THE EXPERT'S VOICE® IN JAVA

Beginning Java 8 Language Features

Lambda Expressions, Inner Classes, Threads, I/O, Collections and Streams

SECOND IN A SERIES OF THREE: CONTINUE YOUR LEARNING, WITH FOCUS ON ADVANCED LANGUAGE FEATURES

Kishori Sharan

Foreword by Jeff Friesen, JavaWorld columnist

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

How This Book Came About

My first encounter with the Java programming language was during a one-week Java training session in 1997. I did not 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 on the test, scoring 95 percent. The three questions that I missed on the test made me realize that the books I read did not adequately cover all of the details on all of the necessary Java topics. I made up my mind to write a book on the Java programming language. So, I formulated a plan to cover most of the topics that a Java developer needs understand 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 700 to 800 pages.

As I progressed, I realized that a book covering most of the Java topics in detail could not be written in 700 to 800 hundred pages. One chapter that covered data types, operators, and statements spanned 90 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 the 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 finish this book because that rush could have compromised the quality and the coverage 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 with the plan that it 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 that make up the building. The same logic applies to most of the things that we want to understand in our lives. It certainly applies to an understanding of the Java programming language. If you want to master the Java programming language, you must start by understanding its basic building blocks. I have used this approach throughout this book, endeavoring to build each topic by describing the basics first. In the 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 books about the Java programming language 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 plenty of illustrations in this book to aid readers in understanding and visualizing the contents. Developers who have little or no programming experience can have difficulty putting things together to make a complete program. Keeping them in mind, the book contains over 290 complete Java programs that are ready to be compiled and run.

I spent countless hours doing research for writing this book. My main sources of research were 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 to research a topic before I could write the first sentence on it. It was always fun to play with Java programs, sometimes for hours, to add them to the book.

Structure of the Book

This is the second book in the three-book Beginning Java series. This book contains 13 chapters. The chapters contain language-level topics of Java such as annotations, generics, lambda expressions, threads, I/O, collections, streams, etc. Chapters introduce Java topics in an increasing order of complexity. The new features of Java 8 are included wherever they fit in the chapter. The lambda expressions and Streams API, which were added in Java 8, are covered in depth.

After finishing this book, take your Java knowledge to the next level by learning the Java APIs, extensions, and libraries; all of this is covered in the last book in this series, *Beginning Java 8 APIs, Extensions, and Libraries* (ISBN 978-1-4302-6661-7).

Audience

This book is designed to be useful for anyone who wants to learn the Java programming language. If you are a beginner, with little or no programming background in Java, you are advised to read the companion book *Beginning Java 8 Fundamentals* before reading this book. This 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 it later when you gain more experience.

If you are a Java developer with an 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.

If you are reading this book to get a certification in the Java programming language, you need to read almost all of the chapters, paying attention to all of 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 290 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 should read the chapters of this book selectively. Some topics such as lambda expressions, collections, and streams are used extensively in developing Java applications, whereas some topics such as threads and archive files are infrequently used. You need to read only those chapters that are covered in your class syllabus. I am sure that you, as a Java student, do not need to read the entire book page by page.

How to Use This Book

This book is the beginning, not the end, of gaining 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, which 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 hard 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.

> *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 where you will find a complete list of everything available in the Java class library. You can download (or view) the Java API documentation from the official web site of Oracle Corporation at 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 that "practice makes perfect," which is also true in learning how to program in Java.

Source Code and Errata

Source code and errata for this book may be downloaded from www.apress.com/source-code.

Questions and Comments

Please direct all your questions and comments for the author to ksharan@jdojo.com.

CHAPTER 1

Annotations

In this chapter, you will learn

- What annotations are
- How to declare annotations
- How to use annotations
- What meta-annotations are and how to use them
- Commonly used annotations
- How to access annotations at runtime
- How to process annotations in source code

What Are Annotations?

Annotations were introduced in Java 5. Before I define annotations and discuss their importance in programming, let's discuss a simple example. Suppose you have an `Employee` class, which has a method called `setSalary()` that sets the salary of an employee. The method accepts a parameter of the type `double`. The following snippet of code shows a trivial implementation for the `Employee` class:

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

A `Manager` class inherits from the `Employee` class. You want to set the salary for managers differently. You decide to override the `setSalary()` method in the `Manager` class. The code for the `Manager` class is as follows:

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

Note that there is a mistake in the above code for the Manager class, when you attempt to override the setSalary() method. (You'll correct the mistake shortly.) You have used the int data type as the parameter type for the incorrectly overridden method. It is time to set the salary for a manager. The following code is used to accomplish this:

```
Employee ken = new Manager();
int salary = 200;
ken.setSalary(salary);
```

```
Employee.setSalary():200.0
```

This snippet of code was expected to call the setSalary() method of the Manager class but the output does not show the expected result.

