Harnessing Java™ 7

A Comprehensive Approach to Learning Java™

Volume - 2

Kishori Sharan

SAMPLE CHAPTERS

Annotations and Enum

Cover Design & Images By: Richard Castillo (http://www.digitizedchaos.com)

Printing History:

September 2011 First Print

ISBN-10: 1-46-624464-X

ISBN-13: 978-1-466-24464-1

Dedicated

To

My parents Ram Vinod Singh and Pratibha Devi

Table of Contents

Preface

My first encounter with the Java programming language was during a one-week Java training session in 1997. I did not then get a chance to use Java in a project until 1999. I read two Java books and took a Java 2 Programmer certification examination. I did very well in the test by scoring ninety five percent. The three questions that I missed on the test made me realize that the books that I had read did not adequately cover details of all the topics necessary about Java. I made up my mind to write a book about the Java programming language. So, in 2001, I formulated a plan to cover most of the topics that a Java developer needs to use the Java programming language effectively in a project, as well as to get a certification. I initially planned to cover all essential topics in Java in seven hundred to eight hundred pages.

As I progressed, I realized that a book covering most of the Java topics in detail could not be written in seven to eight hundred pages. One chapter alone that covered data types, operators, and statements spanned ninety pages. I was then faced with the question, “Should I shorten the content of the book or include all the details that I think a Java developer needs?” I opted for including all the details in the book, rather than shortening its content to keep the number of pages low. It has never been my intent to make lots of money from this book. I was never in a hurry to publish the book, because it could have compromise the quality of its contents. In short, I wrote this book to help the Java community understand and use the Java programming language effectively, without having to read many books on the same subject. I wrote this book keeping in mind that this book would be a comprehensive one-stop reference for everyone who wants to learn and grasp the intricacies of the Java programming language.

One of my high school teachers used to tell us that if one wanted to understand a building, one must first understand the bricks, steel and mortar, because a building is made up of these smaller things. The same logic applies to most of the things that we want to understand in our lives. It also applies to an understanding of the Java programming language. If you want to master the Java programming language, you must start with understanding its basic building blocks. I have used this approach throughout this book endeavoring to build each topic by describing the basics first. In this book, you will rarely find a topic described without first learning its background. Wherever possible, I have tried to correlate the programming practices with activities in our daily-life. Most of the books about the Java programming language available in the market, either do not include any pictures at all, or have only a few. I believe in the adage, “A picture is worth a thousand words.” To a reader, a picture makes a topic easier to understand and remember. I have included over two hundred and sixty graphical representations in this book (spanning three volumes) to aid readers in understanding and visualizing its contents. Developers who have little or no programming experience have difficulty in putting things together to make it a complete program. Keeping those developers in mind, I have included over five hundred complete Java programs that are ready to be compiled and run.

As I finished a chapter, I distributed copies to Java students and developers to get their feedback. Their feedback included a common observation that the material in this book is simple yet detailed. That kept me motivated to write the succeeding chapters. One feedback received in the beginning was about the coverage of setting the classpath in this book. I have seen some Java developers with quite a bit programming experience struggle with setting the classpath for a Java application. Most of the developers start programming using a Java editor. Java editors make it easy for the developers to set the classpath. Most of the time, a developer is unaware of the classpath settings when he uses a Java editor. In reality, a Java developer has to debug issues that are related to classpath settings on a machine or an application server. Keeping the principle of describing the basics of a topic, I devoted a chapter on writing a very basic Java program (the chapter name is *Writing Java Programs*) that also describes setting the classpath in detail. I gave this chapter to

many Java students and some Java developers with over five years of experience. All of them reported, “Now, I know how to work with the classpath.”

I spent countless hours doing research for writing this book. My main source of research was the Java Language Specification, white papers and articles on Java topics, and Java Specification Requests (JSRs). I also spent quite a bit of time reading the Java source code to learn more about some of the Java topics. Sometimes, it took a few months researching a topic, before I could write the first sentence of the topic. Finally, it was always fun to play with Java programs, sometimes for hours, to add them to the book.

I encountered many hurdles and pauses (some long ones) along the way of writing this book. I registered for a master degree program after I finished a few chapters. As I was working on my master program, I could not work on the book for over two years. Sometimes, the extra workload at work prevented me from doing any work on this book for months. It took me ten years (You read it right. Ten years is called a decade.) to finish this book. If I had devoted all my time on writing this book, I could say that it would have taken me about two years to finish it. I also had to keep adding new material to cover the newer versions of Java. I started writing this book using Java 1.2 and finished it using Java 1.7. Finally, it turned out to be almost a two thousand page book, which had to be split in three volumes, because of the restrictions on the number of pages a print-on-demand book can have. At this point, all I can say is, “All’s well that ends well.”

Kishori Sharan

Structure of the Book

This book contains thirty-four chapters and three appendixes spread across three volumes. The print-on-demand technology puts a restriction on the maximum number of pages in a book. This made me divide the book into three volumes. To get the most out of this book, a reader is suggested to read from the first chapter to the last. Each chapter builds upon the previous chapters. Volumes and chapters inside a volume have been arranged in a way that presents the most basic material about the Java programming language first. Sections in a chapter are arranged in an order of increasing complexity, the least complex section being the first. The following is the list of topics covered in the three volumes.

Volume - 1

- Programming Concepts
- Data Types
- Operators
- Statements
- Classes & Objects
- Object and Objects Classes
- AutoBoxing
- Exception Handling
- Assertions
- Strings & Dates
- Formatting Objects
- Regular Expressions
- Arrays
- Garbage Collection
- Inheritance

Volume - 2

- Interfaces
- Annotations
- Inner Classes
- Enum
- Reflection
- Generics
- Threads
- Input/Output
- Archive Files
- Collections

Volume - 3

- Swing
- Applets
- Network Programming
- JDBC API
- Remote Method Invocation
- Java Native Interface

Audience

This book is designed to be useful for anyone who wants to learn about the Java programming language. If you are a beginner, with little or no programming background, you need to read from the first chapter to the last, in order. The book contains topics of various degrees of complexity. As a beginner, if you find yourself overwhelmed while reading a section in a chapter, you can skip to the next section or the next chapter and revisit them later, when you gain more experience.

If you are a Java developer with intermediate or advanced level of experience, you can jump to a chapter or to a section in a chapter directly. If a section uses an unfamiliar topic, you need to visit that topic before continuing the current one. You may only want to read volumes of this book that cover the topics of your interest.

If you are reading this book to get a certification in the Java programming language, you need to read almost all chapters paying attention to all the detailed descriptions and rules. Most of the certification programs test your fundamental knowledge of the language, not the advanced knowledge. You need to read only those topics that are part of your certification test. Compiling and running over five hundred complete Java programs will help you prepare for your certification.

If you are a student who is attending a class in the Java programming language, you need to read the first six chapters in Volume 1, thoroughly. These chapters cover the basics of the Java programming languages in detail. You cannot do well in a Java class unless you first master the basics. After covering the basics, you need to read only those chapters that are covered in your class syllabus. I am sure, as a Java student, you do not need to read the entire book page-by-page.

How to Use This Book

This book is the beginning, not the end, for you to gain the knowledge of the Java programming language. If you are reading this book, it means you are heading in the right direction to learn the Java programming language that will enable you to excel in your academic and professional career. However, there is always a higher goal for you to achieve and you must constantly work harder to achieve it. The following quotations from some great thinkers may help you understand the importance of working hard and constantly looking for knowledge with both your eyes and mind open.

> *Be curious always, for knowledge will not acquire you; you must acquire it. - Sudie Back*
>
> *Knowledge comes by eyes always open and working hard, and there is no knowledge that is not power. - Jeremy Taylor*
>
> *The learning and knowledge that we have, is, at the most, but little compared with that of which we are ignorant. - Plato*
>
> *True knowledge exists in knowing that you know nothing. And in knowing that you know nothing, that makes you the smartest of all. - Socrates*

Readers are advised to use the API documentation for the Java programming language, as much as possible, while using this book. The Java API documentation is the place where you will find a complete list of documentation for everything available in the Java class library. You can download (or view) the Java API documentation from the official website of Oracle Corporation at *http://www.oracle.com*. While you read this book, you need to practice writing Java programs

yourself. You can also practice by tweaking the programs provided in the book. It does not help much in your learning process, if you just read this book and do not practice by writing your own programs. Remember - “Practice makes a person perfect”, which is also true in learning how to program in Java.

Java 7 New Features

Java 7 has added many new language level features. This book covers all Java 7 language level new features. A complete chapter in Volume - 2 has been devoted to discuss the NIO 2.0 in detail. The new features in Java 7 have been discussed in the related chapters. The following is the list of the new features of Java 7 covered in three volumes of this book.

Volume - 1

- Binary Numeric Literals
- Underscores in Numeric Literals
- Strings in a switch Statement
- try-with-resources Statement
- Catching Multiple Exception Types
- Re-throwing Exceptions with Improved Type Checking
- The java.util.Objects class

Volume - 2

- Generic Type Inference
- Heap Pollution
- Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods
- Improved File and Channel Closing Mechanism using a try-with-resources Statement
- New Input/Output 2.0 (NIO 2.0)
- Fork/Join Framework
- Phaser Synchronization Barrier
- TransferQueue Collection

Volume - 3

- JLayer Swing Component
- Translucent Window
- Shaped Window
- Asynchronous Socket IO
- Multicast DatagramChannel
- RowSetFactory

Acknowledgements

This book could not have been written without the encouragements, supports and contributions from many people.

First, I would like to thank the authors of all the books, articles, white papers and Java Specification Requests (JSRs) related to the Java programming language that I read and consulted to gain my own knowledge of the Java programming language.

My heartfelt thanks are due to my father-in-law Mr. Jim Baker for displaying extraordinary patience in proof reading the book. I am very grateful to him for spending so much of his valuable time teaching me quite a bit of English grammar that helped me in producing better material, and hence less work for him during his proof reading sessions. I would also like to thank my mother-in-law Ms. Kim Baker for providing him delicious food (including letting him eat ice cream) and regularly reminding him to finish proof reading this book.

My wife Ellen was always patient when I spent long hours at my computer desk working on this book. She would happily bring me snacks, fruit, and a glass of water every thirty minutes or so to sustain me during that period. I want to thank her for all her support in writing this book. She also deserves my sincere thanks for proofreading many of the chapters and providing valuable feedback.

I would like to thank my sister-in-law Patty Boyd for cooking delicious food for me while I worked on the book. Thanks also go to my brother-in-law Jeff Boyd and my nephew Christopher Estes for helping me in many ways to save my time, so that I can focus on the book.

I would like to thank my friend Kannan Somasekar for his support and hard work to get an appropriate subtitle for this book. I would also like to thank him for time spent in researching the possible publishing options. His research helped me choose the print-on-demand publishing option. I would like to thank Kannan's wife, Divya Somasekar, for taking care of their two lovely sons, Krish and Rishi, while Kannan spent time at his computer desk to help me finish this book.

My special thanks go to my friend and colleague, Richard Castillo, for proof reading this book very thoroughly. He deserves a big thank-you for designing the cover pages and suggesting the title of the book.

I would like to thank my friends and colleagues Christopher Coley, Tanu Mutreja, Rahul Jain, Raju Mudunuri, Willie Baptiste, Tejas Dholakia, Amita Dholakia, Lorenzo Braxter, and Mahbub Chowdhury, for providing valuable feedback. Willie Baptiste, Tejas Dholakia, Amita Dholakia, Lorenzo Braxter, and Mahbub Chowdhury deserve little extra thanks for giving me the opportunity to teach them Java, which gave me more insight on how to explain things in the book to make it easier for the readers to understand.

There is one good thing about members of anybody's family, including mine. Once you tell them that you are working on writing a book, they will remind you periodically about the status of your book, which keeps you always aware that you have one unfinished job at your hand and you must finish it sometime in your life! Finishing this book was not possible without the blessings of my parents, Ram Vinod Singh and Pratibha Devi, and my elder brothers, Janki Sharan and Dr. Sita Sharan. My most sincere and heartfelt thanks go to them for their support and encouragement during the entire period I was working on the book. I would also like to thank my sister Ratna Kumari and brother-in-law Abhay Kumar Singh, my nephew Navin Kumar and daughter-in-law Anjali Singh, my niece Vandana Kumari and son-in-law Prem Prakash, my nephew Neeraj Kumar and daughter-in-law Pallavi, and my nephews Gaurav Sharan and Saurav Kumar, for their interest.

I would like to thank Chitranjan Sharma, my nephew and a student of Bachelor of Computer Applications at Gaya College Gaya, for his diligent efforts to learn Java using this book and providing me valuable feedback.

I would like to thank the following colleagues for their support, encouragement and for being always supportive, while I worked with the Department of Children Service, State of Tennessee: John Jacobs, Reddy Matta, Donna Duncan, Katrina Hills, LaTondra Okeke, Prasanna Despande, Geshan Alvis, Buddy Rice, Jack Parker, Jags.Pinni, Bettye Clark, Charles O' Riley, Basir Kabir, Barbara Gentry, Paula Daugherty, Dinesh Sankala, Elaine Blaylock, Lisa Simmons, Deborah Hurd, Wallace Inman, Pravin Lokhande, Priyanka Sharma, Amitabh Sharma, and Wanda Jackson. I know that some of them did not know that I had been writing a book. But, I feel they deserve mention, because it helped me in the writing of this book at home, when they all were nice and supportive at work. After all, good attitude is contagious!

I would like to thank the following colleagues for their support and helping me learn the intricacies of Java while I worked for Kingsway America in Mobile, Alabama: Larry Brewster, Biju Nair, Jim Jacobs, Ram Atmakuri, Srinivas Kakerra, James Pham, Megan Bodiford, Udhaya Kumar, Matt Flowers, and Greg Langham.

I would like to thank the following colleagues for their support while I worked at ProAssurance in Birmingham, Alabama: LaRonda Lanier, Christy Mueller-Smith, Darren Jackson, Bob Waldrop, Tatiana Telioukova, Russell Thomas, Cameron Ellison, Brandon Russell, Devaraj Rajan, Frank Lay, Andriy Shlykov, Andy Purvis, Rob Ballard, Marty Heim, and Troy Dotson. As I have mentioned earlier, some of my colleagues may not even be aware that I have been writing a book. However,

their support in creating a cordial and helping working environment at work helped me carrying the good frame of mind to home in the evening to spend a few hours of my time on this book. So, all of you deserve my sincere thanks.

I would like to thank the following managers for their support while I worked in different projects: Robert Holloway, Connie Spradlin, Ed Bennett, Kieran Cloonan, and Donovan Fitzgerald at Department of Children Services, Nashville, TN; Leslie Zanders, Cheryl Lawrence, and Kelly Dumas at Kingsway America in Mobile, AL; Kirby Sims, Douglas Dyer, Brian Russell, Lael Boyd, and Vivi Gin at ProAssurance in Birmingham, AL; Heath Wade and Amy Gartman at Doozer Inc.

I would like to thank the following friends for their support and encouragement: Rahul Nagpal, Ravi Datla, Anil Kumar Singh, Balram Kumar, Dilip Kumar, Ramta Prasad Singh, Pratap Chandra, Sanjeev Choudhary, Pramod Kumar, Prakash Chandra, Dharmendhra Kumar Mishra, Rajeev Kumar Verma, Randy Lucas, Sanjay Pandey, Suman Kumar Singh, Amarjeet Kumar, Vijay Kumar Tarun, Raju Mishra, Jayshankar Prasad Singh, Mukesh Sinha, Rajesh Kumar, Vishwa Mohan, Ranjeet Ekka, Sanjay Singh, Kamal Singh, Pankaj Kumar, Ranjeet Kumar, Krishna Kumar, and Anuj Sinha.

