от того, будете вы синхронизировать класс или нет, крайне важно, чтобы в документации вы отразили его возможности при работе в многопоточном режиме (статья 52).

Не всегда понятно, следует ли в указанном классе выполнять внутреннюю синхронизацию. Перечень, приведенный в статье 52, не дает четкого ответа на вопрос, следует ли *поддерживать в классе MH020поточность* или же его нужно сделать *совместимым* с *многопоточностью.* Приведем несколько рекомендаций, которые помогут вам в выборе.

Если вы пишите' класс, который будет интенсивно использоваться в условиях, требующих синхронизации, а также в условиях, когда синхронизация не нужна, правильный подход заключается в обеспечении обоих вариантов: с синхронизацией (с поддержкой многопоточности - thread-safe) и без синхронизации (совместимый с многопоточностью - thread-compatible). Одно из возможных решений - создание *класса-оболочки* (статья 14), в котором реализован соответствующий этому классу интерфейс, а перед передачей вызова внутреннего объекта соответствующему методу выполняется необходимая синхронизация. Такой подход при меняется в Collections

Framework, а также в классе jaya.util.Random. Второй вариант решения, который можно использовать для классов, не предназначенных для расширения или повторной реализации, заключается в предоставлении класса без синхронизации, а также подкласса, состоящего исключительно из синхронизированных методов, которые вызывают соответствующие методы из суперкласса.

Хорошим поводом для внутренней синхронизации класса является то, что он будет интенсивно использоваться параллельными потоками, а также то, что вы можете добиться большего· параллелизма, выполняя тонкую внутреннюю синхронизацию. Например, можно реализовать хэш-таблицу постоянного размера, в которой доступ к каждому сегменту синхронизируется отдельно. Это обеспечивает более высокую степень распараллеливания, чем в случае блокирования всей таблицы при доступе к одному элементу.

Если класс или статический метод связан с изменяемым *статическим* полем, он должен иметь внутреннюю синхронизацию, даже если обычно применяется только с одним потоком. В противоположность совместно используемому экземпляру, здесь клиент не имеет возможности произвести внешнюю синхронизацию, поскольку нет никакой гарантии, что другие клиенты будут делать то же самое. Эту ситуацию иллюстрирует статический метод Math.random.

Подведем итоги. Во избежание взаимной блокировки потоков и разрушения данных никогда не вызывайте чужие методы из синхронизированной области. Постарайтесь ограничить объем работы, выполняемой вами в синхронизированных областях. Проектируя изменяемый класс, подумайте о том, не должен ли он иметь свою собственную синхронизацию. Выигрыш, который вы рассчитываете получить, отказываясь от синхронизации, теперь уже не такой громадный, а вполне умеренный. Ваше решение должно исходить из того, будет ли ваша абстракция использоваться для работы с несколькими потоками. Четко документируйте свое решение.

Никогда не вызывайте wait вне цикла

Метод Object.wait применяется в том случае, когда нужно заставить поток дождаться некоторого условия. Метод должен вызываться из синхронизированной области, блокирующей объект, для которого был сделан вызов. Стандартная схема использования метода wai t:

```
synchronized (obj) {
      while (<условие не выполнено>)
       obj.wait ();
           // Выполнение действия, соответствующего условию

}
```

При вызове метода wait всегда используйте идиому цикла ожидания. Никогда не вызывайте его вне цикла. Цикл нужен для проверки соответствующего условия до и после ожидания.

Проверка условия перед ожиданием и отказ от него, если условие уже выполнено, необходимы для обеспечения *живучести.* Если условие уже выполнено и перед переходом потока в состояние ожидания был вызван метод notify (или not1fyAll), нет никакой гарантии, что поток когда-нибудь выйдет из этого состояния.

Проверка условия по завершении ожидания и вновь ожидание, если условие не выполнено, необходимы для обеспечения *безопасности.* Если поток производит операцию, когда условие не выполнено, он может нарушить инварианты, защищенные блокировкой. Существует несколько причин, по которым поток может "проснуться" при невыполненном условии:

- За время от момента, когда поток вызывает метод notify, и до того момента, когда ожидающий поток проснется, другой поток может успеть заблокировать объект и поменять его защищенное состояние.
- Другой поток может случайно или умышленно вызвать метод notify, когда условие еще не выполнено. Классы подвергают себя такого рода неприятностям, если в общедоступных объектах присутствует ожидание. К этой проблеме восприимчив любой вызов wait в синхронизированном методе общедоступного объекта.
- Во время "побудки" потоков извещающий поток может вести себя слишком "щедро". Например, извещающий поток должен вызывать notifyAll, даже если условие пробуждения выполнено лишь
  для некоторых ожидающих потоков.
- Ожидающий поток может проснуться и в отсутствие извещения.
  Это называется *ложным пробуждением* (spurious wakeup).
  Хотя в *"The Java Language Specification"* [JLS] такая возможность не упоминается, во многих реализациях JVM применяются механизмы управления потоками, у которых ложные пробуждения хотя и редко, но случаются [Posix, 11.4.3.6.1].

Возникает еще один вопрос: для пробуждения ОЖИ4ающих потоков следует использовать метод notify или notifyAll? (Напомним, что метод notify будит ровно один ожидающий поток в предположении, что такой поток существует, а notifyAll будит все ожидающие потоки.) Часто говорится, что во всех случаях лучше применять метод notifyAll. Это разумный осторожный совет, который исходит из предположения, что все вызовы wait находятся в циклах while. Результаты вызова всегда будут правильными, поскольку гарантируется, что вы разбудите все требуемые потоки. Заодно вы можете разбудить еще несколько других потоков, но это не повлияет на правильность вашей программы. Эти потоки проверят условие, которого они дожидаются, и, обнаружив, что оно не выполнено, продолжат ожидание.

Метод notify можно выбрать в целях оптимизации, когда все потоки, находящиеся в состоянии ожидания, ждут одного и того же условия, однако при его выполнении в данный момент времени может пробудиться только один поток. Оба эти условия заведомо выполняются, если в каждом конкретном объекте в состоянии ожидания находится только один поток (как это было в при мере WorkQueue из статьи 49).

Но даже если эти условия справедливы, может потребоваться использование notifyAll. Точно так же, как помещение вызова wait в цикл защищает общедоступный объект от случайных и злонамеренных извещений, применение notifyAll вместо notify защищает от случайного и злонамеренного ожидания в постороннем потоке. Иначе посторонний поток может "проглотить" важное извещение, оставив его действительного адресата в ожидании на неопределенное время. В примере WorkQueue причина, по которой не *использован* метод notifyAll, заключается в том, что поток, обрабатывающий очередь, ждет своего условия в *закрытом* объекте (queue), а потому опасности случайных или злонамеренных ожиданий в других потоках здесь нет.

Следует сделать ~ДНО предупреждение относительно использования notifyAll вместо notify. Хотя метод notifyAll не нарушает корректности приложения, он ухудшает его производительность: для определенных структур данных производительность снижается с линейной до квадратичной зависимости от числа ждущих потоков. Это касается тех структур данных, при работе с которыми в любой момент времени в не котором особом состоянии находится определенное количество потоков, а остальные потоки должны ждать. Среди примеров таких структур: *семафоры* (setaphore), *буферы* с *ограничениями* (bounded ЬиНег), а также *блокировка чтения-записи* (read-write lock).

Если вы реализуете структуру данных подобного типа и будите каждый поток, только когда он становится приемлем для "особого статуса", то каждый поток вы будите один раз и потребуется *п* операций пробуждения потоков. Если же вы будите все *п* потоков, то лишь один из них может получить особый статус, а оставшиеся *п* –1 потоков возвращаются в состояние ожидания. К тому времени как все потоки из очереди ожидания получат особый статус, количество пробуждений составит n + *(п* -1) + *(п* -2) ... + 1. Сумма этого ряда равна $O(n^2)$. Если вы знаете, что потоков всегда будет немного, проблем практически не возникают. Однако если такой уверенности нет, важно использовать более избирательную стратегию пробуждения потоков.

