Broadview
www.broadview.com.cn
PEARSON

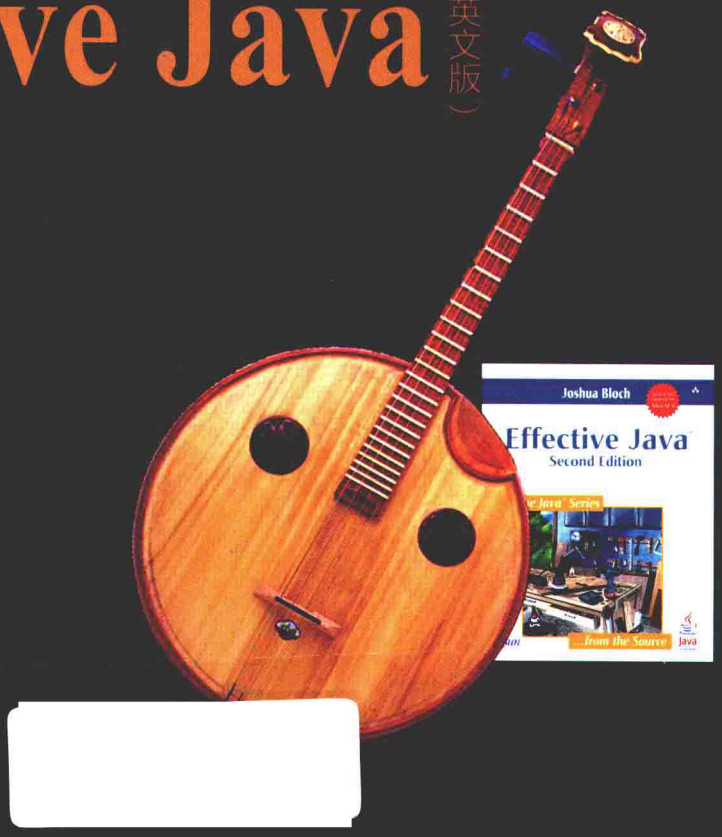
· 原味精品书系 ·

Effective Java（英文版）

（第2版）

Effective Java，2E

[美] Joshua Bloch 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

· 原味精品书系 ·

Effective Java（英文版）（第2版）

Effective Java，2E

[美] Joshua Bloch 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书介绍了在 Java 编程中的 78 条非常具有实用价值的经验规则，这些经验规则涵盖了大部分开发人员每天所面临的问题的解决方案。通过对 Java 平台设计专家所使用的技术的全面描述，揭示了应该做什么，不应该做什么，以及怎样才能编写出清晰、健壮和高效的代码。

本书中的每条规则都以简短、独立的小文章形式出现，并通过示例代码进一步进行说明。本书内容全面，结构清晰，讲解详细，可作为技术人员的参考用书。

版权贸易合同登记号　图字：01-2015-6096

图书在版编目（CIP）数据

Effective Java：第 2 版：英文 /（美）布洛赫（Bloch,J.）著. — 北京：电子工业出版社，2016.4
（原味精品书系）
ISBN 978-7-121-27314-8
I. ① E…II. ① 布…III. ① JAVA 语言－程序设计－英文 IV. ① TP312

中国版本图书馆 CIP 数据核字（2015）第 233695 号

责任编辑：张　玲
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980 1/16　　印张：22.5　　字数：432 千字
版　　次：2016 年 4 月第 1 版
印　　次：2016 年 4 月第 1 次印刷
定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

推荐序

如果有同事对你说，“Spouse of me this night today manufactures the unusual meal in a home. You will join?”这时候你脑子里可能会浮现出三件事情：同事在邀请你参加他的家庭晚宴，英语肯定不是你这位同事的母语，更多的则是满脑子的疑惑。

如果你曾经学习过第二种语言，并且尝试过在课堂之外使用这种语言，你就该知道有三件事情是必须掌握的：这门语言的结构如何（语法）、如何命名你想谈论的事物（词汇），以及如何以惯用和高效的方式来表达日常的事物（用法）。在课堂上大多只涉及到前面两点，当你使出浑身解数想让对方明白你的意思时，常常会发现当地人对你的表述忍俊不禁。