Source Code and Errata

Source code and errata for this volume may be downloaded from *http://www.jdojo.com*.

Questions and Comments

Please direct all your questions and comments to *ksharan@jdojo.com*

.

Chapter 2. Annotations

Annotation was introduced in Java 5. Before we define annotation and discuss its importance in programming, let us discuss a simple example. Suppose we have an `Employee` class, which has one method `setVacationHours()` that lets you set the vacation hours for an employee. This method accepts a parameter of `double` data type. The following snippet of code shows a trivial implementation for an `Employee` class. The method in this class displays a message on the standard output.

```
public class Employee {
     public void setVacationHours(double hours) {
          System.out.println("Employee.setVacationHours(): " + hours);
     }
}
```

A `Manager` class inherits the `Employee` class. We want to deal with vacation hours for managers differently. We decide to override the `setVacationHours()` method in the `Manager` class. Note that there is a mistake made in the code for the `Manager` class, when we attempt to override the `setVacationHours()` method. We will correct the mistake shortly. We have used the `int` data type as the parameter type for our incorrectly overridden method. The code for the `Manager` class is shown below.

```
public class Manager extends Employee {
     // Override setVacationHours() in the Employee class
     public void setVacationHours(int hours) {
          System.out.println("Manager.setVacationHours():" + hours);
     }
}
```

It is time to set the vacation hours for a manager. The following code is used to accomplish this.

```
Employee emp = new Manager();
int hours = 200;
emp.setVacationHours(hours);
```

```
Output:

Employee.setVacationHours():200.0
```

The above snippet of code was expected to call the `setVacationHours()` method of the `Manager` class The output does not show the expected result.

What went wrong in our code? The intention of defining the `setVacationHours()` method in the `Manager` class was to override the `setVacationHours()` method of the `Employee` class and not to overload it. We made a mistake. We used the `int` data type as the parameter type in the `setVacationHours()` method, instead of a `double` type, in the `Manager` class. We did put comments indicating our intention to override the method in the `Manager` class. However, comments are dumb! Comments do not stop us from making logical mistakes in our programs. You

might have spent, as every programmer does, hours and hours debugging errors resulting from this kind of logical mistakes. Who can help us in such situations? Annotations might help us in a few situations like this. Let us rewrite our `Manager` class using an annotation. You do not need to know anything about annotations at this point. All we are going to do is add one word to our program. Here is the modified version of the `Manager` class.

```
public class Manager extends Employee {
     @Override
     public void setVacationHours(int hours) {
          System.out.println("Manager.setVacationHours():" + hours);
     }
}
```

All we have added is a `@Override` annotation to the old `Manager` class and we have removed the "dumb" comments. Just make sure you have placed the `@Override` annotation as part of the `setVacationHours()` method's declaration. If you try to compile the `Manager` class, the compiler refuses do so with the following compiler error.

```
Error(6,17): method setVacationHours(int) does not override any method in
its superclass
```

The use of the `@Override` annotation did the trick. The `@Override` annotation is used with a non-static method to indicate the programmer's intention to override the method in the superclass. At source code level, it serves the purpose of documentation. When the compiler comes across the `@Override` annotation, it makes sure that the method really overrides the method in the superclass. If the method annotated with `@Override` does not override a method in its superclass, the compiler generates an error. In our case, the `setVacationHours(int hours)` method in the `Manager` class does not override any method in its superclass `Employee`. This is the reason why we got an error. You may realize that using an annotation is as simple as documenting the source code. However, it has the compiler support. You can use it to instruct the compiler to enforce some rules. Annotations provide benefits much more than we have seen in this example.

Let us go back to the compiler error that was generated. We can fix the compiler error by doing one of the following two things.

- We can remove the `@Override` annotation from the `setVacationHours(int hours)` method in the `Manager` class. It will make the method a regular method, which it is, and not a method that overrides its superclass method.
- We can change the method signature from `setVacationHours(int hours)` to `setVacationHours(double hours)` in the `Manager` class.

Since we wanted to override the `setVacationHours()` method in the `Manager` class, we will go for the second option and modify the `Manager` class as follows.

```
public class Manager extends Employee {
     @Override
     public void setVacationHours(double hours) {
          System.out.println("Manager.setVacationHours():" + hours);
     }
}
```

Now, the following code will work as expected.

```
Employee emp = new Manager();
int hours = 200;
emp.setVacationHours(hours);
```

```
Output:

Manager.setVacationHours():200.0
```

Note that the `@Override` annotation in the `setVacationHours()` method of the `Manager` class saves you debugging time. Suppose you change this method signature in the `Employee` class. If the changes in the `Employee` class make this method no longer overridden in the `Manager` class, you will get the same error when you compile the `Manager` class again. Did you start realizing the power of annotations? With this background in mind, let us start digging deep into annotations.

What is an Annotation?

According to the Merriam Webster dictionary, the meaning of annotation is:

> *"a note added by way of comment or explanation".*

This is exactly what an annotation is in Java. It lets you associate (or annotate) metadata (or notes) to the program elements in a Java program. The program elements may be a package, a class, an interface, a field of a class, a local variable, a method, a parameter of a method, an enum, an annotation, etc. In other words, you can annotate any declaration in a Java program with an annotation. An annotation is used as a modifier in a declaration of a program element like any other modifiers `public`, `private`, `final`, `static`, etc. Unlike a modifier, an annotation does not modify the meaning of the program elements. It acts like a decoration or a note for the program element that it annotates.

An annotation differs from regular documentation in many ways. A regular documentation is only for humans to read and it is "dumb". It has no intelligence associated with it. If you misspell a word, or state something in the documentation and do just the opposite in the code, you are on your own. It is very difficult and impractical to read the elements of documentation programmatically at runtime. Java lets you generate Javadoc from your documentation and that is where everything stops for the regular documentation. This does not mean that we do not need to document our programs. We do need regular documentation. At the same time, we need a way to enforce our intent using a documentation-like mechanism. Our documentation should be available to the compiler and to the runtime. An annotation serves this purpose. It is human readable, which serves as documentation. It is compiler readable, which lets the compiler verify the intention of the programmer; for example, the compiler makes sure that the programmer has really overridden the method if it comes across a `@Override` annotation for a method. Annotations are also available at runtime, so that a program can read and use it for any purpose it wants. For example, a tool can read annotations and generate supporting boilerplate code. If you have worked with Enterprise JavaBeans (EJB), you know the pain of keeping all the interfaces and classes in sync and adding entries to XML configuration files. EJB 3.0 uses annotations to generate the boilerplate code, which makes EJB development painless for programmers. Another example of annotation being used in a framework/tool is JUnit version 4.0. JUnit is a unit test framework for Java programs. It uses annotations to mark methods that are test cases. Before that, you had to follow a naming convention for the test case methods. Annotations have a variety of uses, which are documentation, verification, and enforcement by the compiler, the runtime validation, code generation by frameworks/tools, etc.

To make an annotation available to the compiler and to the runtime systems, an annotation has to follow rules. In fact, an annotation is another type like a class and an interface. As we have to declare a class type or an interface type, before we can use it, we must also declare an annotation type, before we can use it to annotate a program element.

An annotation does not change the semantics (or meaning) of the program element that it annotates. In that sense, an annotation is like a comment, which does not affect the way the annotated program element works. For example, the `@override` annotation for the `setVacationHours()` method did not change the way this method works. You (or a tool/framework) can change the behavior of a program based on an annotation. In such cases, you make use of the annotation rather than the annotation doing anything on its own. The point is that an annotation by itself is always passive.

Declaring an Annotation Type

Declaring an annotation type is similar to declaring an interface type with some restrictions. According to Java specification, an annotation type declaration is a special kind of interface type declaration. You use the `interface` keyword, which is preceded by the `@` sign (at sign), to declare an annotation type. Here is the general syntax for declaring an annotation type.

```
<<modifiers>> @ interface <<annotation type name>> {
     // Annotation type body goes here
}
```

The `<<modifiers>>` that you can use for an annotation declaration are the same as for an interface declaration. For example, you can declare an annotation type as `public` or package level. The `@` sign (at sign) and the `interface` keyword may be separated by whitespaces, or they can be placed together. By convention, they are placed together as `@interface`. The `interface` keyword is followed by a user defined annotation type name. It should be a valid Java identifier. The annotation type body is placed within braces (`{}`).

Suppose we want to annotate our program elements with the version information, so we can prepare a report about new program elements added in a specific release of our product. To use a custom annotation type (as opposed to built-in annotation, e.g., `@Override`) you must declare it first. We want to include the major and the minor versions of the release in the version information. Listing 2-1 has the complete code for our first annotation declaration.

Listing 2-1: The declaration of the Version annotation

```
// Version.java
package com.jdojo.chapter2;

public @interface Version {
     int major();
     int minor();
}
```

Compare the declaration of the `Version` annotation with the declaration of an interface. It differs from an interface definition only in one aspect in that it uses the `@` sign (at sign) before its name. We have declared two abstract methods in the `Version` annotation type: `major()` and `minor()`. Abstract methods in an annotation type are known as its elements. You can think about it in another way too. An annotation can declare zero, or more elements and they are declared as

abstract methods. The abstract method names are the names of the elements of the annotation type. We have declared two elements – `major` and `minor`, for our `Version` annotation type. The data types of both elements are `int`.

You need to compile the annotation type. When *Version.java* file is compiled, it will produce a *Version.class* file. The simple name of our annotation type is `Version` and its fully qualified name is `com.jdojo.chapter2.Version`. When you use the `Version` annotation type, you must import it to use its simple name. Otherwise, you will have to use its fully qualified name.

How do we use an annotation type? You might be thinking that we will declare a new class that will implement the `Version` annotation type, and we will create an object of that class. You might be relieved to know that you do not need to take any additional steps to use the `Version` annotation type. An annotation type is ready to be used as soon as it is declared and compiled. To create an instance of an annotation type and use it to annotate a program element you need to use the following syntax.

```
@annotationType(name1=value1, name2=value2, names3=values3, ...)
```

The annotation type is preceded by a `@` sign (at sign). The annotation type is followed by a list of comma-separated `name=value` pairs enclosed in parentheses. The name in `name=value` pair is the name of the element declared in the annotation type and the value is the user supplied value for that element. The `name=value` pairs do not have to appear in the same order as they are declared in the annotation type. Although, by convention `name=value` pairs are used in the same order as the declaration of the elements in the annotation type.

Let us use an annotation of the `Version` type, which has the major element value as 1 and the minor element value as 0. The following is an instance of our `Version` annotation type.

```
@Version(major=1, minor=0)
```

We can rewrite the above annotation as `@Version(minor=0, major=1)` without changing its meaning. You can also use the annotation type's fully qualified name as:

```
@com.jdojo.chapter2.Version(major=0, minor=1)
```

You use as many instances of the `Version` annotation type in your program as you want. For example, we have a `VersionTest` class, which was added to our application since its release 1.0. We have added some methods and instance variables in release 1.1. We can use our `Version` annotation to document additions to the `VersionTest` class in different releases. We can annotate our class declaration as:

```
@Version(major=1, minor=0)
public class VersionTest {
     // Code goes here
}
```

An annotation acts as a modifier for the program element it annotates. You can mix the annotation for a program element with its other modifiers. You can place annotations in the same line as other modifiers or in a separate line. It is just a matter of personal choice whether you use a separate line to place the annotations or you mix them with other modifiers. By convention, annotations for a program element are placed before all other modifiers. We will follow the convention and place the annotation in a separate line by itself as shown above. Both of the following declarations are technically the same.

```
// Style - 1
@Version(major=1, minor=0) public class VersionTest {
     // Code goes here
}

// Style - 2
public @Version(major=1, minor=0) class VersionTest {
     // Code goes here
}
```

Listing 2-2 shows the sample code for the `VersionTest` class. We have used `@Version` annotation to annotate the class declaration, the class field, the constructors, and the methods. There is nothing extraordinary in the code for the `VersionTest` class. We have just added `@Version` annotation to various elements of this class. The `VersionTest` class would work the same, even if you remove all `@Version` annotations. It is to be emphasized that using annotations in your program does not change the behavior of the program at all. The real benefit of annotations comes from reading it during compilation and runtime.

Listing 2-2: A VersionTest class with annotated elements

```
// VersionTest.java
package com.jdojo.chapter2;

// Annotation for class VersionTest
@Version(major = 1, minor = 0)
public class VersionTest {
     // Annotation for instance variable xyz
     @Version(major = 1, minor = 1)
     private int xyz = 110;

     // Annotation for constructor VersionTest()
     @Version(major = 1, minor = 0)
     public VersionTest() {
     }

     // Annotation for constructor VersionTest(int xyz)
     @Version(major = 1, minor = 1)
     public VersionTest(int xyz) {
          this.xyz = xyz;
     }

     // Annotation for the printData() method
     @Version(major = 1, minor = 0)
     public void printData() {
          // ...
     }

     // Annotation for the setXyz() method
     @Version(major = 1, minor = 1)
     public void setXyz(int xyz) {
          // Annotation for local variable newValue
          @Version(major = 1, minor = 2)
          int newValue = xyz;
          this.xyz = xyz;
     }
}
```

What do we do next with our `Version` annotation type? We have declared it as a type. We have used it in our `VersionTest` class. Our next step would be to read it at runtime. We will defer this step. We will cover it in detail in a section described later.

Restrictions on Annotations

An annotation is a special type of interface with the following restrictions.

Restriction #1

An annotation type cannot inherit from another annotation type. That is, you cannot use the `extends` clause in an annotation type declaration. The following declaration will not compile, because we have used the `extends` clause to declare `WrongVersion` annotation type.

```
// Won't compile
public @interface WrongVersion extends BasicVersion {
      int extended();
}
```

Every annotation type implicitly inherits the `java.lang.annotation.Annotation` interface, whose declaration is as follows.

```
package java.lang.annotation;

public interface Annotation {
     boolean equals(Object obj);
     int hashCode();
     String toString();
     Class<? extends Annotation> annotationType();
}
```

This implies that all of the four methods declared in the `Annotation` interface are available in all user-defined annotation types. A word of caution needs to be mentioned here. We declare elements for an annotation type using abstract method declarations. The methods declared in the `Annotation` interface do not declare elements in an annotation type. Our `Version` annotation type has only two elements – `major` and `minor`, which are declared in the type itself. You cannot use annotation type `Version` as `@Version(major=1, minor=2, toString="Hello")`. The `Version` annotation type does not declare `toString` as an element. It inherits the `toString()` method from the `Annotation` interface. An annotation type has elements that are declared in its definition. An annotation does not inherit any elements from the `Annotation` interface, which is the implicit ancestor of all annotation types.

