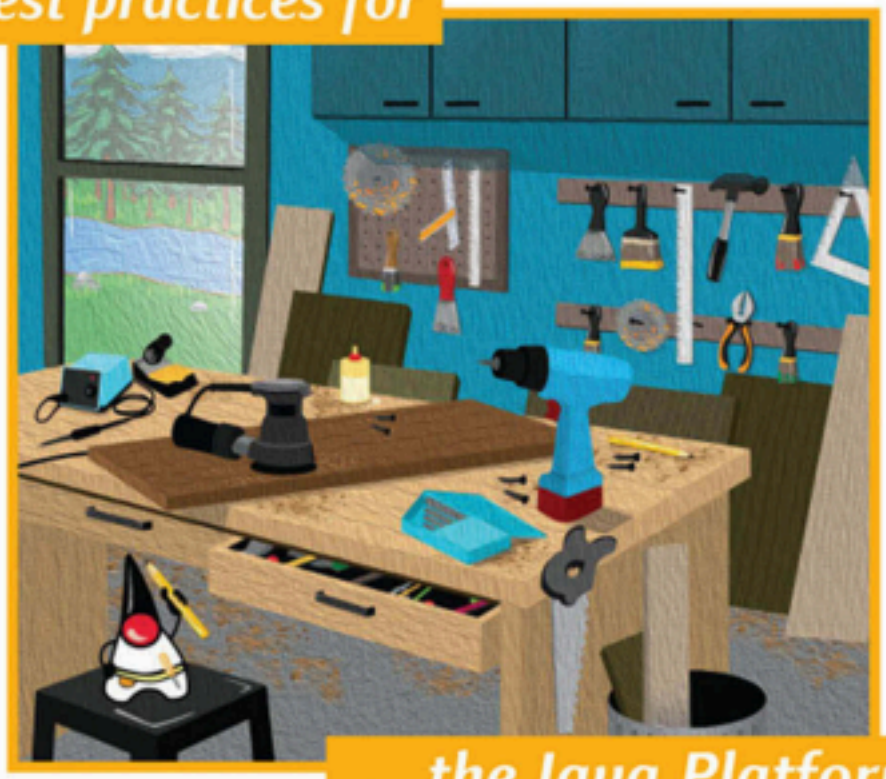
Joshua Bloch

Updated for Java 9

Effective Java

Third Edition

Best practices for



...the Java Platform

Effective Java

Third Edition

This page intentionally left blank

Effective Java

Third Edition

Joshua Bloch

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017956176

ISBN-13: 978-0-13-468599-1
ISBN-10: 0-13-468599-7

1 17

To my family: Cindy, Tim, and Matt
This page intentionally left blank

Contents

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 col league'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 proper ties 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, distrib uted 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 per sons; 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, or *The Java™ Language Specification* by Gosling, Joy, yours truly, and Bracha. 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 Third Edition

IN 1997, when Java was new, James Gosling (the father of Java), described it as a "blue collar language" that was "pretty simple" [Gosling97]. At about the same time, Bjarne Stroustrup (the father of C++) described C++ as a "multi-paradigm language" that "deliberately differs from languages designed to support a single way of writing programs" [Stroustrup95]. Stroustrup warned:

> Much of the relative simplicity of Java is—like for most new languages—partly an illusion and partly a function of its incompleteness. As time passes, Java will grow significantly in size and complexity. It will double or triple in size and grow implementation-dependent extensions or libraries. [Stroustrup]

Now, twenty years later, it's fair to say that Gosling and Stroustrup were both right. Java is now large and complex, with multiple abstractions for many things, from parallel execution, to iteration, to the representation of dates and times.

I still like Java, though my ardor has cooled a bit as the platform has grown. Given its increased size and complexity, the need for an up-to-date best-practices guide is all the more critical. With this third edition of *Effective Java*, I did my best to provide you with one. I hope this edition continues to satisfy the need, while staying true to the spirit of the first two editions.

Small is beautiful, but simple ain't easy.

San Jose, California
November 2017

P.S. I would be remiss if I failed to mention an industry-wide best practice that has occupied a fair amount of my time lately. Since the birth of our field in the 1950's, we have freely reimplemented each others' APIs. This practice was critical to the meteoric success of computer technology. I am active in the effort to preserve this freedom [CompSci17], and I encourage you to join me. It is crucial to the continued health of our profession that we retain the right to reimplement each others' APIs.

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 exam ples 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 contin-

ued 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, imple mented, 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 imi tate my successes while avoiding my failures. I borrowed the format from Scott Meyers's *Effective C++*, which consists of fifty items, each conveying one spe cific rule for improving your programs and designs. I found the format to be sin gularly 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 compo-

nent 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*. 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 refac toring are the improvement of system structure and the avoidance of code duplica

tion. 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

Acknowledgments

Acknowledgments for the Third Edition

I thank the readers of the first two editions of this book for giving it such a kind and enthusiastic reception, for taking its ideas to heart, and for letting me know what a positive influence it had on them and their work. I thank the many professors who used the book in their courses, and the many engineering teams that adopted it.

I thank the whole team at Addison-Wesley and Pearson for their kindness, pro fessionalism, patience, and grace under extreme pressure. Through it all, my editor Greg Doench remained unflappable: a fine editor and a perfect gentleman. I'm afraid his hair may have turned a bit gray as a result of this project, and I humbly apologize. My project manager, Julie Nahil, and my project editor, Dana Wilson, were all I could hope for: diligent, prompt, organized, and friendly. My copy editor, Kim Wimpsett, was meticulous and tasteful.

I have yet again been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them. The core team, who reviewed most every chapter, consisted of Cindy Bloch, Brian Kernighan, Kevin Bourrillion, Joe Bowbeer, William Chargin, Joe Darcy, Brian Goetz, Tim Halloran, Stuart Marks, Tim Peierls, and Yoshiki Shibata, Other reviewers included Marcus Biel, Dan Bloch, Beth Bottos, Martin Buchholz, Michael Diamond, Charlie Garrod, Tom Hawtin, Doug Lea, Aleksey Shipilëv, Lou Wasserman, and Peter Weinberger. These reviewers made numerous suggestions that led to great improvements in this book and saved me from many embarrassments.

I give special thanks to William Chargin, Doug Lea, and Tim Peierls, who served as sounding boards for many of the ideas in this book. William, Doug, and Tim were unfailingly generous with their time and knowledge.

Finally, I thank my wife, Cindy Bloch, for encouraging me to write, for read ing each item in raw form, for writing the index, for helping me with all of the

things that invariably come up when you take on a big project, and for putting up with me while I wrote.

Acknowledgments for the Second Edition

I thank the readers of the first edition of this book for giving it such a kind and enthusiastic reception, for taking its ideas to heart, and for letting me know what a positive influence it had on them and their work. I thank the many professors who used the book in their courses, and the many engineering teams that adopted it.

I thank the whole team at Addison-Wesley for their kindness, professionalism, patience, and grace under pressure. Through it all, my editor Greg Doench remained unflappable: a fine editor and a perfect gentleman. My production man ager, Julie Nahil, was everything that a production manager should be: diligent, prompt, organized, and friendly. My copy editor, Barbara Wood, was meticulous and tasteful.

I have once again been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them. The core team, who reviewed every chapter, consisted of Lexi Baugher, Cindy Bloch, Beth Bottos, Joe Bowbeer, Brian Goetz, Tim Halloran, Brian Kernighan, Rob Konigsberg, Tim Peierls, Bill Pugh, Yoshiki Shibata, Peter Stout, Peter Weinberger, and Frank Yellin. Other reviewers included Pablo Bellver, Dan Bloch, Dan Bornstein, Kevin Bourrillion, Martin Buchholz, Joe Darcy, Neal Gafter, Laurence Gonsalves, Aaron Green house, Barry Hayes, Peter Jones, Angelika Langer, Doug Lea, Bob Lee, Jeremy Manson, Tom May, Mike McCloskey, Andriy Tereshchenko, and Paul Tyma. Again, these reviewers made numerous suggestions that led to great improve ments in this book and saved me from many embarrassments. And again, any remaining embarrassments are my responsibility.

I give special thanks to Doug Lea and Tim Peierls, who served as sounding boards for many of the ideas in this book. Doug and Tim were unfailingly gener ous with their time and knowledge.

I thank my manager at Google, Prabha Krishna, for her continued support and encouragement.

Finally, I thank my wife, Cindy Bloch, for encouraging me to write, for read ing each item in raw form, for helping me with Framemaker, for writing the index, and for putting up with me while I wrote.

Acknowledgments for the First Edition

I thank Patrick Chan for suggesting that I write this book and for pitching the

idea to Lisa Friendly, the series managing editor; Tim Lindholm, the series technical editor; and Mike Hendrickson, executive editor of Addison-Wesley. I thank Lisa, Tim, and Mike for encouraging me to pursue the project and for their superhuman patience and unyielding faith that I would someday write this book.

I thank James Gosling and his original team for giving me something great to write about, and I thank the many Java platform engineers who followed in James's footsteps. In particular, I thank my colleagues in Sun's Java Platform Tools and Libraries Group for their insights, their encouragement, and their sup port. The team consists of Andrew Bennett, Joe Darcy, Neal Gafter, Iris Garcia, Konstantin Kladko, Ian Little, Mike McCloskey, and Mark Reinhold. Former members include Zhenghua Li, Bill Maddox, and Naveen Sanjeeva.

I thank my manager, Andrew Bennett, and my director, Larry Abrahams, for lending their full and enthusiastic support to this project. I thank Rich Green, the VP of Engineering at Java Software, for providing an environment where engi neers are free to think creatively and to publish their work.