Если все потоки, претендующие на получение особого статуса, логически эквивалентны, то все, что нужно сделать,- это аккуратно использовать notify вместо notifyAll. Однако если в любой момент времени к получению особого статуса готовы лишь некоторые потоки из находящихся в состоянии ожидания, то вы должны применять шаблон, который называется *Speci/ic Noti/ication* [Cargill96, Lea99]. Описание указанного шаблона выходит за рамки этой книги.

Подведем итоги. Всегда вызывайте метод wait только из цикла, применяя стандартную идиому. Поступать иначе нет причин. Как правило, методу notify лучше

предпочитать noti fyAll. Однако в ряде ситуаций следование этому совету будет сопровождаться значительным падением производительности. При использовании notify нужно уделить особое внимание обеспечению живучести приложения.

Не попадайте в зависимость от планировщика потоков

При выполнении в системе нескольких потоков соответствующий планировщик определяет, какие из них будут выполняться и в течение какого времени. Каждая правильная реализация JVM пытается при этом добиться какой-то справедливости, однако конкретные стратегии диспетчеризации в различных реализациях сильно отличаются. Хорошо написанные многопоточные приложения не должны зависеть от особенностей этой стратегии. Любая программа, чья корректность или производительность зависит от планировщика потоков, скорее всего окажется не переносимой.

Лучший способ написать устойчивую, гибкую и переносимую многопоточную программу - обеспечить условия, при которых в любой момент времени может выполняться несколько потоков. В этом случае планировщику потоков остается совсем небольшой выбор: он лишь передает управление выполняемым потокам, пока те еще могут выполняться. Как следствие, поведение программы не будет сильно меняться даже при выборе совершенно других алгоритмов диспетчеризации потоков.

Основной прием, позволяющий сократить количество запущенных потоков, заключается в том, что каждый поток должен выполнять небольшую порцию работы, а затем ждать наступления некоего условия (используя Object.wait) либо истечения не которого интервала времени (используя Thread.sleep). Потоки не должны находиться в состоянии *активного ожидания* (busy-wait), регулярно проверяя структуру данных и ожидая, пока с теми что-то произойдет. 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 процессы.

Указанным рекомендациям отвечает пример с очередью заданий (статья 49): если предоставляемый клиентом метод processItem имеет правильное поведение, то поток, обрабатывающий очередь, большую часть своего времени, пока очередь пуста, будет проводить в ожидании монитора. В качестве яркого примера того, как поступать не следует, рассмотрим еще одну неправильную реализацию класса WorkQueue, в которой вместо работы с монитором используется активное ожидание:

```
//Ужасная программа: использует активное ожидание
// вместо метода Object.wait!
public abstract class WorkQueue {
    private final List queue = new LinkedList();
    private boolean stopped = false;
```

```
import java.util.*;
public abstract class WorkQueue {
   private final List queue = new LinkedList();
   private boolean stopped = false;
protected WorkQueue() { new WorkerThread().start(); }
   public final void enqueue(Object workItem) {
      synchronized (queue) { queue.add(workItem); }
   }
   public final void stop()  {
      synchronized (queue) { stopped = true; }
  }
   protected abstract void processItem(Object workItem)
      throws InterruptedException;
   private class WorkerThread extends Thread {
      public void run() {
         final Object QUEUE_IS_EMPTY = new Object();
         while (true) {  // Главный цикл
            Object workItem = QUEUE_IS_EMPTY;
            synchronized (queue) {
               if (stopped)
                  return;
               if (!queue.isEmpty())
                  workItem = queue.remove(0);
            }
                if (workItem != QUEUE_IS_EMPTY) {
               try {
                  processItem(workItem);
               } catch (InterruptedException e) {
                  return;
               }
            }
         }
      }
   }
}
```

Чтобы дать некоторое представление о цене, которую вам придется платить за такую реализацию, рассмотрим микротест, в котором создаются две очереди заданий и затем некое задание передается между ними в ту и другую сторону. (Запись о задании, передаваемая из одной очереди в другую,- это ссылка на первую очередь, которая служит адресом возврата.) Перед началом измерений программа выполняется десять секунд, 'чтобы система "разогрелась"; в течение следующих десяти секунд подсчитывается количество циклических переходов из очереди в очередь. На моей

машине окончательный вариант WorkQueue (статья 49) показал 23 000 циклических переходов в секунду, тогда как представленная выше некорректная реализация демонстрирует 17 переходов в секунду.

```
class PingPongQueue extends WorkQueue {
   volatile int count = 0;

   protected void processItem(final Object sender) {
      count++;
      WorkQueue recipient = (WorkQueue) sender;
      recipient.enqueue(this);
   }
}


public class WaitQueuePerf {
   public static void main(String[] args) {
      PingPongQueue q1 = new PingPongQueue();
      PingPongQueue q2 = new PingPongQueue();
      q1.enqueue(q2); // Запускаем систему

      // Дадим системе 10 с на прогрев
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      // Подсчитаем количество переходов за 10 с
      int count = q1.count;
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }
      System.out.println(q1.count - count);

      q1.stop();
      q2.stop();
   }
}
```

Приведенная реализация WorkQueue может показаться немного надуманной, тем не менее нередко можно встретить многопоточные системы, в которых безо всякой необходимости запускается один или несколько лишних потоков. И хотя результат этого может быть не таким экстремальным, как здесь продемонстрировано, производительность и переносимость приложения, по-видимому, все же страдают.

Столкнувшись с тем, что программа едва работает из-за того, что некоторые потоки, по сравнению с остальными, не получают достаточно процессорного времени, не поддайтесь искушению “исправить" программу, добавив в нее вызовы Thread.yield. Вы можете заставить программу работать, однако полученное приложение не будет переносимым с точки зрения производительности. Вызовы yield, улучшающие производительность в одной реализации JVM, в другой ее ухудшают, а в третьей не оказывают никакого влияния. У Thread.yield нет строгой семантики. Лучше измените структуру приложения таким образом, чтобы сократить количество параллельно выполняемых потоков.

Схожий прием состоит в регулировании *приоритетов потоков.* приоритеты потоков числятся среди наименее переносимых характеристик платформы Java. Нельзя отрицать, что быстроту реагирования приложения можно настроить, отрегулировав приоритеты нескольких потоков, но необходимость в этом возникает редко, а полученные результаты будут меняться от одной реализации JVM к другой. Серьезную проблему живучести не решить с помощью приоритетов потоков. Проблема скорее всего вернется, пока вы не найдете и не устраните основную причину.

Метод Thread.yield следует использовать для того, чтобы искусственно увеличить степень распараллеливания программы на время тестирования. Благодаря просмотру большей части пространства состояний программы, это помогает; найти ошибки и удостовериться в правильности системы. Этот прием доказал свою высокую эффективность в выявлении скрытых ошибок многопоточной обработки.

Подведем итоги. Ваше приложение не должно зависеть от планировщика потоков. Иначе оно не будет ни устойчивым, ни переносимым. Как следствие, лучше не связывайтесь с методом Thread.yield и приоритетами. Эти функции предназначены единственно для планировщика. Их можно дозировано при менять для улучшения качества сервиса в уже работающей реализации, но ими нельзя пользоваться для "исправления" программы, которая едва работает.

При работе с потоками документируйте уровень безопасно

То, как класс работает, когда его экземпляры и статические методы одновременно используются в нескольких потоках, является важной частью соглашений, устанавливаемых классом для своих клиентов. Если вы не отразите эту сторону поведения класса в документации, использующие его программисты будут вынуждены делать Допущения. И если эти допущения окажутся неверными, полученная программа может иметь либо недостаточную (статья 48), либо избыточную (статья 49) синхронизацию. В любом случае это способно привести к серьезным ошибкам.

