FUTURE VISION BIE

One Stop for All Study Materials
& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU, Currently for CSE – Computer Science Engineering...

Join Telegram to get Instant Updates: https://bit.ly/2GKiHnJ

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

Enumerations, Autoboxing, and Annotations (Metadata)

T his chapter examines three recent additions to the Java language: enumerations, autoboxing, and annotations (also referred to as metadata). Each expands the power of the language by offering a streamlined approach to handling common programming tasks. This chapter also discusses Java's type wrappers and introduces reflection.

Enumerations

Versions prior to JDK 5 lacked one feature that many programmers felt was needed: enumerations. In its simplest form, an *enumeration* is a list of named constants. Although Java offered other features that provide somewhat similar functionality, such as **final** variables, many programmers still missed the conceptual purity of enumerations—especially because enumerations are supported by most other commonly used languages. Beginning with JDK 5, enumerations were added to the Java language, and they are now available to the Java programmer.

In their simplest form, Java enumerations appear similar to enumerations in other languages. However, this similarity is only skin deep. In languages such as C++, enumerations are simply lists of named integer constants. In Java, an enumeration defines a class type. By making enumerations into classes, the concept of the enumeration is greatly expanded. For example, in Java, an enumeration can have constructors, methods, and instance variables. Therefore, although enumerations were several years in the making, Java's rich implementation made them well worth the wait.

Enumeration Fundamentals

An enumeration is created using the **enum** keyword. For example, here is a simple enumeration that lists various apple varieties:

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

255

The identifiers **Jonathan**, **GoldenDel**, and so on, are called *enumeration constants*. Each is implicitly declared as a public, static final member of **Apple**. Furthermore, their type is the type of the enumeration in which they are declared, which is **Apple** in this case. Thus, in the language of Java, these constants are called *self-typed*, in which "self" refers to the enclosing enumeration.

Once you have defined an enumeration, you can create a variable of that type. However, even though enumerations define a class type, you do not instantiate an **enum** using **new**. Instead, you declare and use an enumeration variable in much the same way as you do one of the primitive types. For example, this declares **ap** as a variable of enumeration type **Apple**:

```
Apple ap;
```

Because **ap** is of type **Apple**, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns **ap** the value **RedDel**:

```
ap = Apple.RedDel;
```

Notice that the symbol **RedDel** is preceded by **Apple**.

Two enumeration constants can be compared for equality by using the = = relational operator. For example, this statement compares the value in **ap** with the **GoldenDel** constant:

```
if(ap == Apple.GoldenDel) // ...
```

An enumeration value can also be used to control a **switch** statement. Of course, all of the **case** statements must use constants from the same **enum** as that used by the **switch** expression. For example, this **switch** is perfectly valid:

```
// Use an enum to control a switch statement.
switch(ap) {
  case Jonathan:
    // ...
  case Winesap:
    // ...
```

Notice that in the **case** statements, the names of the enumeration constants are used without being qualified by their enumeration type name. That is, **Winesap**, not **Apple.Winesap**, is used. This is because the type of the enumeration in the **switch** expression has already implicitly specified the **enum** type of the **case** constants. There is no need to qualify the constants in the **case** statements with their **enum** type name. In fact, attempting to do so will cause a compilation error.

When an enumeration constant is displayed, such as in a **println( )** statement, its name is output. For example, given this statement:

```
System.out.println(Apple.Winesap);
```

the name **Winesap** is displayed.

The following program puts together all of the pieces and demonstrates the **Apple** enumeration:

```
// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
  public static void main(String args[])
  {
    Apple ap;

    ap = Apple.RedDel;

    // Output an enum value.
    System.out.println("Value of ap: " + ap);
    System.out.println();

    ap = Apple.GoldenDel;

    // Compare two enum values.
    if(ap == Apple.GoldenDel)
      System.out.println("ap contains GoldenDel.\n");

    // Use an enum to control a switch statement.
    switch(ap) {
      case Jonathan:
        System.out.println("Jonathan is red.");
        break;
      case GoldenDel:
        System.out.println("Golden Delicious is yellow.");
        break;
      case RedDel:
        System.out.println("Red Delicious is red.");
        break;
      case Winesap:
        System.out.println("Winesap is red.");
        break;
      case Cortland:
        System.out.println("Cortland is red.");
        break;
    }
  }
}
```

The output from the program is shown here:

```
Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.
```

The values( ) and valueOf( ) Methods

All enumerations automatically contain two predefined methods: **values( )** and **valueOf( )**. Their general forms are shown here:

public static *enum-type*[ ] values( )
public static *enum-type* valueOf(String *str*)

The **values( )** method returns an array that contains a list of the enumeration constants. The **valueOf( )** method returns the enumeration constant whose value corresponds to the string passed in *str*. In both cases, *enum-type* is the type of the enumeration. For example, in the case of the **Apple** enumeration shown earlier, the return type of **Apple.valueOf("Winesap")** is **Winesap**.