I have been blessed with the best team of reviewers imaginable, and I give my sincerest thanks to each of them: Andrew Bennett, Cindy Bloch, Dan Bloch, Beth Bottos, Joe Bowbeer, Gilad Bracha, Mary Campione, Joe Darcy, David Eckhardt, Joe Fialli, Lisa Friendly, James Gosling, Peter Haggar, David Holmes, Brian Kernighan, Konstantin Kladko, Doug Lea, Zhenghua Li, Tim Lindholm, Mike McCloskey, Tim Peierls, Mark Reinhold, Ken Russell, Bill Shannon, Peter Stout, Phil Wadler, and two anonymous reviewers. They made numerous suggestions that led to great improvements in this book and saved me from many embarrassments. Any remaining embarrassments are my responsibility.

Numerous colleagues, inside and outside Sun, participated in technical discussions that improved the quality of this book. Among others, Ben Gomes, Steffen Grarup, Peter Kessler, Richard Roda, John Rose, and David Stoutamire contributed useful insights. A special thanks is due Doug Lea, who served as a sounding board for many of the ideas in this book. Doug has been unfailingly generous with his time and his knowledge.

I thank Julie Dinicola, Jacqui Doucette, Mike Hendrickson, Heather Olszyk, Tracy Russ, and the whole team at Addison-Wesley for their support and profes sionalism. Even under an impossibly tight schedule, they were always friendly and accommodating.

I thank Guy Steele for writing the Foreword. I am honored that he chose to participate in this project.

Finally, I thank my wife, Cindy Bloch, for encouraging and occasionally threatening me to write this book, for reading each item in its raw form, for help ing me with Framemaker, for writing the index, and for putting up with me while I wrote.

CHAPTER 1

Introduction

THIS book is designed to help you make effective use of the Java programming language and its fundamental libraries: java.lang, java.util, and java.io, and subpackages such as java.util.concurrent and java.util.function. Other libraries are discussed from time to time.

This book consists of ninety 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 eleven chapters, each covering 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 since the last edition of this book was published. Most of the items in this book use these features in some way. This table shows you where to go for primary coverage of key features:

Feature Items Release Lambdas Items 42–44 Java 8 Streams Items

45–48 Java 8 Optionals Item 55 Java 8 Default methods in interfaces

Item 21 Java 8 try-with-resources Item 9 Java 7 @SafeVarargs Item 32

Java 7 Modules Item 15 Java 9

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 Java. If you are not, consider one of the many fine introductory texts, such as Peter Sestoft's *Java Precisely* [Sestoft16]. While *Effective Java* 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 component should never be surprised by its behavior. Components should be as small as possible but no smaller. (As used in this book, the term *component* refers to any reusable software element, from an individual method to a complex framework consisting of multiple packages.) Code should be reused rather than copied. The dependencies between components 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 67). 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 3.5GHz quad-core Intel Core i7-4770K with 16 gigabytes of DDR3-1866 CL9 RAM, running Azul's Zulu 9.0.0.15 release of OpenJDK, atop Microsoft Windows 7 Professional SP1 (64-bit).

When discussing features of the Java programming language and its libraries, it is sometimes necessary to refer to specific releases. For convenience, this book uses nicknames in preference to official release names. This table shows the map ping between release names and nicknames:

Official Release Name	Nickname
JDK 1.0.*x*	Java 1.0
JDK 1.1.*x*	Java 1.1
Java 2 Platform, Standard Edition, v1.2	Java 2
Java 2 Platform, Standard Edition, v1.3	Java 3
Java 2 Platform, Standard Edition, v1.4	Java 4

Java 2 Platform, Standard Edition, v5.0 Java 5

Java Platform, Standard Edition 6 Java 6

Java Platform, Standard Edition 7 Java 7

Java Platform, Standard Edition 8 Java 8

Java Platform, Standard Edition 9 Java 9

The examples are reasonably complete, but favor readability over complete ness. They freely use classes from packages java.util and java.io. In order to compile examples, you may have to add one or more import declarations, or other such boilerplate. The book's website, http://joshbloch.com/effectivejava, 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, Java SE 8 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 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 traditional *package-private* instead of the technically correct *package access* [JLS, 6.6.1].

This book uses a few technical terms that are not defined in *The Java Lan guage 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 *applica tion 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 collec tively 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.

In Java 9, a *module system* was added to the platform. If a library makes use of the module system, its exported API is the union of the exported APIs of all the packages exported by the library's module declaration.

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 traditional way for a class to allow a client to obtain an instance 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 `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]. 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, public 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 added in Java 4.)

A class can have only a single constructor with a given signature. Program mers 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 17) 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 illus trates this technique: it *never* creates an object. This technique is similar to the *Flyweight* pattern [Gamma95]. 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 guar antee that it is a singleton (Item 3) or noninstantiable (Item 4). Also, it allows an immutable value class (Item 17) to make the guarantee that no two equal instances exist: `a.equals(b)` if and only if `a == b`. This is the basis of the *Flyweight* pattern [Gamma95]. Enum types (Item 34) 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 20), where interfaces provide natural return types for static factory methods.

Prior to Java 8, interfaces couldn't have static methods. By convention, static factory methods for an interface named *Type* were put in a *noninstantiable com panion class* (Item 4) named *Types*. For example, the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implemen tations 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 forty-five separate public classes, one for each convenience implemen tation. It is not just the *bulk* of the API that is reduced but the *conceptual weight:* the number and difficulty of the concepts that programmers must master in order to use the API. The programmer knows that the returned object has precisely the API specified by its interface, so there is no need to read additional class docu mentation for the implementation class. Furthermore, using such a static factory method requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice (Item 64).

As of Java 8, the restriction that interfaces cannot contain static methods was eliminated, so there is typically little reason to provide a noninstantiable compan ion class for an interface. Many public static members that would have been at home in such a class should instead be put in the interface itself. Note, however, that it may still be necessary to put the bulk of the implementation code behind these static methods in a separate package-private class. This is because Java 8 requires all static members of an interface to be public. Java 9 allows private static methods, but static fields and static member classes are still required to be public.

A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters. Any sub type of the declared return type is permissible. The class of the returned object can also vary from release to release.

The `EnumSet` class (Item 36) has no public constructors, only static factories. In the OpenJDK implementation, they return an instance of one of two subclasses, 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. **A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written.** Such flexible static factory methods form the basis of *service provider frameworks*, like the Java Database Connectivity API (JDBC). A service provider framework is a system in which providers implement a service, and the system makes the implementations available to clients, decoupling the clients from the implementations. There are three essential components in a service provider framework: a *service interface*, which represents an implementation; a *provider registration API*, which providers use to register implementations; and a *service access API*, which clients use to obtain instances of the service. The service access API may allow clients to specify criteria for choosing an implementation. In the absence of such criteria, the API returns an instance of a default implementation, or allows the client to cycle through all available implementations. 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 describes a factory object that produce instances of the service interface. In the absence of a service provider interface, implementations must be instantiated reflectively (Item 65). 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 many variants of the service provider framework pattern. For exam ple, the service access API can return a richer service interface to clients than the one furnished by providers. This is the *Bridge* pattern [Gamma95]. Dependency injection frameworks (Item 5) can be viewed as powerful service providers. Since Java 6, the platform includes a general-purpose service provider framework, java.util.ServiceLoader, so you needn't, and generally shouldn't, write your own (Item 59). JDBC doesn't use ServiceLoader, as the former predates the latter.

The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed. 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 because it encourages programmers to use composition instead of inheritance (Item 18), and is required for immutable types (Item 17).

A second shortcoming of static factory methods is that they are hard for programmers to find. 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 problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

- **from**—A *type-conversion method* that takes a single parameter and returns a corresponding instance of this type, for example:

```
Date d = Date.from(instant);
```

- **of**—An *aggregation method* that takes multiple parameters and returns an instance of this type that incorporates them, for example:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf**—A more verbose alternative to from and of, for example:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- **instance** or **getInstance**—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example:

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** or **newInstance**—Like instance or getInstance, except that the method guarantees that each call returns a new instance, for example:

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **get*Type***—Like getInstance, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

```
FileStore fs = Files.getFileStore(path);
```

- **new*Type***—Like newInstance, but used if the factory method is in a different class. *Type* is the type of object returned by the factory method, for example:

```
BufferedReader br = Files.newBufferedReader(path);
```

- ***type***—A concise alternative to get*Type* and new*Type*, for example:

```
List<Complaint> litany = Collections.list(legacyLitany);
```

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 more than 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, culmi nating 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; // (per serving)
    optional private final int fat; // (g/serving) optional private final int
    sodium; // (mg/serving) optional private final int carbohydrate; //
    (g/serving) 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 51).