程序设计语言也是如此。你需要理解语言的核心：它是面向算法的，还是面向函数的，或者是面向对象的？你需要知道词汇表：标准类库提供了哪些数据结构、操作和功能设施（Facility）？你还需要熟悉如何用习惯和高效的方式来构建代码。也许是因为前面两点比较容易编写，所以关于程序设计语言的书籍通常只涉及这两点，或者只是蜻蜓点水般地介绍一下用法。语法和词汇是语言本身固有的特性，但是用法则反映了使用这门语言的群体特征。

例如，Java 程序设计语言是一门支持单继承的面向对象程序设计语言，在每个方法的内部，它都支持命令式的（面向语句的，Statement-Oriented）编码风格。Java 类库提供了对图形显示、网络、分布式计算和安全性的支持。但是，如何把这门语言以最佳的方式运用到实践中呢？

还有一点，程序与口头的句子以及大多数书籍和杂志都不同，它会随着时间的推移而发生变化。仅仅编写出能够有效地工作并且能够被别人理解的代码往往是不够的，我们还必须把代码组织成易于修改的形式。针对某个任务可能会有 10 种不同的编码方法，而在这 10 种编码方法中，有 7 种编码方法是笨拙的、低效的或者是难以理解的。而在剩下的 3 种编码方法中，哪一种会是最接近该任务的下一年度发行版本的代码呢？

目前有大量的书籍可以供你学习 Java 程序设计语言的语法，包括 *The Java Programming Language* [Arnold05]（作者 Arnold、Gosling 和 Holmes），以及 *The Java Language Specification* [JLS]（作者 Gosling、Joy 和 Bracha）。同样地，与 Java 程序设计语言相关的类库和 API 的书籍也有很多。

本书解决了你的第三种需求，即如何以惯用和高效的方式来表达日常的事物（用法）。多年来，作者 Joshua Bloch 在 Sun Microsystems 公司一直从事 Java 语言的扩展、实现和使用的工作；他还大量地阅读了其他人的代码，包括我的代码。他在本书中提出了许多很好的建议，他系统地把这些建议组织起来，旨在告诉读者如何更好地构造代码以便它们能工作

得更好，也便于其他人能够理解这些代码，而且将来对代码进行修改和改善时也不至于那么头疼。甚至，你的程序也会因此而变得更加令人愉悦、更加优美和雅致。

Guy L. Steele Jr.

Burlington, Massachusetts

2001 年 4 月

前言

我于 2001 年写了本书的第一版之后，Java 平台又发生了很多变化，所以是该出第二版的时候了。Java 5 中最为重要的变化是增加了泛型、枚举类型、注解、自动装箱和 for-each 循环。然后是增加了新的并发类库：java.util.concurrent，并且在 Java 5 中进行了发布。我和 Gilad Bracha 一起，有幸带领团队设计了最新的语言特性，还参加了设计和开发并发类库的团队，这个团队由 Doug Lea 领导。

Java 平台中另一个大的变化在于广泛采用了现代的 IDE（Integrated Development Environment），例如 Eclipse、IntelliJ IDEA 和 NetBeans，以及静态分析工具的 IDE，如 FindBugs。虽然我未参与这部分工作，但已经从中受益匪浅，并且很清楚它们对 Java 开发体验所带来的影响。

2004 年，我从 Sun 公司换到了 Google 公司，但在过去的 4 年中，我仍然继续参与 Java 平台的开发，在 Google 公司和 JCP（Java Community Process）的大力帮助下，继续并发和集合 API 的开发。我还有幸利用 Java 平台去开发供 Google 内部使用的类库。现在我了解了作为一名用户的感受。

我在 2001 年编写第一版的时候，主要目的是与读者分享我的经验，以便让大家避免我所走过的弯路，帮助大家更容易地走向成功。第二版仍然大量采用来自 Java 平台类库的真实范例。