The following program demonstrates the **values( )** and **valueOf( )** methods:

```
// Use the built-in enumeration methods.

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;

    System.out.println("Here are all Apple constants:");

    // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);

    System.out.println();

    // use valueOf()
    ap = Apple.valueOf("Winesap");
    System.out.println("ap contains " + ap);

  }
}
```

The output from the program is shown here:

```
   Here are all Apple constants:
   Jonathan
   GoldenDel
   RedDel
   Winesap
   Cortland

   ap contains Winesap
```

Notice that this program uses a for-each style **for** loop to cycle through the array of constants obtained by calling **values( )**. For the sake of illustration, the variable **allapples** was created and assigned a reference to the enumeration array. However, this step is not necessary because the **for** could have been written as shown here, eliminating the need for the **allapples** variable:

```
for(Apple a : Apple.values())
  System.out.println(a);
```

Now, notice how the value corresponding to the name **Winesap** was obtained by calling **valueOf( )**.

```
ap = Apple.valueOf("Winesap");
```

As explained, **valueOf( )** returns the enumeration value associated with the name of the constant represented as a string.

NOTE *C/C++ programmers will notice that Java makes it much easier to translate between the human-readable form of an enumeration constant and its binary value than do these other languages. This is a significant advantage to Java's approach to enumerations.*

Java Enumerations Are Class Types

As explained, a Java enumeration is a class type. Although you don't instantiate an **enum** using **new**, it otherwise has much the same capabilities as other classes. The fact that **enum** defines a class gives powers to the Java enumeration that enumerations in other languages simply do not have. For example, you can give them constructors, add instance variables and methods, and even implement interfaces.

It is important to understand that each enumeration constant is an object of its enumeration type. Thus, when you define a constructor for an **enum**, the constructor is called when each enumeration constant is created. Also, each enumeration constant has its own copy of any instance variables defined by the enumeration. For example, consider the following version of **Apple**:

```
// Use an enum constructor, instance variable, and method.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }

  int getPrice() { return price; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
    Apple ap;
```

```
        // Display price of Winesap.
        System.out.println("Winesap costs " +
                           Apple.Winesap.getPrice() +
                           " cents.\n");

        // Display all apples and prices.
        System.out.println("All apple prices:");
        for(Apple a : Apple.values())
          System.out.println(a + " costs " + a.getPrice() +
                             " cents.");
    }
}
```

The output is shown here:

```
    Winesap costs 15 cents.

    All apple prices:
    Jonathan costs 10 cents.
    GoldenDel costs 9 cents.
    RedDel costs 12 cents.
    Winesap costs 15 cents.
    Cortland costs 8 cents.
```

This version of **Apple** adds three things. The first is the instance variable **price**, which is used to hold the price of each variety of apple. The second is the **Apple** constructor, which is passed the price of an apple. The third is the method **getPrice( )**, which returns the value of **price**.

When the variable **ap** is declared in **main( )**, the constructor for **Apple** is called once for each constant that is specified. Notice how the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

These values are passed to the **p** parameter of **Apple( )**, which then assigns this value to **price**. Again, the constructor is called once for each constant.

Because each enumeration constant has its own copy of **price**, you can obtain the price of a specified type of apple by calling **getPrice( )**. For example, in **main( )** the price of a Winesap is obtained by the following call:

```
Apple.Winesap.getPrice()
```

The prices of all varieties are obtained by cycling through the enumeration using a **for** loop. Because there is a copy of **price** for each enumeration constant, the value associated with one constant is separate and distinct from the value associated with another constant. This is a powerful concept, which is only available when enumerations are implemented as classes, as Java does.

Although the preceding example contains only one constructor, an **enum** can offer two or more overloaded forms, just as can any other class. For example, this version of **Apple** provides a default constructor that initializes the price to –1, to indicate that no price data is available:

```
// Use an enum constructor.
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }

  // Overloaded constructor
  Apple() { price = -1; }

  int getPrice() { return price; }
}
```

Notice that in this version, **RedDel** is not given an argument. This means that the default constructor is called, and **RedDel**'s price variable is given the value 1.

Here are two restrictions that apply to enumerations. First, an enumeration can't inherit another class. Second, an **enum** cannot be a superclass. This means that an **enum** can't be extended. Otherwise, **enum** acts much like any other class type. The key is to remember that each of the enumeration constants is an object of the class in which it is defined.