Иногда говорится, что пользователи могут сами определить, безопасен ли метод при работе с несколькими потоками, если проверят, присутствует ли модификатор synchronized в документации, генерируемой утилитой Javadoc. Это неверно по

нескольким причинам. Хотя в ранних версиях утилита javadoc действительно указывала в создаваемом документе модификатор synchronized, это было ошибкой, и в версии 1.2 она была устранена. Наличие в декларации метода модификатора synchronized - это деталь реализации, а не часть внешнего API. Присутствие модификатора не является надежной гарантией того, что метод безопасен при работе с несколькими потоками. От версии к версии ситуация может меняться.

Более того, само утверждение о том, что наличия ключевого слова synchronized достаточно для того, чтобы говорить о безопасности при работе с несколькими потоками, содержит в себе распространенную микро-концепцию о категоричности этого свойства. На самом деле, класс может иметь несколько уровней безопасности. Чтобы класс можно было безопасно использовать в среде со многими потоками, в документации к нему должно быть четко указано, какой уровень безопасности он поддерживает.

В следующем списке приводятся уровни безопасности, которых может придерживаться класс при работе с несколькими потоками. Этот список не претендует на полноту, однако в нем представлены самые распространенные случаи. Используемые здесь названия не являются стандартными, поскольку в этой области нет общепринятых соглашений.

- Неизменяемый (immutable). Экземпляры такого класса выглядят для своих клиентов как константы. Никакой внешней синхронизации не требуется. Примерами являются String, Integer и BigInteger (статья 13).
- С поддержкой многопоточности (thread-safe). Экземпляры такого класса могут изменяться, однако все методы имеют довольно надежную внутреннюю синхронизацию, чтобы эти экземпляры могли параллельно использовать несколько потоков безо всякой внешней синхронизации. Параллельные вызовы будут обрабатываться последовательно внекотором глобально согласованном порядке. Примеры: Random и java. util. Titer.
- С условной поддержкой многопоточности (conditionally thread-safe). т о же, что и с поддержкой многопоточности, за исключением того, что класс (или ассоциированный класс) содержит методы, которые должны вызываться один за другим без взаимного влияния со стороны других потоков. Для исключения возможности такого влияния клиент должен установить соответствующую блокировку на время выполнения этой последовательности. Примеры: Hashtable и Vector, чьи итераторы требуют внешней синхронизации.
- Совместимый с многопоточностью (thread-compatible). Экземпляры такого класса можно безопасно использовать при работе с параллельными потоками, если каждый вызов метода (а в некоторых случаях, каждую последовательность вызовов) окружить внешней синхронизацией. Примерами являются реализации коллекций общего назначения, такие как ArrayList и HashMap.

- Несовместимый с многопоточностью (thread-hostile). Этот класс небезопасен при параллельной работе с несколькими потоками, даже если вызовы всех методов окружены внешней синхронизацией. Обычно несовместимость связана с тем обстоятельством, что эти методы меняют некие статические данные, которые оказывают влияние на другие потоки. К счастью, в библиотеках платформы Java лишь очень немногие классы и методы несовместимы с многопоточностью. Так, метод System.runFinalizersOnExit несовместим с многопоточностью и признан устаревшим.

Документированию класса с условной поддержкой многопоточности нужно уделять особое внимание. Вы должны указать, какие последовательности вызовов требуют внешней синхронизации и какую блокировку (в редких случаях, блокировки) необходимо поставить, чтобы исключить одновременный доступ. Обычно это блокировка самого экземпляра, но не всегда. Если объект является альтернативным представлением какого-либо другого объекта, клиент должен получить блокировку для основного объекта с тем, чтобы воспрепятствовать его непосредственной модификации со стороны других потоков. Например, в документации к методу Hashtable.keys нужно сказать примерно следующее:

> Если есть какая-либо опасность того, что хэш-таблица будет изменена из другого потока, то для получения безопасно перечня ее ключей' требуется, чтобы перед вызовом метода keys был блокирован соответствующий экземпляр класса Hashtable и эта блокировка сохранялась до тех пор, пока вы не закончите работу с полученным объектом Enumeration. Описанную схему демонстрирует фрагмент кода:

```
Hashtable h = ... ;
Synchronized (h) {
    for (Enumeration e = h.keys(); e.hasMoreElements(); )
    f(e.nextElement()); }
```

В версии 1.3 нет такого текста в документации к классу Hashtable. Однако будем надеяться, что эта ситуация скоро будет исправлена. Вообще же в библиотеках для платформы Java безопасность работы с потоками можно было' бы документировать получше.

Хотя объявление об общедоступном блокировании объекта позволяет клиентам выполнять последовательность вызовов как неделимую,. за эту гибкость приходится платить. Клиент, имеющий злой умысел, может предпринять атаку "отказ в обслуживании" (denial-of-service, DOS attack), установив блокировку объекта:

```
// DOS-атака
synchronized (importantObject)
    { Thread.sleep(Integer,MAX_VALUE);
        // ВЫВОДИТ из строя importantObject
```

Если вас беспокоят DOS - атаки, то используйте для синхронизации операций *закрытый объект блокировки* (private lock object):

```
// Идиома закрытого объекта блокировки, препятствует DOS-атаке
private Object lock = new Object();
public void foo() {
    synchronized (lock) {  }
    }
```

Поскольку создаваемая этим объектом блокировка недоступна клиентам, данный объект-контейнер неуязвим для представленной ранее DOS - атаки. Заметим, что классы с условной поддержкой многопоточности всегда неустойчивы по отношению к такой атаке, в документации к ним должно быть указано, что выполнение последовательности операций атомарным образом требует получения блокировки. Однако классы с поддержкой многопоточности могут быть защищены от DOS -атаки при помощи идиомы закрытого объекта блокировки.

Применение внутренних объектов для блокировки особенно подходит классам, предназначенным для наследования (статья 15), таким как класс WorkQueue (статья 49). Если бы суперкласс использовал для блокировки собственные экземпляры, подкласс мог бы непреднамеренно повлиять на их работу. Применяя одну и ту же блокировку для разных целей, суперкласс и подкласс стали бы, в конце концов, "наступать друг другу на пятки".

Подведем итоги. для каждого класса необходимо четко документировать возможность работы с несколькими потоками. Единственная возможность сделать это представить аккуратно составленный текст описания. К описанию того, как класс поддерживает многопоточность, наличие модификатора synchronized отношения не имеет. для классов с условной поддержкой многопоточности важно указывать в документации, какой объект следует заблокировать, чтобы последовательность обращений к методам стала неделимой. Описание того, как класс поддерживает многопоточность, обычно располагается в doc - комментарии ко всему классу. Однако для методов, имеющих особый режим работы с потоками, это должно быть отражено в их собственном doc - комментарии.

Избегайте группировки потоков

Помимо потоков, блокировок и мониторов, система многопоточной обработки предлагает еще одну базовую абстракцию: *группа потоков* (thread group). Первоначально группировка потоков рассматривалась как механизм изоляции апплетов

в целях безопасности. В действительности своих обязательств они так и не выполнили, а их роль в системе безопасности упала до такой степени, что в работе, где выстраивается модель безопасности для платформы Java 2 (Gong99], они даже не упоминаются.

Но если группировка потоков I:Ie несет никакой функциональной нагрузки в системе безопасности, то какие же функции она выполняет? В общих словах, она позволяет применять примитивы класса Thread сразу к целой группе потоков. Некоторые из этих примитивов уже устарели, остальные используются нечасто. В итоге группировка потоков не может дать достаточного количества полезной функциональности.

По иронии судьбы, API ThreadGroup слаб с точки зрения поддержки многопоточности. Чтобы для некоей группы получить перечень активных потоков, вы должны вызвать метод enumerate. В качестве параметра ему передается массив, достаточно большой, чтобы в него можно было записать все активные потоки. Метод activeCount возвращает количество активных потоков в группе, однако нет никакой' гарантии, что это количество не изменится в то время, пока вы создаете массив и передаете его методу enumerate. Если указанный массив окажется слишком мал, метод enumerate без каких-либо предупреждений игнорирует потоки, не поместившиеся в массив.

Точно так же API некорректен, когда ему передается список подгрупп, входящих в группу потоков. И хотя указанные проблемы можно было решить, добавив в класс ThreadGroup новые методы, этого не было сделано из-за отсутствия реальной потребности. Группировка потоков сильно устарела.