A second alternative when you're faced with many optional parameters in a constructor is the *JavaBeans* pattern, in which you call a parameterless construc tor 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; // Required; no default value 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 and hence difficult to debug. A related disadvantage is that **the JavaBeans pattern precludes the possibility of making a class immutable** (Item 17) 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 because 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]. 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 typically immutable. The builder is typically a static member class (Item 24) 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 sodium = 0;
        private int carbohydrate = 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 sodium(int val)
                                { sodium = val; return this; }
        public Builder carbohydrate(int val)
                            { carbohydrate = 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;
        }
    }
```


The NutritionFacts class is immutable, and all parameter default values are in one place. The builder's setter methods return the builder itself so that invocations can be chained, resulting in a *fluent API*. 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, easy to read. **The Builder pattern simulates named optional parameters** as found in Python and Scala. Validity checks were omitted for brevity. To detect invalid parameters as soon as possible, check parameter validity in the builder's constructor and methods. Check invariants involving multiple parameters in the constructor invoked by the build method. To ensure these invariants against attack, do the checks on object fields after copying parameters from the builder (Item 50). If a check fails, throw an IllegalArgumentException (Item 72) whose detail message indicates which parameters are invalid (Item 75).

The Builder pattern is well suited to class hierarchies. Use a parallel hierarchy of builders, each nested in the corresponding class. Abstract classes have abstract builders; concrete classes have concrete builders. For example, consider an abstract class at the root of a hierarchy representing various kinds of pizza:

```
// Builder pattern for class hierarchies
public abstract class Pizza {
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping) {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }

        abstract Pizza build();

        // Subclasses must override this method to return "this"
        protected abstract T self();
    }
    Pizza(Builder<?> builder) {
        toppings = builder.toppings.clone(); // See Item 50
    }
}
```

Note that Pizza.Builder is a *generic type* with a *recursive type parameter* (Item 30). This, along with the abstract self method, allows method chaining to work properly in subclasses, without the need for casts. This workaround for the

fact that Java lacks a self type is known as the *simulated self-type* idiom.

Here are two concrete subclasses of Pizza, one of which represents a standard New-York-style pizza, the other a calzone. The former has a required size parame ter, while the latter lets you specify whether sauce should be inside or out:

```
public class NyPizza extends Pizza {
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> { private final
        Size size;

        public Builder(Size size) {
                    this.size = Objects.requireNonNull(size);
        }

        @Override public NyPizza build() {
            return new NyPizza(this);
        }

                    @Override protected Builder self() { return this; }
    }

    private NyPizza(Builder builder) {
        super(builder);
        size = builder.size;
    }
}

public class Calzone extends Pizza {
    private final boolean sauceInside;

        public static class Builder extends Pizza.Builder<Builder> { private
                boolean sauceInside = false; // Default

        public Builder sauceInside() {
            sauceInside = true;
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }

                    @Override protected Builder self() { return this; }
    }

    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}
```


Note that the build method in each subclass's builder is declared to return the correct subclass: the build method of NyPizza.Builder returns NyPizza, while the

one in Calzone.Builder returns Calzone. This technique, wherein a subclass method is declared to return a subtype of the return type declared in the super class, is known as *covariant return typing*. It allows clients to use these builders without the need for casting.

The client code for these "hierarchical builders" is essentially identical to the code for the simple NutritionFacts builder. The example client code shown next assumes static imports on enum constants for brevity:

```
NyPizza pizza = new NyPizza.Builder(SMALL)
        .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
        .addTopping(HAM).sauceInside().build();
```

A minor advantage of builders over constructors is that builders can have mul tiple varargs parameters because each parameter is specified in its own method. Alternatively, builders can aggregate the parameters passed into multiple calls to a method into a single field, as demonstrated in the addTopping method earlier.

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

The Builder pattern has disadvantages as well. In order to create an object, you must first create its builder. While the cost of creating this builder is unlikely to be noticeable in practice, it could be a problem in 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 to make it worthwhile, say four or more. But keep in mind that you may want to add more parameters in the future. But if you start out with constructors or static factories and switch to a builder when the class evolves to the point where the number of parameters gets 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 many of the parameters are optional or of identical type. Client code is much easier to read and write with builders than with telescoping constructors, 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]. Single tons typically represent either a stateless object such as a function (Item 24) or a system component that is intrinsically unique. **Making a class a singleton can make it difficult to test its clients** because it's impossible to substitute a mock

implementation for a singleton unless it implements an interface that serves as its type.

There are two common 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 65) 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 earlier).

The main advantage of the public field approach is that the API makes it clear that the class is a singleton: the public static field is final, so it will always contain the same object reference. The second advantage is that it's simpler.

One advantage of the static factory approach is that it gives you the flexibility to change your mind about whether the class is a singleton without changing its API. The factory method returns the sole instance, but it could be modified to return, say, a separate instance for each thread that invokes it. A second advantage is that you can write a *generic singleton factory* if your application requires it (Item 30). A final advantage of using a static factory is that a *method reference* can be used as a supplier, for example Elvis::instance is a Supplier<Elvis>. Unless one of these advantages is relevant, the public field approach is preferable.

To make a singleton class that uses either of these approaches *serializable* (Chapter 12), it is not sufficient merely to add implements Serializable to its

declaration. To maintain the singleton guarantee, declare all instance fields transient and provide a readResolve method (Item 89). 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 from happening, 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;
}
```

A third way to implement a singleton is to declare a single-element enum:

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

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

This approach is similar to the public field approach, but 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. This approach may feel a bit unnatural, but **a single-element enum type is often the best way to implement a singleton**. Note that you can't use this approach if your singleton must extend a superclass other than Enum (though you *can* declare an enum to implement interfaces).

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 factories (Item 1), for objects that implement some interface, in the manner of java.util.Collections. (As of Java 8, you can also put such methods *in* the interface, assuming it's yours to modify.) Lastly, such classes can be used to group methods on a final class, since you can't put them in a subclass.

Such *utility classes* were not designed to be instantiated: an instance would be nonsensical. In the absence of explicit constructors, however, the compiler pro vides 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 19). There is, however, a simple idiom to ensure noninstantiability. A default construc tor 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 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 the class will never be instantiated under any circumstances. This idiom is mildly counter intuitive because the constructor is provided expressly so that it cannot be invoked. It is therefore wise to include a comment, as shown earlier.

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: Prefer dependency injection to hardwiring resources

Many classes depend on one or more underlying resources. For example, a spell checker depends on a dictionary. It is not uncommon to see such classes imple mented as static utility classes (Item 4):

```
// Inappropriate use of static utility - inflexible & untestable! public
class SpellChecker {
    private static final Lexicon dictionary = ...;

    private SpellChecker() {} // Noninstantiable

    public static boolean isValid(String word) { ... }
    public static List<String> suggestions(String typo) { ... } }
```

Similarly, it's not uncommon to see them implemented as singletons (Item 3):

```
// Inappropriate use of singleton - inflexible & untestable! public
class SpellChecker {
    private final Lexicon dictionary = ...;

    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);

    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Neither of these approaches is satisfactory, because they assume that there is

only one dictionary worth using. In practice, each language has its own dictionary, and special dictionaries are used for special vocabularies. Also, it may be desirable to use a special dictionary for testing. It is wishful thinking to assume that a single dictionary will suffice for all time.

You could try to have SpellChecker support multiple dictionaries by making the dictionary field nonfinal and adding a method to change the dictionary in an existing spell checker, but this would be awkward, error-prone, and unworkable in a concurrent setting. **Static utility classes and singletons are inappropriate for classes whose behavior is parameterized by an underlying resource.**

What is required is the ability to support multiple instances of the class (in our example, SpellChecker), each of which uses the resource desired by the client (in our example, the dictionary). A simple pattern that satisfies this requirement is to **pass the resource into the constructor when creating a new instance**. This is one form of *dependency injection*: the dictionary is a *dependency* of the spell checker and is *injected* into the spell checker when it is created.


```
// Dependency injection provides flexibility and testability public
class SpellChecker {
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary) {
                this.dictionary = Objects.requireNonNull(dictionary);
    }

    public boolean isValid(String word) { ... }
                public List<String> suggestions(String typo) { ... }
}
```

The dependency injection pattern is so simple that many programmers use it for years without knowing it has a name. While our spell checker example had only a single resource (the dictionary), dependency injection works with an arbitrary number of resources and arbitrary dependency graphs. It preserves immutability (Item 17), so multiple clients can share dependent objects (assuming the clients desire the same underlying resources). Dependency injection is equally applicable to constructors, static factories (Item 1), and builders (Item 2).

A useful variant of the pattern is to pass a resource *factory* to the constructor. A factory is an object that can be called repeatedly to create instances of a type. Such factories embody the *Factory Method* pattern [Gamma95]. The Supplier<T> interface, introduced in Java 8, is perfect for representing factories. Methods that take a Supplier<T> on input should typically constrain the factory's type parameter using a *bounded wildcard type* (Item 31) to allow the client to pass in a factory that creates any subtype of a specified type. For example, here is a method that makes a mosaic using a client-provided factory to produce each tile:

```
Mosaic create(Supplier<? extends Tile> tileFactory) { ... }
```

Although dependency injection greatly improves flexibility and testability, it can clutter up large projects, which typically contain thousands of dependencies.

This clutter can be all but eliminated by using a *dependency injection framework*, such as Dagger [Dagger], Guice [Guice], or Spring [Spring]. The use of these frameworks is beyond the scope of this book, but note that APIs designed for manual dependency injection are trivially adapted for use by these frameworks.

In summary, do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

Item 6: Avoid creating unnecessary objects

It is often appropriate to reuse a single object instead of creating a new function ally 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 17). As an extreme example of what not to do, consider this statement:

```
String s = new String("bikini"); // 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 (`"bikini"`) 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 = "bikini";
```

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 meth ods* (Item 1) in preference to constructors on immutable classes that provide both. For example, the factory method `Boolean.valueOf(String)` is preferable to the constructor `Boolean(String)`, which was deprecated in Java 9. The constructor *must* create a new object each time it's called, while the 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.

Some object creations are much more expensive than others. If you're going to need such an "expensive object" repeatedly, it may be advisable to cache it for reuse. Unfortunately, it's not always obvious when you're creating such an object. Suppose you want to write a method to determine whether a string is a valid Roman numeral. Here's the easiest way to do this using a regular expression:

```
// Performance can be greatly improved!
static boolean isRomanNumeral(String s) {
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```


The problem with this implementation is that it relies on the `String.matches` method. **While `String.matches` is the easiest way to check if a string matches a regular expression, it's not suitable for repeated use in performance-critical situations.** The problem is that it internally creates a `Pattern` instance for the regular expression and uses it only once, after which it becomes eligible for garbage collection. Creating a `Pattern` instance is expensive because it requires compiling the regular expression into a finite state machine.

To improve the performance, explicitly compile the regular expression into a `Pattern` instance (which is immutable) as part of class initialization, cache it, and reuse the same instance for every invocation of the `isRomanNumeral` method:

```
// Reusing expensive object for improved performance
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
        "^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    static boolean isRomanNumeral(String s) {
        return ROMAN.matcher(s).matches();
    }
}
```

The improved version of `isRomanNumeral` provides significant performance gains if invoked frequently. On my machine, the original version takes 1.1 μs on an 8-character input string, while the improved version takes 0.17 μs, which is 6.5 times faster. Not only is the performance improved, but arguably, so is clarity. Making a static final field for the otherwise invisible `Pattern` instance allows us to give it a name, which is far more readable than the regular expression itself.