Enumerations Inherit Enum

Although you can't inherit a superclass when declaring an **enum**, all enumerations automatically inherit one: **java.lang.Enum**. This class defines several methods that are available for use by all enumerations. The **Enum** class is described in detail in Part II, but three of its methods warrant a discussion at this time.

You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its *ordinal value,* and it is retrieved by calling the **ordinal( )** method, shown here:

   final int ordinal( )

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the **Apple** enumeration, **Jonathan** has an ordinal value of zero, **GoldenDel** has an ordinal value of 1, **RedDel** has an ordinal value of 2, and so on.

You can compare the ordinal value of two constants of the same enumeration by using the **compareTo( )** method. It has this general form:

   final int compareTo(*enum-type e*)

Here, *enum-type* is the type of the enumeration, and *e* is the constant being compared to the invoking constant. Remember, both the invoking constant and *e* must be of the same enumeration. If the invoking constant has an ordinal value less than *e*'s, then **compareTo( )** returns a negative value. If the two ordinal values are the same, then zero is returned. If the invoking constant has an ordinal value greater than *e*'s, then a positive value is returned.

You can compare for equality an enumeration constant with any other object by using **equals( )**, which overrides the **equals( )** method defined by **Object**. Although **equals( )** can compare an enumeration constant to any other object, those two objects will only be equal if

they both refer to the same constant, within the same enumeration. Simply having ordinal values in common will not cause **equals( )** to return true if the two constants are from different enumerations.

Remember, you can compare two enumeration references for equality by using **= =**.

The following program demonstrates the **ordinal( )**, **compareTo( )**, and **equals( )** methods:

```
// Demonstrate ordinal(), compareTo(), and equals().

// An enumeration of apple varieties.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all apple constants" +
                       " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());

    ap =  Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    System.out.println();

    // Demonstrate compareTo() and equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);

    System.out.println();

    if(ap.equals(ap2))
      System.out.println("Error!");

    if(ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);

    if(ap == ap3)
      System.out.println(ap + " == " + ap3);

  }
}
```

The output from the program is shown here:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel comes before RedDel
RedDel equals RedDel

RedDel equals RedDel
RedDel == RedDel
```

Another Enumeration Example

Before moving on, we will look at a different example that uses an **enum**. In Chapter 9, an automated "decision maker" program was created. In that version, variables called **NO**, **YES**, **MAYBE**, **LATER**, **SOON**, and **NEVER** were declared within an interface and used to represent the possible answers. While there is nothing technically wrong with that approach, the enumeration is a better choice. Here is an improved version of that program that uses an **enum** called **Answers** to define the answers. You should compare this version to the original in Chapter 9.

```java
// An improved version of the "Decision Maker"
// program from Chapter 9. This version uses an
// enum, rather than interface variables, to
// represent the answers.

import java.util.Random;

// An enumeration of the possible answers.
enum Answers {
  NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
  Random rand = new Random();
  Answers ask() {
    int prob = (int) (100 * rand.nextDouble());

    if (prob < 15)
      return Answers.MAYBE; // 15%
    else if (prob < 30)
      return Answers.NO;    // 15%
    else if (prob < 60)
      return Answers.YES;   // 30%
    else if (prob < 75)
      return Answers.LATER; // 15%
    else if (prob < 98)
      return Answers.SOON;  // 13%
```

```
      else
        return Answers.NEVER; // 2%
    }
}

class AskMe {
  static void answer(Answers result) {
    switch(result) {
      case NO:
        System.out.println("No");
        break;
      case YES:
        System.out.println("Yes");
        break;
      case MAYBE:
        System.out.println("Maybe");
        break;
      case LATER:
        System.out.println("Later");
        break;
      case SOON:
        System.out.println("Soon");
        break;
      case NEVER:
        System.out.println("Never");
        break;
    }
  }

  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

Type Wrappers

As you know, Java uses primitive types (also called simple types), such as **int** or **double**, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the sake of performance. Using objects for these values would add an unacceptable overhead to even the simplest of calculations. Thus, the primitive types are not part of the object hierarchy, and they do not inherit **Object**.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation. For example, you can't pass a primitive type by reference to a method. Also, many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types. To handle these (and other) situations, Java provides *type wrappers,* which are classes that encapsulate a primitive type within an object. The type wrapper classes are described

in detail in Part II, but they are introduced here because they relate directly to Java's autoboxing feature.

The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy. Each is briefly examined next.

