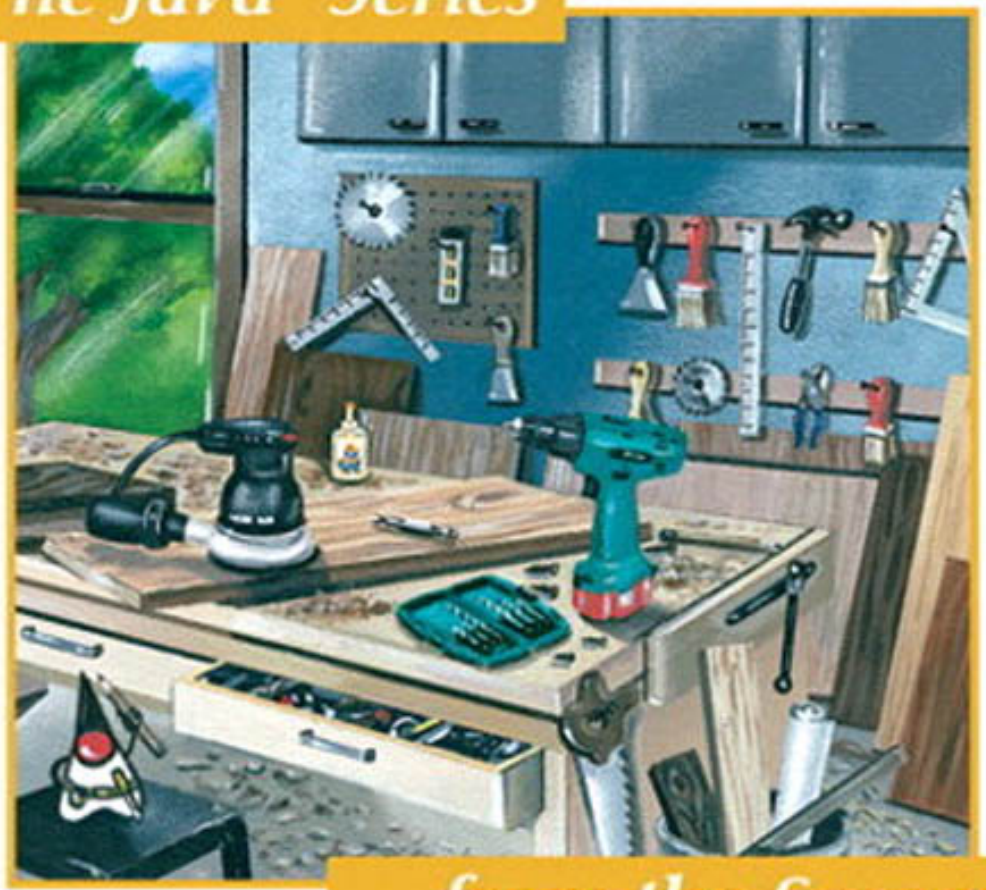
Joshua Bloch

Revised and Updated for Java SE 6

Effective Java™

Second Edition

The Java™ Series



...from the Source

Sun microsystems

Java Sun Microsystems

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

This Book Is Safari Enabled

The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days.

Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:
- Go to www.informit.com/onlineedition
- Complete the brief registration form
- Enter the coupon code I1DV-B4HC-5AP3-6WCL-R4MC

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2008926278

ISBN-13: 978-0-321-35668-0
ISBN-10: 0-321-35668-3

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
Second printing, June 2008

Foreword

IF a colleague were to say to you, "Spouse of me this night today manufactures the unusual meal in a home. You will join?" three things would likely cross your mind: third, that you had been invited to dinner; second, that English was not your colleague's first language; and first, a good deal of puzzlement.

If you have ever studied a second language yourself and then tried to use it outside the classroom, you know that there are three things you must master: how the language is structured (grammar), how to name things you want to talk about (vocabulary), and the customary and effective ways to say everyday things (usage). Too often only the first two are covered in the classroom, and you find native speakers constantly suppressing their laughter as you try to make yourself understood.

It is much the same with a programming language. You need to understand the core language: is it algorithmic, functional, object-oriented? You need to know the vocabulary: what data structures, operations, and facilities are provided by the standard libraries? And you need to be familiar with the customary and effective ways to structure your code. Books about programming languages often cover only the first two, or discuss usage only spottily. Maybe that's because the first two are in some ways easier to write about. Grammar and vocabulary are properties of the language alone, but usage is characteristic of a community that uses it.

The Java programming language, for example, is object-oriented with single inheritance and supports an imperative (statement-oriented) coding style within each method. The libraries address graphic display support, networking, distributed computing, and security. But how is the language best put to use in practice?

There is another point. Programs, unlike spoken sentences and unlike most books and magazines, are likely to be changed over time. It's typically not enough to produce code that operates effectively and is readily understood by other persons; one must also organize the code so that it is easy to modify. There may be ten ways to write code for some task *T*. Of those ten ways, seven will be awkward, inefficient, or puzzling. Of the other three, which is most likely to be similar to the code needed for the task *T′* in next year's software release?

There are numerous books from which you can learn the grammar of the Java Programming Language, including *The Java™ Programming Language* by Arnold, Gosling, and Holmes [Arnold05] or *The Java™ Language Specification* by Gosling, Joy, yours truly, and Bracha [JLS]. Likewise, there are dozens of books on the libraries and APIs associated with the Java programming language.