If the class containing the improved version of the `isRomanNumeral` method is initialized but the method is never invoked, the field `ROMAN` will be initialized needlessly. It would be possible to eliminate the initialization by *lazily initializing* the field (Item 83) the first time the `isRomanNumeral` method is invoked, but this is *not* recommended. As is often the case with lazy initialization, it would compli cate the implementation with no measurable performance improvement (Item 67).

When an object is immutable, it is obvious it can be reused safely, but there are other situations where it is far less obvious, even counterintuitive. Consider the case of *adapters* [Gamma95], also known as *views*. An adapter is an object that delegates to a backing object, providing an alternative interface. 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 largely harmless to create multiple instances of the keySet view object, it is unnecessary and has no benefits.

Another way to create unnecessary objects is *autoboxing*, which 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 distinc tions and not-so-subtle performance differences (Item 61). Consider the following method, 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! Can you spot the object creation?
private static long sum() {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;

    return 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 6.3 seconds to 0.59 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 expen sive 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, simplic ity, 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. 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 out perform such object pools on lightweight objects.

The counterpoint to this item is Item 50 on *defensive copying*. The present item says, "Don't create a new object when you should reuse an existing one," while Item 50 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 7: Eliminate obsolete object references

If you switched from a language with manual memory management, such as C or C++, to a garbage-collected language such as Java, your job as a programmer was 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 29 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 main tains *obsolete references* to these objects. An obsolete reference is simply a refer ence 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 *unin tentional 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 refer ences 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 subse quently 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; 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 57).

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 refer ence 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 pro grammer 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 pro grammer 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 `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. One 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 8: Avoid finalizers and cleaners

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, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. The Java 9 replacement for finalizers is *cleaners*. **Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.**

C++ programmers are cautioned not to think of finalizers or cleaners as Java's analogue 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, a `try`-with-resources or `try-finally` block is used for this purpose (Item 9).

One shortcoming of finalizers and cleaners 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 its finalizer or cleaner runs. This means that you should **never do anything time-critical in a finalizer or cleaner.** For example, it is a grave error to depend on a finalizer or cleaner to close files because open file descriptors are a limited resource. If many files are left open as a result of the system's tardiness in running finalizers or cleaners, a program may fail because it can no longer open files.

The promptness with which finalizers and cleaners are executed is primarily a function of the garbage collection algorithm, which varies widely across imple mentations. The behavior of a program that depends on the promptness of finalizer or cleaner 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 one favored by your most important customer.

Tardy finalization is not just a theoretical problem. Providing a finalizer for a class can 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 guar-

antees 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. Cleaners are a bit better than finalizers in this regard because class authors have control over their own cleaner threads, but cleaners still run in the background, under the control of the garbage collector, so there can be no guarantee of prompt cleaning.

Not only does the specification provide no guarantee that finalizers or cleaners will run promptly; it provides no guarantee that they'll run at all. It is entirely possible, even likely, that a program terminates without running them on

some objects that are no longer reachable. As a consequence, you should **never depend on a finalizer or cleaner to update persistent state.** For example, depending on a finalizer or cleaner 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 or cleaners getting executed, but they don't guarantee it. Two methods once claimed to make this guarantee: `System.runFinalizersOnExit` and its evil twin, `Runtime.runFinalizersOnExit`. These methods are fatally flawed and have been deprecated for decades [ThreadStop].

Another problem with finalizers is that an uncaught exception thrown during finalization is ignored, and finalization of that object terminates [JLS, 12.6]. Uncaught exceptions can leave other 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. Cleaners do not have this problem because a library using a cleaner has control over its thread.

There is a *severe* performance penalty for using finalizers and cleaners. On my machine, the time to create a simple `AutoCloseable` object, to close it using `try`-with-resources, and to have the garbage collector reclaim it is about 12 ns. Using a finalizer instead increases the time to 550 ns. In other words, it is about 50 times slower to create and destroy objects with finalizers. This is primarily because finalizers inhibit efficient garbage collection. Cleaners are comparable in speed to finalizers if you use them to clean all instances of the class (about 500 ns per instance on my machine), but cleaners are much faster if you use them only as a safety net, as discussed below. Under these circumstances, creating, cleaning, and destroying an object takes about 66 ns on my machine, which means you pay a factor of five (not fifty) for the insurance of a safety net *if* you don't use it.

Finalizers have a serious security problem: they open your class up to *finalizer attacks*. The idea behind a finalizer attack is simple: If an exception is

thrown from a constructor or its serialization equivalents—the `readObject` and `readResolve` methods (Chapter 12)—the finalizer of a malicious subclass can run on the partially constructed object that should have "died on the vine." This finalizer can record a reference to the object in a static field, preventing it from being garbage collected. Once the malformed object has been recorded, it is a simple matter to invoke arbitrary methods on this object that should never have been allowed to exist in the first place. **Throwing an exception from a constructor should be sufficient to prevent an object from coming into existence; in the presence of finalizers, it is not.** Such attacks can have dire consequences. Final classes are immune to finalizer attacks because no one can write a malicious subclass of a final class. **To protect nonfinal classes from finalizer attacks, write a final `finalize` method that does nothing.**

So what should you do instead of writing a finalizer or cleaner for a class whose objects encapsulate resources that require termination, such as files or threads? Just **have your class implement AutoCloseable,** and require its clients to invoke the close method on each instance when it is no longer needed, typically using try-with-resources to ensure termination even in the face of exceptions (Item 9). One detail worth mentioning is that the instance must keep track of whether it has been closed: the close method must record in a 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 closed.

So what, if anything, are cleaners and finalizers good for? They have perhaps two legitimate uses. One is to act as a safety net in case the owner of a resource neglects to call its close method. While there's no guarantee that the cleaner or finalizer will run promptly (or at all), it is better to free the resource late than never if the client fails to do so. If you're considering writing such a safety-net finalizer, think long and hard about whether the protection is worth the cost. Some Java library classes, such as FileInputStream, FileOutputStream, ThreadPoolExecutor, and java.sql.Connection, have finalizers that serve as safety nets.

A second legitimate use of cleaners concerns objects with *native peers*. A native peer is a native (non-Java) 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 cleaner or finalizer may be an appropriate vehicle for this task, assuming the performance is acceptable and the native peer holds no critical resources. If the performance is unacceptable or the native peer holds resources that must be reclaimed promptly, the class should have a close method, as described earlier.

Cleaners are a bit tricky to use. Below is a simple Room class demonstrating the facility. Let's assume that rooms must be cleaned before they are reclaimed. The Room class implements AutoCloseable; the fact that its automatic cleaning safety net uses a cleaner is merely an implementation detail. Unlike finalizers, cleaners do not pollute a class's public API:

```
// An autocloseable class using a cleaner as a safety net public
class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();

    // Resource that requires cleaning. Must not refer to Room! private
    static class State implements Runnable {
                int numJunkPiles; // Number of junk piles in this room

        State(int numJunkPiles) {
            this.numJunkPiles = numJunkPiles;
        }

        // Invoked by close method or cleaner
        @Override public void run() {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
```

```
        }
    }

    // The state of this room, shared with our cleanable
    private final State state;

    // Our cleanable. Cleans the room when it's eligible for gc private final
    Cleaner.Cleanable cleanable;

    public Room(int numJunkPiles) {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }

    @Override public void close() {
        cleanable.clean();
    }
}
```

The static nested State class holds the resources that are required by the cleaner to clean the room. In this case, it is simply the numJunkPiles field, which represents the amount of mess in the room. More realistically, it might be a final long that contains a pointer to a native peer. State implements Runnable, and its run method is called at most once, by the Cleanable that we get when we register our State instance with our cleaner in the Room constructor. The call to the run method will be triggered by one of two things: Usually it is triggered by a call to

Room's close method calling Cleanable's clean method. If the client fails to call the close method by the time a Room instance is eligible for garbage collection, the cleaner will (hopefully) call State's run method.

It is critical that a State instance does not refer to its Room instance. If it did, it would create a circularity that would prevent the Room instance from becoming eligible for garbage collection (and from being automatically cleaned). Therefore, State must be a *static* nested class because nonstatic nested classes contain references to their enclosing instances (Item 24). It is similarly inadvisable to use a lambda because they can easily capture references to enclosing objects.

As we said earlier, Room's cleaner is used only as a safety net. If clients surround all Room instantiations in try-with-resource blocks, automatic cleaning will never be required. This well-behaved client demonstrates that behavior:

```
public class Adult {
 public static void main(String[] args) {
 try (Room myRoom = new Room(7)) {
 System.out.println("Goodbye");
 }
 }
}
```

As you'd expect, running the Adult program prints Goodbye, followed by Clean ing room. But what about this ill-behaved program, which never cleans its room?

```
public class Teenager {
 public static void main(String[] args) {
```

```
        new Room(99);
        System.out.println("Peace out");
        }
    }
```

You might expect it to print `Peace out`, followed by `Cleaning room`, but on my machine, it never prints `Cleaning room`; it just exits. This is the unpredictability we spoke of earlier. The `Cleaner` spec says, "The behavior of cleaners during `System.exit` is implementation specific. No guarantees are made relating to whether cleaning actions are invoked or not." While the spec does not say it, the same holds true for normal program exit. On my machine, adding the line `System.gc()` to `Teenager`'s `main` method is enough to make it print `Cleaning room` prior to exit, but there's no guarantee that you'll see the same behavior on your machine.

In summary, don't use cleaners, or in releases prior to Java 9, finalizers, except as a safety net or to terminate noncritical native resources. Even then, beware the indeterminacy and performance consequences.