Character

Character is a wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.
To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.
To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

boolean booleanValue( )

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values. These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**. All of the numeric type wrappers inherit the abstract class **Number**. **Number** declares methods that return the value of an object in each of the different number formats. These methods are shown here:

byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )

For example, **doubleValue( )** returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are implemented by each of the numeric type wrappers.

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for **Integer**:

Integer(int *num*)
Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

All of the type wrappers override **toString( )**. It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to **println( )**, for example, without having to convert it into its primitive type.

The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// Demonstrate a type wrapper.
class Wrap {
  public static void main(String args[]) {

    Integer iOb = new Integer(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*. Thus, in the program, this line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

The process of extracting a value from a type wrapper is called *unboxing*. For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, with the release of JDK 5, Java fundamentally improved on this through the addition of autoboxing, described next.

Autoboxing

Beginning with JDK 5, Java added two important features: *autoboxing* and *auto-unboxing*. Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed. There is no need to explicitly construct an object. Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as **intValue( )** or **doubleValue( )**.

The addition of autoboxing and auto-unboxing greatly streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values. It also helps prevent errors. Moreover, it is very important to generics, which operates only on objects. Finally, autoboxing makes working with the Collections Framework (described in Part II) much easier.

With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you. For example, here is the modern way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Notice that no object is explicitly created through the use of **new**. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox **iOb**, you can use this line:

```
int i = iOb; // auto-unbox
```

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// Demonstrate autoboxing/unboxing.
class AutoBox {
  public static void main(String args[]) {

    Integer iOb = 100; // autobox an int

    int i = iOb; // auto-unbox

    System.out.println(i + " " + iOb);  // displays 100 100
  }
}
```

Autoboxing and Methods

In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object; auto-unboxing takes place whenever an object must be converted into a primitive type. Thus, autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this example:

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {
  // Take an Integer parameter and return
  // an int value;
  static int m(Integer v) {
    return v ; // auto-unbox to int
  }
```

```
   public static void main(String args[]) {
      // Pass an int to m() and assign the return value
      // to an Integer.  Here, the argument 100 is autoboxed
      // into an Integer.   The return value is also autoboxed
      // into an Integer.
      Integer iOb = m(100);

      System.out.println(iOb);
   }
}
```

This program displays the following result:

```
100
```

In the program, notice that **m( )** specifies an **Integer** parameter and returns an **int** result. Inside **main( )**, **m( )** is passed the value 100. Because **m( )** is expecting an **Integer**, this value is automatically boxed. Then, **m( )** returns the **int** equivalent of its argument. This causes **v** to be auto-unboxed. Next, this **int** value is assigned to **iOb** in **main( )**, which causes the **int** return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.

class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    // The following automatically unboxes iOb,
    // performs the increment, and then reboxes
    // the result back into iOb.
    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    // Here, iOb is unboxed, the expression is
    // evaluated, and the result is reboxed and
    // assigned to iOb2.
    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);

    // The same expression is evaluated, but the
```

```
    // result is not reboxed.
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

  }
}
```

The output is shown here:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in **iOb** to be incremented. It works like this: **iOb** is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
  public static void main(String args[]) {

    Integer iOb = 100;
    Double dOb = 98.6;


    dOb = dOb + iOb;
    System.out.println("dOb after expression: " + dOb);
  }
}
```

The output is shown here:

```
dOb after expression: 198.6
```

As you can see, both the **Double** object **dOb** and the **Integer** object **iOb** participated in the addition, and the result was reboxed and stored in **dOb**.

Because of auto-unboxing, you can use integer numeric objects to control a **switch** statement. For example, consider this fragment:

```
Integer iOb = 2;

switch(iOb) {
  case 1: System.out.println("one");
    break;
  case 2: System.out.println("two");
    break;
```

```
    default: System.out.println("error");
}
```

When the **switch** expression is evaluated, **iOb** is unboxed and its **int** value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy. In the past, such code would have involved casts and calls to methods such as **intValue( )**.

Autoboxing/Unboxing Boolean and Character Values

As described earlier, Java also supplies wrappers for **boolean** and **char**. These are **Boolean** and **Character**. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {
  public static void main(String args[]) {

    // Autobox/unbox a boolean.
    Boolean b = true;

    // Below, b is auto-unboxed when used in
    // a conditional expression, such as an if.
    if(b) System.out.println("b is true");

    // Autobox/unbox a char.
    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char

    System.out.println("ch2 is " + ch2);
  }
}
```

The output is shown here:

```
b is true
ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of **b** inside the **if** conditional expression. As you should recall, the conditional expression that controls an **if** must evaluate to type **boolean**. Because of auto-unboxing, the **boolean** value contained within **b** is automatically unboxed when the conditional expression is evaluated. Thus, with the advent of autoboxing/unboxing, a **Boolean** object can be used to control an **if** statement.