Подведем итоги. Группировка потоков практически не имеет сколь-нибудь полезной функциональности, и большинство предоставляемых ею возможностей имеет дефекты. Группировку потоков следует рассматривать как неудачный эксперимент, а существование групп можно игнорировать. Если вы проектируете класс, который работает с логическими группами потоков, вам нужно записывать ссылки Thread, соответствующие каждой логической группе, в массив или коллекцию. Вы могли заметить, что этот совет вступает в противоречие со статьей 30 "Изучите библиотеки и пользуйтесь ими". Однако в данном случае статья 30 не права.

В большинстве случаев следует игнорировать группировку потоков. Однако есть одно небольшое исключение. Нечто полезное можно найти в интерфейсе класса ThreadGroup. Когда какой-либо поток в группе инициирует исключение, не отлавливаемое в приложении, автоматически вызывается метод ThreadGroup.uncaughtException. Этот метод используется "рабочим окружением" для того, чтобы должным образом реагировать на необработанные исключения. Реализация, предлагаемая по умолчанию, печатает трассировку стека в стандартный поток сообщений об ошибках. Возможно, вы захотите поменять такую реализацию, направив, например, трассировку стека в определенный журнал регистрации.

Глава 10

Сериализация

В этой главе описывается API *сериализации объекта* (object serialization), который формирует среду для представления объекта в виде потока байтов и, наоборот, для восстановления объекта из соответствующего потока байтов. Процедура представления объекта в виде потока байтов называется *сериализациеи* объекта (serializing), обратный процесс называется его *десериализациеи* (deserializing). Как только объект сериализован, его представление можно передавать с одной работающей виртуальной машины Java на другую или сохранять на диске для последующей десериализации. Сериализация обеспечивает стандартное представление объектов на базовом уровне, которое используется для взаимодействия с удаленными машинами, а также как стандартный формат для сохранения данных при работе с компонентами JavaBeans TM.

Соблюдайте осторожность при реализации интерфейса Serializable

Чтобы сделать экземпляры класса сериализуемыми, достаточно добавить в его декларацию 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.

Значительная доля затрат на реализацию интерфейса Serializable связана с тем, что уменьшается возможность изменения реализации класса в последующих версиях. Когда класс реализует интерфейс Serializable, соответствующий ему поток байтов *(сериализованная форма* - serialized Form) становится частью его внешнего API. И как только ваш класс получит широкое распространение, вам придется поддерживать соответствующую сериализованную форму точно так же, как вы обязаны поддерживать все остальные части интерфейса, предоставляемого клиентам. Если вы не приложите усилий к построению *специальной сериализованнои 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 12) потеряет свою эффективность как средство скрытия информации.

Если вы принимаете сериализованную форму, предлагаемую по умолчанию, а затем меняете 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 ObjectOutputSt reat. putFields и ObjectOutputSt reat, readF1elds), но этот механизм довольно сложен и оставляет в исходном коде программы видимые изъяны. Поэтому тщательно выстраивайте качественную сериализованную форму, с которой вы сможете отправиться в долгий путь (статья 55). Эта работа усложняет создание приложения, но дело того стоит. Даже хорошо спроектированная сериализованная форма ограничивает дальнейшее развитие класса, плохо же спроектированная форма может его искалечить.

Простым примером того, какие ограничения на изменение класса накладывает сериализация, могут служить *уникальные идентификаторы потока* (stream *unique* identifier), более известные как *seria/ versio'n u 1* D. С каждым сериализуемым классом связан уникальный идентификационный номер. Если вы не указываете этот идентификатор явно, декларируя поле private stat ic final long с названием serialVersionUID, система генерирует его автоматически, используя для класса сложную схему расчетов. При этом на автоматически генерируемое значение оказывают влияние название класса, названия реализуемых им интерфейсов, а также все открытые и защищенные члены. Если вы каким-то образом поменяете что-либо в этом наборе, например, добавите простой и удобный метод, изменится и автоматически генерируемый serial version UID. Следовательно, если вы не будете явным образом декларировать этот идентификатор, совместимость с предыдущими версиями будет потеряна.

Второе неудобство от реализации интерфейса Serializable заключается в том, что повышается вероятность появления ошибок и дыр в защите. Объекты обычно создаются с помощью конструкторов, сериализация же представляет собой *механизм* создания объектов, который *выходит за рамки языка /ava.* Принимаете ли вы схему, которая предлагается по умолчанию, или переопределяете ее, десериализация - это "скрытый конструктор", имеющий все те же проблемы, что и остальные конструкторы. Поскольку явного конструктора здесь нет, легко упустить из виду то, что при десериализации вы должны гарантировать сохранение всех инвариантов, устанавливаемых настоящими конструкторами, и исключить возможность получения злоумышленником доступа к внутреннему содержимому создаваемого объекта. Понадеявшись на механизм десериализации, предоставляемый по умолчанию, вы можете получить объекты, которые не препятствуют несанкционированному доступу к внутренним частям и разрушению инвариантов (статья 56).

Третье неудобство реаливации интерфейса Serializable связано с тем, что выпуск новой версии класса сопряжен с большой работой по тестированию. При пересмотре сериализуемого класса важно проверить возможность сериализации объекта в новой версии и последующей его десериализации в старой и наоборот. Таким образом, объем необходимого тестирования прямо пропорционален произведению числа сериализуемых классов и числа имеющихся версий, что может быть большой величиной., К подготовке таких тестов нельзя подходить формально, поскольку, помимо *совместимости* на *бинарном уровне,* вы должны проверять *совместимость* на *уровне семантики.* Иными словами, необходимо гарантировать .. не только успешность процесса сериализации-десериализации, но и то, что он будет создавать точную копию первоначального объекта. И чем больше изменяется сериализуемый класс, тем сильнее потребность в тестировании. Если при написании класса специальная сериализованная форма была спроектирована тщательно (статья 55), потребность в проверке уменьшается, но полностью не исчезает.

Реализация интерфейса Serializable должна быть хорошо продумана. У этого интерфейса есть реальные преимущества: его реализация играет важную роль, если класс должен участвовать в какой-либо схеме, которая для передачи или обеспечения живучести объекта использует сериализацию. Более того, это упрощает применение класса как составной части другого класса, который должен реализовать интерфейс Serializable. Однако с реализацией интерфейса Serializable связано и множество неудобств. Реализуя класс, соотносите неудобства с преимуществами. Практическое правило таково: классы значений, такие как Date и BigInteger, и большинство классов коллекций обязаны реализовывать этот интерфейс. Классы, представляющие активные сущности, например пул потоков, должны реализовывать интерфейс Serializable крайне редко. Так, в версии 1.4 появился механизм сохранения компонентов JavaBeап, который использует стандарт XML, а потому этим компонентам больше не нужно реализовывать интерфейс Serializable.

Классы, предназначенные для наследования (статья 15), редко должны реализовывать Ser1alizable, а интерфейсы - редко его расширять. Нарушение этого правила связано с большими затратами для любого, кто пытается расширить такой класс или реализовать интерфейс. В ряде случаев это правило можно нарушать. Например, если класс или интерфейс создан в первую очередь для использования

в некоторой системе, требующей, чтобы все ее участники реализовывали интерфейс Ser1al1zable, то лучше всего, чтобы этот класс (интерфейс) реализовывал (расширял) Ser1a11zable.

Нужно сделать одно *предупреждение* относительно реализации интерфейса Ser1a11zable. Если класс предназначен для наследования и не является сериализуемы, может оказаться, что для него невозможно написать сериализуемый подкласс. В частности, этого нельзя сделать, если у суперкласса нет доступного конструктора без параметров. Следовательно, для несериализуемого класса, который предназначен для наследования, **Вы** должны рассмотреть возможность создания конструктора без параметров. Часто это не требует особых усилий, поскольку многие классы, предназначенные для наследования, не имеют состояния. Но так бывает не всегда.

