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.

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 the Java enumeration extraordinary power. 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 be equal only 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.

```
// 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, since 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 operate 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 the object is not 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:

```
// 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 is written.

A Word of Warning

Because of 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)

Since JDK 5, Java has supported a feature 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 something. Prior to JDK 8, annotations could be used only on declarations, and that is where we will begin. (JDK 8 adds the ability to annotate type use, and this is described later in this chapter. However, the same basic techniques apply to both kinds of annotations.) 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 declaration:

```
// 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.

NOTE An annotation on a local variable declaration is not retained in the **.class** file.

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.

NOTE Notice the **<?>** that follows **Class** in the declaration of **getClass( )** just shown. This is related to Java's generics feature. **getClass( )** and several other reflection-related methods discussed in this chapter make use of generics. Generics are described in Chapter 14. However, an understanding of generics is not needed to grasp the fundamental principles of reflection.

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:

<A extends Annotation> getAnnotation(Class<A> *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. (Remember, the **<?>** syntax relates to Java's generics feature. It is described in Chapter 14.)

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:

```
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:

```
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**, among others.

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**, among others.

In addition to **getAnnotation( )** and **getAnnotations( )**, **AnnotatedElement** defines several other methods. Two have been available since JDK 5. 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<? extends Annotation> annoType)
```

It returns **true** if the annotation specified by *annoType* is associated with the invoking object. It returns **false** otherwise. To these, JDK 8 adds **getDeclaredAnnotation( )**,

getAnnotationsByType( ), and **getDeclaredAnnotationsByType( )**. Of these, the last two automatically work with a repeated annotation. (Repeated annotations are discussed at the end of this chapter.)

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 an item. 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 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 nine are general purpose. Of these, four are imported from **java.lang.annotation**: **@Retention**, **@Documented**, **@Target**, and **@Inherited**. Five—**@Override**, **@Deprecated**, **@FunctionalInterface**, **@SafeVarargs**, and **@SuppressWarnings**—are included in **java.lang**. Each is described here.

NOTE To **java.lang.annotation**, JDK 8 adds the annotations **Repeatable** and **Native**. **Repeatable** supports repeatable annotations, as described later in this chapter. **Native** annotates a field that can be accessed by native code.

@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 items 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 is an array of constants of 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
TYPE_PARAMETER	Type parameter (Added by JDK 8.)
TYPE_USE	Type use (Added by JDK 8.)

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 } )
```

If you don't use **@Target**, then, except for type parameters, the annotation can be used on any declaration. For this reason, it is often a good idea to explicitly specify the target or targets so as to clearly indicate the intended uses of an annotation.

@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.

@FunctionalInterface

@FunctionalInterface is a marker annotation added by JDK 8 and designed for use on interfaces. It indicates that the annotated interface is a functional interface. A *functional interface* is an interface that contains one and only one abstract method. Functional interfaces are used by lambda expressions. (See Chapter 15 for details on functional interfaces and lambda expressions.) If the annotated interface is not a functional interface, a compilation error will be reported. It is important to understand that **@FunctionalInterface** is not needed to create a functional interface. Any interface with exactly one abstract method is, by definition, a functional interface. Thus, **@FunctionalInterface** is purely informational.

@SafeVarargs

@SafeVarargs is a marker annotation that can be applied to methods and constructors. It indicates that no unsafe actions related to a varargs parameter occur. It is used to suppress unchecked warnings on otherwise safe code as it relates to non-reifiable vararg types and parameterized array instantiation. (A non-reifiable type is, essentially, a generic type. Generics are described in Chapter 14.) It must be applied only to vararg methods or constructors that are **static** or **final**.

@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.

Type Annotations

Beginning with JDK 8, the places in which annotations can be used has been expanded. As mentioned earlier, annotations were originally allowed only on declarations. However, with the advent of JDK 8, annotations can also be specified in most cases in which a type is used. This expanded aspect of annotations is called *type annotation*. For example, you can annotate the return type of a method, the type of **this** within a method, a cast, array levels, an inherited class, and a **throws** clause. You can also annotate generic types, including generic type parameter bounds and generic type arguments. (See Chapter 14 for a discussion of generics.)

Type annotations are important because they enable tools to perform additional checks on code to help prevent errors. Understand that, as a general rule, **javac** will not perform these checks, itself. A separate tool is used for this purpose, although such a tool might operate as a compiler plug-in.

A type annotation must include **ElementType.TYPE_USE** as a target. (Recall that valid annotation targets are specified using the **@Target** annotation, as previously described.) A type annotation applies to the type that the annotation precedes. For example, assuming some type annotation called **@TypeAnno**, the following is legal:

```
void myMeth() throws @TypeAnno NullPointerException { // ...
```

Here, **@TypeAnno** annotates **NullPointerException** in the **throws** clause.

You can also annotate the type of **this** (called the *receiver*). As you know, **this** is an implicit argument to all instance methods and it refers to the invoking object. To annotate its type requires the use of another new JDK 8 feature. Beginning with JDK 8, you can explicitly declare **this** as the first parameter to a method. In this declaration, the type of **this** must be the type of its class; for example:

```
class SomeClass {
  int myMeth(SomeClass this, int i, int j) { // ...
```

Here, because **myMeth( )** is a method defined by **SomeClass**, the type of **this** is **SomeClass**. Using this declaration, you can now annotate the type of **this**. For example, again assuming that **@TypeAnno** is a type annotation, the following is legal:

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { // ...
```

It is important to understand that it is not necessary to declare **this** unless you are annotating it. (If **this** is not declared, it is still implicitly passed. JDK 8 *does not* change this fact.) Also, explicitly declaring **this** does not change any aspect of the method's signature because **this** is implicitly declared, by default. Again, you will declare **this** only if you want to apply a type annotation to it. If you do declare **this**, it *must* be the first parameter.

The following program shows a number of the places that a type annotation can be used. It defines several annotations, of which several are for type annotation. The names and targets of the annotations are shown here:

Annotation	Target
@TypeAnno	ElementType.TYPE_USE
@MaxLen	ElementType.TYPE_USE
@NotZeroLen	ElementType.TYPE_USE
@Unique	ElementType.TYPE_USE
@What	ElementType.TYPE_PARAMETER
@EmptyOK	ElementType.FIELD
@Recommended	ElementType.METHOD

Notice that **@EmptyOK**, **@Recommended**, and **@What** are not type annotations. They are included for comparison purposes. Of special interest is **@What**, which is used to annotate a generic type parameter declaration and is another new annotation feature added by JDK 8. The comments in the program describe each use.

```
// Demonstrate several type annotations.
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface TypeAnno { }

// Another marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface NotZeroLen {
}

// Still another marker annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface Unique { }

// A parameterized annotation that can be applied to a type.
@Target(ElementType.TYPE_USE)
@interface MaxLen {
  int value();
}

// An annotation that can be applied to a type parameter.
@Target(ElementType.TYPE_PARAMETER)
@interface What {
  String description();
}

// An annotation that can be applied to a field declaration.
@Target(ElementType.FIELD)
@interface EmptyOK { }

// An annotation that can be applied to a method declaration.
@Target(ElementType.METHOD)
@interface Recommended { }


// Use an annotation on a type parameter.
class TypeAnnoDemo<@What(description = "Generic data type") T> {

  // Use a type annotation on a constructor.
  public @Unique TypeAnnoDemo() {}

  // Annotate the type (in this case String), not the field.
  @TypeAnno String str;
```

```
  // This annotates the field test.
  @EmptyOK String test;

  // Use a type annotation to annotate this (the receiver).
  public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
    return 10;
  }

  // Annotate the return type.
  public @TypeAnno Integer f2(int j, int k) {
    return j+k;
  }

  // Annotate the method declaration.
  public @Recommended Integer f3(String str) {
    return str.length() / 2;
  }

  // Use a type annotation with a throws clause.
  public void f4() throws @TypeAnno NullPointerException {
    // ...
  }

  // Annotate array levels.
  String @MaxLen(10) [] @NotZeroLen [] w;

  // Annotate the array element type.
  @TypeAnno Integer[] vec;

  public static void myMeth(int i) {

    // Use a type annotation on a type argument.
    TypeAnnoDemo<@TypeAnno Integer> ob =
                              new TypeAnnoDemo<@TypeAnno Integer>();

    // Use a type annotation with new.
    @Unique TypeAnnoDemo<Integer> ob2 = new @Unique TypeAnnoDemo<Integer>();

    Object x = new Integer(10);
    Integer y;

    // Use a type annotation on a cast.
    y = (@TypeAnno Integer) x;
  }

  public static void main(String args[]) {
    myMeth(10);
  }

  // Use type annotation with inheritance clause.
  class SomeClass extends @TypeAnno TypeAnnoDemo<Boolean> {}
}
```