Because of auto-unboxing, a **Boolean** object can now also be used to control any of Java's loop statements. When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do**/**while**, it is automatically unboxed into its **boolean** equivalent. For example, this is now perfectly valid code:

```
Boolean b;
// ...
while(b) { // ...
```

Autoboxing/Unboxing Helps Prevent Errors

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.
class UnboxingError {
  public static void main(String args[]) {

    Integer iOb = 1000; // autobox the value 1000

    int i = iOb.byteValue(); // manually unbox as byte !!!

    System.out.println(i);  // does not display 1000 !
  }
}
```

This program displays not the expected value of 1000, but 24! The reason is that the value inside **iOb** is manually unboxed by calling **byteValue( )**, which causes the truncation of the value stored in **iOb**, which is 1,000. This results in the garbage value of 24 being assigned to **i**. Auto-unboxing prevents this type of error because the value in **iOb** will always auto-unbox into a value compatible with **int**.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, new code should employ autoboxing/unboxing. It is the way that modern Java code will be written.

A Word of Warning

Now that Java includes autoboxing and auto-unboxing, some might be tempted to use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether. For example, with autoboxing/unboxing it is possible to write code like this:

```
// A bad use of autoboxing/unboxing!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

In this example, objects of type **Double** hold values that are used to calculate the hypotenuse of a right triangle. Although this code is technically correct and does, in fact, work properly, it is a very bad use of autoboxing/unboxing. It is far less efficient than the equivalent code written using the primitive type **double**. The reason is that each autobox and auto-unbox adds overhead that is not present if the primitive type is used.

In general, you should restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required. Autoboxing/unboxing was not added to Java as a "back door" way of eliminating the primitive types.

Annotations (Metadata)

Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file. This information, called an *annotation*, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged. However, this information can be used by various tools during both development and deployment. For example, an annotation might be processed by a source-code generator. The term *metadata* is also used to refer to this feature, but the term *annotation* is the most descriptive and more commonly used.

Annotation Basics

An annotation is created through a mechanism based on the **interface**. Let's begin with an example. Here is the declaration for an annotation called **MyAnno**:

```
// A simple annotation type.
@interface MyAnno {
  String str();
  int val();
}
```

First, notice the @ that precedes the keyword **interface**. This tells the compiler that an annotation type is being declared. Next, notice the two members **str( )** and **val( )**. All annotations consist solely of method declarations. However, you don't provide bodies for these methods. Instead, Java implements these methods. Moreover, the methods act much like fields, as you will see.

An annotation cannot include an **extends** clause. However, all annotation types automatically extend the **Annotation** interface. Thus, **Annotation** is a super-interface of all annotations. It is declared within the **java.lang.annotation** package. It overrides **hashCode( )**, **equals( )**, and **toString( )**, which are defined by **Object**. It also specifies **annotationType( )**, which returns a **Class** object that represents the invoking annotation.

Once you have declared an annotation, you can use it to annotate a declaration. Any type of declaration can have an annotation associated with it. For example, classes, methods, fields, parameters, and **enum** constants can be annotated. Even an annotation can be annotated. In all cases, the annotation precedes the rest of the declaration.

When you apply an annotation, you give values to its members. For example, here is an example of **MyAnno** being applied to a method:

```
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() { // ...
```

This annotation is linked with the method **myMeth( )**. Look closely at the annotation syntax. The name of the annotation, preceded by an @, is followed by a parenthesized list of member initializations. To give a member a value, that member's name is assigned a value. Therefore, in the example, the string "Annotation Example" is assigned to the **str** member of **MyAnno**.

Notice that no parentheses follow **str** in this assignment. When an annotation member is given a value, only its name is used. Thus, annotation members look like fields in this context.

Specifying a Retention Policy

Before exploring annotations further, it is necessary to discuss *annotation retention policies.* A retention policy determines at what point an annotation is discarded. Java defines three such policies, which are encapsulated within the **java.lang.annotation.RetentionPolicy** enumeration. They are **SOURCE**, **CLASS**, and **RUNTIME**.

An annotation with a retention policy of **SOURCE** is retained only in the source file and is discarded during compilation.

An annotation with a retention policy of **CLASS** is stored in the **.class** file during compilation. However, it is not available through the JVM during run time.

An annotation with a retention policy of **RUNTIME** is stored in the **.class** file during compilation and is available through the JVM during run time. Thus, **RUNTIME** retention offers the greatest annotation persistence.