Самое лучшее _ это создавать объекты, у которых все инварианты уже установлены (статья 13). Если для установки инвариантов необходима информация от клиента, это будет препятствовать использованию конструктора без параметров. Бесхитростное добавление конструктора без параметров и метода инициализации в класс, остальные конструкторы которого устанавливают инварианты, усложняет пространство состояний этого класса и увеличивает вероятность появления ошибки.

Приведем вариант добавления конструктора без параметров в несериализуемый расширяемый класс, свободный от этих пороков. Предположим, что в этом классе есть один конструктор:

```
public AbstractFoo(int x, int y) { ... }
```

Следующее преобразование добавляет защищенный конструктор без параметров и отдельный метод инициализации. Причем метод инициализации имеет те же параметры и устанавливает те же инварианты, что и обычный конструктор:

```
public abstract class AbstractFoo {
    private int x, y; // Состояние
    private boolean initialized = false;

    public AbstractFoo(int x, int y) { initialize(x, y); }

    /**
     * Данный конструктор и следующий за ним метод позволяют методу
     * readObject в подклассе инициализировать наше внутреннее состояние.
     */
    protected AbstractFoo() { }

    protected final void initialize(int x, int y) {
        if (initialized)
            throw new IllegalStateException(
                "Already initialized");
        this.x = x;
        this.y = y;
        // ... // Делает все остальное, что делал прежний конструктор
        initialized = true;
    }

    /**
     *Эти методы предоставляют доступ к внутреннему состоянию класса,
     * и потому его можно сериализовать вручную, используя
     * метод writeObject из подкласса
     */
```

```
  protected final int getX() { return x; }
    protected final int getY() { return y; }

    // Должен вызываться для всех открытых методов экземпляра
    private void checkInit() throws IllegalStateException {
        if (!initialized)
            throw new IllegalStateException("Uninitialized");
    }
    // ... // Остальное опущено
}
```

Все методы в экземпляре AbstractFoo, прежде чем выполнять свою работу, должны вызывать checkInit. Тем самым гарантируется быстрое и четкое аварийное завершение этих методов в случае, если неудачно написанный подкласс не инициализировал соответствующий экземпляр. Имея этот механизм взамен прежнего, можно перейти к реализации сериалнзуемого подкласса:

```
import java.io.*;

public class Foo extends AbstractFoo implements Serializable {
    private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
        s.defaultReadObject();

        // Ручная десериализация и инициализация состояния суперкласса
        int x = s.readInt();
        int y = s.readInt();
        initialize(x, y);
    }

    private void writeObject(ObjectOutputStream s)
            throws IOException {
        s.defaultWriteObject();

        // Ручная сериализация состояния суперкласса
        s.writeInt(getX());
        s.writeInt(getY());
    }

    // Конструктор не использует никаких причудливых механизмов
    public Foo(int x, int y) { super(x, y); }
}
```

Внутренние классы (статья 18) редко должны (если вообще должны) реализовывать интерфейс Serializable. Для размещения ссылок на *экземпляры контейнера* (enclosing instance) и значений локальных переменных из окружения они

используют *искусственные поля* (synthetic field), генерируемые компилятором. Как именно эти поля соотносятся с декларацией класса, не конкретизируется. Не конкретизируются также названия анонимных и локальных классов. Поэтому выбор для внутреннего класса серилизованной формы, предлагаемой по умолчанию, плохое дизайнерское решение. Однако *статический класс-член* вполне может реализовывать интерфейс Serializable.

Подведем итоги. Легкость реализации интерфейса Serializable обманчива. Реализация интерфейса Serializable - серьезное обязательство, которое следует брать на себя с осторожностью, если только не предполагается выбросить класс после недолгого использования. Особое внимание требует класс, предназначенный для наследования. Для таких классов границей между реализацией интерфейса Serializable в подклассах и его запретом является создание доступного конструктора без параметров. Это позволяет реализовать в подклассе интерфейс Serializable, хотя и не является обязательным условием.

Рассмотрите возможность использования специализированн сериализованной формы

Если вы создаете класс в условиях дефицита времени, то, как правило, имеет смысл сконцентрировать усилия на построении самого лучшего *API.* Иногда это означает создание "одноразовой" реализации, которая в следующей версии поменяется. Обычно это проблем не вызывает, однако если данный класс реализует интерфейс Serializable и использует при этом сериализованную форму, предоставленную по умолчанию, вам уже никогда не удастся полностью избавиться от этой временной реализации, и она всегда будет навязывать вам именно эту сериализованную форму. Это не теоретическая проблема. Такое уже происходило с несколькими классами из библиотек для платформы Java, такими как B1gInteger.

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 той кодировкой, которую вы бы выбрали, если бы проектировали сериализованную форму сами,

Сериализованная форма представления объекта, предлагаемая по умолчанию,это довольно Эффективное *физическое* представление графа объектов, имеющего корнем данный объект. Другими словами, эта форма описывает данные, содержащиеся как в самом объекте, так и во всех доступных из него объектах. Она также отражает топологию взаимосвязи этих объектов. Идеальная же сериализованная форма, описывающая объект, содержит только представляемые им логические данные, От физического представления она не зависит,

Сериализованная форма, предлагаемая по умолчанию, по-видимому, будет приемлема в том случае, если физическое представление объекта 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
     /**
      * Последнее имя (англ. ). Не должно быть пустым (non-null)
       * @serial
      */
     private String lastName;
     /**
      * Первое имя. Не должно быть пустым.
      *'@serial
      */
     private String firstName;
    /**
      * Средний инициал или '\uOOOO', если инициал отсутствует
      * @serial
      */
    private char middlelnitial;
            // Остальное опущено
```

Логически имя человека в английском языке состоит из двух строк, представляющих последнее и первое имя, а также из некоего символа, соответствующего среднему инициалу. Экземпляры полей в классе Name в точности воспроизводят это .логическое содержание,

Если вы решите принять сериализованную форму, предлагаемую по умолчанию, во многих случаях сохранение инвариантов и безопасность требуют реализации метода readObj ect. 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 статье 56.

Заметим, что поля lastName, firstName и middlelnitial сопровождает комментарий к документации, хотя все они являются закры~ыми. Это необходимо поскольку эти закрытые поля определяют открытый *AP!:* сериализованную форму класса, а всякий открытый AP! должен быть документирован. Наличие тега @serial говорит утилите Javadoc о том, что эту информацию необходимо поместить на специальную страницу, где описываются сериализованные формы.

Теперь рассмотрим класс, который представляет набор строк (забудем на минуту о том, что для этого лучше было бы взять в библиотеке одну из стандартных реализаций интерфейса List):

```
// Ужасный кандидат на использование сериализованной формы,
// предлагаемой по умолчанию
public class StringList implements Serializable {
 private int size = 0
     private Entry head = null;
     private static class Entry implements Serializable {
     String data;
          Entry next;
          Entry previous;  }