第一版所带来的反应远远超出了我最大的预期。我在收集所有新的资料以使本书保持最新时，尽可能地保持了资料的真实。毫无疑问，本书的篇幅肯定会增加，从 57 个条目发展到了 78 个。我不仅增加了 23 个条目，并且修改了原来的所有资料，并删去了一些已经过时的条目。在附录中，你可以看到本书内容与第一版内容的对照情况。

在第一版的前言中，我说过：Java 程序设计语言和它的类库非常有益于代码质量和效率的提高，并且使得用 Java 进行编码成为一种乐趣。Java 5 和 Java 6 发行版本中的变化是好事，也使得 Java 平台日趋完善。现在这个平台比 2001 年的要大得多，也复杂得多，但是一旦掌握了使用新特性的模式和惯用模式，它们就会使你的程序变得更完美，使你的工作变得更轻松。我希望第二版能够体现出我对 Java 平台持续的热情，并将这种热情传递给你，帮助你更加高效和愉快地使用 Java 平台及其新的特性。

San Jose, California

2008 年 4 月

目录

推荐序 ix

前言 xi

致谢 xiii

1 Introduction 1

2 Creating and Destroying Objects 5

Item 1: Consider static factory methods instead of constructors ... 5
Item 2: Consider a builder when faced with many constructor parameters 11
Item 3: Enforce the singleton property with a private constructor or an enum type 17
Item 4: Enforce noninstantiability with a private constructor 19
Item 5: Avoid creating unnecessary objects 20
Item 6: Eliminate obsolete object references 24
Item 7: Avoid finalizers 27

3 Methods Common to All Objects 33

Item 8: Obey the general contract when overriding `equals` 33
Item 9: Always override `hashCode` when you override `equals` 45
Item 10: Always override `toString` 51
Item 11: Override `clone` judiciously 54
Item 12: Consider implementing `Comparable` 62

4 Classes and Interfaces.....67

Item 13: Minimize the accessibility of classes and members.....67
Item 14: In public classes, use accessor methods, not public fields.....71
Item 15: Minimize mutability.....73
Item 16: Favor composition over inheritance.....81
Item 17: Design and document for inheritance or else prohibit it..87
Item 18: Prefer interfaces to abstract classes.....93
Item 19: Use interfaces only to define types.....98
Item 20: Prefer class hierarchies to tagged classes.....100
Item 21: Use function objects to represent strategies.....103
Item 22: Favor static member classes over nonstatic.....106

5 Generics.....109

Item 23: Don't use raw types in new code.....109
Item 24: Eliminate unchecked warnings.....116
Item 25: Prefer lists to arrays.....119
Item 26: Favor generic types.....124
Item 27: Favor generic methods.....129
Item 28: Use bounded wildcards to increase API flexibility.....134
Item 29: Consider typesafe heterogeneous containers.....142

6 Enums and Annotations.....147

Item 30: Use enums instead of `int` constants.....147
Item 31: Use instance fields instead of ordinals.....158
Item 32: Use `EnumSet` instead of bit fields.....159
Item 33: Use `EnumMap` instead of ordinal indexing.....161
Item 34: Emulate extensible enums with interfaces.....165
Item 35: Prefer annotations to naming patterns.....169
Item 36: Consistently use the `Override` annotation.....176
Item 37: Use marker interfaces to define types.....179

7 Methods.....181

Item 38: Check parameters for validity.....181
Item 39: Make defensive copies when needed.....184
Item 40: Design method signatures carefully.....189
Item 41: Use overloading judiciously.....191

Item 42: Use varargs judiciously 197
Item 43: Return empty arrays or collections, not nulls 201
Item 44: Write doc comments for all exposed API elements 203

8 General Programming 209

Item 45: Minimize the scope of local variables 209
Item 46: Prefer for-each loops to traditional `for` loops 212
Item 47: Know and use the libraries 215
Item 48: Avoid `float` and `double` if exact answers are required 218
Item 49: Prefer primitive types to boxed primitives 221
Item 50: Avoid strings where other types are more appropriate . . 224
Item 51: Beware the performance of string concatenation 227
Item 52: Refer to objects by their interfaces 228
Item 53: Prefer interfaces to reflection 230
Item 54: Use native methods judiciously 233
Item 55: Optimize judiciously 234
Item 56: Adhere to generally accepted naming conventions 237