A retention policy is specified for an annotation by using one of Java's built-in annotations: **@Retention**. Its general form is shown here:

@Retention(*retention-policy*)

Here, *retention-policy* must be one of the previously discussed enumeration constants. If no retention policy is specified for an annotation, then the default policy of **CLASS** is used.

The following version of **MyAnno** uses **@Retention** to specify the **RUNTIME** retention policy. Thus, **MyAnno** will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

Obtaining Annotations at Run Time by Use of Reflection

Although annotations are designed mostly for use by other development or deployment tools, if they specify a retention policy of **RUNTIME**, then they can be queried at run time by any Java program through the use of *reflection.* Reflection is the feature that enables information about a class to be obtained at run time. The reflection API is contained in the **java.lang.reflect** package. There are a number of ways to use reflection, and we won't examine them all here. We will, however, walk through a few examples that apply to annotations.

The first step to using reflection is to obtain a **Class** object that represents the class whose annotations you want to obtain. **Class** is one of Java's built-in classes and is defined in **java.lang**. It is described in detail in Part II. There are various ways to obtain a **Class** object. One of the easiest is to call **getClass( )**, which is a method defined by **Object**. Its general form is shown here:

final Class getClass( )

It returns the **Class** object that represents the invoking object. (**getClass( )** and several other reflection-related methods make use of the generics feature. However, because generics are not discussed until Chapter 14, these methods are shown and used here in their raw form. As a result, you will see a warning message when you compile the following programs. In Chapter 14, you will learn about generics in detail.)

After you have obtained a **Class** object, you can use its methods to obtain information about the various items declared by the class, including its annotations. If you want to obtain the annotations associated with a specific item declared within a class, you must first obtain an object that represents that item. For example, **Class** supplies (among others) the **getMethod( )**, **getField( )**, and **getConstructor( )** methods, which obtain information about a method, field, and constructor, respectively. These methods return objects of type **Method**, **Field**, and **Constructor**.

To understand the process, let's work through an example that obtains the annotations associated with a method. To do this, you first obtain a **Class** object that represents the class, and then call **getMethod( )** on that **Class** object, specifying the name of the method. **getMethod( )** has this general form:

Method getMethod(String *methName*, Class ... *paramTypes*)

The name of the method is passed in *methName*. If the method has arguments, then **Class** objects representing those types must also be specified by *paramTypes*. Notice that *paramTypes* is a varargs parameter. This means that you can specify as many parameter types as needed, including zero. **getMethod( )** returns a **Method** object that represents the method. If the method can't be found, **NoSuchMethodException** is thrown.

From a **Class**, **Method**, **Field**, or **Constructor** object, you can obtain a specific annotation associated with that object by calling **getAnnotation( )**. Its general form is shown here:

Annotation getAnnotation(Class *annoType*)

Here, *annoType* is a **Class** object that represents the annotation in which you are interested. The method returns a reference to the annotation. Using this reference, you can obtain the values associated with the annotation's members. The method returns **null** if the annotation is not found, which will be the case if the annotation does not have **RUNTIME** retention.

Here is a program that assembles all of the pieces shown earlier and uses reflection to display the annotation associated with a method.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // Annotate a method.
  @MyAnno(str = "Annotation Example", val = 100)
  public static void myMeth() {
    Meta ob = new Meta();

    // Obtain the annotation for this method
    // and display the values of the members.
    try {
```

```
      // First, get a Class object that represents
      // this class.
      Class c = ob.getClass();

      // Now, get a Method object that represents
      // this method.
      Method m = c.getMethod("myMeth");

      // Next, get the annotation for this class.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Finally, display the values.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

The output from the program is shown here:

```
Annotation Example 100
```

This program uses reflection as described to obtain and display the values of **str** and **val** in the **MyAnno** annotation associated with **myMeth( )** in the **Meta** class. There are two things to pay special attention to. First, in this line

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

notice the expression **MyAnno.class**. This expression evaluates to a **Class** object of type **MyAnno**, the annotation. This construct is called a *class literal.* You can use this type of expression whenever a **Class** object of a known class is needed. For example, this statement could have been used to obtain the **Class** object for **Meta**:

```
Class c = Meta.class;
```

Of course, this approach only works when you know the class name of an object in advance, which might not always be the case. In general, you can obtain a class literal for classes, interfaces, primitive types, and arrays.

The second point of interest is the way the values associated with **str** and **val** are obtained when they are output by the following line:

```
System.out.println(anno.str() + " " + anno.val());
```

Notice that they are invoked using the method-call syntax. This same approach is used whenever the value of an annotation member is required.