              // Остальное опущено

              }
```

Логически этот класс представляет последовательность строк. Физически последовательность представлена им как дважды связный список. Если вы примите сериализованную форму, предлагаемую по умолчанию, она старательно отразит каждый элемент в этом связном списке, а также все связи между этими элементами в обоих направлениях.

В случае, когда физическое представление объекта существенно отличается от содержащихся в нем логических данных, сериализованная форма, предлагаемая по умолчанию, имеет четыре недостатка:

- Она навсегда связывает внешний API класса с его текущим внутренним представлением. В приведенном примере закрытый класс StringList.Entry становится частью открытого API. Даже если в будущей версии внутреннее представление StringList поменяется, он все равно должен будет получать на входе представление в виде связного списка и генерировать его же на выходе. Этот класс уже никогда не избавится от кода, необходимого для манипулирования связными списками, даже если он ими уже не пользуется.
- Она может занимать чрезвычайно много места. В приведенном примере в сериализованной форме без всякой на то надобности представлен каждый элемент связанного списка со всеми его связями. Эти элементы и связи являются всего лишь деталями реализации, не стоящими включения в сериализованную форму. Из-за того, что полученная форма слишком велика, ее запись на диск или передача по сети будет выполняться слишком медленно.

- Она может обрабатываться чрезвычайно долго. Логика сериализации не содержит информации о топологии графа объекта, а потому приходится выполнять дорогостоящий обход вершин графа. В приведенном Примере достаточно было просто идти по ссылкам next.
- Она может вызвать переполнение стека. Процедура сериализации, реализуемая по умолчанию, выполняет рекурсивный обход графа объектов, что может вызвать переполнение стека даже при обработке графов среднего размера. На моей машине к переполнению стека приводит сериализация экземпляра StringList с 1200 элементами. Количество элементов, вызывающее эту проблему, меняется в зависимости от реализации JVM. В некоторых реализациях этой проблемы вообще не существует.

Правильная сериализованная форма для класса StringList - это количество строк в списке, за которым следуют сами строки. Это соответствует освобожденным от деталей физической реализации логическим данным, представляемым классом StringList. Приведем исправленный вариант StringList, содержащий методы writeObject и readObject, которые реализуют правильную сериализованную форму. Напомним, что модификатор transient указывает на то, что экземпляр поля должен быть исключен из сериализованной формы, применяемой по умолчанию:

```
// Класс StringList с правильной сериализованной формой

public class StringList implements Serializable
    private transient int size = 0;
    private transient Entry head = null;

    // Больше нет реализации Serializable!

    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // Добавляет указанную строку в конец списка

    public void add(String s) { ... }

    /**
     * Сериализует данный экземпляр <tt>StringList</tt>.
     *
     * @serialData Показывается размер списка (количество
     * содержащихся в нем строк) <tt>int</tt>, за которым
     * следуют все
     * элементы списка (каждый в виде <tt>String</tt>). */
```

```
private void writeObject(ObjectOutputStream s)
                throws IOException {
        s.defaultWriteObject();
         s.writeInt(size);
// Выписываем все элементы в правильном порядке
        for (Entry e = head; e != null; e = e.next)
                s.writeObject(e.data);
}
private void readObject(ObjectInputStream s)
                throws IOException, ClassNotFoundException {
        s.defaultReadObject();
        int size = s. readInt();
// Считываем все элементы и вставляем
        for (int i= 0; i < size; i++)
                add((String)s. readObject())}
// Остальное опускаем
}
```

Заметим, что из метода writeObject вызывается defaultWriteObject, а из метода readObject делается вызов defaultReadObject, несмотря на то, что ни одно из полей класса StringList не попадает в сериализованную форму. Если все экземпляры полей имеют модификатор transient, то формально можно обойтись без вызова методов defaultWriteObject и defaultReadObject, но это не рекомендуется. Даже если все экземпляры полей имеют модификатор transient, вызов defaultWriteObject оказывает влияние на сериализованную форму, в результате чего значительно повышается гибкость сериализации. Полученная форма оставляет возможность в последующих версиях добавлять в форму новые экземпляры полей, сохраняя при этом прямую и обратную совместимость с предыдущими версиями. Так, если сериализовать экземпляр класса в более поздней версии, а десериализовать в более ранней версии, появившиеся поля будут проигнорированы. Если бы более ранняя версия метода readObject не вызывала метод defaultReadObject, десериализация закончилась бы 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 @seria1 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. 11 наконец, у исправленного варианта не возникает проблем с переполнением стека, а потому практически нет верхнего ограничения на размер StringList, для которого можно выполнить сериализацию.

Сериализованная форма, предлагаемая по умолчанию, плохо подходит для класса StringList, но есть классы, для которых она подходит еще меньше. Для StringList сериализованная форма, применяемая по умолчанию, не имеет гибкости и работает медленно. Однако она является *правильнои* в том смысле, что в результате сериализации и десериализации экземпляра StringList получается точная копия исходного объекта, и все его инварианты будут сохранены. Но для любого объекта, чьи инварианты при вязаны к деталям реализации, это не так.

Например, рассмотрим случай с хэш-таблицей. Ее физическим представлением является набор сегментов, содержащих записи ключ/значение. Сегмент, куда будет помещена запись, определяется функцией, которая для представленного ключа вычисляет хэш-код. Вообще говоря, нельзя гарантировать, что в различных реализациях jVM эта функция будет одной и той же. В действительности нельзя даже гарантировать, что она будет оставаться той же самой, если одну и ту же jVM запускать несколько раз. Следовательно, использование для хэш-таблицы сериализованной формы, предлагаемой по умолчанию, может стать серьезной ошибкой: сериализация и десериализация хэш-таблицы могут привести к созданию объекта, инварианты которого будут серьезно нарушены.

11спользуете вы или нет сериализованную форму, предлагаемую по умолчанию, каждый экземпляр поля, не помеченный модификатором transient, будет сериализован при вызове метода defaultWriteObject. Поэтому каждое поле, которое можно не заносить в форму, нужно пометить этим модификатором. К таковым относятся избыточные поля, чьи значения можно вычислить по таким "первичным полям данных", как кэшированное значение хэша. Сюда также относятся поля, чьи значения меняются при повторном запуске JVM. Например, это может быть поле типа 10ng, в котором хранится указатель на местную' (native) структуру данных. Прежде чем согласиться на запись какого-либо поля в сериализованной форме, убедитесь в том, что его значение является частью логического состояния данного объекта. Если вы пользуетесь специальной сериализованной формой, большинство или даже все экземпляры полей нужно пометить модификатором transient, как в примере с классом StringList.

Если вы пользуетесь сериализованной формой, предлагаемой по умолчанию, и к тому же пометили одно или несколько полей как transient, помните о том, что при десериализации экземпляра эти поля получат *значения по умолчанию:* nu11 для полей ссылок на объекты, нуль для простых числовых полей и false для полей типа

boolean [JLS, 4.5.5]. 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 56). Альтернативный подход заключается в том, чтобы отложить инициализацию этих полей до первого вызова.

Независимо от того, какую сериализованную форму вы выберете, в каждом сериализуемом классе, который вы пишите, явным образом декларируйте serial version UID. Тем самым вы исключите этот идентификатор из числа возможных причин несовместимости (статья 54). Это также даст некоторый выигрыш в производительности. Если serial version UID не представлен, то при выполнении программы для его генерации потребуется выполнить трудоемкие вычисления. Для декларации serial version UID добавьте в ваш класс строку:

```
private static final long serialVersionUID = randomLongValue;
// Произвольное число типа long
```

Не важно, какое значение вы выберете для *randomLongValue.* Общепринятая практика предписывает генерировать это число, запуская для класса утилиту serialver. Однако можно взять число просто из "воздуха". Если вам когда-нибудь захочется создать новую версию класса, которая *несовместима* с имеющимися версиями, достаточно будет поменять значение в этой декларации. В результате попытки десериализировать экземпляры, сериализованные в предыдущих версиях, будут заканчиваться инициированием исключения InvalidClassException.

Подведем итоги. Если решено, что класс должен быть сериализуемым (статья 54), подумайте над тем, какой должна быть сериализованная форма. Форму, предлагаемую по умолчанию, используйте, только если она правильно описывает логическое состояние объекта. В противном случае создайте специальную сериализованную форму, которая надлежащим образом описывает этот объект. На разработку сериализованной формы для класса вы должны выделить не меньше времени, чем на разработку его методов, предоставляемых клиентам. Точно так же, как из последующих версий нельзя изъять те методы класса, которые были доступны клиентам, нельзя изымать поля из сериализованной формы. Чтобы при сериализации сохранялась совместимость, эти поля должны оставаться в форме навсегда. Неверный выбор сериализованной формы может иметь постоянное отрицательное влияние на сложность и производительность класса.

Метод readObject должен создаваться с защитой

В статье 24 представлен неизменяемый класс для интервалов времени, который содержит изменяемые закрытые поля даты. Чтобы сохранить свои инварианты и неизменяемость, этот класс создает резервную копию объектов Date в конструкторе и методах доступа. Приведем этот класс:

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

public final class Period implements Serializable {
   private final Date start;
   private final Date end;

   /**
    * @param  start - начало периода
    * @param  end  - конец периода, не должен предшествовать началу периода
    * @throws IllegalArgument – если начало периода указано после конца
    * @throws NullPointerException – если начало или конец периода нулевые
    */
   public Period(Date start, Date end) {
      this.start = new Date(start.getTime());
      this.end   = new Date(end.getTime());

      if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start +" > "+ end);
   }

   public Date start () { return (Date) start.clone(); }

   public Date end ()   { return (Date) end.clone(); }

   public String toString() { return start + " - " + end; }

   // ... // Остальное опущено
```

Предположим, что вам необходимо сделать этот класс сериализуемым. Поскольку физическое представление объекта Period в точности отражает его логическое содержание, вполне можно воспользоваться сериализованной формой, предлагаемой по умолчанию (статья 55). Может показаться, что все, что вам нужно для того, чтобы класс был сериализуемым,- это добавить в его декларацию слова "implements Serializable". Если вы поступите таким образом, то гарантировать классу сохранение его критически важных инвариантов будет невозможно.