Although what most of the annotations in the preceding program refer to is clear, four uses require a bit of discussion. The first is the annotation of a method return type versus the annotation of a method declaration. In the program, pay special attention to these two method declarations:

```
// Annotate the return type.
public @TypeAnno Integer f2(int j, int k) {
  return j+k;
}

// Annotate the method declaration.
public @Recommended Integer f3(String str) {
  return str.length() / 2;
}
```

Notice that in both cases, an annotation precedes the method's return type (which is **Integer**). However, the two annotations annotate two different things. In the first case, the **@TypeAnno** annotation annotates **f2( )**'s return type. This is because **@TypeAnno** has its target specified as **ElementType.TYPE_USE**, which means that it can be used to annotate type uses. In the second case, **@Recommended** annotates the method declaration, itself. This is because **@Recommended** has its target specified as **ElementType.METHOD**. As a result, **@Recommended** applies to the declaration, not the return type. Therefore, the target specification is used to eliminate what, at first glance, appears to be ambiguity between the annotation of a method declaration and the annotation of the method's return type.

One other thing about annotating a method return type: You cannot annotate a return type of **void**.

The second point of interest are the field annotations, shown here:

```
// Annotate the type (in this case String), not the field.
@TypeAnno String str;

// This annotates the field test.
@EmptyOK String test;
```

Here, **@TypeAnno** annotates the type **String**, but **@EmptyOK** annotates the field **test**. Even though both annotations precede the entire declaration, their targets are different, based on the target element type. If the annotation has the **ElementType.TYPE_USE** target, then the type is annotated. If it has **ElementType_FIELD** as a target, then the field is annotated. Thus, the situation is similar to that just described for methods, and no ambiguity exists. The same mechanism also disambiguates annotations on local variables.

Next, notice how **this** (the receiver) is annotated here:

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

Here, **this** is specified as the first parameter and is of type **TypeAnnoDemo** (which is the class of which **f( )** is a member). As explained, beginning with JDK 8, an instance method declaration can explicitly specify the **this** parameter for the sake of applying a type annotation.

Finally, look at how array levels are annotated by the following statement:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

In this declaration, **@MaxLen** annotates the type of the first level and **@NotZeroLen** annotates the type of the second level. In this declaration

```
@TypeAnno Integer[] vec;
```

the element type **Integer** is annotated.

Repeating Annotations

Another new JDK 8 annotation feature enables an annotation to be repeated on the same element. This is called *repeating annotations*. For an annotation to be repeatable, it must be annotated with the **@Repeatable** annotation, defined in **java.lang.annotation**. Its **value** field specifies the *container* type for the repeatable annotation. The container is specified as an annotation for which the **value** field is an array of the repeatable annotation type. Thus, to create a repeatable annotation, you must create a container annotation and then specify that annotation type as an argument to the **@Repeatable** annotation.

To access the repeated annotations using a method such as **getAnnotation( )**, you will use the container annotation, not the repeatable annotation. The following program shows this approach. It converts the version of **MyAnno** shown previously into a repeatable annotation and demonstrates its use.

```
// Demonstrate a repeated annotation.

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

// Make MyAnno repeatable.
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}

// This is the container annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
  MyAnno[] value();
}

class RepeatAnno {

  // Repeat MyAnno on myMeth().
  @MyAnno(str = "First annotation", val = -1)
  @MyAnno(str = "Second annotation", val = 100)
  public static void myMeth(String str, int i)
```

```
  {
    RepeatAnno ob = new RepeatAnno();

    try {
      Class<?> c = ob.getClass();

      // Obtain the annotations for myMeth().
      Method m = c.getMethod("myMeth", String.class, int.class);

      // Display the repeated MyAnno annotations.
      Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
      System.out.println(anno);

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

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

The output is shown here:

```
@MyRepeatedAnnos(value=[@MyAnno(str=First annotation, val=-1),
@MyAnno(str=Second annotation, val=100)])
```

As explained, in order for **MyAnno** to be repeatable, it must be annotated with the **@Repeatable** annotation, which specifies its container annotation. The container annotation is called **MyRepeatedAnnos**. The program accesses the repeated annotations by calling **getAnnotation( )**, passing in the class of the container annotation, not the repeatable annotation, itself. As the output shows, the repeated annotations are separated by a comma. They are not returned individually.

Another way to obtain the repeated annotations is to use one of the new methods added to **AnnotatedElement** by JDK 8, which can operate directly on a repeated annotation. These are **getAnnotationsByType( )** and **getDeclaredAnnotationsByType( )**. Here, we will use the former. It is shown here:

```
<T extends Annotation> T[ ] getAnnotationsByType(Class<T> annoType)
```

It returns an array of the annotations of *annoType* associated with the invoking object. If no annotations are present, the array will be of zero length. Here is an example. Assuming the preceding program, the following sequence uses **getAnnotationsByType( )** to obtain the repeated **MyAnno** annotations:

```
Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
  System.out.println(a);
```

Here, the repeated annotation type, which is **MyAnno**, is passed to **getAnnotationsByType( )**. The returned array contains all of the instances of **MyAnno** associated with **myMeth( )**, which, in this example, is two. Each repeated annotation can be accessed via its index in the array. In this case, each **MyAnno** annotation is displayed via a for-each loop.

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.

This page has been intentionally left blank

CHAPTER

13 I/O, Applets, and Other Topics

This chapter introduces two of Java's most important packages: **io** and **applet**. The **io** package supports Java's basic I/O (input/output) system, including file I/O. The **applet** package supports applets. Support for both I/O and applets comes from Java's core API libraries, not from language keywords. For this reason, an in-depth discussion of these topics is found in Part II of this book, which examines Java's API classes. This chapter discusses the foundation of these two subsystems so that you can see how they are integrated into the Java language and how they fit into the larger context of the Java programming and execution environment. This chapter also examines the **try**-with-resources statement and the last of Java's keywords: **transient**, **volatile**, **instanceof**, **native**, **strictfp**, and **assert**. It concludes by discussing static import, describing another use for the **this** keyword, and introducing compact profiles (a feature added by JDK 8).

I/O Basics

As you may have noticed while reading the preceding 12 chapters, not much use has been made of I/O in the example programs. In fact, aside from **print( )** and **println( )**, none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are either graphically oriented programs that rely on one of Java's graphical user interface (GUI) frameworks, such as Swing, the AWT, or JavaFX, for user interaction, or they are Web applications. Although text-based, console programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not that useful in real-world Java programming.

The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master. A general overview of I/O is presented here. A detailed description is found in Chapters 20 and 21.

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package.

NOTE In addition to the stream-based I/O defined in **java.io**, Java also provides buffer- and channel-based I/O, which is defined in **java.nio** and its subpackages. They are described in Chapter 21.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn't use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in **java.io** are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import **java.io**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 13-1 The Byte Stream Classes in **java.io**

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in **java.io** are shown in Table 13-2.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 13-2 The Character Stream I/O Classes in **java.io**

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.

System.out refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

The preceding chapters have been using **System.out** in their examples. You can use **System.err** in much the same way. As explained in the next section, use of **System.in** is a little more complicated.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still acceptable. However, for commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. A commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
```

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

Reading Characters

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is

int read( ) throws IOException

Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an **IOException**.

The following program demonstrates **read( )** by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of **main( )**. Such an approach is common when reading from the console

in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```
// Use a BufferedReader to read characters from the console.
import java.io.*;

class BRRead {
  public static void main(String args[]) throws IOException
  {
    char c;
    BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");
    // read characters
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read( )** particularly valuable for interactive console input.

Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

String readLine( ) throws IOException

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine( )** method; the program reads and displays lines of text until you enter the word "stop":

```
// Read a string from console using a BufferedReader.
import java.io.*;
```

```
class BRReadLines {
  public static void main(String args[]) throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str;
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("stop"));
  }
}
```

The next example creates a tiny text editor. It creates an array of **String** objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop." It uses a **BufferedReader** to read from the console.

```
// A tiny editor.
import java.io.*;

class TinyEdit {
  public static void main(String args[]) throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                            InputStreamReader(System.in));
    String str[] = new String[100];
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'stop' to quit.");
    for(int i=0; i<100; i++) {
      str[i] = br.readLine();
      if(str[i].equals("stop")) break;
    }
    System.out.println("\nHere is your file:");
    // display the lines
    for(int i=0; i<100; i++) {
      if(str[i].equals("stop")) break;
      System.out.println(str[i]);
    }
  }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

```
stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

Writing Console Output

Console output is most easily accomplished with **print( )** and **println( )**, described earlier, which are used in most of the examples in this book. These methods are defined by the class **PrintStream** (which is the type of object referenced by **System.out**). Even though **System.out** is a byte stream, using it for simple program output is still acceptable. However, a character-based alternative is described in the next section.