Restriction #2

Method declarations in an annotation type cannot specify any parameters. A method in an annotation type declares an element for the annotation type. An element in an annotation type lets you associate a data value to an annotation's instance. A method declaration in an annotation is not called to perform any kind of processing. Think of an element as an instance variable in a class having two methods - one setter method, and one getter method, for that instance variable. For an

annotation, the Java runtime creates a proxy class that implements the annotation type (which is an interface). Each annotation instance is an object of that proxy class. The method you declare in your annotation type becomes the getter method for the value of that element you specify in the annotation. Java runtime will take care of setting the specified value for your annotation element. Since the goal of declaring a method in an annotation type is to work with a data element, we do not need to (and are not allowed to) specify any parameters in a method declaration. The following declaration of an annotation type would not compile, because it declares a `concatenate()` method, which accepts two parameters.

```
// Won't compile
public @interface WrongVersion {
     // Cannot have parameters
     String concatenate(int major, int minor);
}
```

Restriction #3

Method declarations in an annotation type cannot have a `throws` clause. A method in an annotation type is defined to represent a data element. Throwing an exception to represent a data value does not make sense. The following declaration of an annotation type would not compile, because the `major()` method has a `throws` clause.

```
// Won't compile
public @interface WrongVersion {
     int major() throws Exception; // Cannot have throws clause
     int minor(); // OK
}
```

Restriction #4

A method declared in an annotation has restrictions for its return type. The return type of a method declared in an annotation type must be one of the following types.

- Any primitive types – `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`.
- `java.lang.String`
- `java.lang.Class`
- An `enum` type
- An annotation type
- An array of any of the above mentioned type. The return type cannot be a nested array. For example, you cannot have a return type of `String[][]` or `int[][]`. However, it is allowed to have a return type of `String[]`, `int[]`, etc.

The return type of `Class` needs a little explanation. Instead of the `Class` type, you can use a generic return type that will return a user defined class type. Suppose you have a `Test` class and you want to declare the return type of a method in an annotation type, which is of type `Test`. You can declare the annotation method as shown below.

```
public @interface GoodOne {
      Class element1();              // Any Class type
      Class<Test> element2();        // Only Test class type
```

```
      Class<? extends Test> element3() ; // Test or its subclass type
}
```

Restriction #5

An annotation type cannot declare a method, which would be equivalent to overriding a method in the `Object` class or the `Annotation` interface.

Restriction #6

An annotation type cannot be generic.

An annotation type declaration declares its elements (which are data elements) using method declarations. The return type of the method represents the data type of the element. The compiler makes sure that you have supplied values of the correct data type for elements of an annotation. For example, the data type of the two elements, `major` and `minor`, in the `Version` annotation is `int`. The compiler makes sure that we have passed an `int` value for the `major` and the `minor` elements when using the `@Version` annotation. For example, none of the following uses of the `@Version` annotation would compile, because they do not use a value of the `int` data type for its elements.

```
@Version(major="1", minor="0");  // Error. (String, String)
@Version(major=1.0, minor="0");  // Error, (double, String)
@Version(major="1.0", minor=0);  // Error, (String, double)
@Version(major=1.0, minor=0.0);  // Error, (double, double)
```

Default Value of an Annotation Element

The syntax for an annotation type declaration lets you specify a default value for its elements. You are not required to specify a value for an annotation element that has a default value specified in its declaration. The default value for an element can be specified using the following general syntax.

```
<<modifiers>> @interface <<annotation type name>> {
     <<data type>> <<element name>>() default <<default value>>;
}
```

The `default` keyword is used to specify the default value. The default value must be of the type compatible to the data type for that element. For example, if the data type of an element (which is the return type of the method) is `int`, its default value must be an integer.

Suppose we have a product that is not frequently released. Therefore, it is less likely that it will have a minor version other than zero. We can simplify our `Version` annotation type by specifying the default value for its `minor` element as zero. The following code shows that the default value for the minor element is set to zero.

```
public @interface Version {
     int major();
     int minor() default 0;    // Set zero as default value for minor
}
```

Once you set the default value for an element, you do not have to pass its value when you use an annotation of this type. Java will use the default value for the missing value of the element.

```
@Version(major=1)  // minor is zero, which is its default value
@Version(major=2)  // minor is zero, which is its default value
@Version(major=2, minor=1) // minor is 1, which is the specified value
```

All default values must be compile-time constants. How do you specify the default value for an array type? You need to use the array initializer syntax to specify the default value for an array type element. The following snippet of code shows how to specify default values for an array and other data types.

```
// Shows how to assign default values to elements of different types
public @interface DefaultTest {
     double d() default 12.89;
     int num() default 12;
     int[] x() default {1, 2};
     String s() default "Hello";
     String[] s2() default { "abc", "xyz"};
     Class c() default Exception.class;
     Class[] c2() default {Exception.class, java.io.IOException.class};
}
```

TIP
If an annotation's element does not have a default value, the element's value must be specified when the annotation type is used. If an element has a default value, passing a value for it is optional. The default value for an element is not compiled with the annotation. It is read from the annotation type definition, when a program attempts to read the value of an element at runtime. For example, when you use `@Version(major=2)`, this annotation instance is compiled as is. It does not add `minor` element with its default value as zero. In other words, this annotation is not modified to `@Version(major=2, minor=0)` at the time of compilation. However, when you read the value of the `minor` element for this annotation at runtime, Java will detect that the value for the `minor` element was not specified. It will consult the `Version` annotation type definition for its default value and return that value to you. The implication of this mechanism is that if you change the default value of an element, the changed default value will be read whenever a program attempts to read it, even if the annotated program was compiled before you changed the default value.

Annotation Type and its Instances

We use the terms "annotation type" and "annotation" frequently. Annotation type is a type like an interface. Theoretically, you can use annotation type wherever you can use an interface type. Practically, we limit its use only to annotate program elements. You can declare a variable of an annotation type as:

```
Version v = null; // Here, Version is an annotation type
```

Like an interface, you can also implement an annotation type in a class. However, you are never supposed to do that, as it will defeat the purpose of having an annotation type as a new construct. You should always implement an interface in a class, not an annotation type. Technically, the following code for the `DoNotUseIt` class is valid.

```
/* This is just for the demonstration purpose.
  Do not implement an annotation in a class even if it works
*/
// DoNotUseIt.java
package com.jdojo.chapter2;

import java.lang.annotation.Annotation;

public class DoNotUseIt implements Version {
     // Implemented method from the Version annotation type
     public int major() {
          return 0;
     }

     // Implemented method from the Version annotation type
     public int minor() {
          return 0;
     }

     // Implemented method from the Annotation annotation type
     // which is the supertype of the Version annotation type
     public Class<? extends Annotation> annotationType() {
          return null;
     }
}
```

The Java runtime implements your annotation type to a proxy class. It provides you with an object of a class that implements your annotation type for each annotation you use in your program. We must distinguish between an annotation type and instances (or objects) of that annotation type. In our example, `Version` is an annotation type. Whenever we use it as `@Version(major=2, minor=4)`, we are creating an instance of the `Version` annotation type. An instance of an annotation type is simply referred to as an "annotation". For example, we say that `@Version(major=2, minor=4)` is an annotation or an instance of the `Version` annotation type. Each time you use `@Version(...)` annotation in your code, Java runtime will create a new object of the class (created as a `Proxy` class at runtime), which implements the `Version` annotation type. An annotation should be easy to use in a program. The syntax, `@Version(...)`, is a shorthand for creating a class, creating an object of that class, and setting the values for its elements. We will cover how we get to the object of an annotation type at runtime later in this chapter.

Using Annotations

In this section, we will go through the details of using annotations of different data types in a program. The supplied value for elements of an annotation must be a compile-time constant expression. You cannot use `null` as the value for any type of element in an annotation.

Primitive Types

The data type of an element in an annotation type could be any of the primitive data types - `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. Our `Version` annotation type uses two elements, `major` and `minor`, and both are of `int` data type. We must use a compile-time constant

expression to specify a value for an element of an annotation. Typically, you would use primitive type literals as the value of the elements. If the data type of an element is `boolean`, its value must be specified either `true` or `false`. Of course, you can specify any compile time expression that evaluates to a `boolean` value of `true` or `false`. The following code snippet declares an annotation type called `PrimitiveAnnTest`.

```
public @interface PrimitiveAnnTest {
     byte a();
     short b();
     int c();
     long d();
     float e();
     double f();
     boolean g();
     char h();
}
```

You can use an instance of the `PrimitiveAnnTest` type as:

```
@PrimitiveAnnTest(a=1, b=2, c=3, d=4, e=12.34F, f=1.89, g=true, h='Y')
```

You can use an expression to specify the value for an element of an annotation. The following two instances of the `Version` annotation are valid, and have the same values for their elements.

```
@Version(major=2+1, minor=(int)13.2)
@Version(major=3, minor=13)
```

String Types

You can use an element of the `String` type in an annotation type. Listing 2-3 has code for a `Name` annotation type. It has two elements, `first` and `last`, which are of the `String` type. You can use this annotation whenever you need to annotate a program element with a name. It lets you specify the first and the last names.

Listing 2-3: Name annotation type, which has two elements – first and last, of the String type

```
package com.jdojo.chapter2;

public @interface Name {
     String first();
     String last();
}
```

The following snippet of code shows how to use the `Name` annotation in a program.

```
@Name(first="John", last="Jacobs")
public class NameTest {
     @Name(first="Wally", last="Inman")
     public void aMethod() {
          //More code goes here...
     }
}
```

It is valid to use the string concatenation operator (+) in the value expression for an element of a `String` type. The following two annotations are equivalent.

```
@Name(first=”Jo” + “hn”, last=”Ja” + “cobs”)
@Name(first=”John”, last=”Jacobs”)
```

The following use of the `@Name` annotation is not valid, because the expression, `new String(“John”)`, is not a compile-time constant expression.

```
@Name(first=new String("John"), last="Jacobs")
```

Class Types

The benefits of using the `Class` type as an element in an annotation type are not obvious. Typically, it is used where a tool/framework reads the annotations with elements of a class type and performs some specialized processing on the element’s value or generates code. We will go through a simple example of using a class type element. Suppose you are writing a test runner tool for running test cases for a Java program. Your annotation will be used in writing test cases. If your test case must throw an exception when it is invoked by the test runner, you need to use an annotation to indicate that. Let us create a default exception class as shown in Listing 2-4.

Listing 2-4: A DefaultException class that is inherited from the Throwable exception class

```
// DefaultException.java
package com.jdojo.chapter2;

public class DefaultException extends java.lang.Throwable {
     public DefaultException() {
     }

     public DefaultException(String msg) {
          super(msg);
     }
}
```

Listing 2-5: A TestCase annotation type whose instances are used to annotate test case methods

```
//TestCase.java
package com.jdojo.chapter2;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface TestCase {
   Class<? extends Throwable> willThrow() default DefaultException.class;
}
```

Listing 2-5 shows the code for our `TestCase` annotation type. The return type of the `willThrow` element is defined as the wild card of the `Throwable` class, so that the user will specify only the `Throwable` class or its subclasses as the element’s value. We could have used the `Class` type as

the type of our `willThrow` element. However, that would have allowed the users of this annotation type to pass any class type as its value. Note that we have used two annotations, `@Retention` and `@Target`, for the `TestCase` annotation type. The `@Retention` annotation type specified that `@TestCase` annotation would be available at runtime. It is necessary to use the retention policy of `RUNTIME` for our `TestCase` annotation type, because it is meant for the test runner tool to read it at runtime. The `@Target` annotation states that the `TestCase` annotation can be used only to annotate methods. We will cover `@Retention` and `@Target` annotation types in detail in later sections when we discuss meta-annotations. The following snippet of code shows the use of our `TestCase` annotation type.

```
package com.jdojo.chapter2;

import java.io.IOException;

public class PolicyTestCases {
     // Must throw IOExceptionn
     @TestCase(willThrow=IOException.class)
     public static void testCase1(){
          // ...
     }

     // We are not expecting any exception
     @TestCase()
     public static void testCase2(){
          // ...
     }
}
```

The `testCase1()` method specifies, using the `@TestCase` annotation, that it will throw an `IOException`. The test runner tool will make sure that when it invokes this method, the method does throw an `IOException`. Otherwise, it will fail the test case. The `testCase2()` method does not specify that it will throw an exception. If it throws an exception, when the test is run, the tool should fail this test case.

Enum Type

An annotation can have elements of an enum type. Suppose you want to declare an annotation type called `Review` that can describe the code review status of a program element. Let us assume that it has a status element and it can have one of the four values – `PENDING`, `FAILED`, `PASSED`, `PASSEDWITHCHANGES`. You use the `String` data type for the status element and pass one of the four values as a `String` for the status element. Using the `String` type will not stop the user from specifying any value other than these four. Using enum type solves this problem. Note that an annotation type is an interface type with some restrictions. As you can declare an enum as an interface's member, so can you declare an enum as an annotation type's member. Your code is more maintainable and readable if you use an enum type element in an annotation and the enum type itself is defined as a member of the annotation type.

Listing 2-6: An annotation type, which uses an enum type element

```
// Review.java
package com.jdojo.chapter2;

public @interface Review {
```

```
    ReviewStatus status() default ReviewStatus.PENDING;
    String comments() default "";

    // ReviewStatus enum is a member of the Review annotation type
    public enum ReviewStatus {PENDING, FAILED,
                              PASSED, PASSEDWITHCHANGES};
}
```

Listing 2-6 shows code for our `Review` annotation type. It declares a `ReviewStatus` enum type and those four statuses are the elements of the enum. It has two elements – `status` and `comments`. The type of status element is `ReviewStatus` enum type. The default value for `status` element is `ReviewStatus.PENDING`. We have an empty string as the default value for the `comments` element. Here are some of the instances of the `Review` annotation type. You will need to import `com.jdojo.chapter2.Review.ReviewStatus` enum in your program to use the simple name of the `ReviewStatus` enum type.

```
// Have default for status and comments. Maybe code is new
@Review()

// Leave status as Pending, but add some comments
@Review(comments="Have scheduled code review on June 3 2009")

// Fail the review with comments
@Review(status=ReviewStatus.FAILED, comments="Need to handle errors")

// Pass the review without changes
@Review(status=ReviewStatus.PASSED)
```

Here is the sample code that annotates a `Test` class indicating that it has passed the code review.

```
import com.jdojo.chapter2.Review.ReviewStatus;

@Review(status=ReviewStatus.PASSED)
public class Test {
    // ...
}
```

Annotation Type

Since an annotation type is a type, it can be used anywhere a type can be used in a Java program. For example, you can use an annotation type as the return type for a method. You can also use an annotation type as the type of an element inside another annotation type's declaration. Suppose we want to have a new annotation type called `Description`, which will include the name of the author, version, and comments for a program element. We can reuse our `Name` and `Version` annotation types as its name and version elements type. Listing 2-7 has code the for `Description` annotation type.

Listing 2-7: An annotation type using other annotation types as data type of its elements

```
// Description.java
package com.jdojo.chapter2;

public @interface Description {
```

```
    Name name();
    Version version();
    String comments() default "";
}
```

To provide a value for an element of an annotation type, you need to use the syntax that is used to create an annotation type instance. For example, `@Version(major=1, minor=2)` creates an instance of the `Version` annotation Note the nesting of an annotation inside another annotation in the following snippet of code.

```
@Description(name=@Name(first="John", last="Jacobs"),
             version=@Version(major=1, minor=2),
             comments="Just a test class")
public class Test {
    // ...
}
```

Array Type Annotation Element

An annotation can have elements of an array type. The array type could be of one of the following types.