9 Exceptions 241

Item 57: Use exceptions only for exceptional conditions 241
Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors 244
Item 59: Avoid unnecessary use of checked exceptions 246
Item 60: Favor the use of standard exceptions 248
Item 61: Throw exceptions appropriate to the abstraction 250
Item 62: Document all exceptions thrown by each method 252
Item 63: Include failure-capture information in detail messages 254
Item 64: Strive for failure atomicity 256
Item 65: Don't ignore exceptions 258

10 Concurrency 259

Item 66: Synchronize access to shared mutable data 259
Item 67: Avoid excessive synchronization 265
Item 68: Prefer executors and tasks to threads 271
Item 69: Prefer concurrency utilities to `wait` and `notify` 273

Item 70: Document thread safety 278
Item 71: Use lazy initialization judiciously 282
Item 72: Don't depend on the thread scheduler 286
Item 73: Avoid thread groups 288

11 Serialization 289

Item 74: Implement `Serializable` judiciously 289
Item 75: Consider using a custom serialized form 295
Item 76: Write `readObject` methods defensively 302
Item 77: For instance control, prefer enum types to `readResolve` 308
Item 78: Consider serialization proxies instead of serialized instances 312

Appendix: Items Corresponding to First Edition 317

References 321

Index 327

CHAPTER 1

Introduction

THIS book is designed to help you make the most effective use of the Java™ programming language and its fundamental libraries, `java.lang`, `java.util`, and, to a lesser extent, `java.util.concurrent` and `java.io`. The book discusses other libraries from time to time, but it does not cover graphical user interface programming, enterprise APIs, or mobile devices.

This book consists of seventy-eight items, each of which conveys one rule. The rules capture practices generally held to be beneficial by the best and most experienced programmers. The items are loosely grouped into ten chapters, each concerning one broad aspect of software design. The book is not intended to be read from cover to cover: each item stands on its own, more or less. The items are heavily cross-referenced so you can easily plot your own course through the book.

Many new features were added to the platform in Java 5 (release 1.5). Most of the items in this book use these features in some way. The following table shows you where to go for primary coverage of these features:

Feature	Chapter or Item
Generics	Chapter 5
Enums	Items 30–34
Annotations	Items 35–37
For-each loop	Item 46
Autoboxing	Items 40, 49
Varargs	Item 42
Static import	Item 19
`java.util.concurrent`	Items 68, 69

Most items are illustrated with program examples. A key feature of this book is that it contains code examples illustrating many design patterns and idioms. Where appropriate, they are cross-referenced to the standard reference work in this area [Gamma95].

Many items contain one or more program examples illustrating some practice to be avoided. Such examples, sometimes known as *antipatterns,* are clearly labeled with a comment such as "`// Never do this!`" In each case, the item explains why the example is bad and suggests an alternative approach.

This book is not for beginners: it assumes that you are already comfortable with the Java programming language. If you are not, consider one of the many fine introductory texts [Arnold05, Sestoft05]. While the book is designed to be accessible to anyone with a working knowledge of the language, it should provide food for thought even for advanced programmers.

Most of the rules in this book derive from a few fundamental principles. Clarity and simplicity are of paramount importance. The user of a module should never be surprised by its behavior. Modules should be as small as possible but no smaller. (As used in this book, the term *module* refers to any reusable software component, from an individual method to a complex system consisting of multiple packages.) Code should be reused rather than copied. The dependencies between modules should be kept to a minimum. Errors should be detected as soon as possible after they are made, ideally at compile time.

While the rules in this book do not apply 100 percent of the time, they do characterize best programming practices in the great majority of cases. You should not slavishly follow these rules, but violate them only occasionally and with good reason. Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.