Because **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write( )**. Thus, **write( )** can be used to write to the console. The simplest form of **write( )** defined by **PrintStream** is shown here:

void write(int *byteval*)

This method writes the byte specified by *byteval*. Although *byteval* is declared as an integer, only the low-order eight bits are written. Here is a short example that uses **write( )** to output the character "A" followed by a newline to the screen:

```
// Demonstrate System.out.write().
class WriteDemo {
  public static void main(String args[]) {
    int b;

    b = 'A';
    System.out.write(b);
    System.out.write('\n');
  }
}
```

You will not often use **write( )** to perform console output (although doing so might be useful in some situations) because **print( )** and **println( )** are substantially easier to use.

The PrintWriter Class

Although using **System.out** to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier.

PrintWriter defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, boolean *flushingOn*)

Here, *outputStream* is an object of type **OutputStream**, and *flushingOn* controls whether Java flushes the output stream every time a **println( )** method (among others) is called. If *flushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

PrintWriter supports the **print( )** and **println( )** methods. Thus, you can use these methods in the same way as you used them with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString( )** method and then display the result.

To write to the console by using a **PrintWriter**, specify **System.out** for the output stream and automatic flushing. For example, this line of code creates a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a **PrintWriter** to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;

public class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);

    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

The output from this program is shown here:

```
This is a string
-7
4.5E-7
```

Remember, there is nothing wrong with using **System.out** to write simple text output to the console when you are learning Java or debugging your programs. However, using a **PrintWriter** makes your real-world applications easier to internationalize. Because no advantage is gained by using a **PrintWriter** in the sample programs shown in this book, we will continue to use **System.out** to write to the console.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although bytes streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:

FileInputStream(String *fileName*) throws FileNotFoundException
FileOutputStream(String *fileName*) throws FileNotFoundException

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**. When an output file is opened, any preexisting file by the same name is destroyed.

NOTE In situations in which a security manager is present, several of the file classes, including **FileInputStream** and **FileOutputStream**, will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, other types of applications (such as applets) will use the security manager, and file I/O performed by such an application could generate a **SecurityException**. In that case, you will need to appropriately handle this exception.

When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**. It is shown here:

```
void close( ) throws IOException
```

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in "memory leaks" because of unused resources remaining allocated.

NOTE Beginning with JDK 7, the **close( )** method is specified by the **AutoCloseable** interface in **java.lang**. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes, including **FileInputStream** and **FileOutputStream**.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which **close( )** is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all pre-JDK 7 legacy code. The second is to use the **try**-with-resources statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to **close( )** is executed. Since there is a large amount of pre-JDK 7 legacy code that is still being used and maintained, it is important that you know and understand the traditional approach. Therefore, we will begin with it. The new automated approach is described in the following section.

To read from a file, you can use a version of **read( )** that is defined within **FileInputStream**. The one that we will use is shown here:

```
int read( ) throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read( )** returns –1 when the end of the file is encountered. It can throw an **IOException**.

The following program uses **read( )** to input and display the contents of a file that contains ASCII text. The name of the file is specified as a command-line argument.

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // Attempt to open the file.
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open File");
      return;
    }

    // At this point, the file is open and can be read.
    // The following reads characters until EOF is encountered.
    try {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Error Reading File");
   }

    // Close the file.
    try {
      fin.close();
    } catch(IOException e) {
        System.out.println("Error Closing File");
    }
  }
}
```

In the program, notice the **try/catch** blocks that handle the I/O errors that might occur. Each I/O operation is monitored for exceptions, and if an exception occurs, it is handled. Be aware that in simple programs or example code, it is common to see I/O exceptions simply thrown out of **main( )**, as was done in the earlier console I/O examples. Also, in some real-world code, it can be helpful to let an exception propagate to a calling routine to let the caller know that an I/O operation failed. However, most of the file I/O examples in this book handle all I/O exceptions explicitly, as shown, for the sake of illustration.

Although the preceding example closes the file stream after the file is read, there is a variation that is often useful. The variation is to call **close( )** within a **finally** block. In this approach, all of the methods that access the file are contained within a **try** block, and the **finally** block is used to close the file. This way, no matter how the **try** block terminates, the file is closed. Assuming the preceding example, here is how the **try** block that reads the file can be recoded:

```
try {
  do {
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);
} catch(IOException e) {
  System.out.println("Error Reading File");
} finally {
  // Close file on the way out of the try block.
  try {
    fin.close();
  } catch(IOException e) {
    System.out.println("Error Closing File");
  }
}
```

Although not an issue in this case, one advantage to this approach in general is that if the code that accesses a file terminates because of some non-I/O related exception, the file is still closed by the **finally** block.