- A primitive type
- `java.lang.String` type
- `java.lang.Class`, type
- An `enum` type
- An annotation type

You need to specify the value for an array element inside braces. Elements of the array are separated by a comma. Suppose you want to annotate your program elements with a short description of a list of things that you need to work on. We will create a `ToDo` annotation type for this purpose as shown in Listing 2-8,

Listing 2-8: ToDo annotation type with String[] as its sole element

```
// ToDo.java
package com.jdojo.chapter2;

public @interface ToDo {
    String[] items();
}
```

The following snippet of code shows how to use a `@ToDo` annotation.

```
@ToDo(items={"Add readFile method", "Add error handling"})
public class Test {
    // ...
}
```

If you have only one element in the array, it is allowed to omit the braces. The following two annotation instances of the `ToDo` annotation type are equivalent.

```
@ToDo(items={"Add error handling"})
@ToDo(items="Add error handling")
```

TIP
If you do not have valid values to pass to an element of an array type, you can use an empty array. For example, `@ToDo(items={})` is a valid annotation where the `items` element has been assigned an empty array.

No null Value in an Annotation

You cannot use a `null` reference as a value for an element in an annotation. Note that it is allowed to use an empty string for the `String` type element and empty array for an array type element. Using the following annotations is invalid.

```
@ToDo(items=null)
@Name(first=null, last="Jacobs")
```

No Multiple Annotations of the Same Type

You cannot annotate a program element with more than one instance of the same annotation type. For example, the following snippet of code would not compile, because it uses the `@Name` annotation twice for the `Test` class. However, it is allowed to annotate a program element with multiple annotations of different types.

```
// Won't compile – Duplicate @Name annotation
@Name(first="John", last="Jacobs")
@Name(first="Wally", last="Inman")
public class WontCompile {
     //...
}

// Valid to have multiple annotation of different types
@Name(first="John", last="Jacobs")
@ToDo(items={"Optimize the code"})
@Version(major=2, minor=0)
public class Test {
     //...
}
```

However, such scenarios are not uncommon where you would like to use multiple annotations of the same type on the same program element. The solution is to create a new annotation type using an array of the same type as its element. Listing 2-9 has the code for a new annotation type named `Authors`. It uses an array of the `Name` annotation type as the type for its `authors` element.

Listing 2-9: An Authors annotation type using an array of Name annotation type as its element's data type

```
// Authors.java
package com.jdojo.chapter2;
```

```
public @interface Authors {
     Name[] authors();
}
```

We can rewrite the annotation for our `Test` class, which has two authors as follows. You need to be careful in positioning the braces and parentheses, when you specify values for an element whose type is an array of another annotation type.

```
@Authors(authors={@Name(first="John", last="Jacobs"),
                  @Name(first="Wally", last="Inman")})
public class Test {
     //...
}
```

Here again, you can omit the braces when you specify only one element in the array for the `authors` elements as shown below. This is the shorthand for specifying the value for an array type element of an annotation.

```
@Authors(authors=@Name(first="John", last="Jacobs"))
public class Test {
     //...
}
```

Shorthand Annotation Syntax

The shorthand annotation syntax is a little easier to use in a few circumstances. Suppose you have an annotation type `ABC`, whose all elements have a default value as shown below.

```
public @interface ABC {
     int e1() default 0;
     String e2 default = “”:
}
```

If you want to annotate a program element with the `ABC` annotation using default values for all its elements, you can use the `@ABC()` syntax. You do not need to specify the values for `e1` and `e2` elements, because they have default values. You can use shorthand in this situation, which allows you to omit the parentheses. You can just use `@ABC` instead of `@ABC()`. The `ABC` annotation can be used in either of the two forms as shown below.

```
@ABC
public class Test {
     //...
}

@ABC()
public class Test {
     //...
}
```

An annotation type with only one element also has a shorthand syntax. You can use this shorthand as long as you adhere to a naming rule for the sole element you have in your annotation type. The name of the element must be `value`. If you name the only element in your annotation type as

value, you can omit the name from name=value pair from your annotation. The following snippet of code declares a Company annotation type, which has only one element named value.

```
public @interface Company {
     String value(); // the element's name is value
}
```

You can omit the name from name=value pair, when you use the Company annotation as shown below. If you want to use the element name with the Company annotation, you can always do so as @Company(value="ABC Inc.").

```
// No need to specify name of the element. It is assumed to be value
@Company("ABC Inc.")
public class Test {
     //...
}
```

You can use this shorthand of omitting the name of the element from annotations, even if the element data type is an array. Let us consider the following annotation type Reviewers.

```
public @interface Reviewers {
     String[] value(); // element name is value
}
```

Since the Reviewers annotation type has only one element, which is named value, you can omit the element name when you are using it as shown below.

```
// No need to specify name of the element
@Reviewers({"John Jacobs", "Wally Inman"})
public class Test {
     //...
}
```

You can also omit the braces, if you specify only one element in the array for the value element of the Reviewers annotation type as:

```
@Reviewers("John Jacobs")
public class Test {
     //...
}
```

We had several examples using the name of the element as value. Here is the general rule of omitting the name of the element in an annotation. If you supply only one value when using an annotation, the name of the element is assumed value. This means that you are not required to have only one element in the annotation type, which is named value to omit its name in the annotations. If you have an annotation type, which has an element named value (with or without a default value) and all other elements have default values, you can still omit the name of the element in annotation instances of this type. Here are some examples to illustrate this rule.

```
public @interface A {
     String value();
     int id() default 10;
}
```

```
// Same as @A(value="Hello", id=10)
@A("Hello")
public class Test {
     //...
}

// Won't compile. Must use only one value to omit the element name
@A("Hello", id=16)
public class WontCompile {
     //...
}

// OK. Must use name=value pair when passing more than one value
@A(value="Hello", id=16)
public class Test {
     //...
}
```

Marker Annotation Types

A marker annotation type is an annotation type that does not declare any elements, not even one with a default value. Typically, a marker annotation is used by the annotation processing tools. The tools generate boilerplate code based on the marker annotation type.

```
public @interface Marker{
     // No elements declarations
}

@Marker
public class Test{
     //...
}
```

Meta-Annotation Types

Meta-annotations types are annotation types, which are used to annotate other annotation types. The following annotation types are meta-annotation types.

- `java.lang.annotation.Target`
- `java.lang.annotation.Retention`
- `java.lang.annotation.Inherited`
- `java.lang.annotation.Documented`

Meta-annotation types are part of the Java class library. They are declared in the `java.lang.annotation` package. Note that using an annotation in your program is the same as using any other type (e.g. class, interface, enum, etc.) in your program. You must either use the fully qualified name of the annotation type or use appropriate import statements to use their simple names.

The java.lang.annotation.Target Annotation

The `Target` annotation is used to annotate an annotation type to specify on what types of program elements the annotation type can be used. It has only one element named `value`. Its `value` element is an array of `java.lang.annotation.ElementType` enum type. Table 2-1 lists all constants in the `ElementType` enum.

Table 2-1: List of constants in the java.lang.annotation.ElementType enum

Constant Name	Description
`ANNOTATION_TYPE`	The annotation can be used to annotate another annotation type declaration. This makes the annotation type a meta-annotation.
`CONSTRUCTOR`	The annotation can be used to annotate constructors.
`FIELD`	The annotation can be used to annotate fields and enum constants.
`LOCAL_VARIABLE`	The annotation can be used to annotate local variables.
`METHOD`	The annotation can be used to annotate methods.
`PACKAGE`	The annotation can be used to annotate package declarations.
`PARAMETER`	The annotation can be used to annotate parameters.
`TYPE`	The annotation can be used to annotate class, interface (including annotation type), or enum declarations.

The following code annotates a `Version` annotation type with the `Target` meta-annotation, which specifies that the `Version` annotation type can be used with program elements of only three types – any types (class, interface, enum, and annotation types), a constructor, and a method.

```
// Version.java
package com.jdojo.chapter2;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Target({ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface Version {
     int major();
     int minor();
}
```

The `Version` annotation cannot be used on any program elements other than the three types specified in its `Target` annotation. The following use of the `Version` annotation is incorrect, because it is being used on an instance variable (a field).

```
public class WontCompile {
     // Version annotation cannot be used on a field
     @Version(major = 1, minor = 1)
     int xyz = 110;
}
```

The following uses of the `Version` annotation are valid.

```
// OK. Class type declaration
```

```
@Version(major = 1, minor = 0)
public class VersionTest {

     // OK. Constructor declaration
     @Version(major = 1, minor = 0)
     public Test(){
          //...
     }

     // OK. Method declaration
     @Version(major = 1, minor = 1)
     public void doSomething() {
          //...
     }
}
```

TIP
If you do not annotate an annotation type with the `Target` annotation, the annotation type can be used to annotate any program element declaration.

The java.lang.annotation.Retention Annotation

You can use annotations for different purposes. You may want to use annotations solely for documentation purposes, to be processed by the compiler, and/or to use them at runtime. An annotation can be retained at three levels.

- Source code only
- Class file only (the default)
- Class file and the runtime

The `Retention` meta-annotation is used to specify how an annotation instance of an annotation type should be retained by Java. This is also known as the retention policy of an annotation type. If an annotation type has a “source code only” retention policy, instances of its type are removed when compiled into a class file. If the retention policy is “class file only”, annotation instances are retained in the class file, but they cannot be read at runtime. If the retention policy is “class file and runtime” (simply known as runtime), the annotation instances are retained in the class file and are available for read at runtime.

The `Retention` meta-annotation type declares one element, which is named `value` and is of the `java.lang.annotation.RetentionPolicy` enum type. The `RetentionPolicy` enum has three constants – `SOURCE`, `CLASS`, and `RUNTIME` and they are used to specify the retention policy of source only, class only, and class-and-runtime respectively. The following code uses the `Retention` meta-annotation on the `Version` annotation type. It specifies that the `Version` annotations should be available at runtime. Note that we have used two meta-annotations on the `Version` annotation type: `Target` and `Retention`.

```
// Version.java
package com.jdojo.chapter2;

import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.TYPE, ElementType.CONSTRUCTOR,
         ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Version {
     int major();
     int minor();
}
```

TIP
If you do not use `Retention` meta-annotation on an annotation type, its retention policy defaults to class file only. This implies that you will not be able to read those annotations at runtime. You will make this common mistake in the beginning. You would try to read annotations and the runtime will not return any values. Make sure that your annotation type has been annotated with the `Retention` meta-annotation with the retention policy of `RetentionPolicy.RUNTIME`, before you attempt to read annotations at runtime. An annotation on a local variable declaration is never available in the class file or at runtime irrespective of the retention policy of the annotation type.

The java.lang.annotation.Inherited Annotation

The `Inherited` is a marker meta-annotation type. If an annotation type is annotated with an `Inherited` meta-annotation, its instances are inherited by a subclass declaration. It has no effect if an annotation type is used to annotate any program elements other than a class declaration. Let us consider two annotation type declarations: `Ann2` and `Ann3`. Note that `Ann2` is not annotated with an `Inherited` meta-annotation, whereas `Ann3` is annotated with an `Inherited` meta-annotation.

```
public @interface Ann2 {
     int id();
}

@Inherited
public @interface Ann3 {
     int id();
}
```

Let us declare two classes – `A` and `B` as shown below. Note that class `B` inherits class `A`.

```
@Ann2(id=505)
@Ann3(id=707)
public class A {
     // Code for class A goes here
}

// Class B inherits Ann3(id=707) annotation from the class A
public class B extends A {
     // Code for class B goes here
}
```

In the above snippet of code, class `B` inherits the `@Ann3(id=707)` annotation from class `A`, because the `Ann3` annotation type has been annotated with an `Inherited` meta-annotation. The class `B` does not inherit the `@Ann2(id=505)` annotation, because the `Ann2` annotation type is not annotated with an `Inherited` meta-annotation.

The java.lang.annotation.Documented Annotation

The `Documented` annotation is a marker meta-annotation type. If an annotation type is annotated with a `Documented` annotation, the Javadoc tool will generate documentation for all of its instances. Listing 2-10 has the code for our final version of the `Version` annotation type, which has been annotated with a `Documented` meta-annotation.

Listing 2-10: Our final version of the Version annotation type

```
// Version.java
package com.jdojo.chapter2;

import java.lang.annotation.Documented;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Version {
     int major();
     int minor();
}
```

Suppose we annotate a `Test` class with our `Version` annotation type as follows.

```
package com.jdojo.chapter2;

@Version(major=1, minor=0)
public class Test {
     // Code for Test class goes here
}
```

When we generate documentation for the `Test` class using the Javadoc tool, the `Version` annotation on the `Test` class declaration is also generated as part of the documentation. If you remove the `Documented` annotation from the `Version` annotation type declaration, the `Test` class documentation would not contain information about its `Version` annotation.

Commonly Used Standard Annotations

Java API defines many standard annotation types. This section discusses three most commonly used standard annotations. They are defined in the `java.lang` package. They are:

- java.lang.Deprecated
- java.lang.Override
- java.lang.SuppressWarnings

The Deprecated Annotation Type

It is a marker annotation type. Developers are discouraged to use a program element annotated with a `Deprecated` annotation, because it is not safe to use the program element anymore or a better alternative exists. If you use a deprecated program element in a non-deprecated code, the compiler will generate a warning. Suppose we have a `DeprecatedTest` class as follows. Note that we have annotated the class with a `@Deprecated` annotation. Its `getInstance()` method uses the class type as its return type, which will not generate a compiler warning, because it is inside the deprecated class.

Listing 2-11: An example of deprecating a class named DeprecatedTest

```
// DeprecatedTest.java
package com.jdojo.chapter2;

@Deprecated
public class DeprecatedTest {
     private DeprecatedTest() {
     }

     public static DeprecatedTest getInstance() {
          // Using the deprecated class inside its own body
          DeprecatedTest dt = new DeprecatedTest();
          return dt;
     }
}
```

Let us attempt to use the `DeprecatedTest` class inside a new class called `Test` as shown below. When you compile the `Test` class, it will generate a compiler warning stating the fact that the deprecated `DeprecatedTest` class should not be used.

```
package com.jdojo.chapter2;

public class Test {
     public static void main(String[] args) {
          DeprecatedTest dt; // Generates a compiler warning
     }
}
```

```
Compiler Output:

Warning(5):  [deprecation] com.jdojo.chapter2.DeprecatedTest in
com.jdojo.chapter2 has been deprecated
```

The Override Annotation Type

It is a marker annotation type. It can only be used on methods. It indicates that a method annotated with this annotation overrides a method declared in its supertype. In Java 5, it could be used only in class methods. In Java 6, it can be used for methods of any type. This is very helpful for developers to avoid typo that leads to logical errors in the program. If you mean to override a method from a supertype, it is recommended to annotate the overridden method with a `@Override` annotation. The compiler will make sure that the annotated method really overrides a method in the supertype. If the annotated method does not override a method in the supertype, the compiler will generate an error.

Consider two classes - `A` and `B`. Class `B` inherits from class `A`. The `m1()` method in the class `B` overrides the `m1()` method in its superclass `A`. The annotation `@Override` on the `m1()` method in the class `B` just makes a statement about this intention. The compiler verifies this statement and finds it to be true in this case.

```
public class A {
     public void m1() {
     }
}

public class B extends A {
     @Override
     public void m1() {
     }
}
```

Let us consider class `C` as declared below.

```
// Won't compile because m2() does not override any method
public class C extends A {
     @Override
     public void m2() {
     }
}
```

The `m2()` method of the class `C` has an `@Override` annotation. However, there is no `m2()` method in its superclass `A`. The `m2()` method is a new method declaration in the class `C`. The compiler finds out that the `m2()` method in the class `C` does not override any superclass method, even though its developer has indicated so. The compiler generates an error in this case.