This book addresses your third need: customary and effective usage. Joshua Bloch has spent years extending, implementing, and using the Java programming language at Sun Microsystems; he has also read a lot of other people's code, including mine. Here he offers good advice, systematically organized, on how to structure your code so that it works well, so that other people can understand it, so that future modifications and improvements are less likely to cause headaches—perhaps, even, so that your programs will be pleasant, elegant, and graceful.

Guy L. Steele Jr.
Burlington, Massachusetts
April 2001

Preface

Preface to the Second Edition

A lot has happened to the Java platform since I wrote the first edition of this book in 2001, and it's high time for a second edition. The most significant set of changes was the addition of generics, enum types, annotations, autoboxing, and the for-each loop in Java 5. A close second was the addition of the new concurrency library, `java.util.concurrent`, also released in Java 5. With Gilad Bracha, I had the good fortune to lead the teams that designed the new language features. I also had the good fortune to serve on the team that designed and developed the concurrency library, which was led by Doug Lea.

The other big change in the platform is the widespread adoption of modern Integrated Development Environments (IDEs), such as Eclipse, IntelliJ IDEA, and NetBeans, and of static analysis tools, such as FindBugs. While I have not been involved in these efforts, I've benefited from them immensely and learned how they affect the Java development experience.

In 2004, I moved from Sun to Google, but I've continued my involvement in the development of the Java platform over the past four years, contributing to the concurrency and collections APIs through the good offices of Google and the Java Community Process. I've also had the pleasure of using the Java platform to develop libraries for use within Google. Now I know what it feels like to be a user.

As was the case in 2001 when I wrote the first edition, my primary goal is to share my experience with you so that you can imitate my successes while avoiding my failures. The new material continues to make liberal use of real-world examples from the Java platform libraries.

The first edition succeeded beyond my wildest expectations, and I've done my best to stay true to its spirit while covering all of the new material that was required to bring the book up to date. It was inevitable that the book would grow, and grow it did, from fifty-seven items to seventy-eight. Not only did I add twenty-three items, but I thoroughly revised all the original material and retired a

few items whose better days had passed. In the Appendix, you can see how the material in this edition relates to the material in the first edition.

In the Preface to the First Edition, I wrote that the Java programming language and its libraries were immensely conducive to quality and productivity, and a joy to work with. The changes in releases 5 and 6 have taken a good thing and made it better. The platform is much bigger now than it was in 2001 and more complex, but once you learn the patterns and idioms for using the new features, they make your programs better and your life easier. I hope this edition captures my continued enthusiasm for the platform and helps make your use of the platform and its new features more effective and enjoyable.

San Jose, California
April 2008

Preface to the First Edition

In 1996 I pulled up stakes and headed west to work for JavaSoft, as it was then known, because it was clear that that was where the action was. In the intervening five years I've served as Java platform libraries architect. I've designed, implemented, and maintained many of the libraries and served as a consultant for many others. Presiding over these libraries as the Java platform matured was a once-in-a-lifetime opportunity. It is no exaggeration to say that I had the privilege to work with some of the great software engineers of our generation. In the process, I learned a lot about the Java programming language—what works, what doesn't, and how to use the language and its libraries to best effect.

This book is my attempt to share my experience with you so that you can imitate my successes while avoiding my failures. I borrowed the format from Scott Meyers's *Effective C++* [Meyers98], which consists of fifty items, each conveying one specific rule for improving your programs and designs. I found the format to be singularly effective, and I hope you do too.

In many cases, I took the liberty of illustrating the items with real-world examples from the Java platform libraries. When describing something that could have been done better, I tried to pick on code that I wrote myself, but occasionally I pick on something written by a colleague. I sincerely apologize if, despite my best efforts, I've offended anyone. Negative examples are cited not to cast blame

but in the spirit of cooperation, so that all of us can benefit from the experience of those who've gone before.

While this book is not targeted solely at developers of reusable components, it is inevitably colored by my experience writing such components over the past two decades. I naturally think in terms of exported APIs (Application Programming Interfaces), and I encourage you to do likewise. Even if you aren't developing reusable components, thinking in these terms tends to improve the quality of the software you write. Furthermore, it's not uncommon to write a reusable component without knowing it: You write something useful, share it with your buddy across the hall, and before long you have half a dozen users. At this point, you no longer have the flexibility to change the API at will and are thankful for all the effort that you put into designing the API when you first wrote the software.

My focus on API design may seem a bit unnatural to devotees of the new lightweight software development methodologies, such as *Extreme Programming* [Beck99]. These methodologies emphasize writing the simplest program that could possibly work. If you're using one of these methodologies, you'll find that a focus on API design serves you well in the *refactoring* process. The fundamental goals of refactoring are the improvement of system structure and the avoidance of code duplication. These goals are impossible to achieve in the absence of well-designed APIs for the components of the system.

No language is perfect, but some are excellent. I have found the Java programming language and its libraries to be immensely conducive to quality and productivity, and a joy to work with. I hope this book captures my enthusiasm and helps make your use of the language more effective and enjoyable.

Cupertino, California
April 2001

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.

There are three essential components of a service provider framework: a *service interface*, which providers implement; a *provider registration API*, which the system uses to register implementations, giving clients access to them; and a *service access API*, which clients use to obtain an instance of the service. The service access API typically allows but does not require the client to specify some criteria for choosing a provider. In the absence of such a specification, the API returns an instance of a default implementation. The service access API is the "flexible static factory" that forms the basis of the service provider framework.