Sometimes it's easier to wrap the portions of a program that open the file and access the file within a single **try** block (rather than separating the two) and then use a **finally** block to close the file. For example, here is another way to write the **ShowFile** program:

```
/* Display a text file.
   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT

   This variation wraps the code that opens and
   accesses the file within a single try block.
   The file is closed by the finally block.
*/
```

```
import java.io.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;
    FileInputStream fin = null;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // The following code opens a file, reads characters until EOF
    // is encountered, and then closes the file via a finally block.
    try {
      fin = new FileInputStream(args[0]);

      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(FileNotFoundException e) {
      System.out.println("File Not Found.");
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      // Close file in all cases.
      try {
        if(fin != null) fin.close();
      } catch(IOException e) {
        System.out.println("Error Closing File");
      }
    }
  }
}
```

In this approach, notice that **fin** is initialized to **null**. Then, in the **finally** block, the file is closed only if **fin** is not **null**. This works because **fin** will be non-**null** only if the file is successfully opened. Thus, **close( )** is not called if an exception occurs while opening the file.

It is possible to make the **try/catch** sequence in the preceding example a bit more compact. Because **FileNotFoundException** is a subclass of **IOException**, it need not be caught separately. For example, here is the sequence recoded to eliminate catching **FileNotFoundException**. In this case, the standard exception message, which describes the error, is displayed.

```
try {
  fin = new FileInputStream(args[0]);

  do {
```

```
    i = fin.read();
    if(i != -1) System.out.print((char) i);
  } while(i != -1);

} catch(IOException e) {
  System.out.println("I/O Error: " + e);
} finally {
  // Close file in all cases.
  try {
    if(fin != null) fin.close();
  } catch(IOException e) {
    System.out.println("Error Closing File");
  }
}
```

In this approach, any error, including an error opening the file, is simply handled by the single **catch** statement. Because of its compactness, this approach is used by many of the I/O examples in this book. Be aware, however, that this approach is not appropriate in cases in which you want to deal separately with a failure to open a file, such as might be caused if a user mistypes a filename. In such a situation, you might want to prompt for the correct name, for example, before entering a **try** block that accesses the file.

To write to a file, you can use the **write( )** method defined by **FileOutputStream**. Its simplest form is shown here:

void write(int *byteval*) throws IOException

This method writes the byte specified by *byteval* to the file. Although *byteval* is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an **IOException** is thrown. The next example uses **write( )** to copy a file:

```
/* Copy a file.
   To use this program, specify the name
   of the source file and the destination file.
   For example, to copy a file called FIRST.TXT
   to a file called SECOND.TXT, use the following
   command line.

   java CopyFile FIRST.TXT SECOND.TXT
*/

import java.io.*;

class CopyFile {
  public static void main(String args[]) throws IOException
  {
    int i;
    FileInputStream fin = null;
    FileOutputStream fout = null;

    // First, confirm that both files have been specified.
    if(args.length != 2) {
      System.out.println("Usage: CopyFile from to");
      return;
```

```
    }

    // Copy a File.
    try {
      // Attempt to open the files.
      fin = new FileInputStream(args[0]);
      fout = new FileOutputStream(args[1]);

      do {
        i = fin.read();
        if(i != -1) fout.write(i);
      } while(i != -1);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    } finally {
      try {
        if(fin != null) fin.close();
      } catch(IOException e2) {
        System.out.println("Error Closing Input File");
      }
      try {
        if(fout != null) fout.close();
      } catch(IOException e2) {
        System.out.println("Error Closing Output File");
      }
    }
  }
}
```

In the program, notice that two separate **try** blocks are used when closing the files. This ensures that both files are closed, even if the call to **fin.close( )** throws an exception.

In general, notice that all potential I/O errors are handled in the preceding two programs by the use of exceptions. This differs from some computer languages that use error codes to report file errors. Not only do exceptions make file handling cleaner, but they also enable Java to easily differentiate the end-of-file condition from file errors when input is being performed.

Automatically Closing a File

In the preceding section, the example programs have made explicit calls to **close( )** to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as *automatic resource management*, or *ARM* for short, is based on an expanded version of the **try** statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.