Item 9: Prefer try-with-resources to try-finally

The Java libraries include many resources that must be closed manually by invoking a `close` method. Examples include `InputStream`, `OutputStream`, and `java.sql.Connection`. Closing resources is often overlooked by clients, with predictably dire performance consequences. While many of these resources use finalizers as a safety net, finalizers don't work very well (Item 8).

Historically, a `try-finally` statement was the best way to guarantee that a resource would be closed properly, even in the face of an exception or return:

```
// try-finally - No longer the best way to close resources! static String
firstLineOfFile(String path) throws IOException { BufferedReader br = new
BufferedReader(new FileReader(path)); try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

This may not look bad, but it gets worse when you add a second resource:

```
// try-finally is ugly when used with more than one resource! static
void copy(String src, String dst) throws IOException { InputStream in = new
FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dst);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        } finally {
            out.close();
        }
```

```
        } finally {
            in.close();
        }
    }
```

It may be hard to believe, but even good programmers got this wrong most of the time. For starters, I got it wrong on page 88 of *Java Puzzlers* [Bloch05], and no one noticed for years. In fact, two-thirds of the uses of the close method in the Java libraries were wrong in 2007.

Even the correct code for closing resources with try-finally statements, as illustrated in the previous two code examples, has a subtle deficiency. The code in both the try block and the finally block is capable of throwing exceptions. For example, in the firstLineOfFile method, the call to readLine could throw an exception due to a failure in the underlying physical device, and the call to close could then fail for the same reason. Under these circumstances, the second exception completely obliterates the first one. There is no record of the first exception in the exception stack trace, which can greatly complicate debugging in real systems—usually it's the first exception that you want to see in order to diagnose the problem. While it is possible to write code to suppress the second exception in favor of the first, virtually no one did because it's just too verbose.

All of these problems were solved in one fell swoop when Java 7 introduced the try-with-resources statement [JLS, 14.20.3]. To be usable with this construct, a resource must implement the AutoCloseable interface, which consists of a single void-returning close method. Many classes and interfaces in the Java libraries and in third-party libraries now implement or extend AutoCloseable. If you write a class that represents a resource that must be closed, your class should implement AutoCloseable too.

Here's how our first example looks using try-with-resources:

```
// try-with-resources - the the best way to close resources! static
String firstLineOfFile(String path) throws IOException { try (BufferedReader br
= new BufferedReader(
                    new FileReader(path))) {
            return br.readLine();
        }
    }
```

And here's how our second example looks using try-with-resources:

```
// try-with-resources on multiple resources - short and sweet static
void copy(String src, String dst) throws IOException { try (InputStream in =
                            new FileInputStream(src);
                        OutputStream out = new FileOutputStream(dst)) {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        }
    }
```

Not only are the try-with-resources versions shorter and more readable than the originals, but they provide far better diagnostics. Consider the firstLineOfFile method. If exceptions are thrown by both the readLine call and the (invisible) close, the latter exception is *suppressed* in favor of the former. In fact, multiple exceptions may be suppressed in order to preserve the exception that you actually want to see. These suppressed exceptions are not merely discarded; they are printed in the stack trace with a notation saying that they were suppressed. You can also access them programmatically with the getSuppressed method, which was added to Throwable in Java 7.

You can put catch clauses on try-with-resources statements, just as you can on regular try-finally statements. This allows you to handle exceptions without sullying your code with another layer of nesting. As a slightly contrived example, here's a version our firstLineOfFile method that does not throw exceptions, but takes a default value to return if it can't open the file or read from it:

```
// try-with-resources with a catch clause
static String firstLineOfFile(String path, String defaultVal) { try
      (BufferedReader br = new BufferedReader(
                  new FileReader(path))) {
            return br.readLine();
      } catch (IOException e) {
            return defaultVal;
      }
}
```

The lesson is clear: Always use try-with-resources in preference to try finally when working with resources that must be closed. The resulting code is shorter and clearer, and the exceptions that it generates are more useful. The try with-resources statement makes it easy to write correct code using resources that must be closed, which was practically impossible using try-finally.

CHAPTER 3

Methods Common to All Objects

ALTHOUGH Object is a concrete class, it is designed primarily for extension. All of its nonfinal methods (equals, hashCode, toString, clone, and finalize) have explicit *general contracts* because they are designed to be overridden. It is the responsibility of any class overriding these methods to obey their general con tracts; failure to do so will prevent other classes that depend on the contracts (such as HashMap and HashSet) from functioning properly in conjunction with the class.

This chapter tells you when and how to override the nonfinal Object methods. The finalize method is omitted from this chapter because it was discussed in Item 8. While not an Object method, Comparable.compareTo is discussed in this chapter because it has a similar character.

Item 10: Obey the general contract when overriding equals

Overriding the equals method seems simple, but there are many ways to get it wrong, and consequences can be dire. The easiest way to avoid problems is not to override the equals method, in which case each instance of the class is equal only to itself. This is the right thing to do if any of the following conditions apply:

- **Each instance of the class is inherently unique.** This is true for classes such as Thread that represent active entities rather than values. The equals imple mentation provided by Object has exactly the right behavior for these classes.
- **There is no need for the class to provide a "logical equality" test.** For example, java.util.regex.Pattern could have overridden equals to check whether two Pattern instances represented exactly the same regular expression, but the designers didn't think that clients would need or want this functionality. Under these circumstances, the equals implementation inherited from Object is ideal.

- **A superclass has already overridden equals, and the superclass behavior is appropriate for this class.** For example, most Set implementations inherit their equals implementation from AbstractSet, List implementations from AbstractList, and Map implementations from AbstractMap.
- **The class is private or package-private, and you are certain that its equals method will never be invoked.** If you are extremely risk-averse, you can over ride the equals method to ensure that it isn't invoked accidentally:

```
@Override public boolean equals(Object o) {
        throw new AssertionError(); // Method is never called
```

```
}
```

So when is it appropriate to override equals? It is when a class has a notion of *logical equality* that differs from mere object identity and a superclass has not already overridden equals. This is generally the case for *value classes.* A value class is simply a class that represents a value, such as Integer or String. A programmer who compares references to value objects using the equals method expects to find out whether they are logically equivalent, not whether they refer to the same object. Not only is overriding the equals method necessary to satisfy programmer expectations, it enables instances to serve as map keys or set elements with predictable, desirable behavior.

One kind of value class that does *not* require the equals method to be overrid den is a class that uses instance control (Item 1) to ensure that at most one object exists with each value. Enum types (Item 34) fall into this category. For these classes, logical equality is the same as object identity, so Object's equals method functions as a logical equals method.

When you override the equals method, you must adhere to its general con tract. Here is the contract, from the specification for Object :

The equals method implements an *equivalence relation.* It has these properties:

- *Reflexive*: For any non-null reference value x, x.equals(x) must return true.
- *Symmetric*: For any non-null reference values x and y, x.equals(y) must re turn true if and only if y.equals(x) returns true.
- *Transitive*: For any non-null reference values x, y, z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
- *Consistent*: For any non-null reference values x and y, multiple invocations of x.equals(y) must consistently return true or consistently return false, provided no information used in equals comparisons is modified.
- For any non-null reference value x, x.equals(null) must return false.

Unless you are mathematically inclined, this might look a bit scary, but do not ignore it! If you violate it, you may well find that your program behaves erratically or crashes, and it can be very difficult to pin down the source of the failure. To paraphrase John Donne, no class is an island. Instances of one class are frequently passed to another. Many classes, including all collections classes, depend on the objects passed to them obeying the equals contract.

Now that you are aware of the dangers of violating the equals contract, let's go over the contract in detail. The good news is that, appearances notwithstanding, it really isn't very complicated. Once you understand it, it's not hard to adhere to it.

So what is an equivalence relation? Loosely speaking, it's an operator that partitions a set of elements into subsets whose elements are deemed equal to one another. These subsets are known as *equivalence classes*. For an equals method to be useful, all of the elements in each equivalence class must be interchangeable from the perspective of the user. Now let's examine the five requirements in turn:

Reflexivity—The first requirement says merely that an object must be equal to itself. It's hard to imagine violating this one unintentionally. If you were to violate it and then add an instance of your class to a collection, the contains method might well say that the collection didn't contain the instance that you just added.

Symmetry—The second requirement says that any two objects must agree on whether they are equal. Unlike the first requirement, it's not hard to imagine violating this one unintentionally. For example, consider the following class, which implements a case-insensitive string. The case of the string is preserved by toString but ignored in equals comparisons:

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        this.s = Objects.requireNonNull(s);
    }

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


The well-intentioned equals method in this class naively attempts to interoperate with ordinary strings. Let's suppose that we have one case-insensitive string and one ordinary one:

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

As expected, cis.equals(s) returns true. The problem is that while the equals method in CaseInsensitiveString knows about ordinary strings, the equals method in String is oblivious to case-insensitive strings. Therefore, s.equals(cis) returns false, a clear violation of symmetry. Suppose you put a case-insensitive string into a collection:

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

What does list.contains(s) return at this point? Who knows? In the current OpenJDK implementation, it happens to return false, but that's just an implementation artifact. In another implementation, it could just as easily return true or throw a runtime exception. **Once you've violated the equals contract, you simply don't know how other objects will behave when confronted with**

your object.

To eliminate the problem, merely remove the ill-conceived attempt to interop erate with `String` from the `equals` method. Once you do this, you can refactor the method into a single return statement:

```
@Override public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
                    ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

Transitivity—The third requirement of the `equals` contract says that if one object is equal to a second and the second object is equal to a third, then the first object must be equal to the third. Again, it's not hard to imagine violating this requirement unintentionally. Consider the case of a subclass that adds a new *value component* to its superclass. In other words, the subclass adds a piece of

information that affects `equals` comparisons. Let's start with a simple immutable two-dimensional integer point class:

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }

    ... // Remainder omitted
}
```

Suppose you want to extend this class, adding the notion of color to a point:

```
public class ColorPoint extends Point {
    private final Color color;

        public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // Remainder omitted
}
```

How should the `equals` method look? If you leave it out entirely, the imple mentation is inherited from `Point` and color information is ignored in `equals`

comparisons. While this does not violate the equals contract, it is clearly unac ceptable. Suppose you write an equals method that returns true only if its argu ment is another color point with the same position and color:

```
// Broken - violates symmetry!
@Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color; }
```


The problem with this method is that you might get different results when com paring a point to a color point and vice versa. The former comparison ignores color, while the latter comparison always returns false because the type of the argument is incorrect. To make this concrete, let's create one point and one color point:

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

Then p.equals(cp) returns true, while cp.equals(p) returns false. You might try to fix the problem by having ColorPoint.equals ignore color when doing "mixed comparisons":

```
// Broken - violates transitivity!
@Override public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison if (!(o
    instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    return super.equals(o) && ((ColorPoint) o).color == color; }
```

This approach does provide symmetry, but at the expense of transitivity:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Now p1.equals(p2) and p2.equals(p3) return true, while p1.equals(p3) returns false, a clear violation of transitivity. The first two comparisons are "color-blind," while the third takes color into account.

Also, this approach can cause infinite recursion: Suppose there are two subclasses of Point, say ColorPoint and SmellPoint, each with this sort of equals method. Then a call to myColorPoint.equals(mySmellPoint) will throw a StackOverflowError.

So what's the solution? It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. **There is no way to extend an instantiable class and add a value component while preserving the equals**

contract, unless you're willing to forgo the benefits of object-oriented abstraction.

You may hear it said that you can extend an instantiable class and add a value component while preserving the equals contract by using a getClass test in place of the instanceof test in the equals method:

```
// Broken - violates Liskov substitution principle (page 43)
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

This has the effect of equating objects only if they have the same implementation class. This may not seem so bad, but the consequences are unacceptable: An instance of a subclass of Point is still a Point, and it still needs to function as one, but it fails to do so if you take this approach! Let's suppose we want to write a method to tell whether a point is on the unit circle. Here is one way we could do it:

```
// Initialize unitCircle to contain all Points on the unit circle private
static final Set<Point> unitCircle = Set.of(
        new Point( 1, 0), new Point( 0, 1),
        new Point(-1, 0), new Point( 0, -1));

public static boolean onUnitCircle(Point p) {
    return unitCircle.contains(p);
}
```

While this may not be the fastest way to implement the functionality, it works fine. Suppose you extend Point in some trivial way that doesn't add a value component, say, by having its constructor keep track of how many instances have been created:

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
            new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated() { return counter.get(); } }
```

The *Liskov substitution principle* says that any important property of a type should also hold for all its subtypes so that any method written for the type should work equally well on its subtypes [Liskov87]. This is the formal statement of our

earlier claim that a subclass of Point (such as CounterPoint) is still a Point and must act as one. But suppose we pass a CounterPoint to the onUnitCircle method. If the Point class uses a getClass-based equals method, the onUnitCircle method will return false regardless of the CounterPoint instance's *x* and *y* coordinates. This is so because most collections, including the HashSet used by the onUnitCircle method, use the equals method to test for containment, and no CounterPoint instance is equal to any Point. If, however, you use a proper instanceof-based equals method on Point, the same onUnitCircle method works fine when presented with a CounterPoint instance.

While there is no satisfactory way to extend an instantiable class and add a value component, there is a fine workaround: Follow the advice of Item 18, "Favor composition over inheritance." Instead of having ColorPoint extend Point, give ColorPoint a private Point field and a public *view* method (Item 6) that returns the point at the same position as this color point:

```
// Adds a value component without violating the equals contract
public class ColorPoint {
    private final Point point;
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = Objects.requireNonNull(color);
    }

    /**
     * Returns the point-view of this color point.
     */
    public Point asPoint() {
        return point;
    }

    @Override public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint) o;
        return cp.point.equals(point) && cp.color.equals(color); }

    ... // Remainder omitted
}
```

There are some classes in the Java platform libraries that do extend an instantiable class and add a value component. For example, java.sql.Timestamp

extends java.util.Date and adds a nanoseconds field. The equals implementation for Timestamp does violate symmetry and can cause erratic behavior if Timestamp and Date objects are used in the same collection or are otherwise intermixed. The Timestamp class has a disclaimer cautioning programmers against mixing dates and timestamps. While you won't get into trouble as long as you

keep them separate, there's nothing to prevent you from mixing them, and the resulting errors can be hard to debug. This behavior of the Timestamp class was a mistake and should not be emulated.

Note that you *can* add a value component to a subclass of an *abstract* class without violating the equals contract. This is important for the sort of class hier archies that you get by following the advice in Item 23, "Prefer class hierarchies to tagged classes." For example, you could have an abstract class Shape with no value components, a subclass Circle that adds a radius field, and a subclass Rectangle that adds length and width fields. Problems of the sort shown earlier won't occur so long as it is impossible to create a superclass instance directly.

Consistency—The fourth requirement of the equals contract says that if two objects are equal, they must remain equal for all time unless one (or both) of them is modified. In other words, mutable objects can be equal to different objects at different times while immutable objects can't. When you write a class, think hard about whether it should be immutable (Item 17). If you conclude that it should, make sure that your equals method enforces the restriction that equal objects remain equal and unequal objects remain unequal for all time.

Whether or not a class is immutable, **do not write an equals method that depends on unreliable resources.** It's extremely difficult to satisfy the consis tency requirement if you violate this prohibition. For example, java.net.URL's equals method relies on comparison of the IP addresses of the hosts associated with the URLs. Translating a host name to an IP address can require network access, and it isn't guaranteed to yield the same results over time. This can cause the URL equals method to violate the equals contract and has caused problems in practice. The behavior of URL's equals method was a big mistake and should not be emulated. Unfortunately, it cannot be changed due to compatibility require ments. To avoid this sort of problem, equals methods should perform only deter ministic computations on memory-resident objects.

Non-nullity—The final requirement lacks an official name, so I have taken the liberty of calling it "non-nullity." It says that all objects must be unequal to null. While it is hard to imagine accidentally returning true in response to the invocation o.equals(null), it isn't hard to imagine accidentally throwing a

NullPointerException. The general contract prohibits this. Many classes have equals methods that guard against it with an explicit test for null:

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

This test is unnecessary. To test its argument for equality, the equals method must first cast its argument to an appropriate type so its accessors can be invoked or its fields accessed. Before doing the cast, the method must use the instanceof oper ator to check that its argument is of the correct type:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
}
```

If this type check were missing and the equals method were passed an argument of the wrong type, the equals method would throw a ClassCastException, which violates the equals contract. But the instanceof operator is specified to return false if its first operand is null, regardless of what type appears in the second operand [JLS, 15.20.2]. Therefore, the type check will return false if null is passed in, so you don't need an explicit null check.

Putting it all together, here's a recipe for a high-quality equals method:

1. **Use the == operator to check if the argument is a reference to this object.** If so, return true. This is just a performance optimization but one that is worth doing if the comparison is potentially expensive.
2. **Use the instanceof operator to check if the argument has the correct type.** If not, return false. Typically, the correct type is the class in which the method occurs. Occasionally, it is some interface implemented by this class. Use an interface if the class implements an interface that refines the equals contract to permit comparisons across classes that implement the interface. Collection interfaces such as Set, List, Map, and Map.Entry have this property.
3. **Cast the argument to the correct type.** Because this cast was preceded by an instanceof test, it is guaranteed to succeed.

4. **For each "significant" field in the class, check if that field of the argument matches the corresponding field of this object.** If all these tests succeed, re turn true; otherwise, return false. If the type in Step 2 is an interface, you must access the argument's fields via interface methods; if the type is a class, you may be able to access the fields directly, depending on their accessibility.

   For primitive fields whose type is not float or double, use the == operator for comparisons; for object reference fields, call the equals method recursively; for float fields, use the static Float.compare(float, float) method; and for double fields, use Double.compare(double, double). The special treat ment of float and double fields is made necessary by the existence of Float.NaN, -0.0f and the analogous double values; see JLS 15.21.1 or the documentation of Float.equals for details. While you could compare float and double fields with the static methods Float.equals and Double.equals, this would entail autoboxing on every comparison, which would have poor performance. For array fields, apply these guidelines to each element. If every element in an array field is significant, use one of the Arrays.equals methods.

   Some object reference fields may legitimately contain null. To avoid the

possibility of a NullPointerException, check such fields for equality using the static method Objects.equals(Object, Object).

For some classes, such as CaseInsensitiveString above, field comparisons are more complex than simple equality tests. If this is the case, you may want to store a *canonical form* of the field so the equals method can do a cheap ex act comparison on canonical forms rather than a more costly nonstandard com parison. This technique is most appropriate for immutable classes (Item 17); if the object can change, you must keep the canonical form up to date.

The performance of the equals method may be affected by the order in which fields are compared. For best performance, you should first compare fields that are more likely to differ, less expensive to compare, or, ideally, both. You must not compare fields that are not part of an object's logical state, such as lock fields used to synchronize operations. You need not compare *derived fields*, which can be calculated from "significant fields," but doing so may improve the performance of the equals method. If a derived field amounts to a summa ry description of the entire object, comparing this field will save you the ex pense of comparing the actual data if the comparison fails. For example, suppose you have a Polygon class, and you cache the area. If two polygons have unequal areas, you needn't bother comparing their edges and vertices.

When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent? And don't just ask yourself; write unit tests to check, unless you used AutoValue (page 49) to gener ate your equals method, in which case you can safely omit the tests. If the proper ties fail to hold, figure out why, and modify the equals method accordingly. Of course your equals method must also satisfy the other two properties (reflexivity and non-nullity), but these two usually take care of themselves.