An optional fourth component of a service provider framework is a *service provider interface*, which providers implement to create instances of their service implementation. In the absence of a service provider interface, implementations are registered by class name and instantiated reflectively (Item 53). In the case of JDBC, `Connection` plays the part of the service interface, `DriverManager.registerDriver` is the provider registration API, `DriverManager.getConnection` is the service access API, and `Driver` is the service provider interface.

There are numerous variants of the service provider framework pattern. For example, the service access API can return a richer service interface than the one required of the provider, using the Adapter pattern [Gamma95, p. 139]. Here is a simple implementation with a service provider interface and a default provider:

```
// Service provider framework sketch

// Service interface
public interface Service {
    ... // Service-specific methods go here
}

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

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

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

```
    // Provider registration API
    public static void registerDefaultProvider(Provider p) {
        registerProvider(DEFAULT_PROVIDER_NAME, p);
    }
    public static void registerProvider(String name, Provider p){
        providers.put(name, p);
    }

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

A fourth advantage of static factory methods is that they reduce the verbosity of creating parameterized type instances. Unfortunately, you must specify the type parameters when you invoke the constructor of a parameterized class even if they're obvious from context. This typically requires you to provide the type parameters twice in quick succession:

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

This redundant specification quickly becomes painful as the length and complexity of the type parameters increase. With static factories, however, the compiler can figure out the type parameters for you. This is known as *type inference*. For example, suppose that `HashMap` provided this static factory:

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

Then you could replace the wordy declaration above with this succinct alternative:

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

Someday the language may perform this sort of type inference on constructor invocations as well as method invocations, but as of release 1.6, it does not.

Unfortunately, the standard collection implementations such as `HashMap` do not have factory methods as of release 1.6, but you can put these methods in your own utility class. More importantly, you can provide such static factories in your own parameterized classes.

The main disadvantage of providing only static factory methods is that classes without public or protected constructors cannot be subclassed. The same is true for nonpublic classes returned by public static factories. For example, it is impossible to subclass any of the convenience implementation classes in the Collections Framework. Arguably this can be a blessing in disguise, as it encourages programmers to use composition instead of inheritance (Item 16).

A second disadvantage of static factory methods is that they are not readily distinguishable from other static methods. They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this disadvantage by drawing attention to static factories in class or interface comments, and by adhering to common naming conventions. Here are some common names for static factory methods:

- `valueOf`—Returns an instance that has, loosely speaking, the same value as its parameters. Such static factories are effectively type-conversion methods.
- `of`—A concise alternative to `valueOf`, popularized by `EnumSet` (Item 32).
- `getInstance`—Returns an instance that is described by the parameters but cannot be said to have the same value. In the case of a singleton, `getInstance` takes no parameters and returns the sole instance.
- `newInstance`—Like `getInstance`, except that `newInstance` guarantees that each instance returned is distinct from all others.
- `get`*Type*—Like `getInstance`, but used when the factory method is in a different class. *Type* indicates the type of object returned by the factory method.
- `new`*Type*—Like `newInstance`, but used when the factory method is in a different class. *Type* indicates the type of object returned by the factory method.

In summary, static factory methods and public constructors both have their uses, and it pays to understand their relative merits. Often static factories are preferable, so avoid the reflex to provide public constructors without first considering static factories.

Item 2: Consider a builder when faced with many constructor parameters

Static factories and constructors share a limitation: they do not scale well to large numbers of optional parameters. Consider the case of a class representing the Nutrition Facts label that appears on packaged foods. These labels have a few required fields—serving size, servings per container, and calories per serving—and over twenty optional fields—total fat, saturated fat, trans fat, cholesterol, sodium, and so on. Most products have nonzero values for only a few of these optional fields.

What sort of constructors or static factories should you write for such a class? Traditionally, programmers have used the *telescoping constructor* pattern, in which you provide a constructor with only the required parameters, another with a single optional parameter, a third with two optional parameters, and so on, culminating in a constructor with all the optional parameters. Here's how it looks in practice. For brevity's sake, only four optional fields are shown:

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

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
            int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }
```

```
    public NutritionFacts(int servingSize, int servings,
           int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

When you want to create an instance, you use the constructor with the shortest parameter list containing all the parameters you want to set:

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

Typically this constructor invocation will require many parameters that you don't want to set, but you're forced to pass a value for them anyway. In this case, we passed a value of `0` for `fat`. With "only" six parameters this may not seem so bad, but it quickly gets out of hand as the number of parameters increases.

In short, **the telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.** The reader is left wondering what all those values mean and must carefully count parameters to find out. Long sequences of identically typed parameters can cause subtle bugs. If the client accidentally reverses two such parameters, the compiler won't complain, but the program will misbehave at runtime (Item 40).

A second alternative when you are faced with many constructor parameters is the *JavaBeans* pattern, in which you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest:

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

    public NutritionFacts() { }
```

```
    // Setters
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)     { servings = val; }
    public void setCalories(int val)     { calories = val; }
    public void setFat(int val)          { fat = val; }
    public void setSodium(int val)       { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

This pattern has none of the disadvantages of the telescoping constructor pattern. It is easy, if a bit wordy, to create instances, and easy to read the resulting code:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, **a JavaBean may be in an inconsistent state partway through its construction.** The class does not have the option of enforcing consistency merely by checking the validity of the constructor parameters. Attempting to use an object when it's in an inconsistent state may cause failures that are far removed from the code containing the bug, hence difficult to debug. A related disadvantage is that **the JavaBeans pattern precludes the possibility of making a class immutable** (Item 15), and requires added effort on the part of the programmer to ensure thread safety.