Проблема заключается в том, что метод readObject фактически является еще одним открытым конструктором и потому требует такого же внимания, как и любой другой конструктор. Точно так же, как конструктор, метод readObject должен проверять правильность своих аргументов (статья 23) и при необходимости создавать для параметров резервные копии. Если метод readObject не выполнит хотя бы одно из этих условий, злоумышленник сможет относительно легко нарушить инварианты этого класса.

Метод readObject - это конструктор, который в качестве единственного входного параметра принимает поток байтов. В нормальных условиях этот поток байтов создается в результате сериализации нормально построенного экземпляра. Проблема возникает, когда метод readObject сталкивается с потоком байтов, полученным искусственно с целью генерации объекта, который нарушает инварианты этого класса. Допустим, что мы лишь добавили "implements Serializable" в декларацию класса Period. Следующая уродливая программа генерирует такой экземпляр класса Period, в котором конец периода предшествует началу:

```
import java.io.*;
public class BogusPeriod {
 // Этот поток байтов не мог быть получен из реального экземпляра Period
   private static final byte[] serializedForm = new byte[] {
   (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
   0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
   0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
   0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
   0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
   0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
   0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
   0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
   0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
   (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
   0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
   0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
   0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
   0x00, 0x78 };
public static void main(String[] args) {
      Period p = (Period) deserialize(serializedForm);
     System.out.println(p); }     // Возвращает объект с указанной сериализованной формой
   public static Object deserialize(byte[] sf) {
      try {
         InputStream is = new ByteArrayInputStream(sf);
         ObjectInputStream ois = new ObjectInputStream(is);
         return ois.readObject();
      } catch (Exception e) {
         throw new IllegalArgumentException(e.toString());
      }  }  }
```

Фиксированный массив байтов, используемый для инициализации массива SerializedForm, был получен путем сериализации обычного экземпляра Period и последующего редактирования потока байтов вручную. Детали построения потока для данного примера значения не имеют, однако если вам это любопытно, формат потока байтов описан в *"Java™ Object Serialization Specification"* [Serialization, 6]. Если вы запустите эту программу, она напечатает: "Fri Jan 01 12: 00: 00 PST 1999 Sun Jan 01 12:00:00 PST 1984". Таким образом, то, что класс Period стал сериализуемым, позволило создать объект, который нарушает инварианты этого класса. Для решения этой проблемы создадим в классе Period метод readObject, который будет вызывать defaultReadObject и проверять правильность десериализованного объекта. Если проверка покажет ошибку, метод readObject инициирует исключение ! nvalidObjectException, 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 еще одна, более тонкая проблема. Можно создать изменяемый экземпляр Pe riod, Сфабриковав поток байтов, который начинается потоком байтов, представляющим прав ильный экземпляр Period, а затем формирует дополнительные ссылки на закрытые поля Date в этом экземпляре. Злоумышленник может прочесть экземпляр Period из ObjectInputStream и получить "неконтролируемые ссылки на объекты", прилагаемые к этому потоку. Имея указанные ссылки, злоумышленник получает доступ к объектам, на которые есть ссылки в закрытых полях Date объекта Period. Меняя эти экземпляры Date, он может менять и сам экземпляр Period. Следующий класс демонстрирует атаку такого рода:

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

public class MutablePeriod {
  // Экземпляр интервала времени
  public final Period period;

  // Поле начала периода, к которому мы не должны иметь доступ
  public final Date start;

  // Поле конца периода, к которому мы не должны иметь
  public final Date end;

  public MutablePeriod() {
    try {
      ByteArrayOutputStream bos =
        new ByteArrayOutputStream();
      ObjectOutputStream out =
        new ObjectOutputStream(bos);
```

```
// Сериализуем правильный экземпляр Period
        out.writeObject(new Period(new Date(), new Date()));


/*
* Добавляем в конец не контролируемые "ссылки
* на предыдущие объекты" для внутренних полей Date
 * в экземпляре Period. Подробнее см. "Java Object * Serialization Specification", раздел
6.4.
*/
byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
        bos.write(ref); // The start field
        ref[4] = 4;     // Ref # 4
        bos.write(ref); // The end field

        // Десериализация экземпляра Period и "украденных" ссылок
          // на экземпляры Date
        ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(bos.toByteArray()));
        period = (Period) in.readObject();
        start  = (Date)   in.readObject();
        end    = (Date)   in.readObject();
      } catch (Exception e) {
        throw new RuntimeException(e.toString());
      }
   }
```

Чтобы увидеть описанную атаку в действии, запустим следующую программу:

```
public static void main(String[] args) {
      MutablePeriod mp = new MutablePeriod();
      Period p = mp.period;
      Date pEnd = mp.end;

      // Let's turn back the clock
      pEnd.setYear(78);
      System.out.println(p);

      // Bring back the 60's!
      pEnd.setYear(69);
      System.out.println(p);
   }
}
```

Запустив эту программу, на выходе получим следующее:

```
Wed Mar 07 23:30:01 PST 2001 - Tue Mar 07 23:30:01 PST 1978
Wed Mar 07 23:30:01 PST 2001 - Fri Mar 07 23:30:01 PST 1969
```

Хотя экземпляр Period создается с неповрежденными инвариантами, при желании его внутренние компоненты можно поменять извне. Завладев изменяемым экземпляром класса Period, злоумышленник может причинить массу вреда, передав этот экземпляр классу, чья безопасность зависит от неизменяемости класса Period. И это не такая уж надуманная тема. Существуют классы, чья безопасность зависит от неизменяемости класса String.

Причина этой проблемы кроется в том, что метод readObject класса Period не выполняет необходимого резервного копирования. При десериализации объекта крайне важно создать резервные копии для всех полей, содержащих ссылки на те объекты, которые не должны попасть в распоряжение клиентов. Поэтому каждый сериализуемый неизменяемый класс, содержащий закрытые изменяемые компоненты, должен в своем методе readObject создавать резервные копии для этих компонентов. Следующий метод readObject достаточен для того, чтобы объект Period оставался неизменяемым и сохранялись его инварианты:

```
private void readObfect(ObjectInputStream s)
throws IOException, ClassNotFoundException {
s.defaultReadObject();
    // Резервное копирование 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 резервное копирование осуществляется перед проверкой корректности и что для резервного копирования не используется метод clone из класса Date. Указанные особенности реализации необходимы для защиты объекта Period (статья 24). Заметим также, что выполнить резервное копирование для полей final невозможно. Следовательно, для того чтобы можно было воспользоваться методом readObject, мы должны сделать поля start и end неокончательными. Это огорчает, но приходится выбирать из двух зол меньшее. Разместив в классе метод readObject и удалив модификатор final из полей start и end, мы обнаруживаем, что класс MutablePeriod потерял свою силу. Приведенная выше программа выводит теперь следующие строки:

```
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
```

Есть простой безошибочный тест, показывающий, приемлем ли метод readObject, предлагаемый по умолчанию. Будете ли вы чувствовать себя уютно, если добавите в класс открытый конструктор, который в качестве параметров принимает значения полей вашего объекта, записываемых в сериализованную форму, а затем заносит эти значения в соответствующие поля без какой-либо проверки? Если вы не можете ответить на этот вопрос утвердительно, вам нужно явным образом реализовать метод readObject, который должен выполнять все необходимые проверки параметров и создавать все резервные копии, как это требуется от конструктора.