For the most part, this book is not about performance. It is about writing programs that are clear, correct, usable, robust, flexible, and maintainable. If you can do that, it's usually a relatively simple matter to get the performance you need (Item 55). Some items do discuss performance concerns, and a few of these items provide performance numbers. These numbers, which are introduced with the phrase "On my machine," should be regarded as approximate at best.

For what it's worth, my machine is an aging homebuilt 2.2 GHz dual-core AMD Opteron™ 170 with 2 gigabytes of RAM, running Sun's 1.6_05 release of the Java SE Development Kit (JDK) atop Microsoft Windows® XP Professional SP2. This JDK has two virtual machines, the Java HotSpot™ Client and Server VMs. Performance numbers were measured on the Server VM.

When discussing features of the Java programming language and its libraries, it is sometimes necessary to refer to specific releases. For brevity, this book uses "engineering version numbers" in preference to official release names. This table shows the mapping between release names and engineering version numbers.

Official Release Name	Engineering Version Number
JDK 1.1.x / JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4
Java 2 Platform, Standard Edition, v 5.0	1.5
Java Platform, Standard Edition 6	1.6

The examples are reasonably complete, but they favor readability over completeness. They freely use classes from the packages `java.util` and `java.io`. In order to compile the examples, you may have to add one or more of these import statements:

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

Other boilerplate is similarly omitted. The book's Web site, `http://java.sun.com/docs/books/effective`, contains an expanded version of each example, which you can compile and run.

For the most part, this book uses technical terms as they are defined in *The Java Language Specification, Third Edition* [JLS]. A few terms deserve special mention. The language supports four kinds of types: *interfaces* (including *annotations*), *classes* (including *enums*), *arrays*, and *primitives*. The first three are known as *reference types*. Class instances and arrays are *objects*; primitive values are not. A class's *members* consist of its *fields*, *methods*, *member classes*, and *member interfaces*. A method's *signature* consists of its name and the types of its formal parameters; the signature does *not* include the method's return type.

This book uses a few terms differently from the *The Java Language Specification*. Unlike *The Java Language Specification*, this book uses *inheritance* as a synonym for *subclassing*. Instead of using the term inheritance for interfaces, this

book simply states that a class *implements* an interface or that one interface *extends* another. To describe the access level that applies when none is specified, this book uses the descriptive term *package-private* instead of the technically correct term *default access* [JLS, 6.6.1].

This book uses a few technical terms that are not defined in *The Java Language Specification*. The term *exported API*, or simply *API*, refers to the classes, interfaces, constructors, members, and serialized forms by which a programmer accesses a class, interface, or package. (The term *API*, which is short for *application programming interface*, is used in preference to the otherwise preferable term *interface* to avoid confusion with the language construct of that name.) A programmer who writes a program that uses an API is referred to as a *user* of the API. A class whose implementation uses an API is a *client* of the API.

Classes, interfaces, constructors, members, and serialized forms are collectively known as *API elements*. An exported API consists of the API elements that are accessible outside of the package that defines the API. These are the API elements that any client can use and the author of the API commits to support. Not coincidentally, they are also the elements for which the Javadoc utility generates documentation in its default mode of operation. Loosely speaking, the exported API of a package consists of the public and protected members and constructors of every public class or interface in the package.

CHAPTER 2

Creating and Destroying Objects

THIS chapter concerns creating and destroying objects: when and how to create them, when and how to avoid creating them, how to ensure they are destroyed in a timely manner, and how to manage any cleanup actions that must precede their destruction.

Item 1: Consider static factory methods instead of constructors

The normal way for a class to allow a client to obtain an instance of itself is to provide a public constructor. There is another technique that should be a part of every programmer's toolkit. A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class. Here's a simple example from `Boolean` (the boxed primitive class for the primitive type `boolean`). This method translates a `boolean` primitive value into a `Boolean` object reference:

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

Note that a static factory method is not the same as the *Factory Method* pattern from *Design Patterns* [Gamma95, p. 107]. The static factory method described in this item has no direct equivalent in *Design Patterns*.