It is possible to reduce these disadvantages by manually "freezing" the object when its construction is complete and not allowing it to be used until frozen, but this variant is unwieldy and rarely used in practice. Moreover, it can cause errors at runtime, as the compiler cannot ensure that the programmer calls the freeze method on an object before using it.

Luckily, there is a third alternative that combines the safety of the telescoping constructor pattern with the readability of the JavaBeans pattern. It is a form of the *Builder* pattern [Gamma95, p. 97]. Instead of making the desired object directly, the client calls a constructor (or static factory) with all of the required parameters and gets a *builder object*. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameterless `build` method to generate the object, which is immutable. The builder is a static member class (Item 22) of the class it builds. Here's how it looks in practice:

```
// Builder Pattern
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories      = 0;
        private int fat           = 0;
        private int carbohydrate  = 0;
        private int sodium        = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;           return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val;  return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

Note that `NutritionFacts` is immutable, and that all parameter default values are in a single location. The builder's setter methods return the builder itself so that invocations can be chained. Here's how the client code looks:

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

This client code is easy to write and, more importantly, to read. **The Builder pattern simulates named optional parameters** as found in Ada and Python.

Like a constructor, a builder can impose invariants on its parameters. The `build` method can check these invariants. It is critical that they be checked after copying the parameters from the builder to the object, and that they be checked on the object fields rather than the builder fields (Item 39). If any invariants are violated, the `build` method should throw an `IllegalStateException` (Item 60). The exception's detail method should indicate which invariant is violated (Item 63).

Another way to impose invariants involving multiple parameters is to have setter methods take entire groups of parameters on which some invariant must hold. If the invariant isn't satisfied, the setter method throws an `IllegalArgumentException`. This has the advantage of detecting the invariant failure as soon as the invalid parameters are passed, instead of waiting for `build` to be invoked.

A minor advantage of builders over constructors is that builders can have multiple varargs parameters. Constructors, like methods, can have only one varargs parameter. Because builders use separate methods to set each parameter, they can have as many varargs parameters as you like, up to one per setter method.

The Builder pattern is flexible. A single builder can be used to build multiple objects. The parameters of the builder can be tweaked between object creations to vary the objects. The builder can fill in some fields automatically, such as a serial number that automatically increases each time an object is created.

A builder whose parameters have been set makes a fine *Abstract Factory* [Gamma95, p. 87]. In other words, a client can pass such a builder to a method to enable the method to create one or more objects for the client. To enable this usage, you need a type to represent the builder. If you are using release 1.5 or a later release, a single generic type (Item 26) suffices for *all* builders, no matter what type of object they're building:

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

Note that our `NutritionFacts.Builder` class could be declared to implement `Builder<NutritionFacts>`.

Methods that take a `Builder` instance would typically constrain the builder's type parameter using a *bounded wildcard type* (Item 28). For example, here is a method that builds a tree using a client-provided `Builder` instance to build each node:

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

The traditional Abstract Factory implementation in Java has been the `Class` object, with the `newInstance` method playing the part of the `build` method. This usage is fraught with problems. The `newInstance` method always attempts to invoke the class's parameterless constructor, which may not even exist. You don't get a compile-time error if the class has no accessible parameterless constructor. Instead, the client code must cope with `InstantiationException` or `IllegalAccessException` at runtime, which is ugly and inconvenient. Also, the `newInstance` method propagates any exceptions thrown by the parameterless constructor, even though `newInstance` lacks the corresponding `throws` clauses. In other words, **`Class.newInstance` breaks compile-time exception checking.** The `Builder` interface, shown above, corrects these deficiencies.

The Builder pattern does have disadvantages of its own. In order to create an object, you must first create its builder. While the cost of creating the builder is unlikely to be noticeable in practice, it could be a problem in some performance-critical situations. Also, the Builder pattern is more verbose than the telescoping constructor pattern, so it should be used only if there are enough parameters, say, four or more. But keep in mind that you may want to add parameters in the future. If you start out with constructors or static factories, and add a builder when the class evolves to the point where the number of parameters starts to get out of hand, the obsolete constructors or static factories will stick out like a sore thumb. Therefore, it's often better to start with a builder in the first place.

In summary, **the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters**, especially if most of those parameters are optional. Client code is much easier to read and write with builders than with the traditional telescoping constructor pattern, and builders are much safer than JavaBeans.

Item 3: Enforce the singleton property with a private constructor or an enum type

A *singleton* is simply a class that is instantiated exactly once [Gamma95, p. 127]. Singletons typically represent a system component that is intrinsically unique, such as the window manager or file system. **Making a class a singleton can make it difficult to test its clients,** as it's impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type.

Before release 1.5, there were two ways to implement singletons. Both are based on keeping the constructor private and exporting a public static member to provide access to the sole instance. In one approach, the member is a final field:

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

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

The private constructor is called only once, to initialize the public static final field `Elvis.INSTANCE`. The lack of a public or protected constructor *guarantees* a "monoelvistic" universe: exactly one `Elvis` instance will exist once the `Elvis` class is initialized—no more, no less. Nothing that a client does can change this, with one caveat: a privileged client can invoke the private constructor reflectively (Item 53) with the aid of the `AccessibleObject.setAccessible` method. If you need to defend against this attack, modify the constructor to make it throw an exception if it's asked to create a second instance.

In the second approach to implementing singletons, the public member is a static factory method:

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }

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

All calls to `Elvis.getInstance` return the same object reference, and no other `Elvis` instance will ever be created (with the same caveat mentioned above).

The main advantage of the public field approach is that the declarations make it clear that the class is a singleton: the public static field is final, so it will always contain the same object reference. There is no longer any performance advantage to the public field approach: modern Java virtual machine (JVM) implementations are almost certain to inline the call to the static factory method.

One advantage of the factory-method approach is that it gives you the flexibility to change your mind about whether the class should be a singleton without changing its API. The factory method returns the sole instance but could easily be modified to return, say, a unique instance for each thread that invokes it. A second advantage, concerning generic types, is discussed in Item 27. Often neither of these advantages is relevant, and the public field approach is simpler.

To make a singleton class that is implemented using either of the previous approaches *serializable* (Chapter 11), it is not sufficient merely to add `implements Serializable` to its declaration. To maintain the singleton guarantee, you have to declare all instance fields `transient` and provide a `readResolve` method (Item 77). Otherwise, each time a serialized instance is deserialized, a new instance will be created, leading, in the case of our example, to spurious `Elvis` sightings. To prevent this, add this `readResolve` method to the `Elvis` class:

```
// readResolve method to preserve singleton property
private Object readResolve() {
     // Return the one true Elvis and let the garbage collector
     // take care of the Elvis impersonator.
    return INSTANCE;
}
```

As of release 1.5, there is a third approach to implementing singletons. Simply make an enum type with one element:

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;

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

This approach is functionally equivalent to the public field approach, except that it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks. While this approach has yet to be widely adopted, **a single-element enum type is the best way to implement a singleton**.

Item 4: Enforce noninstantiability with a private constructor

Occasionally you'll want to write a class that is just a grouping of static methods and static fields. Such classes have acquired a bad reputation because some people abuse them to avoid thinking in terms of objects, but they do have valid uses. They can be used to group related methods on primitive values or arrays, in the manner of `java.lang.Math` or `java.util.Arrays`. They can also be used to group static methods, including factory methods (Item 1), for objects that implement a particular interface, in the manner of `java.util.Collections`. Lastly, they can be used to group methods on a final class, instead of extending the class.

Such *utility classes* were not designed to be instantiated: an instance would be nonsensical. In the absence of explicit constructors, however, the compiler provides a public, parameterless *default constructor*. To a user, this constructor is indistinguishable from any other. It is not uncommon to see unintentionally instantiable classes in published APIs.

Attempting to enforce noninstantiability by making a class abstract does not work. The class can be subclassed and the subclass instantiated. Furthermore, it misleads the user into thinking the class was designed for inheritance (Item 17). There is, however, a simple idiom to ensure noninstantiability. A default constructor is generated only if a class contains no explicit constructors, so **a class can be made noninstantiable by including a private constructor**:

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    ...  // Remainder omitted
}
```

Because the explicit constructor is private, it is inaccessible outside of the class. The `AssertionError` isn't strictly required, but it provides insurance in case the constructor is accidentally invoked from within the class. It guarantees that the class will never be instantiated under any circumstances. This idiom is mildly counterintuitive, as the constructor is provided expressly so that it cannot be invoked. It is therefore wise to include a comment, as shown above.

As a side effect, this idiom also prevents the class from being subclassed. All constructors must invoke a superclass constructor, explicitly or implicitly, and a subclass would have no accessible superclass constructor to invoke.

Item 5: Avoid creating unnecessary objects

It is often appropriate to reuse a single object instead of creating a new functionally equivalent object each time it is needed. Reuse can be both faster and more stylish. An object can always be reused if it is *immutable* (Item 15).

As an extreme example of what not to do, consider this statement:

```
String s = new String("stringette");  // DON'T DO THIS!
```

The statement creates a new `String` instance each time it is executed, and none of those object creations is necessary. The argument to the `String` constructor (`"stringette"`) is itself a `String` instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of `String` instances can be created needlessly.

The improved version is simply the following:

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

This version uses a single `String` instance, rather than creating a new one each time it is executed. Furthermore, it is guaranteed that the object will be reused by any other code running in the same virtual machine that happens to contain the same string literal [JLS, 3.10.5].

You can often avoid creating unnecessary objects by using *static factory methods* (Item 1) in preference to constructors on immutable classes that provide both. For example, the static factory method `Boolean.valueOf(String)` is almost always preferable to the constructor `Boolean(String)`. The constructor creates a new object each time it's called, while the static factory method is never required to do so and won't in practice.

In addition to reusing immutable objects, you can also reuse mutable objects if you know they won't be modified. Here is a slightly more subtle, and much more common, example of what not to do. It involves mutable `Date` objects that are never modified once their values have been computed. This class models a person and has an `isBabyBoomer` method that tells whether the person is a "baby boomer," in other words, whether the person was born between 1946 and 1964:

```
public class Person {
    private final Date birthDate;

    // Other fields, methods, and constructor omitted
```

```
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
        // Unnecessary allocation of expensive object
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
               birthDate.compareTo(boomEnd)   <  0;
    }
}
```