The SuppressWarnings Annotation Type

The `SuppressWarnings` is used to suppress named compiler warnings. It declares one element, which is named `value` whose data type is an array of `String`. Let us consider the code for the `SuppressWarningsTest` class, which uses the `List` interface as its raw type instead of its generic type in the `m1()` method. The compiler generates `unchecked` named warning, when you use a raw collection type.

```
// SuppressWarningsTest.java
package com.jdojo.chapter2;

import java.util.ArrayList;
```

```
import java.util.List;

public class SuppressWarningsTest {
     public void m1() {
          List list = new ArrayList();
          list.add("Hello"); // The compiler issues an unchecked warning
     }
}
```

Let us compile our `SuppressWarningsTest` class with an option to generate an unchecked warning as shown below.

```
javac -Xlint:unchecked com/jdojo/chapter2/SuppressWarningsTest.java
```

```
Compiler Warnings:

com\jdojo\chapter2\SuppressWarningsTest.java:10: warning: [unchecked]
unchecked
call to add(E) as a member of the raw type java.util.List
        list.add("Hello"); // Compiler will issue unchecked warning
                ^
1 warning
```

As a developer, sometimes you are aware of such compiler warnings and you want to suppress them, when your code is compiled. You can do so by using a `@SuppressWarnings` annotation on your program element by supplying a list of the names of the warnings to be suppressed. For example, if you use it on a class declaration, all specified warnings will be suppressed from all methods inside that class declaration. It is recommended that you use this annotation on the inner most program element on which you want to suppress the warnings. The following snippet of code uses a `SuppressWarnings` annotation on the `m1()` method. It specifies two named warnings: `unchecked` and `deprecated`. The `m1()` method does not contain code that will generate a `deprecated` warning. It was included here to show you that you could suppress multiple named warnings using a `SuppressWarnings` annotation. If you recompile the `SuppressWarningsTest` class with the same options as shown above, it will not generate any compiler warnings.

```
// SuppressWarningsTest.java
package com.jdojo.chapter2;

import java.util.ArrayList;
import java.util.List;

public class SuppressWarningsTest {
     @SuppressWarnings({"unchecked", "deprecation"})
     public void m1() {
          List list = new ArrayList();
          list.add("Hello"); // Will not issue unchecked warning
     }
}
```

Annotating a Java Package

Annotating program elements such as classes and fields is intuitive as we annotate them when they are declared. How do we annotate a package? A package declaration appears as part of a top-level type declaration. Further, the same package declaration occurs multiple times at different places. The question arises - how and where do we annotate a package declaration?

You need to create a file, which should be named `package-info.java`, and place the annotated package declaration in it. Listing 2-12 shows the content of the `package-info.java` file. When you compile the `package-info.java` file, a class file will be created.

Listing 2-12: Contents of a package-info.java file

```
// package-info.java
@Version(major=1, minor=0)
package com.jdojo.chapter2;
```

You may need some import statement to import annotation types or you can use the fully qualified names of the annotation types in the `package-info.java` file. Even though the import statement appears after the package declaration, you should be fine to use the imported types. You can have contents like the one shown below in a `package-info.java` file.

```
// package-info.java
@com.jdojo.myannotations.Author("John Jacobs")
@Reviewer("Wally Inman")
package com.jdojo.chapter2;

import com.jdojo.myannotations.Reviewer;
```

Accessing Annotations at Runtime

Accessing annotation on a program element is easy. Annotations on a program element are Java objects. All you need to know is how to get the reference of objects of an annotation type at runtime. Program elements, which let you access their annotations, implement the `java.lang.reflect.AnnotatedElement` interface. There are four methods in the `AnnotatedElement` interface. You will need to use one or more of these to access annotations of a program element. The following classes implement the `AnnotatedElement` interface.

- `java.lang.Class`
- `java.lang.reflect.Constructor`
- `java.lang.reflect.Field`
- `java.lang.reflect.Method`
- `java.lang.Package`

Table 2-2 lists the four methods of the `AnnotatedElement` interface with their short descriptions. Note that three of the four methods return an object or an array of objects of the `Annotation` interface type. Recall that `Annotation` is an interface and it is the implicit supertype of all annotation types in Java.

Table 2-2: List of methods in the java.lang.reflect.AnnotatedElement interface and their descriptions

Method Name	Description
`<T extends Annotation> T getAnnotation(Class<T> annotationClass)`	It returns a program element's annotation of the specified `annotationClass` type. If the program element does have an annotation of the specified `annotationClass` type, it returns `null`.
`Annotation[] getAnnotations()`	It returns all annotations present on a program element. If the program element does not have any annotations, it returns an array of length zero.
`Annotation[] getDeclaredAnnotations()`	It returns all annotations that are directly present on a program element. It ignores inherited annotations. If a program element does not have any annotation present on it, it returns an array of length zero.
`boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)`	It returns `true` if an annotation of the specified `annotationClass` is present on a program element. Otherwise, it returns `false`. In the case of marker annotations, all you need to know is if they are present on a program element or not. This is the method that you need to use if you just want to check if a marker annotation is present on a program element.

TIP
It is very important to note that an annotation type must be annotated with the `Retention` meta-annotation with retention policy of runtime to access them at runtime. If a program element has multiple annotations, you would be able to access only annotations, which have runtime as their retention policy.

Suppose you have a `Test` class and you want to print all its annotations. The following snippet of code will print all annotations on the class declaration of the `Test` class.

```
// Get the class object reference
Class c = Test.class;

// Get all annotations on the class declaration
Annotation[] ann = c.getAnnotations();
System.out.println("Annotation count: " + ann.length);

// Print all annotations
for(int i = 0; i < ann.length; i++) {
     System.out.println(ann[i]);
}
```

The `toString()` method of the `Annotation` interface returns the string representation of an annotation. Suppose you want to print the `Version` annotation on the `Test` class. You can do so

as follows. The following code shows that you can use the `major()` and `minor()` methods. It also shows that you can declare a variable of an annotation type (e.g. `Version v`), which can refer to an instance of that annotation type. The instances of an annotation type are created by the Java runtime. You never create an instance of an annotation type using the `new` operator.

```
Class<Test> c = Test.class;

// Get the instance of the Version annotation of Test class
Version v = c.getAnnotation(Version.class);
if (v == null) {
     System.out.println("Version annotation is not present.");
}
else {
     int major = v.major();
     int minor = v.minor();
     System.out.println("Version: major=" + major + ", minor=" + minor);
}
```

We will use the `Version` and `Deprecated` annotation types to annotate our program elements, and access those annotations at runtime. We will also annotate a package declaration and a method declaration. We will use the code for the `Version` annotation type as listed in Listing 2-13. Note that it uses `@Retention(RetentionPolicy.RUNTIME)` annotation, which is needed to read its instances at runtime. Listing 2-14 shows the code that you need to save in a `package-info.java` file and compile it along with other programs. It annotates `com.jdojo.chapter2` package. Listing 2-15 has the code for a class for our demonstration purpose that has some annotations. Listing 2-16 is the program that demonstrates how to access annotations at runtime. Its output shows that we are able to read all annotations used in the `AccessAnnotation` class successfully. The `printAnnotations()` method accesses the annotations. It accepts a parameter of the `AnnotatedElement` type and prints all annotations of its parameter. If the annotation is of the `Version` annotation type, it prints the values for its major and minor versions.

Listing 2-13: A Version annotation type

```
// Version.java
package com.jdojo.chapter2;

import java.lang.annotation.Documented;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Target({ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Version {
     int major();
     int minor();
}
```

Listing 2-14: Contents of package-info.java file

```
//package-info.java
@Version(major=1, minor=0)
package com.jdojo.chapter2;
```

Listing 2-15: AccessAnnotation class has some annotations, which will be accessed at runtime

```
// AccessAnnotation.java
package com.jdojo.chapter2;

@Version(major=1, minor=0)
public class AccessAnnotation {
     @Version(major=1, minor=1)
     public void testMethod1() {
          // ...
     }

     @Version(major=1, minor=2)
     @Deprecated
     public void testMethod2() {
          // ...
     }
}
```

Listing 2-16: Using the AccessAnnotationTest class to access annotations

```
// AccessAnnotationTest.java
package com.jdojo.chapter2;

import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
import java.lang.reflect.Method;

public class AccessAnnotationTest {
     public static void main(String[] args) {
          // Read annotation of class declaration
          Class<AccessAnnotation> c = AccessAnnotation.class;
          System.out.println("Annotations for class:" + c.getName());
          printAnnotations(c);

          // Read annotation of package declaration
          Package p = c.getPackage();
          System.out.println("Annotations for package:" + p.getName());
          printAnnotations(p);

          // Read annotation of method declaration
          System.out.println("Method annotations:");
          Method[] m = c.getDeclaredMethods();
          for (int i = 0; i < m.length; i++) {
               System.out.println("Annotations for method:" +
                                  m[i].getName());
               printAnnotations(m[i]);
          }
     }

     public static void printAnnotations(
               AnnotatedElement programElement) {
          Annotation[] annList = programElement.getAnnotations();
          for (int i = 0; i < annList.length; i++) {
               System.out.println(annList[i]);
               if (annList[i] instanceof Version) {
                    Version v = (Version)annList[i];
```

```
                    int major = v.major();
                    int minor = v.minor();
                    System.out.println("Found Version annotation: "  +
                          "major =" + major + ", minor=" + minor);
               }
          }
          System.out.println();
     }
}
```

```
Output:

Annotations for class:com.jdojo.chapter2.AccessAnnotation
@com.jdojo.chapter2.Version(major=1, minor=0)
Found Version annotation: major =1, minor=0

Annotations for package:com.jdojo.chapter2
@com.jdojo.chapter2.Version(major=1, minor=0)
Found Version annotation: major =1, minor=0

Method annotations:
Annotations for method:testMethod1
@com.jdojo.chapter2.Version(major=1, minor=1)
Found Version annotation: major =1, minor=1

Annotations for method:testMethod2
@com.jdojo.chapter2.Version(major=1, minor=2)
Found Version annotation: major =1, minor=2
@java.lang.Deprecated()
```

TIP
You can also access annotations of program elements at compile time. Java 5 introduced a tool called Annotation Processing Tool (APT) to process annotations at compile-time. Java 6 has included APT in Java API library in `javax.annotation.processing` package.

An Annotation Type may Grow

Unlike an interface type, an annotation type can grow without breaking the existing code that uses it. If you add a new element to an annotation type, you must supply its default value. All existing instances of the annotation will use the default value for the new elements. If you add a new element to an existing annotation type without specifying a default value for the element, the code that that uses the annotation will break.

Annotation Processing at Source Code Level

Java lets you process annotations at runtime as well as at compile-time. We have already seen how to process annotations at runtime. Now, we will discuss, in brief, how to process annotations at compile-time (or at source code level).

Java 5 shipped with a command-line utility called Annotation Processing Tool (APT) that let you process annotations at source code level at compile-time. You needed to write custom annotation processor using non-public API that was part of the APT. APT would let you hook the custom processor to it. Java 6 has APT incorporated in the `javac` command-line compiler. The API to write your custom processor is part of the standard Java library in Java 6. This section discusses how to write and use a custom annotation processor using Java 6.

Why would you want to process annotations at compile-time? Processing annotations at compile-time opens up a wide variety of possibilities that can help Java programmers in an application development. It also helps developers of Java tools immensely. For example, boilerplate code and configuration files can be generated based on annotations in the source code; custom annotation-based rules can be validated at the compiled-time, etc.

Annotation processing at compile time is a two-step process. First, you need to write a custom annotation processor. Second, you need to use the `javac` command line utility tool. You need to pass your custom annotation processor to the `javac` compiler using the `-processor` option. You can pass multiple custom annotation processors to `javac` separating them by a comma. The following command compiles the Java source file, `MySourceFile.java`, and passes two custom annotation processors: `MyProcessOr1` and `MyProcessor2`.

```
javac -processor MyProcessor1,MyProcessor2 MySourceFile.java
```

Using `-proc` option, the `javac` command-line utility lets you specify if you want to process annotation and/or compile the source files. You can use `-proc` option as `-proc:none` or `-proc:only`. The `-proc:none` option does not perform annotation processing. It only compiles source files. The `-proc:only` option performs only annotation processing and skips the source files compilation. If the `-proc:none` and the `-processor` options are specified in the same command, the `-processor` option is ignored. The following command processes annotations in the source file `MySourceFile.java` using custom processors: `MyProcessor1` and `MyProcessors2`. It does not compile the source code in the `MySourceFile.java` file.

```
javac -proc:only -processor MyProcessor1,MyProcessor2 MySourceFile.java
```

To see the compile-time annotation processing in action, you must write an annotation processor using the classes in the `javax.annotation.processing` package.

While writing a custom annotation processor, you often need to access the elements from the source code, e.g., name of a class and its modifiers, name of a method and its return types, etc. You will need to use classes in the `javax.lang.model` package and its subpackages to work with the elements of the source code. In our example, we will write an annotation processor for our `@Version` annotation. It will validate all `@Version` annotations that are used in the source code to make sure the `major` and `minor` values for a `Version` are always zero or greater than zero. For example, if `@Version(major=-1, minor=0)` is used in a source code, our annotation processor will print an error message, because the `major` value for the version is negative.

An annotation processor is an object of a class, which implements the `Processor` interface. The `AbstractProcessor` class is an abstract annotation processor, which provides a default implementation for all methods of the `Processor` interface, except an implementation for the `process()` method. The default implementation in the `AbstractProcessor` class is good enough in most of the circumstances. To create your own processor, you need to inherit your processor class from the `AbstractProcessor` class and provide an implementation for the `process()` method. If the `AbstractProcessor` class does not suit your need, you can create

your own processor class, which implements the `Processor` interface. Let us call our processor class `VersionProcessor`, which inherits the `AbstractProcessor` class as shown below.

```
public class VersionProcessor extends AbstractProcessor {
     // Code goes here
}
```

The annotation processor object is instantiated by the compiler using a no-args constructor. You must have a no-args constructor for your processor class, so that the compiler can instantiate it. The default constructor for our `VersionProcessor` class will meet this requirement.