A Second Reflection Example

In the preceding example, **myMeth( )** has no parameters. Thus, when **getMethod( )** was called, only the name **myMeth** was passed. However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to **getMethod( )**. For example, here is a slightly different version of the preceding program:

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {

  // myMeth now has two arguments.
  @MyAnno(str = "Two Parameters", val = 19)
  public static void myMeth(String str, int i)
  {
    Meta ob = new Meta();

    try {
      Class c = ob.getClass();

      // Here, the parameter types are specified.
      Method m = c.getMethod("myMeth", String.class, int.class);

      MyAnno anno = m.getAnnotation(MyAnno.class);

      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

  public static void main(String args[]) {
    myMeth("test", 10);
  }
}
```

The output from this version is shown here:

```
Two Parameters 19
```

In this version, **myMeth( )** takes a **String** and an **int** parameter. To obtain information about this method, **getMethod( )** must be called as shown here:

```java
Method m = c.getMethod("myMeth", String.class, int.class);
```

Here, the **Class** objects representing **String** and **int** are passed as additional arguments.

Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling **getAnnotations( )** on that item. It has this general form:

Annotation[ ] getAnnotations( )

It returns an array of the annotations. **getAnnotations( )** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**.

Here is another reflection example that shows how to obtain all annotations associated with a class and with a method. It declares two annotations. It then uses those annotations to annotate a class and a method.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
  String description();
}

@What(description = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

  @What(description = "An annotation test method")
  @MyAnno(str = "Testing", val = 100)
  public static void myMeth() {
    Meta2 ob = new Meta2();

    try {
      Annotation annos[] = ob.getClass().getAnnotations();

      // Display all annotations for Meta2.
      System.out.println("All annotations for Meta2:");
      for(Annotation a : annos)
        System.out.println(a);

      System.out.println();

      // Display all annotations for myMeth.
      Method m = ob.getClass( ).getMethod("myMeth");
      annos = m.getAnnotations();

      System.out.println("All annotations for myMeth:");
      for(Annotation a : annos)
```

```
        System.out.println(a);

    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)

All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The program uses **getAnnotations( )** to obtain an array of all annotations associated with the **Meta2** class and with the **myMeth( )** method. As explained, **getAnnotations( )** returns an array of **Annotation** objects. Recall that **Annotation** is a super-interface of all annotation interfaces and that it overrides **toString( )** in **Object**. Thus, when a reference to an **Annotation** is output, its **toString( )** method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods **getAnnotation( )** and **getAnnotations( )** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for annotations and is implemented by the classes **Method**, **Field**, **Constructor**, **Class**, and **Package**.

In addition to **getAnnotation( )** and **getAnnotations( )**, **AnnotatedElement** defines two other methods. The first is **getDeclaredAnnotations( )**, which has this general form:

Annotation[ ] getDeclaredAnnotations( )

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent( )**, which has this general form:

boolean isAnnotationPresent(Class *annoType*)

It returns true if the annotation specified by *annoType* is associated with the invoking object. It returns false otherwise.

NOTE *The methods **getAnnotation( )** and **isAnnotationPresent( )** make use of the generics feature to ensure type safety. Because generics are not discussed until Chapter 14, their signatures are shown in this chapter in their raw forms.*

Using Default Values

You can give annotation members default values that will be used if no value is specified when the annotation is applied. A default value is specified by adding a **default** clause to a member's declaration. It has this general form:

   type member( ) default *value*;

Here, *value* must be of a type compatible with *type*.

Here is **@MyAnno** rewritten to include default values:

```
// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}
```

This declaration gives a default value of "Testing" to **str** and 9000 to **val**. This means that neither value needs to be specified when **@MyAnno** is used. However, either or both can be given values if desired. Therefore, following are the four ways that **@MyAnno** can be used:

```
  @MyAnno() // both str and val default
  @MyAnno(str = "some string") // val defaults
  @MyAnno(val = 100) // str defaults
  @MyAnno(str = "Testing", val = 100) // no defaults
```

The following program demonstrates the use of default values in an annotation.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration that includes defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}

class Meta3 {

  // Annotate a method using the default values.
  @MyAnno()
  public static void myMeth() {
    Meta3 ob = new Meta3();

    // Obtain the annotation for this method
    // and display the values of the members.
    try {
      Class c = ob.getClass();

      Method m = c.getMethod("myMeth");
```

```
        MyAnno anno = m.getAnnotation(MyAnno.class);

        System.out.println(anno.str() + " " + anno.val());
      } catch (NoSuchMethodException exc) {
        System.out.println("Method Not Found.");
      }
  }

  public static void main(String args[]) {
    myMeth();
  }
}
```

The output is shown here:

```
    Testing 9000
```

Marker Annotations

A *marker* annotation is a special kind of annotation that contains no members. Its sole purpose
is to mark a declaration. Thus, its presence as an annotation is sufficient. The best way to
determine if a marker annotation is present is to use the method **isAnnotationPresent( )**,
which is a defined by the **AnnotatedElement** interface.