The `isBabyBoomer` method unnecessarily creates a new `Calendar`, `TimeZone`, and two `Date` instances each time it is invoked. The version that follows avoids this inefficiency with a static initializer:

```
public class Person {
    private final Date birthDate;
    // Other fields, methods, and constructor omitted

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

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

The improved version of the `Person` class creates `Calendar`, `TimeZone`, and `Date` instances only once, when it is initialized, instead of creating them every time `isBabyBoomer` is invoked. This results in significant performance gains if the

method is invoked frequently. On my machine, the original version takes 32,000 ms for 10 million invocations, while the improved version takes 130 ms, which is about 250 times faster. Not only is performance improved, but so is clarity. Changing `boomStart` and `boomEnd` from local variables to static final fields makes it clear that these dates are treated as constants, making the code more understandable. In the interest of full disclosure, the savings from this sort of optimization will not always be this dramatic, as `Calendar` instances are particularly expensive to create.

If the improved version of the `Person` class is initialized but its `isBabyBoomer` method is never invoked, the `BOOM_START` and `BOOM_END` fields will be initialized unnecessarily. It would be possible to eliminate the unnecessary initializations by *lazily initializing* these fields (Item 71) the first time the `isBabyBoomer` method is invoked, but it is not recommended. As is often the case with lazy initialization, it would complicate the implementation and would be unlikely to result in a noticeable performance improvement beyond what we've already achieved (Item 55).

In the previous examples in this item, it was obvious that the objects in question could be reused because they were not modified after initialization. There are other situations where it is less obvious. Consider the case of *adapters* [Gamma95, p. 139], also known as *views*. An adapter is an object that delegates to a backing object, providing an alternative interface to the backing object. Because an adapter has no state beyond that of its backing object, there's no need to create more than one instance of a given adapter to a given object.

For example, the `keySet` method of the `Map` interface returns a `Set` view of the `Map` object, consisting of all the keys in the map. Naively, it would seem that every call to `keySet` would have to create a new `Set` instance, but every call to `keySet` on a given `Map` object may return the same `Set` instance. Although the returned `Set` instance is typically mutable, all of the returned objects are functionally identical: when one of the returned objects changes, so do all the others because they're all backed by the same `Map` instance. While it is harmless to create multiple instances of the `keySet` view object, it is also unnecessary.

There's a new way to create unnecessary objects in release 1.5. It is called *autoboxing*, and it allows the programmer to mix primitive and boxed primitive types, boxing and unboxing automatically as needed. Autoboxing blurs but does not erase the distinction between primitive and boxed primitive types. There are subtle semantic distinctions, and not-so-subtle performance differences (Item 49). Consider the following program, which calculates the sum of all the positive `int`

values. To do this, the program has to use `long` arithmetic, because an `int` is not big enough to hold the sum of all the positive `int` values:

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

This program gets the right answer, but it is *much* slower than it should be, due to a one-character typographical error. The variable `sum` is declared as a `Long` instead of a `long`, which means that the program constructs about 2^{31} unnecessary `Long` instances (roughly one for each time the `long i` is added to the `Long sum`). Changing the declaration of `sum` from `Long` to `long` reduces the runtime from 43 seconds to 6.8 seconds on my machine. The lesson is clear: **prefer primitives to boxed primitives, and watch out for unintentional autoboxing.**

This item should not be misconstrued to imply that object creation is expensive and should be avoided. On the contrary, the creation and reclamation of small objects whose constructors do little explicit work is cheap, especially on modern JVM implementations. Creating additional objects to enhance the clarity, simplicity, or power of a program is generally a good thing.

Conversely, avoiding object creation by maintaining your own *object pool* is a bad idea unless the objects in the pool are extremely heavyweight. The classic example of an object that *does* justify an object pool is a database connection. The cost of establishing the connection is sufficiently high that it makes sense to reuse these objects. Also, your database license may limit you to a fixed number of connections. Generally speaking, however, maintaining your own object pools clutters your code, increases memory footprint, and harms performance. Modern JVM implementations have highly optimized garbage collectors that easily outperform such object pools on lightweight objects.

The counterpoint to this item is Item 39 on *defensive copying*. Item 5 says, "Don't create a new object when you should reuse an existing one," while Item 39 says, "Don't reuse an existing object when you should create a new one." Note that the penalty for reusing an object when defensive copying is called for is far greater than the penalty for needlessly creating a duplicate object. Failing to make defensive copies where required can lead to insidious bugs and security holes; creating objects unnecessarily merely affects style and performance.

Item 6: Eliminate obsolete object references

When you switch from a language with manual memory management, such as C or C++, to a garbage-collected language, your job as a programmer is made much easier by the fact that your objects are automatically reclaimed when you're through with them. It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn't quite true.

Consider the following simple stack implementation:

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

There's nothing obviously wrong with this program (but see Item 26 for a generic version). You could test it exhaustively, and it would pass every test with flying colors, but there's a problem lurking. Loosely speaking, the program has a "memory leak," which can silently manifest itself as reduced performance due to

increased garbage collector activity or increased memory footprint. In extreme cases, such memory leaks can cause disk paging and even program failure with an `OutOfMemoryError`, but such failures are relatively rare.

So where is the memory leak? If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains *obsolete references* to these objects. An obsolete reference is simply a reference that will never be dereferenced again. In this case, any references outside of the "active portion" of the element array are obsolete. The active portion consists of the elements whose index is less than `size`.

Memory leaks in garbage-collected languages (more properly known as *unintentional object retentions*) are insidious. If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on. Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.

The fix for this sort of problem is simple: null out references once they become obsolete. In the case of our `Stack` class, the reference to an item becomes obsolete as soon as it's popped off the stack. The corrected version of the `pop` method looks like this:

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

An added benefit of nulling out obsolete references is that, if they are subsequently dereferenced by mistake, the program will immediately fail with a `NullPointerException`, rather than quietly doing the wrong thing. It is always beneficial to detect programming errors as quickly as possible.

When programmers are first stung by this problem, they may overcompensate by nulling out every object reference as soon as the program is finished using it. This is neither necessary nor desirable, as it clutters up the program unnecessarily. **Nulling out object references should be the exception rather than the norm.** The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope (Item 45).

So when should you null out a reference? What aspect of the `Stack` class makes it susceptible to memory leaks? Simply put, it *manages its own memory.* The *storage pool* consists of the elements of the `elements` array (the object reference cells, not the objects themselves). The elements in the active portion of the array (as defined earlier) are *allocated*, and those in the remainder of the array are *free*. The garbage collector has no way of knowing this; to the garbage collector, all of the object references in the `elements` array are equally valid. Only the programmer knows that the inactive portion of the array is unimportant. The programmer effectively communicates this fact to the garbage collector by manually nulling out array elements as soon as they become part of the inactive portion.

Generally speaking, **whenever a class manages its own memory, the programmer should be alert for memory leaks**. Whenever an element is freed, any object references contained in the element should be nulled out.

Another common source of memory leaks is caches. Once you put an object reference into a cache, it's easy to forget that it's there and leave it in the cache long after it becomes irrelevant. There are several solutions to this problem. If you're lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a `WeakHashMap`; entries will be removed automatically after they become obsolete. Remember that `WeakHashMap` is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