What went wrong in your code? The intention of defining the setSalary() method in the Manager class was to override the setSalary() method of the Employee class, not to overload it. You made a mistake. You used the type int as the parameter type in the setSalary() method, instead of the type double, in the Manager class. You put comments indicating your intention to override the method in the Manager class. However, comments do not stop you from making logical mistakes. You might spend, as every programmer does, hours and hours debugging errors resulting from this kind of logical mistake. Who can help you in such situations? Annotations might help you in a few situations like this. Let's rewrite your Manager class using an annotation. You do not need to know anything about annotations at this point. All you are going to do is add one word to your program. The following code is the modified version of the Manager class:

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

All you have added is a @Override annotation to the Manager class and removed the "dumb" comments. Trying to compile the revised Manager class results in a compile-time error that points to the use of the @Override annotation for the setSalary() method of the Manager class:

```
Manager.java:2: error: method does not override or implement a method from a supertype
        @Override
        ^
1 error
```

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 does not override a method in the superclass, the compiler generates an error. In your case, the setSalary(int salary) method in the Manager class does not override any method in the superclass Employee. This is the reason that you got the error. You may realize that using an annotation is as simple as documenting the source code. However, they have compiler support. You can use them to instruct the compiler to enforce some rules. Annotations provide benefits much more than you have seen in this example.

Let's go back to the compile-time error. You can fix the error by doing one of the following two things:

- You can remove the `@Override` annotation from the `setSalary(int salary)` method in the `Manager` class. It will make the method an overloaded method, not a method that overrides its superclass method.
- You can change the method signature from `setSalary(int salary)` to `setSalary(double salary)`.

Since you want to override the `setSalary()` method in the `Manager` class, use the second option and modify the `Manager` class as follows:

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

Now the following code will work as expected:

```
Employee ken = new Manager();
int salary = 200;
ken.setSalary(salary);
```

```
Manager.setSalary():200.0
```

Note that the `@Override` annotation in the `setSalary()` method of the `Manager` class saves you debugging time. Suppose you change the 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. Are you starting to understand the power of annotations? With this background in mind, let's start digging deep into annotations.

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, a type parameter in a generic type/method declaration, a type use, etc. In other words, you can annotate any declaration or type use in a Java program. 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 Javadocs from your documentation and that's it for regular documentation. This does not mean that you do not need to document your programs. You do need regular documentation. At the same time, you need a way to enforce your intent using a documentation-like mechanism. Your documentation should be available to the compiler and 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 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 an 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 the runtime, an annotation has to follow rules. In fact, an annotation is another type like a class and an interface. As you have to declare a class type or an interface type before you can use it, you must also declare an annotation type.

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 `setSalary()` method did not change the way the 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, except for 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. The following is the general syntax for declaring an annotation type:

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

The `<modifiers>` for an annotation declaration is the same as for an interface declaration. For example, you can declare an annotation type as `public` or package level. The `@` 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 an annotation type name. It should be a valid Java identifier. The annotation type body is placed within braces.

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

Listing 1-1. The Declaration of an Annotation Type Named Version

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

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: it uses the `@` sign before its name. You 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: 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. You have declared two elements, `major` and `minor`, for the `Version` annotation type. The data types of both elements are `int`.

■ **Note** Although it is allowed to declare static and default methods in interface types, they are not allowed in annotation types.

You need to compile the annotation type. When `Version.java` file is compiled, it will produce a `Version.class` file. The simple name of your annotation type is `Version` and its fully qualified name is `com.jdojo.annotation.Version`. Using the simple name of an annotation type follows the rules of any other types (e.g. classes, interfaces, etc.). You will need to import an annotation type the same way you import any other types.

How do you use an annotation type? You might be thinking that you will declare a new class that will implement the `Version` annotation type, and you 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 an `@` sign. It is followed by a list of comma-separated `name=value` pairs enclosed in parentheses. The name in a `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's 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 your `Version` annotation type:

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

You 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.annotation.Version(major=0, minor=1)
```

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

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

An annotation is added in the same way you add a modifier for a program element. 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 a 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. Let's follow this 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 1-2 shows the sample code for the `VersionTest` class.

Listing 1-2. A VersionTest Class with Annotated Elements

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

// 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;
        }
}
```

In Listing 1-2, you use `@Version` annotation to annotate the class declaration, class field, constructors, and methods. There is nothing extraordinary in the code for the `VersionTest` class. You just added the `@Version` annotation to various elements of the 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.