The next step is to add two pieces of information to the processor class. The first one is about what kind of annotations processing are supported by this processor. You can specify the supported annotation type using `@SupportedAnnotationTypes` annotation at class level. The following snippet of code shows that the `VersionProcessor` supports processing of `com.jdojo.chapter2.Version` annotation type.

```
@SupportedAnnotationTypes({"com.jdojo.chapter2.Version"})
public class VersionProcessor extends AbstractProcessor {
     // Code goes here
}
```

You can use an asterisk (`*`) by itself or as a part of the annotation name of the supported annotation types. The asterisk works as a wild card. For example, `"com.jdojo.*"` means any annotation types whose names start with `"com.jdojo."`. An asterisk only, `"*"`, means all annotation types. Note that when an asterisk is used as part of the name, the name must be of the form `PartialName.*`. For example, `"com*"` and `"com.*jdojo"` are invalid uses of an asterisk in the supported annotation types. You can pass multiple supported annotation types using an `SupportedAnnotationTypes` annotation. The following snippet of code shows that the processor supports processing for the `com.jdojo.Ann1` annotation and any annotations whose name begins with `com.jdojo.chapter2`.

```
@SupportedAnnotationTypes({"com.jdojo.Ann1", "com.jdojo.chapter2.*"})
```

You need to specify the latest source code version that is supported by your processor using an `@SupportedSourceVersion` annotation. If you do not specify the source code version, the `AbstractProcessor` class defaults it to the source code version 6 and prints a warning. The following snippet of code specifies the source code version 5 as the supported source code version for the `VersionProcessor` class.

```
@SupportedAnnotationTypes({"com.jdojo.chapter2.Version"})
@SupportedSourceVersion(SourceVersion.RELEASE_5)
public class VersionProcessor extends AbstractProcessor {
     // Code goes here
}
```

The next step is to provide the implementation for the `process()` method in your processor. Annotation processing is performed in rounds. An instance of the `RoundEnvironment` interface represents a round. The `javac` compiler calls the `process()` method of your processor by passing all annotations that the processor declares to support and a `RoundEnvironment` object. The return type of the `process()` method is `boolean`. If it returns `true`, the annotations passed to it are considered to be claimed by the processor. The claimed annotations are not passed to other processors. If it returns `false`, the annotations passed to it are considered as not claimed

and other processor will be asked to process them. The following snippet of code shows the skeleton of the `process()` method.

```
public boolean process(Set<? extends TypeElement> annotations,
                       RoundEnvironment roundEnv) {
     // The processor code goes here
}
```

The code you write inside the `process()` method of your processor depends on your requirements. In our case, we want to look at the major and minor values for each `@Version` annotation in the source code. If either of them is less than zero, we want to print an error message. To process each `Version` annotation, we will iterate through all `Version` annotation instances passed to the `process()` method as:

```
for (TypeElement currentAnnotation: annotations) {
     // Code to validate each Version annotation goes here
}
```

You can get the fully qualified name of an annotation using the `getQualifiedName()` method of the `TypeElement` interface as:

```
Name qualifiedName = currentAnnotation.getQualifiedName();

// Check if it is a Version annotation
if (qualifiedName.contentEquals("com.jdojo.chapter2.Version")) {
     // Get Version annotation values to validate
}
```

Once you are sure that you have a `Version` annotation, you need to get all its instances from the source code. To get information from the source code, you need to use the `RoundEnvironment` object passed to the `process()` method. The following snippet of code will get all elements of the source code (e.g. classes, methods, constructors, etc.) that are annotated with a `Version` annotation.

```
Set<? extends Element> annotatedElements
          = roundEnv.getElementsAnnotatedWith(currentAnnotation);
```

At this point, we need to iterate through all elements that are annotated with a `Version` annotation; get the instance of the `Version` annotation present on them; and, validate the values of the `major` and `minor` elements. We can perform this logic as follows.

```
for (Element element : annotatedElements) {
     Version v = element.getAnnotation(Version.class);
     int major = v.major();
     int minor = v.minor();
     if (major < 0 || minor < 0) {
          // Print the error message here
     }
}
```

You can print the error message using the `printMessage()` method of the `Messager` object. The `processingEnv` is an instance variable defined in the `AbstractProcessor` class that you can use inside your processor to get the `Messager` object reference as shown below. If you pass

the source code element's reference to the `printMessage()` method, your message will be formatted to include the source code file name and the line number in the source code for that element. The first argument to the `printMessage()` method indicates the type of the message. You can use `Kind.NOTE` and `Kind.WARNING` as the first argument to print a note and warning respectively.

```
String errorMsg = "Version cannot be negative. " +
                  "major=" + major + " minor=" + minor;
Messager messager = this.processingEnv.getMessager();
messager.printMessage(Kind.ERROR, errorMsg, element);
```

Finally, you need to return `true` or `false` from the `process()` method. If a processor returns `true` it means it claimed all the annotations that were passed to it. Otherwise, those annotations are considered unclaimed and they will be passed to other processors. Listing 2-17 has the complete code for the `VersionProcessor` class.

Listing 2-17: An annotation processor to process Version annotations using the AbstractProcessor class in Java 6

```
// VersionProcessor.java
package com.jdojo.chapter2;

import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;

import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.Name;
import javax.lang.model.element.TypeElement;

import javax.tools.Diagnostic.Kind;

@SupportedAnnotationTypes({"com.jdojo.chapter2.Version"})
@SupportedSourceVersion(SourceVersion.RELEASE_5)
public class VersionProcessor extends AbstractProcessor {

     // A no-args constructor is required for an annotation processor
     public VersionProcessor() {
     }

     public boolean process(Set<? extends TypeElement> annotations,
               RoundEnvironment roundEnv) {

          // Process all annotations
          for (TypeElement currentAnnotation: annotations) {
               Name qualifiedName =
                    currentAnnotation.getQualifiedName();

               // check if it is a Version annotation
               if (qualifiedName.contentEquals(
                    "com.jdojo.chapter2.Version" )) {
```

```
                    // Look at all elements that have Version annotations
                    Set<? extends Element> annotatedElements;
                    annotatedElements =
                         roundEnv.getElementsAnnotatedWith(
                                   currentAnnotation);
                    for (Element element: annotatedElements) {
                         Version v =  element.getAnnotation(Version.class);
                         int major = v.major();
                         int minor = v.minor();
                         if (major < 0 || minor < 0) {
                              // Print the error message
                              String errorMsg = "Version cannot" +
                                             " be negative." +
                                             " major=" + major +
                                             " minor=" + minor;

                              Messager messager =
                                   this.processingEnv.getMessager();

                              messager.printMessage(Kind.ERROR,
                                             errorMsg, element);
                         }
                    }
               }
          }

          return true;
     }
}
```

Now, we have an annotation processor. It is time to see it in action. We need to have a source code that uses invalid values for the major and minor elements in the `Version` annotation. The `VersionProcessorTest` class in Listing 2-18 uses the `Version` annotation three times. It uses negative values for major and minor elements for the class itself and for the method `m2()`. Our processor should catch these two errors, when we compile the source code for the `VersionProcessorTest` class.

Listing 2-18: A test class to test VersionProcessor. It uses negative values for major and minor values for the Version annotation, so that the processor should generate error messages.

```
// VersionProcessorTest.java
package com.jdojo.chapter2;

import java.lang.annotation.Annotation;

@Version(major = -1, minor = 2)
public class VersionProcessorTest {
     @Version(major = 1, minor = 1)
     public void m1() {
     }

     @Version(major = -2, minor = 1)
     public void m2() {
     }
}
```

To see the processor in action, we need to run the following command. Note that the command text is entered on one line and not two lines as shown below. Make sure the `VersionProcessor` class is compiled and it is available in the `CLASSPATH`.

```
javac -processor com.jdojo.chapter2.VersionProcessor
com\jdojo\chapter2\VersionProcessorTest.java
```

The output of the above command is as follows. The output displays two errors with the source file name and the line number at which errors were found in the source file.

```
Output:

com\jdojo\chapter2\VersionProcessorTest.java:10: Version cannot be
negative. major=-1 minor=2
public class VersionProcessorTest {
       ^
com\jdojo\chapter2\VersionProcessorTest.java:16: Version cannot be
negative. major=-2 minor=1
    public void m2() {
                ^
2 errors
```

Chapter 4. Enum

What is an Enum Type?

Java 5.0 introduced a new type called enum (enumeration or enumerated data type), which lets you create an ordered list of constants as a type. Before we discuss what an enum is and why we need it, let us consider a problem and solve it using Java features that were available before version 5.0. Suppose you are working on a defect tracking application in which you need to represent the severity of a defect. There are four types of severity that your application can handle. The four types of severity are `LOW`, `MEDIUM`, `HIGH` and `URGENT`. The typical way to represent the four types of severity before Java 5.0 was to declare four `int` constants in a class, say `Severity`, as shown below.

```
public class Severity {
     public static final int LOW = 0;
     public static final int MEDIUM = 1;
     public static final int HIGH = 2;
     public static final int URGENT = 3;
}
```

Suppose we want to write a utility class named `DefectUtil` that has a method to compute the projected turnaround days for a defect based on its severity. The code for the `DefectUtil` class may look as shown below.

```
public class DefectUtil {
     public static int getProjectedTurnAroundDays(int severity) {
          int days = 0;

          switch (severity) {
               case Severity.LOW:
                    days = 30;
                    break;
               case Severity.MEDIUM:
                    days = 15;
                    break;
               case Severity.HIGH:
                    days = 7;
                    break;
               case Severity.URGENT:
                    days = 1;
                    break;
          }

          return days;
     }

     // Other code for the DefectUtil class goes here
}
```

The following are a few problems with this approach in handling the severity of a defect.

- Since a severity is represented as an integer constant, you can pass any integer value to the `getProjectedTurnAroundDays()` method and not just 0, 1, 2, and 3, which are the valid values for the severity type. You may want to add a check inside this method, so that only valid severity values can be passed to it. Otherwise, the method may throw an exception. However, that does not solve the problem forever. You will need to keep updating your code that checks for valid severity values whenever you change the list of the severity types.
- If you change the value for a severity constant, you must recompile the code that uses it to reflect the changes. When you compile `DefectUtil` class, `Severity.LOW` is replaced with 0, `Severity.MEDIUM` is replaced with 1 and so on. If you change the value for the constant `LOW` in the `Severity` class to 10, you must recompile the `DefectUtil` class to reflect this change. Otherwise, `DefectUtil` class will still keep using the value `1`.
- When you save the value of the severity on the disk, its corresponding integer value will be saved, e.g., `0`, `1`, `2`, etc., and not its string value, e.g., `LOW`, `MEDIUM`, `HIGH`, etc. You must maintain a separate map to convert an integer value to its corresponding string representation for all severity types.
- When you print the severity value of a defect, it will print an integer value, e.g., 0, 1, 2, etc. An integer value for a severity does not mean anything to the users.
- Note that the severity types of the defects have a specific order. For example, a `LOW` severity defect is given less priority than a `MEDIUM` severity defect. Since a severity is being represented by an arbitrary number, you must write code using hard-coded values to maintain the order of the constants defined in the `Severity` class. Suppose we add another severity type `VERY_HIGH`, which has less priority than `URGENT` and more priority than `HIGH`. Now you must change the code that handles ordering of severity type because you have added one in the middle of the existing severity types.
- There is no automatic way (except by hard coding) that will let you list all severity types.

You would agree that representing the severity types using integer constants is difficult to maintain. That was the only easily implemented solution available in Java before version 5.0 to define enumerated constants. You could have solved this problem effectively before Java 5.0. However, the amount of code you had to write was disproportionate to the problem. The enum type in Java 5.0 solves this problem in a simple and effective way.

According to Merriam-Webster online dictionary, the term “enumerate” means “to specify one after another”. This is exactly what the enum type lets you do. It lets you specify constants in a specific order. The constants defined in an enum are instances of that enum type. You define an enum type using the `enum` keyword. Its simplest general syntax is:

```
<<access modifier>> enum <<enum type name>> {
     // List of comma separated enum constant names
}
```

The value for `<<access modifiers>` could be the same as the access modifier for a class - `public`, `private`, `protected` or package-level (no modifier). The valid value for `<<access modifier>>` that could be used with enum type depends on the context in which the enum type is defined. When an enum type is declared as a top-level enum type, it can use a `public` or package-level access modifier. If you declare an enum type inside a class or inside another enum, you can use any of the four access modifiers. The `<<enum type name>>` is any valid Java identifier. The body of the enum type is placed within braces following its name. The body of the enum type can have a list of comma-separated constants and other elements that are similar to the elements you have in a class, e.g., instance variables, methods, etc. Most of the time, the enum body includes only constants. The following code declares an enum type called `Gender`, which

declares two constants `MALE` and `FEMALE`. It is a convention to name the enum constants in uppercase. The semi-colon after the last enum constant is optional if there is no code that follows the list of constants.

```
public enum Gender {
     MALE, FEMALE;  // The semi-colon is optional in this case
}
```

Listing 4-1 declares a `public` enum type called `Severity` with four enum constants – `LOW`, `MEDIUM`, `HIGH`, and `URGENT`. A `public` enum type can be accessed from anywhere in the application. You need to save the code in Listing 4-1 in a file named Severity.java. When you compile the code, the compiler will create a *Severity.class* file. Note that except for the use of the `enum` keyword and the body part (within braces `{}`), everything for the `Severity` enum type looks exactly as if it is a class declaration. In fact, Java implements an enum type as a class. The compiler does a lot of work for an enum type and generates code for it that is essentially a class. You need to place an enum type in a package as we have been placing all classes in a package. You can use an import statement to import an enum type, as we import a class type, into a compilation unit.

Listing 4-1: Declaration of a Severity enum

```
// Severity.java
package com.jdojo.chapter4;

public enum Severity {
     LOW, MEDIUM, HIGH, URGENT;
}
```

You declare a variable of an enum type the same way, you declare a variable of a class type.

```
// Declare defectSeverity variable of the Severity enum type
Severity defectSeverity;
```

You can assign `null` to an enum type variable as:

```
Severity defectSeverity = null;
```

What other values can we assign to an enum type variable? An enum type defines two things – the enum constants, which are the only valid values for its type, and the order for those constants. The `Severity` enum type defines fours enum constants – `LOW`, `MEDIUM`, `HIGH` and `URGENT`. Therefore, a variable of the `Severity` enum type can have only one of the four values - `LOW`, `MEDIUM`, `HIGH` and `URGENT`. You can use dot notation to refer to the enum constants by using the enum type name as the qualifier. The following snippet of code assigns values to a variable of `Severity` enum type.

```
Severity defectSeverity1 = Severity.LOW;
Severity defectSeverity2 = Severity.MEDIUM;
Severity defectSeverity3 = Severity.HIGH;
Severity defectSeverity4 = Severity.URGENT;
```

You cannot instantiate an enum type. The following code that attempts to instantiate the `Severity` enum type will generate a compiler error.

```
Severity badAttempt = new Severity(); // A compiler error
```

An enum type acts as a type as well as a factory. It declares a new type and a list of valid instances of that type as its constants. For example, the `Severity` enum type declares four constants. These four constants are the only valid values for the `Severity` type.

An enum type also assigns an order number (or position number), called ordinal, to all of its constants. The ordinal starts with zero and is incremented by one as we move from left-to-right in the list of constants. The first enum constant is assigned the ordinal value of zero, the second of 1, the third of 2, and so on. The ordinal values assigned to constants declared in `Severity` enum type are – 0 to `LOW`, 1 to `MEDIUM`, 2 to `HIGH`, and 3 to `URGENT`. If you change the order of the constants in the enum type body, their ordinal values will change accordingly.

Each enum constant has a name. The name of an enum constant is the same as the identifier specified for the constant in its declaration. For example, the name for the `LOW` constant in the `Severity` enum type is "`LOW`".

You can read the name and the ordinal of an enum constant using the `name()` and `ordinal()` methods respectively. Each enum type has a static method named `values()` that returns an array of constants in the order they are declared in its body. Listing 4-2 has code that prints the name and ordinal of all enum constants declared in `Severity` enum type. It uses an enhanced for-each loop for an array, which is part of Java 5.0. You could also rewrite the for-each loop as a simple for-loop as follows. Note that the use of the enhanced for-each loop is compact and more readable than the use of the simple for-loop. The output of Listing 4-2 shows the name and the ordinal of the constants declared in the `Severity` enum type.

```
Severity[] list = Severity.values();
for(int i = 0; i < list.length; i++) {
     String name = list[i].name();
     int ordinal = list[i].ordinal();
     System.out.println(name + "\t" + ordinal);
}
```

Listing 4-2: Listing name and ordinal of enum type constants

```
// EnumConstantsList.java
package com.jdojo.chapter4;

public class EnumConstantsList {
     public static void main(String[] args)  {
          for(Severity s : Severity.values()) {
               String name = s.name();
               int ordinal = s.ordinal();
               System.out.println(name + "\t" + ordinal);
          }
     }
}
```

```
Output:

LOW       0
MEDIUM    1
HIGH      2
URGENT    3
```

Every Enum Type Extends java.lang.Enum Class

An enum type is similar to a Java class type. In fact, the compiler creates a class, when an enum type is compiled. You can treat an enum type as a class type for all practical purposes. However, there are some rules that apply only to the enum type. An enum type can also have constructors, fields and methods. Did we not say that an enum type cannot be instantiated (e.g. `new Severity()` is valid)? Why do we need constructors for an enum type, if it cannot be instantiated?