Между конструкторами и методами readObject существует еще одно сходство, касающееся расширяемых сериализуемых классов: метод readObject не должен вызывать переопределяемые методы ни прямо, ни косвенно (статья 15). Если это правило нарушено и вызываемый метод переопределен, то он будет вызван прежде, чем будет десериализовано состояние соответствующего подкласса. Скорее всего, это приведет к сбою программы.

Подведем итоги. Всякий раз, когда вы пишите 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. Приведем в кратком изложении рекомендации по написанию "пуленепробиваемого" метода readObject:

- для классов, где есть поля, которые хранят ссылки на объект
  и при этом должны оставаться закрытыми, для каждого объекта, который должен быть помещен в такое поле, необходимо создавать резервную копию. В эту категорию попадают также изменяемые компоненты неизменяемых классов.

- Для классов, где есть инварианты, выполняйте проверку этих инвариантов и в случае ошибки инициируйте исключение InvalidObjectException. Проверка должна производиться после создания всех резервных копий.

- Если после десериализации необходимо проверить целый гр~ф объектов, следует использовать интерфейс ObjectInputValidation.
  Порядок применения этого интерфейса выходит за рамки данной книги. Пример можно найти в *The Java Class Libraries. Second Edition. Volume 1* [Chan98, стр. 1256].

- Ни прямо, ни косвенно не используйте в этом классе переопределяемых методов.

Как альтернативу защищенному методу readObject можно использовать метод readResolve. Об этом говорится в статье 57.

При необходимости создавайте метод readResolue

В статье 2 описывается шаблон *Singleton* и приводится следующий пример класса. синглтона. Этот класс ограничивает доступ к конструктору с тем, чтобы гарантировать создание только одного экземпляра:

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {  }
        // Остальное опущено
        }
```

Как отмечалось в статье 2, этот класс перестает быть синглтоном, если в его декларацию добавляются слова "implements Serializable". Не имеет значения, использует этот класс сериализованную форму, предлагаемую по умолчанию, или специальную форму (статья 55). Не имеет также значения, предоставляется ли пользователю в этом классе явный метод readObject (статья 56). Любой метод readObject, явный или применяемый по умолчанию, возвращает вновь созданный экземпляр, а не тот, что был сформирован в момент инициализации класса. До выхода версии 1.2 написать сериализуемый класс-синглтон было невозможно.

В версии 1.2 механизм сериализации пополнился функцией readResolve [Serialization, 3.6]. Если класс десериализуемого объекта имеет метод readResolve с соответствующей декларацией, то по завершении десериализации этот метод будет вызван для вновь созданного объекта. Вместо ссылки на вновь созданный объект клиенту передается ссылка, возвращаемая этим методом. В большинстве случаев использования этого механизма ссылка на новый объект не сохраняется, а сам объект фактически является мертворожденным и немедленно становится объектом для сборки мусора.

Если класс Elvis создан так, чтобы реализовать интерфейс Serializable, то для обеспечения свойства синглтона достаточно будет создать следующий метод readResolve:

```
private Object readResolve() throws ObjectStreamException {
    // Возвращает только истинный экземпляр Elvis и дает возможность
    // сборщику мусора позаботиться об Elvis - самозванце
    return INSTANCE;  }
```

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.Это относится не только к классу Elvis, но и к любым другим синглтонам.

Метод readResolve необходим не только для синглтонов, но и для всех остальных классов, *контролирующих свои экземпляры* (instance-controlled), т. е. для тех классов, в которых сохранение некоего инварианта требует строгого контроля над процедурой создания экземпляров. Еще один пример класса, контролирующего свои экземпляры,- *перечисление типов* (статья 21), чей метод readResolve должен возвращать канонический экземпляр, соответствующий указанной константе в перечислении. Главное правило таково, что если вы пишете сериализуемый класс, не имеющий ни открытых, ни защищенных конструкторов, проверьте, нужен ли этому классу метод r~adResolve.

Второй вариант применения метода readResolve - в качестве безопасной альтернативы защищенному методу readObject(статья 56). В этом случае из метода readObjectизымаются все проверки параметров, а также процедуры создания резервных копий, и применяются проверки и резервное копирование, предоставляемые обычным конструктором. При использовании сериализованной формы, предлагаемой по умолчанию, метод readObjectможет быть полностью исключен. Как объясняется в статье 56, это дает возможность клиенту, имеющему злой умысел, создать экземпляр с испорченными инвариантами. Однако потенциально испорченный десериализованный экземпляр никогда не будет передан в активное использование. Из него просто будут извлечены данные для открытого конструктора или статического метода генерации, после чего он будет отброшен.

Изящество этого подхода заключается в том, что фактически устраняются составные части сериализации, выходящие за пределы языка, что делает невозможным нарушение каких-либо инвариантов класса, имевшихся до того, как этот класс был сделан сериализуемым. Чтобы продемонстрировать эту методику, в примере с классом Period (статья 56) вместо защищенного метода readObjectможно воспользоваться следующим методом readResolve:

```
// Идиома защищенного метода readResolve
private Object readResolve() throws ObjectStreamException {
return new Period(start, end);
}
```

Этот метод останавливает обе атаки, описанные в статье 56. Идиома защищенного метода readResolve имеет несколько преимуществ перед защищенным readObject. Это чисто *механический* прием, позволяющий делать класс сериализуемым, не подвергая риску его инварианты. Он требует небольшого объема кода, немного размышлений и работает надежно. Наконец, он устраняет искусственные ограничения, накладываемые сериализацией на использование окончательных полей.

Хотя идиома защищенного метода readResolveне имеет широкого применения, она заслуживает серьезного рассмотрения. Главный ее недостаток состоит в том, что она не годится для класса, который можно наследовать за пределами соответствующего пакета. Это не касается неизменяемых классов, поскольку они обычно являются окончательными (статья 13). Недостатком этой идиомы является и некоторое снижение скорости десериализации, поскольку она сопровождается созданием дополнительного объекта. На моей машине десериализация экземпляров Period замедлил ась примерно на один процент по сравнению с защищенным методом readObj ect.

Большое значение имеет доступность метода readResolve. Если вы помещаете его в окончательный класс, например синглтон, он должен быть закрытым. Помещая метод readResolveв расширяемый класс, вы должны внимательно изучить его доступность. Если он будет закрытым, его не удастся использовать ни в одном из подклассов. Если он будет доступен только в пакете, его применение ограничится подклассами из того же пакета. Защищенный или открытый метод будет доступен во всех подклассах, если только они его не переопределят. Если метод readResolveявляется защищенным или открытым и не переопределяется в подклассе, то десериализация сериализованного экземпляра подкласса создаст экземпляр суперкласса, а это, вероятно, будет совсем не то, чего вы ожидали.

Предыдущее рассуждение наводит на мысль о причине, по которой в классах, допускающих наследование, метод readResolveне может служить заменой защищенному методу readObject.Если бы в суперклассе метод readResolve был помечен как final, это не позволило бы правильно десериализовать экземпляры подкласса. Если бы он был переопределяемым, то подкласс, написанный злоумышленником, мог бы его переопределить таким методом, который возвращает испорченный экземпляр.

Подведем итоги. Вы должны использовать метод readResolveдля защиты "инвариантов контроля над экземплярами" в синглтонах и других классах, которые контролируют свои экземпляры. По существу, метод readResolveпревращает метод readObjectиз де-факто открытого конструктора в де-факто открытый статический метод генерации. Метод readResolveтакже полезен в качестве простой альтернативы защищенному методу readObjectв классах, для которых запрещено наследование за пределами соответствующего пакета.