More commonly, the useful lifetime of a cache entry is less well defined, with entries becoming less valuable over time. Under these circumstances, the cache should occasionally be cleansed of entries that have fallen into disuse. This can be done by a background thread (perhaps a `Timer` or `ScheduledThreadPoolExecutor`) or as a side effect of adding new entries to the cache. The `LinkedHashMap` class facilitates the latter approach with its `removeEldestEntry` method. For more sophisticated caches, you may need to use `java.lang.ref` directly.

A third common source of memory leaks is listeners and other callbacks. If you implement an API where clients register callbacks but don't deregister them explicitly, they will accumulate unless you take some action. The best way to ensure that callbacks are garbage collected promptly is to store only *weak references* to them, for instance, by storing them only as keys in a `WeakHashMap`.

Because memory leaks typically do not manifest themselves as obvious failures, they may remain present in a system for years. They are typically discovered only as a result of careful code inspection or with the aid of a debugging tool known as a *heap profiler.* Therefore, it is very desirable to learn to anticipate problems like this before they occur and prevent them from happening.

Item 7: Avoid finalizers

Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we'll cover later in this item, but as a rule of thumb, you should avoid finalizers.

C++ programmers are cautioned not to think of finalizers as Java's analog of C++ destructors. In C++, destructors are the normal way to reclaim the resources associated with an object, a necessary counterpart to constructors. In Java, the garbage collector reclaims the storage associated with an object when it becomes unreachable, requiring no special effort on the part of the programmer. C++ destructors are also used to reclaim other nonmemory resources. In Java, the `try-finally` block is generally used for this purpose.

One shortcoming of finalizers is that there is no guarantee they'll be executed promptly [JLS, 12.6]. It can take arbitrarily long between the time that an object becomes unreachable and the time that its finalizer is executed. This means that you should **never do anything time-critical in a finalizer.** For example, it is a grave error to depend on a finalizer to close files, because open file descriptors are a limited resource. If many files are left open because the JVM is tardy in executing finalizers, a program may fail because it can no longer open files.

The promptness with which finalizers are executed is primarily a function of the garbage collection algorithm, which varies widely from JVM implementation to JVM implementation. The behavior of a program that depends on the promptness of finalizer execution may likewise vary. It is entirely possible that such a program will run perfectly on the JVM on which you test it and then fail miserably on the JVM favored by your most important customer.

Tardy finalization is not just a theoretical problem. Providing a finalizer for a class can, under rare conditions, arbitrarily delay reclamation of its instances. A colleague debugged a long-running GUI application that was mysteriously dying with an `OutOfMemoryError`. Analysis revealed that at the time of its death, the application had thousands of graphics objects on its finalizer queue just waiting to be finalized and reclaimed. Unfortunately, the finalizer thread was running at a lower priority than another application thread, so objects weren't getting finalized at the rate they became eligible for finalization. The language specification makes no guarantees as to which thread will execute finalizers, so there is no portable way to prevent this sort of problem other than to refrain from using finalizers.

Not only does the language specification provide no guarantee that finalizers will get executed promptly; it provides no guarantee that they'll get executed at

all. It is entirely possible, even likely, that a program terminates without executing finalizers on some objects that are no longer reachable. As a consequence, you should **never depend on a finalizer to update critical persistent state**. For example, depending on a finalizer to release a persistent lock on a shared resource such as a database is a good way to bring your entire distributed system to a grinding halt.

Don't be seduced by the methods `System.gc` and `System.runFinalization`. They may increase the odds of finalizers getting executed, but they don't guarantee it. The only methods that claim to guarantee finalization are `System.runFinalizersOnExit` and its evil twin, `Runtime.runFinalizersOnExit`. These methods are fatally flawed and have been deprecated [ThreadStop].

In case you are not yet convinced that finalizers should be avoided, here's another tidbit worth considering: if an uncaught exception is thrown during finalization, the exception is ignored, and finalization of that object terminates [JLS, 12.6]. Uncaught exceptions can leave objects in a corrupt state. If another thread attempts to use such a corrupted object, arbitrary nondeterministic behavior may result. Normally, an uncaught exception will terminate the thread and print a stack trace, but not if it occurs in a finalizer—it won't even print a warning.

Oh, and one more thing: **there is a *severe* performance penalty for using finalizers.** On my machine, the time to create and destroy a simple object is about 5.6 ns. Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers.

So what should you do instead of writing a finalizer for a class whose objects encapsulate resources that require termination, such as files or threads? Just **provide an *explicit termination method***, and require clients of the class to invoke this method on each instance when it is no longer needed. One detail worth mentioning is that the instance must keep track of whether it has been terminated: the explicit termination method must record in a private field that the object is no longer valid, and other methods must check this field and throw an `IllegalStateException` if they are called after the object has been terminated.

Typical examples of explicit termination methods are the `close` methods on `InputStream`, `OutputStream`, and `java.sql.Connection`. Another example is the `cancel` method on `java.util.Timer`, which performs the necessary state change to cause the thread associated with a `Timer` instance to terminate itself gently. Examples from `java.awt` include `Graphics.dispose` and `Window.dispose`. These methods are often overlooked, with predictably dire performance consequences. A related method is `Image.flush`, which deallocates all the

resources associated with an `Image` instance but leaves it in a state where it can still be used, reallocating the resources if necessary.

Explicit termination methods are typically used in combination with the `try-finally` construct to ensure termination. Invoking the explicit termination method inside the `finally` clause ensures that it will get executed even if an exception is thrown while the object is being used:

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

So what, if anything, are finalizers good for? There are perhaps two legitimate uses. One is to act as a "safety net" in case the owner of an object forgets to call its explicit termination method. While there's no guarantee that the finalizer will be invoked promptly, it may be better to free the resource late than never, in those (hopefully rare) cases when the client fails to call the explicit termination method. But **the finalizer should log a warning if it finds that the resource has not been terminated**, as this indicates a bug in the client code, which should be fixed. If you are considering writing such a safety-net finalizer, think long and hard about whether the extra protection is worth the extra cost.

The four classes cited as examples of the explicit termination method pattern (`FileInputStream`, `FileOutputStream`, `Timer`, and `Connection`) have finalizers that serve as safety nets in case their termination methods aren't called. Unfortunately these finalizers do not log warnings. Such warnings generally can't be added after an API is published, as it would appear to break existing clients.

A second legitimate use of finalizers concerns objects with *native peers*. A native peer is a native object to which a normal object delegates via native methods. Because a native peer is not a normal object, the garbage collector doesn't know about it and can't reclaim it when its Java peer is reclaimed. A finalizer is an appropriate vehicle for performing this task, *assuming the native peer holds no critical resources*. If the native peer holds resources that must be terminated promptly, the class should have an explicit termination method, as described above. The termination method should do whatever is required to free the critical resource. The termination method can be a native method, or it can invoke one.

It is important to note that "finalizer chaining" is not performed automatically. If a class (other than `Object`) has a finalizer and a subclass overrides it, the subclass finalizer must invoke the superclass finalizer manually. You should finalize the subclass in a `try` block and invoke the superclass finalizer in the corresponding `finally` block. This ensures that the superclass finalizer gets executed even if the subclass finalization throws an exception and vice versa. Here's how it looks. Note that this example uses the `Override` annotation (`@Override`), which was added to the platform in release 1.5. You can ignore `Override` annotations for now, or see Item 36 to find out what they mean:

```
// Manual finalizer chaining
@Override protected void finalize() throws Throwable {
    try {
        ... // Finalize subclass state
    } finally {
        super.finalize();
    }
}
```

If a subclass implementor overrides a superclass finalizer but forgets to invoke it, the superclass finalizer will never be invoked. It is possible to defend against such a careless or malicious subclass at the cost of creating an additional object for every object to be finalized. Instead of putting the finalizer on the class requiring finalization, put the finalizer on an anonymous class (Item 22) whose sole purpose is to finalize its enclosing instance. A single instance of the anonymous class, called a *finalizer guardian*, is created for each instance of the enclosing class. The enclosing instance stores the sole reference to its finalizer guardian in a private instance field so the finalizer guardian becomes eligible for finalization at the same time as the enclosing instance. When the guardian is finalized, it performs the finalization activity desired for the enclosing instance, just as if its finalizer were a method on the enclosing class:

```
// Finalizer Guardian idiom
public class Foo {
   // Sole purpose of this object is to finalize outer Foo object
   private final Object finalizerGuardian = new Object() {
      @Override protected void finalize() throws Throwable {
         ... // Finalize outer Foo object
      }
   };
   ...  // Remainder omitted
}
```

Note that the public class, `Foo`, has no finalizer (other than the trivial one it inherits from `Object`), so it doesn't matter whether a subclass finalizer calls `super.finalize` or not. This technique should be considered for every nonfinal public class that has a finalizer.

In summary, don't use finalizers except as a safety net or to terminate noncritical native resources. In those rare instances where you do use a finalizer, remember to invoke `super.finalize`. If you use a finalizer as a safety net, remember to log the invalid usage from the finalizer. Lastly, if you need to associate a finalizer with a public, nonfinal class, consider using a finalizer guardian, so finalization can take place even if a subclass finalizer fails to invoke `super.finalize`.

Index

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U