Here is the reason why we need constructors for an enum type. An enum type is instantiated only in the code generated by the compiler. All enum constants are instances (or objects) of the same enum type. These instances are created and named the same as the enum constants in the code generated by the compiler. The compiler is playing tricks. The compiler generates code for an enum type similar to the one shown below. The following sample code is just to give you an idea what goes on behind the scenes. The actual code generated by the compiler may be different from the one shown below. For example, the code for the `valueOf()` method gives you a sense that it compares the name with enum constant names and returns the matching constant instance. In reality, the compiler generates code for a `valueOf()` method that makes a call to the `valueOf()` method in the `Enum` superclass.

```
/* Transformed code for Severity enum type declaration*/
package com.jdojo.chapter4;

public final class Severity extends Enum {
     public static final Severity LOW;
     public static final Severity MEDIUM;
     public static final Severity HIGH;
     public static final Severity URGENT;

     // Create constants when class is loaded
     static {
          LOW    = new Severity("LOW", 0);
          MEDIUM = new Severity("MEDIUM", 1);
          HIGH   = new Severity("HIGH", 2);
          URGENT = new Severity("URGENT", 3);
     }

     // The private constructor to prevent direct instantiation
     private Severity(String name, int ordinal) {
          super(name, ordinal);
     }

     public static Severity[] values() {
          return new Severity[] { LOW, MEDIUM, HIGH, URGENT };
     }

     public static Severity valueOf(String name) {
          if (LOW.name().equals(name)) {
               return LOW;
          }

          if (MEDIUM.name().equals(name)) {
               return MEDIUM;
          }

          if (HIGH.name().equals(name)) {
```

```
                return HIGH;
            }

            if (URGENT.name().equals(name)) {
                return URGENT;
            }

            throw new IllegalArgumentException("Invalid enum constant " +
                                               name);
        }
}
```

By looking at the transformed code for the `Severity` enum declaration, we can conclude the following points.

- Every enum type implicitly extends `java.lang.Enum` class. This means that all methods defined in the `Enum` class can be used with all enum types. Table 4-1 lists the methods that are defined in the `Enum` class. We have seen the use of the `name()` and `ordinal()` methods in Listing 4-2 with the `Severity` enum type. Note that all instance methods in the `Enum` class are declared `final`, except for the `toString()` method. The `toString()` method in the `Enum` class returns the name of the enum constant as specified in the constant declaration. Its return value is the same as the return value from the `name()` method. You can override the `toString()` method in your enum type to provide a more meaningful textual representation for each enum constant. For example, you can override the `toString()` method in the `Severity` enum to return "Low Priority" string for the `LOW` enum constant. Listing 4-5 has an example of how to override the `toString()` method in an enum type.
- An enum type is implicitly `final`. In some situations, the compiler cannot declare it as `final` as it has done in the sample code for the `Severity` class. You can also associate a body to each constant in the enum type declaration. When at least one of the constants in an enum type has a body, the compiler does not declare the enum type `final`, because it inherits an anonymous class that extends the enum type to represent the body of that constant. It is the responsibility of the compiler and you should not worry about it. The compiler does not allow you to declare an enum type as `final` explicitly.
- The compiler adds two static methods, `values()` and `valueOf()`, to every enum type. The `values()` method returns the array of enum constants in the same order they are declared in the enum type. We have seen the use of the `values()` method in Listing 4-2. The `valueOf()` method is used to get the instance of an enum type using the constant name as a string. For example, `Severity.valueOf("LOW")` will return `Severity.LOW` constant. The `valueOf()` method facilitates reverse lookup – from string value to enum type value.
- The `Enum` class implements the `java.lang.Comparable` and `java.io.Serializable` interfaces. This means, instances of every enum type can be compared and serialized. The `Enum` class makes sure that during the deserialization process no other instances of an enum type is created than the ones declared as the enum constants. You can use the `compareTo()` method to determine if one enum constant is declared before or after another enum constant. Note that you can also determine the order of two enum constants by comparing their ordinals. The `compareTo()` method does the same, with one more check, that the enum constants being compared must be of the same enum type. The following code snippet shows how to compare two enum constants.

```
Severity s1 = Severity.LOW;
Severity s2 = Severity.HIGH;

// s1.compareTo(s2) returns s1.ordinal() - s2.ordinal()
int diff = s1.compareTo(s2);
```

```
if (diff > 0) {
    System.out.println(s1 + " occurs after " + s2);
}
else {
   System.out.println(s1 + " occurs before " + s2);
}
```

Table 4-1: List of methods in the Enum class that are available in all enum type

Method Name	Description
`public final String name()`	It returns the name of the enum constant exactly as declared in the enum type declaration.
`public final int ordinal()`	It returns the order (or position) of the enum constant as declared in the enum type declaration. The first enum constant is given the position zero, the second as 1, and so on.
`public final boolean equals(Object other)`	It returns `true` if the specified object is equal to the enum constant. Otherwise, it returns `false`. Note that an enum type cannot be instantiated directly and it has a fixed number of instances, which are equal to the number of enum constants it declares. It implies that the == operator and the `equals()` method return the same result, when they are used on two enum constants.
`public final int hashCode()`	It returns the hash code value for an enum constant.
`public final int compareTo(E o)`	It compares the order of this enum constant with the order of the specified enum constant. It returns the difference in ordinal value (as returned by `ordinal()` method) on this enum constant and the specified enum constant. Note that to compare two enum constants they must be of the same enum type. Otherwise, a runtime exception is thrown.
`public final Class<E> getDeclaringClass()`	It returns the class object for the class that declares the enum constant. Two enum constants are considered to be of the same enum type if this method returns the same class object for both. Note that the class object returned by the `getClass()` method, which every enum type inherits from the `Object` class, might not be the same as the class object returned by this method. When an enum constant has a body, the actual class of the object for that enum constant is not the same as the declaring class.
`public String toString()`	It returns the name of the enum constant, which is the same as the return value of the `name()` method. Note that this method is not declared `final` and hence you can override it to return a more meaningful string representation for each enum constant.
`public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)`	It returns an enum constant of the specified enum type and name. For example, you can use the following code to get the `LOW` enum constant value of the `Severity` enum type in your code. `Severity lowSeverity = Enum.valueOf(Severity.class, "LOW")`

`protected final Object clone() throws CloneNotSupportedException`	The `Enum` class redefines the `clone()` method. It declares the `clone()` method `final`, so that it cannot be overridden by any enum type. The `clone()` method always throws an exception from its body. This is done intentionally to prevent cloning of enum constants. This makes sure that only one set of enum constants exists for each enum type.
`protected final void finalize()`	The `Enum` class declares it final so that it cannot be overridden by any enum type. It provides an empty body. Since you cannot create an instance of an enum type, except its constants, it makes no sense to have a `finalize`() method for your enum type.

Enhanced switch Statement for Enum Types

The general syntax of a switch statement is:

```
switch(expression) {
     case constant1:
          // do something

     case constant2:
          // do something

     case constantN:
          // do something

     default:
          // do something
}
```

Before Java 5.0, the expression for the switch statement must be of only `int` type. Java 5.0 has enhanced the switch statement to accommodate enum types. It allows you to have an enum type as a switch expression. When the switch expression is of an enum type, all case labels must be unqualified enum constants of the same enum type. The switch statement deduces the enum type name from the type of its expression. You may include a default label in the switch block with an enum type switch expression. The following is the version of our `DefectUtil` class that has been rewritten using enum type and enhanced switch statement. Now you do not need to handle the exceptional case of receiving a `null` value in the `severity` parameter inside `getProjectedTurnAroundDays()` method. If the enum expression of the switch statement evaluates to `null`, it throws `NullPointerException`. When you need to use code like the one shown in the `getProjectedTurnAroundDays()` method, which uses a switch statement for an enum type, you can improve the code by moving it to the enum type declaration. We will look at this feature in the next section.

```
// DefectUtil.java
package com.jdojo.chapter4;

public class DefectUtil {
     public static int getProjectedTurnAroundDays(Severity severity) {
          int days = 0;
```

```
            switch (severity) {
                 case LOW: // Must use the unqualified name LOW
                           // and not the qualified name Severity.LOW
                      days = 30;
                      break;
                 case MEDIUM:
                      days = 15;
                      break;
                 case HIGH:
                      days = 7;
                      break;
                 case URGENT:
                      days = 1;
                      break;
            }

            return days;
       }
}
```

Associating Data and Methods to Enum Constants

Generally, you declare an enum type just to have some enum constants as we have done in the `Severity` enum type. Since an enum type is actually a class type you can declare pretty much everything inside an enum type body that you can declare inside a class body. You can declare constructors, fields and methods for an enum type. Let us associate one data element, projected turnaround days, with each of our `Severity` enum constant. We will name our enhanced `Severity` enum type as `SmartSeverity`. Listing 4-3 has code for `SmartSeverity` enum type, which is very different from the code for `Severity` enum type.

Listing 4-3: A SmartSeverity enum type declaration, which uses fields, constructors, and methods

```
// SmartSeverity.java
package com.jdojo.chapter4;

public enum SmartSeverity {
     LOW(30), MEDIUM(15), HIGH(7), URGENT(1);

     // Declare an instance variable
     private int projectedTurnaroundDays;

     // Declare a private constructor
     private SmartSeverity(int projectedTurnaroundDays) {
          this.projectedTurnaroundDays = projectedTurnaroundDays;
     }

     // Declare a public method to get the turnaround days
     public int getProjectedTurnAroundDays() {
          return projectedTurnaroundDays;
     }
}
```

Let us discuss new things that are in `SmartSeverity` enum type.

- It declares an instance variable `projectedTurnaroundDays`, which will store the value of the projected turnaround days for each enum constant.

```
// Declare an instance variable
private int projectedTurnaroundDays;
```

- It defines a `private` constructor, which accepts an `int` parameter. It stores the value of its parameter in the instance variable. You can add multiple constructors to an enum type. If you do not add a constructor, a no-args constructor is added. You cannot add a `public` or `protected` constructor to an enum type. All constructors in an enum type declaration go through parameter and code transformation by the compiler and their access levels are changed to `private`. Many things are added or changed in the constructor of an enum type by the compiler. As a programmer, you do not need to know the details of the changes made by the compiler.

```
// Declare a private constructor
private SmartSeverity(int projectedTurnaroundDays){
    this.projectedTurnaroundDays = projectedTurnaroundDays;
}
```

- It declares a method `getProjectedTurnAroundDays()`, which returns the value of the projected turnaround days for an enum constant (or the instance of the enum type).
- The enum constant declarations have changed to:

```
LOW(30), MEDIUM(15), HIGH(7), URGENT(1);
```

  This change is not obvious. Now every enum constant name is followed by an integer value in parentheses e.g. `LOW(30)`. This syntax is shorthand for calling the constructor with an `int` parameter type. When an enum constant is created, the value inside the parentheses will be passed to the constructor that we have added. By simply using the name of the enum constant, e.g., `LOW` in the constant declaration, you invoke a default no-args constructor.

Listing 4-4 has code that tests the `SmartSeverity` enum type. It prints the names of the constants, their ordinals and their projected turnaround days. Note that the logic to compute the projected turnaround days is encapsulated inside the definition of the enum type itself. The `SmartSeverity` enum type definition combines the code for the `Severity` enum type and the `getProjectedTurnAroundDays()` method in the `DefectUtil` class. We do not have to write a switch statement anymore to get the projected turnaround days. Each enum constant knows about its projected turnaround days.

Listing 4-4: A test class to test the SmartSeverity enum type

```
// SmartSeverityTest.java
package com.jdojo.chapter4;

public class SmartSeverityTest {
     public static void main(String[] args) {
          for(SmartSeverity s : SmartSeverity.values()) {
               String name = s.name();
               int ordinal = s.ordinal();
               int days = s.getProjectedTurnAroundDays();
               System.out.println(name + "\t" + ordinal + "\t" + days);
          }
     }
}
```

```
Output:

LOW       0    30
MEDIUM    1    15
HIGH      2    7
URGENT    3    1
```

Associating a Body to an Enum Constant

The `SmartSeverity` is an example of adding data and methods to an enum type. The code in the `getProjectedTurnAroundDays()` method is the same for all enum constants. You can also associate a different body to each enum constant. The body can have fields and methods. The body for an enum constant is placed inside braces (`{ }`) following its name. If the enum constant accepts arguments, its body follows its argument list. The syntax for associating a body to an enum constant is as follows.

```
<<access modifier> enum <<enum type name>>{
     CONST1 {
          // Body for CONST1 goes here
     },
     CONST2 {
          // Body for CONST2 goes here
     },
     CONST3(arguments-list) {
          // Body of CONST3 goes here
     };

     // Other code goes here
}
```

It is a little different game, when you add a body to an enum constant. The compiler creates an anonymous class, which inherits from the enum type. It moves the body of the enum constant to the body of that anonymous class. Consider an `ETemp` enum type as shown below.

```
public enum ETemp {
     C1 {
          // Body of constant C1
          public int getValue() {
               return 100;
          }
     },
     C2,
     C3;
}
```

The body of the `ETemp` enum type declares three constants: `C1`, `C2`, and `C3`. We have added a body to the `C1` constant. The compiler will transform the code for `ETemp` into something like the code shown below.

```
public enum ETemp {
     public static final ETemp C1 = new ETemp() {
          // Body of constant C1
```

```
          public int getValue() {
               return 100;
          }
     };
     public static final ETemp C2 = new ETemp();
     public static final ETemp C3 = new ETemp();

     // Other code goes here
}
```

Note that the constant `C1` is declared of the `ETemp` type and assigned an object, which is of an anonymous class type. The `ETemp` enum type has no knowledge of the `getValue()` method defined in the anonymous class. Therefore, it is useless for all practical purposes, because you cannot call the method as `ETemp.C1.getValue()`.

To let the client code use the `getValue()` method, you must declare a `getValue()` method for the `ETemp` enum type. If you want all constants of `ETemp` to override and provide implementation for this method, you need to declare it as `abstract`. If you want it to be overridden by some, but not all constants, you need to declare it non-abstract and provide a default implementation for it. The following code declares a `getValue()` method for the `ETemp` enum type, which returns 0. The `C1` constant has its body, which overrides the `getValue()` method and returns `100`. Note that the constants `C2` and `C3` do not have to have a body; they do not need to override the `getValue()` method. Now, you can use the `getValue()` method on the `ETemp` enum type.

```
public enum ETemp {
     C1 {
          // Body of constant C1
          public int getValue() {
               return 100;
          }
     },
     C2,
     C3;

     // Provide the default implementation for the getValue() method
     public int getValue() {
          return 0;
     }
}
```

The following code rewrites the above version of `ETemp` and declares the `getValue()` method `abstract`. An `abstract` method for an enum type forces you to provide a body for all constants and override that method. The following code provides the body for all constants, `C1`, `C2`, and `C3`. The body of each constant overrides and provides implementation for the `getValue()` method.

```
public enum ETemp {
     C1 {
          // Body of constant C1
          public int getValue() {
               return 100;
          }
     },
     C2 {
          // Body of constant C2
          public int getValue() {
```

```
                return 0;
            }
        },
        C3 {
            // Body of constant C3
            public int getValue() {
                return 0;
            }
        };

        // Provide default implementation for getValue() method
        public abstract int getValue();
}
```

Let us enhance our `SmartSeverity` enum type. We are running out of good names for our enum type. We will name the new one `SuperSmartSeverity`. Listing 4-5 has its code.

