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 mapping 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 completeness. 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 Language Specification*. The term *exported API*, or simply *API*, refers to the classes, interfaces, constructors, members, and serialized forms by which a programmer accesses a class, interface, or package. (The term *API*, which is short for *application programming interface*, is used in preference to the otherwise preferable term *interface* to avoid confusion with the language construct of that name.) A programmer who writes a program that uses an API is referred to as a *user* of the API. A class whose implementation uses an API is a *client* of the API.

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

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.