A class can provide its clients with static factory methods instead of, or in addition to, constructors. Providing a static factory method instead of a public constructor has both advantages and disadvantages.

One advantage of static factory methods is that, unlike constructors, they have names. If the parameters to a constructor do not, in and of themselves, describe the object being returned, a static factory with a well-chosen name is easier to use and the resulting client code easier to read. For example, the constructor

`BigInteger(int, int, Random)`, which returns a `BigInteger` that is probably prime, would have been better expressed as a static factory method named `BigInteger.probablePrime`. (This method was eventually added in release 1.4.)

A class can have only a single constructor with a given signature. Programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. The user of such an API will never be able to remember which constructor is which and will end up calling the wrong one by mistake. People reading code that uses these constructors will not know what the code does without referring to the class documentation.

Because they have names, static factory methods don't share the restriction discussed in the previous paragraph. In cases where a class seems to require multiple constructors with the same signature, replace the constructors with static factory methods and carefully chosen names to highlight their differences.

A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they're invoked. This allows immutable classes (Item 15) to use preconstructed instances, or to cache instances as they're constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The `Boolean.valueOf(boolean)` method illustrates this technique: it never creates an object. This technique is similar to the *Flyweight* pattern [Gamma95, p. 195]. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

The ability of static factory methods to return the same object from repeated invocations allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be *instance-controlled.* There are several reasons to write instance-controlled classes. Instance control allows a class to guarantee that it is a singleton (Item 3) or noninstantiable (Item 4). Also, it allows an immutable class (Item 15) to make the guarantee that no two equal instances exist: `a.equals(b)` if and only if `a==b`. If a class makes this guarantee, then its clients can use the `==` operator instead of the `equals(Object)` method, which may result in improved performance. Enum types (Item 30) provide this guarantee.

A third advantage of static factory methods is that, unlike constructors, they can return an object of any subtype of their return type. This gives you great flexibility in choosing the class of the returned object.

One application of this flexibility is that an API can return objects without making their classes public. Hiding implementation classes in this fashion leads to a very compact API. This technique lends itself to *interface-based frameworks* (Item 18), where interfaces provide natural return types for static factory methods.

Interfaces can't have static methods, so by convention, static factory methods for an interface named *Type* are put in a noninstantiable class (Item 4) named *Types*.

For example, the Java Collections Framework has thirty-two convenience implementations of its collection interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (`java.util.Collections`). The classes of the returned objects are all nonpublic.

The Collections Framework API is much smaller than it would have been had it exported thirty-two separate public classes, one for each convenience implementation. It is not just the bulk of the API that is reduced, but the *conceptual weight*. The user knows that the returned object has precisely the API specified by its interface, so there is no need to read additional class documentation for the implementation classes. Furthermore, using such a static factory method requires the client to refer to the returned object by its interface rather than its implementation class, which is generally good practice (Item 52).

Not only can the class of an object returned by a public static factory method be nonpublic, but the class can vary from invocation to invocation depending on the values of the parameters to the static factory. Any class that is a subtype of the declared return type is permissible. The class of the returned object can also vary from release to release for enhanced software maintainability and performance.

The class `java.util.EnumSet` (Item 32), introduced in release 1.5, has no public constructors, only static factories. They return one of two implementations, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a `RegularEnumSet` instance, which is backed by a single `long`; if the enum type has sixty-five or more elements, the factories return a `JumboEnumSet` instance, backed by a `long` array.

The existence of these two implementation classes is invisible to clients. If `RegularEnumSet` ceased to offer performance advantages for small enum types, it could be eliminated from a future release with no ill effects. Similarly, a future release could add a third or fourth implementation of `EnumSet` if it proved beneficial for performance. Clients neither know nor care about the class of the object they get back from the factory; they care only that it is some subclass of `EnumSet`.

The class of the object returned by a static factory method need not even exist at the time the class containing the method is written. Such flexible static factory methods form the basis of *service provider frameworks*, such as the Java Database Connectivity API (JDBC). A service provider framework is a system in which multiple service providers implement a service, and the system makes the implementations available to its clients, decoupling them from the implementations.