An equals method constructed according to the previous recipe is shown in this simplistic PhoneNumber class:

```
// Class with a typical equals method
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix = rangeCheck(prefix, 999, "prefix");
                this.lineNum = rangeCheck(lineNum, 9999, "line num");
    }

    private static short rangeCheck(int val, int max, String arg) { if (val < 0 ||
        val > max)
                    throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
```

```
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
                return pn.lineNum == lineNum && pn.prefix == prefix
                 && pn.areaCode == areaCode;
    }
    ... // Remainder omitted
}
```

Here are a few final caveats:

- **Always override `hashCode` when you override `equals`** (Item 11).
- **Don't try to be too clever.** If you simply test fields for equality, it's not hard to adhere to the `equals` contract. If you are overly aggressive in searching for equivalence, it's easy to get into trouble. It is generally a bad idea to take any form of aliasing into account. For example, the `File` class shouldn't attempt to equate symbolic links referring to the same file. Thankfully, it doesn't.

- **Don't substitute another type for `Object` in the `equals` declaration.** It is not uncommon for a programmer to write an `equals` method that looks like this and then spend hours puzzling over why it doesn't work properly:

```
// Broken - parameter type must be Object!
public boolean equals(MyClass o) {
    ...
}
```

The problem is that this method does not *override* `Object.equals`, whose argument is of type `Object`, but *overloads* it instead (Item 52). It is unacceptable to provide such a "strongly typed" `equals` method even in addition to the normal one, because it can cause `Override` annotations in subclasses to generate false positives and provide a false sense of security.

Consistent use of the `Override` annotation, as illustrated throughout this item, will prevent you from making this mistake (Item 40). This `equals` method won't compile, and the error message will tell you exactly what is wrong:

```
// Still broken, but won't compile
@Override public boolean equals(MyClass o) {
    ...
}
```

Writing and testing `equals` (and `hashCode`) methods is tedious, and the result ing code is mundane. An excellent alternative to writing and testing these methods manually is to use Google's open source AutoValue framework, which automati cally generates these methods for you, triggered by a single annotation on the class . In most cases, the methods generated by AutoValue are essentially identical to those you'd write yourself.

IDEs, too, have facilities to generate `equals` and `hashCode` methods, but the resulting source code is more verbose and less readable than code that uses

AutoValue, does not track changes in the class automatically, and therefore requires testing. That said, having IDEs generate equals (and hashCode) methods is generally preferable to implementing them manually because IDEs do not make careless mistakes, and humans do.

In summary, don't override the equals method unless you have to: in many cases, the implementation inherited from Object does exactly what you want. If you do override equals, make sure to compare all of the class's significant fields and to compare them in a manner that preserves all five provisions of the equals contract.

Item 11: Always override hashCode when you override equals

You must override hashCode in every class that overrides equals. If you fail to do so, your class will violate the general contract for hashCode, which will prevent it from functioning properly in collections such as HashMap and HashSet. Here is the contract, adapted from the Object specification :

- When the hashCode method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value, provided no information used in equals comparisons is modified. This value need not remain consistent from one execution of an application to another.
- If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result.
- If two objects are unequal according to the equals(Object) method, it is *not* required that calling hashCode on each of the objects must produce distinct results. However, the programmer should be aware that producing distinct results for unequal objects may improve the performance of hash tables.

The key provision that is violated when you fail to override hashCode is the second one: equal objects must have equal hash codes. Two distinct instances may be logically equal according to a class's equals method, but to Object's hashCode method, they're just two objects with nothing much in common. Therefore, Object's hashCode method returns two seemingly random numbers instead of two equal numbers as required by the contract.

For example, suppose you attempt to use instances of the PhoneNumber class from Item 10 as keys in a HashMap:

```
Map<PhoneNumber, String> m = new HashMap<>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

At this point, you might expect m.get(new PhoneNumber(707, 867, 5309)) to return "Jenny", but instead, it returns null. Notice that two PhoneNumber instances are involved: one is used for insertion into the HashMap, and a second, equal instance is used for (attempted) retrieval. The PhoneNumber class's failure to override hashCode causes the two equal instances to have unequal hash codes, in violation of the hashCode contract. Therefore, the get method is likely to look for

the phone number in a different hash bucket from the one in which it was stored by the put method. Even if the two instances happen to hash to the same bucket, the get method will almost certainly return null, because HashMap has an optimization that caches the hash code associated with each entry and doesn't bother checking for object equality if the hash codes don't match.

Fixing this problem is as simple as writing a proper hashCode method for PhoneNumber. So what should a hashCode method look like? It's trivial to write a bad one. This one, for example, is always legal but should never be used:

```
// The worst possible legal hashCode implementation - never use!
@Override public int hashCode() { return 42; }
```

It's legal because it ensures that equal objects have the same hash code. It's atrocious because it ensures that *every* object has the same hash code. Therefore, every object hashes to the same bucket, and hash tables degenerate to linked lists. Programs that should run in linear time instead run in quadratic time. For large hash tables, this is the difference between working and not working.

A good hash function tends to produce unequal hash codes for unequal instances. This is exactly what is meant by the third part of the hashCode contract. Ideally, a hash function should distribute any reasonable collection of unequal instances uniformly across all int values. Achieving this ideal can be difficult. Luckily it's not too hard to achieve a fair approximation. Here is a simple recipe:

1. Declare an int variable named result, and initialize it to the hash code c for the first significant field in your object, as computed in step 2.a. (Recall from Item 10 that a significant field is a field that affects equals comparisons.)
2. For every remaining significant field f in your object, do the following:
   a. Compute an int hash code c for the field:
      i. If the field is of a primitive type, compute *Type*.hashCode(f), where *Type* is the boxed primitive class corresponding to f's type.
      ii. If the field is an object reference and this class's equals method compares the field by recursively invoking equals, recursively invoke hashCode on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke hashCode on the canonical representation. If the value of the field is null, use 0 (or some other constant, but 0 is traditional).
      iii. If the field is an array, treat it as if each significant element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine the values per step 2.b. If the array has no significant elements, use a constant, preferably not 0. If all elements are significant, use Arrays.hashCode.
   b. Combine the hash code c computed in step 2.a into result as follows:
      `result = 31 * result + c;`
3. Return result.

When you are finished writing the hashCode method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition (unless you used AutoValue to generate your equals and hashCode methods, in which case you can safely omit these tests). If equal instances have unequal hash codes, figure out why and fix the problem.

You may exclude *derived fields* from the hash code computation. In other words, you may ignore any field whose value can be computed from fields included in the computation. You *must* exclude any fields that are not used in equals com parisons, or you risk violating the second provision of the hashCode contract.

The multiplication in step 2.b makes the result depend on the order of the fields, yielding a much better hash function if the class has multiple similar fields. For example, if the multiplication were omitted from a String hash function, all anagrams would have identical hash codes. The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, because multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance on some architectures: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimi zation automatically.

Let's apply the previous recipe to the PhoneNumber class:

```
// Typical hashCode method
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

Because this method returns the result of a simple deterministic computation whose only inputs are the three significant fields in a PhoneNumber instance, it is clear that equal PhoneNumber instances have equal hash codes. This method is, in fact, a perfectly good hashCode implementation for PhoneNumber, on par with those in the Java platform libraries. It is simple, is reasonably fast, and does a reasonable job of dispersing unequal phone numbers into different hash buckets.

While the recipe in this item yields reasonably good hash functions, they are not state-of-the-art. They are comparable in quality to the hash functions found in the Java platform libraries' value types and are adequate for most uses. If you have a bona fide need for hash functions less likely to produce collisions, see Guava's com.google.common.hash.Hashing [Guava].

The Objects class has a static method that takes an arbitrary number of objects and returns a hash code for them. This method, named hash, lets you

write one-line hashCode methods whose quality is comparable to those written accord ing to the recipe in this item. Unfortunately, they run more slowly because they entail array creation to pass a variable number of arguments, as well as boxing and unboxing if any of the arguments are of primitive type. This style of hash function is recommended for use only in situations where performance is not critical. Here is a hash function for PhoneNumber written using this technique:

```
// One-line hashCode method - mediocre performance
@Override public int hashCode() {
 return Objects.hash(lineNum, prefix, areaCode);
}
```

If a class is immutable and the cost of computing the hash code is significant, you might consider caching the hash code in the object rather than recalculating it each time it is requested. If you believe that most objects of this type will be used as hash keys, then you should calculate the hash code when the instance is created. Otherwise, you might choose to *lazily initialize* the hash code the first time hash Code is invoked. Some care is required to ensure that the class remains thread-safe in the presence of a lazily initialized field (Item 83). Our PhoneNumber class does not merit this treatment, but just to show you how it's done, here it is. Note that the initial value for the hashCode field (in this case, 0) should not be the hash code of a commonly created instance:

```
// hashCode method with lazily initialized cached hash code
private int hashCode; // Automatically initialized to 0

@Override public int hashCode() {
      int result = hashCode;
      if (result == 0) {
            result = Short.hashCode(areaCode);
                        result = 31 * result + Short.hashCode(prefix);
                       result = 31 * result + Short.hashCode(lineNum);
            hashCode = result;
      }
      return result;
}
```

Do not be tempted to exclude significant fields from the hash code com putation to improve performance. While the resulting hash function may run faster, its poor quality may degrade hash tables' performance to the point where they become unusable. In particular, the hash function may be confronted with a

large collection of instances that differ mainly in regions you've chosen to ignore. If this happens, the hash function will map all these instances to a few hash codes, and programs that should run in linear time will instead run in quadratic time.

This is not just a theoretical problem. Prior to Java 2, the String hash func tion used at most sixteen characters evenly spaced throughout the string, starting with the first character. For large collections of hierarchical names, such as URLs, this function displayed exactly the pathological behavior described earlier.