What do you do next with the `Version` annotation type? You have declared it as a type. You have used it in your `VersionTest` class. Your next step is to read it at runtime. Let's defer this step for now; I will cover it in detail in a later section.

Restrictions on Annotation Types

An annotation type is a special type of interface with some restrictions. I will cover some of the restrictions in the sections to follow.

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 you 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, which is declared 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 annotation types. A word of caution needs to be mentioned here. You 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. Your `Version` annotation type has only two elements, `major` and `minor`, which are declared in the `Version`

type itself. You cannot use the 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.

Restriction #2

Method declarations in an annotation type cannot specify any parameters. A method 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, a setter and a getter, 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. The Java runtime will take care of setting the specified value for the annotation elements. Since the goal of declaring a method in an annotation type is to work with a data element, you 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 a throws clause
        int minor(); // OK
}
```

Restriction #4

The return type of a method declared in an annotation type must be one of the following types:

- Any primitive type: `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, for example, `String[]`, `int[]`, etc. The return type cannot be a nested array. For example, you cannot have a return type of `String[][]` or `int[][]`.

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 of type `Test`. You can declare the annotation method as shown:

```
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.

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 keyword `default` is used to specify the default value. The default value must be of the type compatible to the data type for the element.

Suppose you have a product that is not frequently released, so it is less likely that it will have a minor version other than zero. You can simplify your `Version` annotation type by specifying a default value for its `minor` element as zero, as shown:

```
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. 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};
}
```

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 the default value. 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

I 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 shown:

```
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 code in Listing 1-3 for the `DoNotUseIt` class is valid. This is just for the purpose of demonstration. Do not implement an annotation in a class even if it works.

Listing 1-3. A Class Implementing an Annotation Type

```
// DoNotUseIt.java
package com.jdojo.annotation;

import java.lang.annotation.Annotation;

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

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

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

The Java runtime implements the 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. You must distinguish between an annotation type and instances (or objects) of that annotation type. In your example, `Version` is an annotation type. Whenever you use it as `@Version(major=2, minor=4)`, you 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. An annotation should be easy to use in a program. The syntax `@Version(...)` is shorthand for creating a class, creating an object of that class, and setting the values for its elements. I will cover how to get to the object of an annotation type at runtime later in this chapter.

Using Annotations

In this section, I will discuss the details of using different types of elements while declaring annotation types. Remember that the supplied value for elements of an annotation must be a compile-time constant expression and 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`. The `Version` annotation type declares two elements, `major` and `minor`, and both are of `int` data type. 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 a compile-time constant 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 1-4 contains the code for an annotation type called `Name`. It has two elements, `first` and `last`, which are of the `String` type.

Listing 1-4. Name Annotation Type, Which Has Two Elements, first and last, of the String Type

```
package com.jdojo.annotation;

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

The following snippet of code shows how to use the `Name` annotation type 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. Let's 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's create a `DefaultException` class as shown in Listing 1-5.

Listing 1-5. A DefaultException Class That Is Inherited from the Throwable Exception Class

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

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

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

Listing 1-6 shows the code for a `TestCase` annotation type.

Listing 1-6. A TestCase Annotation Type Whose Instances Are Used to Annotate Test Case Methods

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

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;
}
```

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. You could have used the `Class` type as the type of your `willThrow` element. However, that would have allowed the users of this annotation type to pass any class type as its value. Note that you have used two annotations, `@Retention` and `@Target`, for the `TestCase` annotation type. The `@Retention` annotation type specified that the `@TestCase` annotation would be available at runtime. It is necessary to use the retention policy of `RUNTIME` for your `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. I will cover the `@Retention` and `@Target` annotation types in detail in later sections when I discuss meta-annotations. Listing 1-7 shows the use of your `TestCase` annotation type.

Listing 1-7. A Test Case That Uses the TestCase Annotations

```
// PolicyTestCases.java
package com.jdojo.annotation;

import java.io.IOException;

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

```
                // Code goes here
        }

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

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's assume that it has a status element and it can have one of the four values: `PENDING`, `FAILED`, `PASSED`, and `PASSEDWITHCHANGES`. You can declare an enum as an annotation type member. Listing 1-8 shows the code for a `Review` annotation type.

Listing 1-8. An Annotation Type, Which Uses an enum Type Element

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

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};
}
```

The `Review` annotation type declares a `ReviewStatus` enum type and the four review statuses are the elements of the enum. It has two elements, `status` and `comments`. The type of status element is the enum type `ReviewStatus`. The default value for the `status` element is `ReviewStatus.PENDING`. You 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 the `com.jdojo.annotation.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 2014")

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