Listing 4-5: A SuperSmartSeverity enum type, which demonstrates the use of a body for enum constants. It also shows how to override the toString() method and provide a custom descriptive name for enum constants

```
// SuperSmartSeverity.java
package com.jdojo.chapter4;

public enum SuperSmartSeverity {
        LOW("Low Priority", 30) {
            public double getProjectedCost() {
                return 1000.0;
            }
        },
        MEDIUM("Medium Priority", 15) {
            public double getProjectedCost() {
                return 2000.0;
            }
        },
        HIGH("High Priority", 7) {
            public double getProjectedCost() {
                return 3000.0;
            }
        },
        URGENT("Urgent Priority", 1) {
            public double getProjectedCost() {
                return 5000.0;
            }
        };

        // Declare instance variables
        private String description;
        private int projectedTurnaroundDays;

        // Declare a private constructor
        private SuperSmartSeverity(String description,
                                   int projectedTurnaroundDays){
            this.description = description;
            this.projectedTurnaroundDays = projectedTurnaroundDays;
        }

        // Declare a public method to get the turn around days
```

```
    public int getProjectedTurnAroundDays() {
        return projectedTurnaroundDays;
    }

    // Override toString() method of Enum class to return description
    @Override
    public String toString() {
        return this.description;
    }

    // Provide getProjectedCost() abstract method, so that all constants
    // override and provide implementation for it in their body
    public abstract double getProjectedCost();
}
```

The following are new things in the `SuperSmartSeverity` enum type. The code in Listing 4-6 demonstrates the use of the new features added to the `SuperSmartSeverity` enum type.

- It has added an `abstract` method `getProjectedCost()` to return the projected cost of each type of severity.
- It has a body for each constant that provides implementation for the `getProjectedCost()` method. Note that declaring an abstract method in an enum type forces you to provide a body for all its constants.
- It has added another parameter to its constructor, which is a nicer name for the severity type.
- It has overridden the `toString()` method of `Enum` class. The `toString()` method in the `Enum` class returns the name of the constant. Our `toString()` method returns a brief and more intuitive name for each constant.

TIP
Typically, you do not need to write this kind of complex code for an enum type. Java enum is very powerful. It has features for you to utilize, if you need them.

Listing 4-6: A test class to test the SuperSmartSeverity enum type

```
// SuperSmartSeverityTest.java
package com.jdojo.chapter4;

public class SuperSmartSeverityTest {
    public static void main(String[] args) {
        for(SuperSmartSeverity s : SuperSmartSeverity.values()) {
            String name = s.name();
            int ordinal = s.ordinal();
            int projectedTurnAroundDays =
                                s.getProjectedTurnAroundDays();
            double projectedCost = s.getProjectedCost();
            System.out.println(name + "\t" + s + "\t" +
                                ordinal + "\t" +
                                projectedTurnAroundDays + "\t" +
                                projectedCost);
        }
    }
}
```

```
Output:

LOW       Low Priority       0    30 1000.0
MEDIUM    Medium Priority    1    15 2000.0
HIGH      High Priority      2    7  3000.0
URGENT    Urgent Priority    3    1  5000.0
```

Comparing Two Enum Constants

You can compare two enum constants in three ways.

- Using the `compareTo()` method of the `Enum` class.
- Using the `equals()` method of the `Enum` class.
- Using the == operator.

The `compareTo()` method of the `Enum` class lets you compare two enum constants of the same enum type. It returns the difference in ordinal for the two enum constants. If both enum constant are the same, it returns zero. The following snippet of code will print -3 because the difference of the ordinals for `LOW` and `URGENT` is -3. Recall that the first enum constant is assigned an ordinal of zero, the second one 1, and so one.

```
Severity s1 = Severity.LOW;
Severity s2 = Severity.URGENT;
int diff = s1.compareTo(s2);
System.out.println(diff);
```

```
Output:

-3
```

The `compareTo()` method works only on two enum constants of the same enum type. Suppose you have another enum called `BasicColor` as:

```
public enum BasicColor {
     RED, GREEN, BLUE;
}
```

The following code will not compile because it tries to compare the two enum constants, which belong to two different enum types.

```
int diff = BasicColor.RED.compareTo(Severity.URGENT); // A compiler error
```

You can use the `equals()` method of the `Enum` class to compare the two enum constants for equality. An enum constant is equal only to itself. Note that the `equals()` method can be invoked on two enum constants of different types. If the two enum constants are from different enum types, it always returns `false`. The following code snippet demonstrates the result of the `equals()` method while comparing two enum constants.

```
Severity s1 = Severity.LOW;
Severity s2 = Severity.URGENT;
```

```
BasicColor c = BasicColor.BLUE;

System.out.println(s1.equals(s1));
System.out.println(s1.equals(s2));
System.out.println(s1.equals(c));
```

```
Output:

true
false
false
```

You can also use the equality operator (==) to compare two enum constants for equality. Both operands to the == operator must be of the same enum type. Otherwise, you get a compile-time error. The following code snippet demonstrates the use of the == operator to compare two enum constants.

```
Severity s1 = Severity.LOW;
Severity s2 = Severity.URGENT;
BasicColor c = BasicColor.BLUE;

System.out.println(s1 == s1);
System.out.println(s1 == s2);

//System.out.println(s1 == c); // A compiler error. Cannot compare
                               // Severity and BasicColor enum types
```

```
Output:

true
false
```

TIP
The `equals()` method in the `Enum` class uses the == operator to compare two enum constants. For example, if you invoke `s1.equals(s2)` for two enum constants `s1` and `s2`, it returns the result of the expression `s1 == s2`. Using the == operator in `equals()` method for the `Enum` class is justified, because two instances of the same enum constant of an enum type cannot exist. If two enum constants are equal, they must be the same instance of the enum type.

Nested Enum Types

You can have a nested enum type declaration. You can declare a nested enum type inside a class, an interface or another enum type. Nested enum types are implicitly `static`. You can also declare a nested enum type `static` explicitly in its declaration. Since an enum type is always `static`, whether you declare it or not, you cannot declare a local enum type (e.g. inside a method's body). Note that an inner class also cannot have static members. Therefore, it implies that you cannot declare an enum type inside an inner class. You can use any of the access modifiers – `public`, `private`, `protected` or package-level for a nested enum type. The following code snippet declares nested `public` enum type named `Gender` inside a `Person` class.

```
// Person.java
package com.jdojo.chapter4;

public class Person {
     public enum Gender {MALE, FEMALE}
}
```

The `Person.Gender` enum type can be accessed from anywhere in the application because it has been declared `public`. You need to import the enum type to use its simple name in other packages as shown in the following code.

```
// Test.java
package com.jdojo.chapter4.pkg1;

import com.jdojo.chapter4.Person.Gender;

public class Test {
     public static void main(String[] args) {
          Gender m = Gender.MALE;
          Gender f = Gender.FEMALE;
          System.out.println(m);
          System.out.println(f);
     }
}
```

You can also use the simple name of en enum constant by importing the enum constants using static imports. The following code snippet uses `MALE` and `FEMALE` simple names of constants of `Person.Gender` enum type. Note that the first import statement is needed to import the `Gender` type itself so that its simple name could be used in that code.

```
// Test.java
package com.jdojo.chapter4.pkg1;

import com.jdojo.chapter4.Person.Gender;
import static com.jdojo.chapter4.Person.Gender.*;

public class Test {
     public static void main(String[] args) {
          Gender m = MALE;
          Gender f = FEMALE;
          System.out.println(m);
          System.out.println(f);
     }
}
```

You can also nest an enum type inside another enum type or an interface. The following are valid enum type declarations.

```
public enum OuterEnum {
     C1, C2, C3;

     public enum NestedEnum {
          C4, C5, C6;
     }
}
```

```
public interface MyInterface {
     int operation1();
     int operation2();

     public enum AnotherNestedEnum {
          CC1, CC2, CC3;
     }
}
```

Implementing an Interface to an Enum Type

An enum type may implement one or more interfaces. The rules for an enum type implementing an interface are the same as the rules for a class implementing an interface. Note that an enum type is never inherited by another enum type. Therefore, you cannot declare an enum type as abstract. This also implies that if an enum type implements an interface, it must also provide implementation for all methods in that interface. The following code snippet shows an enum type called `CommandList` that implements an interface called `Command`. Note that each enum constant implements the `execute()` method of the `Command` interface. Alternatively, you can implement the `execute()` method in the enum type body and omit the implementations from some or all enum constants.

```
// Command interface
public interface Command {
     void execute();
}
// CommandList enum type implementing Command interface
public enum CommandList implements Command {
     RUN {
          public void execute() {
               System.out.println("Run");
          }
     },
     JUMP {
          public void execute() {
               System.out.println("Jump");
          }
     };

     // Force all constants to implement the execute() method.
     public abstract void execute();
}
```

Behind the Scenes

Reverse Lookup for Enum Constants

We have seen how to loop through the list of enum constants of an enum type by first getting the list of enum constants in an array using the `values()` static method. Every enum constant has a

name and an ordinal. You can perform reverse lookup based on the name or the ordinal. That is, you can get the value of an enum constant if you know its name or position in the list. You can use the `valueOf()` method, which is added by the compiler to an enum type, to perform reverse lookup based on a name. You can use the array returned by the `values()` method, which is added by the compiler to an enum type, to perform reverse lookup by ordinal (or position). Note that the order of the values in the array that is returned by `values()` method is the same as the order in which the enum constants are declared. The ordinal of enum constants starts with zero. This implies that the ordinal value of an enum constant can be used as the index in the array that is returned by the `values()` method to get to the value of the enum constant. The following code snippet demonstrates how to reverse look up an enum constant.

```
Severity low1 = Severity.valueOf("LOW"); // Reverse lookup using a name
Severity low2 = Severity.values()[0]; // Reverse lookup using an ordinal

System.out.println(low1);
System.out.println(low2);
System.out.println(low1 == low2);
```

```
Output:

LOW
LOW
true
```

Working with Enum Constants Ranges - EnumSet

Java API provides the `java.util.EnumSet` collection class to work with ranges of enum constants of an enum type. The implementation of the `EnumSet` class is very efficient. Suppose we have an enum type called `Day` as shown in Listing 4-7. You can work with a range of days using the `EnumSet` class, for example, get all days between `MONDAY` and `FRIDAY`. An `EnumSet` iterator always returns the enum constants in the same order they are declared in the enum type. An `EnumSet` can contain enum constants only from one enum type. Listing 4-8 demonstrates how to use the `EnumSet` class to work with the range for enum constants.

Listing 4-7: Declaration of a Day enum type, which represents seven days of a week as enum constants

```
// Day.java
package com.jdojo.chapter4;

public enum Day {
     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

Listing 4-8: A test class to demonstrate how to use the EnumSet class to loop through the elements of an enum type

```
// EnumSetTest.java
package com.jdojo.chapter4;

import java.util.EnumSet;
import java.util.Iterator;

public class EnumSetTest {
```

```
    public static void main(String[] args) {
        // Print all enum constants of Day enum type
        System.out.println("All days using a for-each loop.");

        for(Day d: EnumSet.allOf(Day.class)) {
            System.out.print(d + " ");
        }

        System.out.println("\n\nOnly weekdays using a for-each loop.");

        for(Day d: EnumSet.range(Day.MONDAY, Day.FRIDAY)) {
            System.out.print(d + " ");
        }

        System.out.println("\n\nOnly weekdays using an iterator.");

        EnumSet<Day> weekDays = EnumSet.range(Day.MONDAY, Day.FRIDAY);
        Iterator<Day> iterator = weekDays.iterator();
        while(iterator.hasNext()) {
            Day d = iterator.next();
            System.out.print(d + " ");
        }

        System.out.println("\n\nOnly weekends using a for-each loop.”);

        for(Day d: EnumSet.complementOf(weekDays)) {
            System.out.print(d + " ");
        }
    }
}
```

Output:

```
All days using a for-each loop.
MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY SATURDAY SUNDAY

Only weekdays using a for-each loop.
MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY

Only weekdays using an iterator.
MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY

Only weekends using a for-each loop.
SATURDAY SUNDAY
```

References

Bloch, Joshua. *Effective Java: Programming Language Guide.* Addison-Wesley, 2001.

Cormen , Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2nd Edition. The MIT Press, 2001.

Cornell, Gary, and Cay Horstmann. *Core Java 1.2.* Vol. 1. Prentice Hall Computer Books, 1998.

—. *Core Java 1.2.* Vol. 2. Prentice Hall Computer Books, 1998.

Downing, Troy Bryan. *Java RMI: Remote Method Invocation.* Wiley Publishing, 1998.

Eckel, Bruce. *Thinking in Java.* 1st Edition. Prentice Hall, 1998.

Freeman, Elisabeth, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns.* O'Reilly Media, 2004.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* 3rd Edition. Addison-Wesley, 2005.

Grosso, William. *Java RMI.* 1st Edition. O'Reilly Media, 2001.

Hall, Marty. *Core Servlets and Javaserver Pages.* Prentice Hall, 2000.

Harold, Elliotte Rusty. *Java I/O.* 1st Edition. O'Reilly Media, 1999.

—. *Java Network Programming.* 2nd Edition. O'Reilly Media, 2000.

Horton, Ivor. *Beginning Java.* Wrox Press, 1997.

Hyde, Paul. *Java Thread Programming.* Sams, 1999.

"Java SE 7 Documentation." *Oracle Corporation Web site.* http://download.oracle.com/javase/7/docs/ (accessed 2011).

Lakshman, Bulusu. *Oracle and Java Development.* Sams, 2001.

Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns.* 2nd Edition. Addison-Wesley, 1999.

Liskov, Barbara, and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Professional, 2000.

Mano, M. Morris. *Computer Engineering: Hardware Design.* Prentice Hall, 198.

Oaks, Scott. *Java Security.* 2nd Edition. O'Reilly Media, 2001.

Oaks, Scott, and Henry Wong. *Java Threads.* 2nd Edition. O'Reilly Media, 1999.

Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* 2nd Edition. Morgan Kaufmann, 1997.

Pawlan, Monica. "Reference Objects and Garbage Collection." *Pawlan Communications.* 8 1998. http://www.pawlan.com/monica/articles/refobjs/ (accessed 7 2011).

Preiss, Bruno R. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java.* Wiley, 1999.

Shirazi, Jack. *Java Performance Tuning.* 1st Edition. O'Reilly Media, 2000.

Topley, Kim. *Core Java Foundation Classes.* Prentice Hall PTR, 1998.

Travis, Greg. "IBM DeveloperWorks." *Getting started with new I/O (NIO).* July 09, 2003. https://www.ibm.com/developerworks/java/tutorials/j-nio/ (accessed September 01, 2008).

Venners, Bill. *Inside the Java 2 Virtual Machine.* 2nd Edition. McGraw-Hill Companies, 2000.

Walsh, Aaron E, Justin Couch, and Daniel H. Steinberg. *Java 2 Bible.* Wiley Publishing, 2000.

Index