Don't provide a detailed specification for the value returned by `hashCode`, so clients can't reasonably depend on it; this gives you the flexibility to change it. Many classes in the Java libraries, such as `String` and `Integer`, specify the exact value returned by their `hashCode` method as a function of the instance value. This is *not* a good idea but a mistake that we're forced to live with: It impedes the ability to improve the hash function in future releases. If you leave the details unspecified and a flaw is found in the hash function or a better hash function is discovered, you can change it in a subsequent release.

In summary, you *must* override `hashCode` every time you override `equals`, or your program will not run correctly. Your `hashCode` method must obey the general contract specified in `Object` and must do a reasonable job assigning unequal hash codes to unequal instances. This is easy to achieve, if slightly tedious, using the recipe on page 51. As mentioned in Item 10, the AutoValue framework provides a fine alternative to writing `equals` and `hashCode` methods manually, and IDEs also provide some of this functionality.

Item 12: Always override `toString`

While `Object` provides an implementation of the `toString` method, the string that it returns is generally not what the user of your class wants to see. It consists of the class name followed by an "at" sign (@) and the unsigned hexadecimal representa tion of the hash code, for example, `PhoneNumber@163b91`. The general contract for `toString` says that the returned string should be "a concise but informative repre sentation that is easy for a person to read." While it could be argued that `PhoneNumber@163b91` is concise and easy to read, it isn't very informative when compared to `707-867-5309`. The `toString` contract goes on to say, "It is recom mended that all subclasses override this method." Good advice, indeed!

While it isn't as critical as obeying the `equals` and `hashCode` contracts (Items 10 and 11), **providing a good `toString` implementation makes your class much more pleasant to use and makes systems using the class easier to debug**. The `toString` method is automatically invoked when an object is passed to `println`, `printf`, the string concatenation operator, or `assert`, or is printed by a debugger. Even if you never call `toString` on an object, others may. For example, a component that has a reference to your object may include the string representa tion of the object in a logged error message. If you fail to override `toString`, the message may be all but useless.

If you've provided a good `toString` method for `PhoneNumber`, generating a useful diagnostic message is as easy as this:

```
System.out.println("Failed to connect to " + phoneNumber);
```

Programmers will generate diagnostic messages in this fashion whether or not you override `toString`, but the messages won't be useful unless you do. The ben efits of providing a good `toString` method extend beyond instances of the class to objects containing references to these instances, especially collections.

Which would you rather see when printing a map, {Jenny=PhoneNumber@163b91} or {Jenny=707-867-5309}?

When practical, the toString method should return *all* of the interesting information contained in the object, as shown in the phone number example. It is impractical if the object is large or if it contains state that is not conducive to string representation. Under these circumstances, toString should return a summary such as Manhattan residential phone directory (1487536 listings) or Thread[main,5,main]. Ideally, the string should be self-explanatory. (The Thread example flunks this test.) A particularly annoying penalty for failing to

include all of an object's interesting information in its string representation is test failure reports that look like this:

```
Assertion failure: expected {abc, 123}, but was {abc, 123}.
```

One important decision you'll have to make when implementing a toString method is whether to specify the format of the return value in the documentation. It is recommended that you do this for *value classes*, such as phone number or matrix. The advantage of specifying the format is that it serves as a standard, unambiguous, human-readable representation of the object. This representation can be used for input and output and in persistent human-readable data objects, such as CSV files. If you specify the format, it's usually a good idea to provide a matching static factory or constructor so programmers can easily translate back and forth between the object and its string representation. This approach is taken by many value classes in the Java platform libraries, including BigInteger, BigDecimal, and most of the boxed primitive classes.

The disadvantage of specifying the format of the toString return value is that once you've specified it, you're stuck with it for life, assuming your class is widely used. Programmers will write code to parse the representation, to generate it, and to embed it into persistent data. If you change the representation in a future release, you'll break their code and data, and they will yowl. By choosing not to specify a format, you preserve the flexibility to add information or improve the format in a subsequent release.

Whether or not you decide to specify the format, you should clearly document your intentions. If you specify the format, you should do so precisely. For example, here's a toString method to go with the PhoneNumber class in Item 11:

```
/**
 * Returns the string representation of this phone number.  * The string
consists of twelve characters whose format is  * "XXX-YYY-ZZZZ", where
XXX is the area code, YYY is the  * prefix, and ZZZZ is the line number.
Each of the capital  * letters represents a single decimal digit.
 *
 * If any of the three parts of this phone number is too small  * to fill up its
field, the field is padded with leading zeros.  * For example, if the value of
the line number is 123, the last  * four characters of the string representation
will be "0123".  */
@Override public String toString() {
      return String.format("%03d-%03d-%04d",
```

```
            areaCode, prefix, lineNum);
    }
```


If you decide not to specify a format, the documentation comment should read something like this:

```
/**
 * Returns a brief description of this potion. The exact details * of the
 representation are unspecified and subject to change, * but the following
 may be regarded as typical:
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]" */
@Override public String toString() { ... }
```

After reading this comment, programmers who produce code or persistent data that depends on the details of the format will have no one but themselves to blame when the format is changed.

Whether or not you specify the format, **provide programmatic access to the information contained in the value returned by toString.** For example, the PhoneNumber class should contain accessors for the area code, prefix, and line number. If you fail to do this, you *force* programmers who need this information to parse the string. Besides reducing performance and making unnecessary work for programmers, this process is error-prone and results in fragile systems that break if you change the format. By failing to provide accessors, you turn the string for mat into a de facto API, even if you've specified that it's subject to change.

It makes no sense to write a toString method in a static utility class (Item 4). Nor should you write a toString method in most enum types (Item 34) because Java provides a perfectly good one for you. You should, however, write a toString method in any abstract class whose subclasses share a common string representation. For example, the toString methods on most collection implemen tations are inherited from the abstract collection classes.

Google's open source AutoValue facility, discussed in Item 10, will generate a toString method for you, as will most IDEs. These methods are great for telling you the contents of each field but aren't specialized to the *meaning* of the class. So, for example, it would be inappropriate to use an automatically generated toString method for our PhoneNumber class (as phone numbers have a standard string representation), but it would be perfectly acceptable for our Potion class. That said, an automatically generated toString method is far preferable to the one inherited from Object, which tells you *nothing* about an object's value.

To recap, override Object's toString implementation in every instantiable class you write, unless a superclass has already done so. It makes classes much more pleasant to use and aids in debugging. The toString method should return a concise, useful description of the object, in an aesthetically pleasing format.

Item 13: Override clone judiciously

The Cloneable interface was intended as a *mixin interface* (Item 20) for classes to advertise that they permit cloning. Unfortunately, it fails to serve this purpose. Its primary flaw is that it lacks a clone method, and Object's clone method is protected. You cannot, without resorting to *reflection* (Item 65), invoke clone on an object merely because it implements Cloneable. Even a reflective invocation may fail, because there is no guarantee that the object has an accessible clone method. Despite this flaw and many others, the facility is in reasonably wide use, so it pays to understand it. This item tells you how to implement a well-behaved clone method, discusses when it is appropriate to do so, and presents alternatives.

So what *does* Cloneable do, given that it contains no methods? It determines the behavior of Object's protected clone implementation: if a class implements Cloneable, Object's clone method returns a field-by-field copy of the object; otherwise it throws CloneNotSupportedException. This is a highly atypical use of interfaces and not one to be emulated. Normally, implementing an interface says something about what a class can do for its clients. In this case, it modifies the behavior of a protected method on a superclass.

Though the specification doesn't say it, **in practice, a class implementing Cloneable is expected to provide a properly functioning public clone method.** In order to achieve this, the class and all of its superclasses must obey a complex, unenforceable, thinly documented protocol. The resulting mechanism is fragile, dangerous, and *extralinguistic*: it creates objects without calling a constructor.

The general contract for the clone method is weak. Here it is, copied from the Object specification :

> Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression
>
> x.clone() != x
>
> will be true, and the expression
>
> x.clone().getClass() == x.getClass()
>
> will be true, but these are not absolute requirements. While it is typically the case that
>
> x.clone().equals(x)
>
> will be true, this is not an absolute requirement.

> By convention, the object returned by this method should be obtained by calling super.clone. If a class and all of its superclasses (except Object) obey this convention, it will be the case that
>
> x.clone().getClass() == x.getClass().
>
> By convention, the returned object should be independent of the object being cloned. To achieve this independence, it may be necessary to modify one or

more fields of the object returned by super.clone before returning it.

This mechanism is vaguely similar to constructor chaining, except that it isn't enforced: if a class's clone method returns an instance that is *not* obtained by call ing super.clone but by calling a constructor, the compiler won't complain, but if a subclass of that class calls super.clone, the resulting object will have the wrong class, preventing the subclass from clone method from working properly. If a class that overrides clone is final, this convention may be safely ignored, as there are no subclasses to worry about. But if a final class has a clone method that does not invoke super.clone, there is no reason for the class to implement Cloneable, as it doesn't rely on the behavior of Object's clone implementation.

Suppose you want to implement Cloneable in a class whose superclass provides a well-behaved clone method. First call super.clone. The object you get back will be a fully functional replica of the original. Any fields declared in your class will have values identical to those of the original. If every field contains a primitive value or a reference to an immutable object, the returned object may be exactly what you need, in which case no further processing is necessary. This is the case, for example, for the PhoneNumber class in Item 11, but note that **immutable classes should never provide a clone method** because it would merely encourage wasteful copying. With that caveat, here's how a clone method for PhoneNumber would look:

```
// Clone method for class with no references to mutable state
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
            throw new AssertionError(); // Can't happen
    }
}
```

In order for this method to work, the class declaration for PhoneNumber would have to be modified to indicate that it implements Cloneable. Though Object's clone method returns Object, this clone method returns PhoneNumber. It is legal