# Содержание

# 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 некоторой задачи $A$ 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 $A'$?

Существует много книг, по которым можно изучать грамматику языка программирования Java, в том числе книги "*The Java Programming Language*" авторов Arnold, Gosling и Holmes [Arnold00] 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 Bloch) 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. В 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 Л. Стил-младший (Guy L. Steele Jr.)*
Берлингтон, шт. Массачусетс

# Предисловие автора

В 1996 г. 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, 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

# 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 6) и Defensive `readResolve` (статья 57). Где это необходимо, шаблоны и идиомы имеют ссылки на основные работы в данной области [Gamma95].

Таблица 1.1

*Версии платформы Java*

| Официальное название версии | Рабочий номер версии |
|---|---|
| JDK 1.1.x / JRE 1.1.x | 1.1 |
| Java 2 Platform, Standard Edition, v 1.2 | 1.2 |
| Java 2 Platform, Standard Edition, v 1.3 | 1.3 |
| Java 2 Platform, Standard Edition, v 1.4 | 1.4 |

Многие статьи содержат примеры 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 [Arnold00, Campione00]. 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 его пользователя. Модули должны быть настолько компактны, насколько это возможно, но не более того. (В этой книге термин "*модуль*" относится к любому программному компоненту, который используется много раз: от отдельного метода до сложной системы, состоящей из нескольких пакетов.) Программный код следует использовать повторно, а не копировать. Взаимозависимость между модулями должна быть сведена к минимуму. Ошибку нужно выявлять как можно 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, правильных, 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 VM 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.

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

```
import java.util.*;
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* (member interfaces). Сигнатура метода (signature) состоит из его названия и типов, которые имеют его формальные параметры. Тип значения, возвращаемого методом, в сигнатуру *не входит*.

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

Статья 1

## 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 (b ? 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.probablePrime`. (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. Те, кто читает программный код, в котором применяются 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 `a==b`. 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 API 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 `java.util.Collections`, для которого невозможно создать экземпляр. Все классы, соответствующие возвращаемым объектам, не являются открытыми.

API Collections Framework имеет гораздо меньшие размеры, чем это было бы, если бы в нем были представлены 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 Java Cryptography Extension (JCE). 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 JCE системный администратор регистрирует класс реализации, редактируя хорошо известный файл Properties: делает в нем запись, которая связывает 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:

```
// 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();
        }
    }
}
```

**Основной недостаток статических методов генерации заключается в том, что классы, не имеющие открытых или защищенных конструк-**

**торов, 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 возмож-

ности и не нашли достаточных доводов в чью-либо пользу, вероятно, лучше всего создать конструктор, хотя бы потому, что этот подход является нормой.

Статья 2

## Свойство синглтона обеспечивайте закрытым конструктором

*Синглтон* (singleton) — это класс, для которого экземпляр создается только один раз [Gamma95, 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
}
```

Закрытый конструктор вызывается только один раз для инициализации поля `Elvis.INSTANCE`. Отсутствие открытых или защищенных конструкторов *гарантирует* "вселенную с одним `Elvis`": после инициализации класса `Elvis` будет существовать ровно один экземпляр `Elvis` — не больше и не меньше. И 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
public class Elvis {
```

```
    private static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    public static Elvis getInstance() {
        return INSTANCE;
    }

    ... // Остальное опущено
}
```

Все вызовы статического метода `Elvis.getInstance` возвращают ссылку на один и тот же объект, и никакие другие экземпляры `Elvis` никогда 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 final, 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 (inlining) 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 API. 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 `readRe-`

solve (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;
}
```

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.

## Статья 3 Отсутствие экземпляров обеспечивает закрытый конструктор

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 использовать 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 `java.util.Collections`. Можно также собрать методы в некоем окончательном (final) классе вместо того, чтобы заниматься расширением класса.

Подобные *классы утилит* (utility class) 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 (default constructor), не имеющий параметров. Для пользователя этот конструктор ничем не будет отличаться от любого другого. В опубликованных API нередко можно встретить классы, непреднамеренно наделенные способностью порождать экземпляры.

**Попытки запретить классу создавать экземпляры, объявив его абстрактным, не работают.** Такой класс может иметь подкласс, для которого можно создавать экземпляры. Более того, это вводит пользователя в заблуждение, заставляя думать, что данный класс был разработан именно для наследования (статья 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
public class UtilityClass {

    // Подавляет появление конструктора по умолчанию,
    // а заодно и создание экземпляров класса
    private UtilityClass() {
        // 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 программы 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.

Статья 4

## 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";
```