Here is an example that uses a marker annotation. Because a marker interface contains
no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Annotate a method using a marker.
  // Notice that no ( ) is needed.
  @MyMarker
  public static void myMeth() {
    Marker ob = new Marker();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      // Determine if the annotation is present.
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("MyMarker is present.");

    } catch (NoSuchMethodException exc) {
      System.out.println("Method Not Found.");
    }
  }
```

```
  public static void main(String args[]) {
    myMeth();
  }
}
```

The output, shown here, confirms that **@MyMarker** is present:

```
MyMarker is present.
```

In the program, notice that you do not need to follow **@MyMarker** with parentheses when it is applied. Thus, **@MyMarker** is applied simply by using its name, like this:

```
@MyMarker
```

It is not wrong to supply an empty set of parentheses, but they are not needed.

Single-Member Annotations

A *single-member* annotation contains only one member. It works like a normal annotation except that it allows a shorthand form of specifying the value of the member. When only one member is present, you can simply specify the value for that member when the annotation is applied—you don't need to specify the name of the member. However, in order to use this shorthand, the name of the member must be **value**.

Here is an example that creates and uses a single-member annotation:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // this variable name must be value
}

class Single {

  // Annotate a method using a single-member annotation.
  @MySingle(100)
  public static void myMeth() {
    Single ob = new Single();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      MySingle anno = m.getAnnotation(MySingle.class);

      System.out.println(anno.value()); // displays 100

    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }
```

```
  public static void main(String args[]) {
    myMeth();
  }
}
```

As expected, this program displays the value 100. In the program, **@MySingle** is used to annotate **myMeth( )**, as shown here:

```
@MySingle(100)
```

Notice that **value =** need not be specified.

You can use the single-value syntax when applying an annotation that has other members, but those other members must all have default values. For example, here the value **xyz** is added, with a default value of zero:

```
@interface SomeAnno {
  int value();
  int xyz() default 0;
}
```

In cases in which you want to use the default for **xyz**, you can apply **@SomeAnno**, as shown next, by simply specifying the value of **value** by using the single-member syntax.

```
@SomeAnno(88)
```

In this case, **xyz** defaults to zero, and **value** gets the value 88. Of course, to specify a different value for **xyz** requires that both members be explicitly named, as shown here:

```
@SomeAnno(value = 88, xyz = 99)
```

Remember, whenever you are using a single-member annotation, the name of that member must be **value**.

The Built-In Annotations

Java defines many built-in annotations. Most are specialized, but seven are general purpose. Of these, four are imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**, and **@Inherited**. Three—**@Override**, **@Deprecated**, and **@SuppressWarnings**—are included in **java.lang**. Each is described here.

@Retention

@Retention is designed to be used only as an annotation to another annotation. It specifies the retention policy as described earlier in this chapter.

@Documented

The **@Documented** annotation is a marker interface that tells a tool that an annotation is to be documented. It is designed to be used only as an annotation to an annotation declaration.

@Target

The **@Target** annotation specifies the types of declarations to which an annotation can be applied. It is designed to be used only as an annotation to another annotation. **@Target** takes

one argument, which must be a constant from the **ElementType** enumeration. This argument specifies the types of declarations to which the annotation can be applied. The constants are shown here along with the type of declaration to which they correspond.

Target Constant	Annotation Can Be Applied To
ANNOTATION_TYPE	Another annotation
CONSTRUCTOR	Constructor
FIELD	Field
LOCAL_VARIABLE	Local variable
METHOD	Method
PACKAGE	Package
PARAMETER	Parameter
TYPE	Class, interface, or enumeration

You can specify one or more of these values in a **@Target** annotation. To specify multiple values, you must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this **@Target** annotation:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

@Inherited
@Inherited is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited**, then that annotation will be returned.

@Override
@Override is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result. It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

@Deprecated
@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.

@SuppressWarnings
@SuppressWarnings specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration.

Some Restrictions

There are a number of restrictions that apply to annotation declarations. First, no annotation can inherit another. Second, all methods declared by an annotation must be without parameters. Furthermore, they must return one of the following:

- A primitive type, such as **int** or **double**
- An object of type **String** or **Class**
- An **enum** type
- Another annotation type
- An array of one of the preceding types

Annotations cannot be generic. In other words, they cannot take type parameters. (Generics are described in Chapter 14.) Finally, annotation methods cannot specify a **throws** clause.