Automatic resource management is based on an expanded form of the **try** statement. Here is its general form:

```
try (resource-specification) {
  // use the resource
}
```

Here, *resource-specification* is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call **close( )** explicitly.) Of course, this form of **try** can also include **catch** and **finally** clauses. This new form of **try** is called the ***try*-*with*-*resources*** statement.

The **try**-with-resources statement can be used only with those resources that implement the **AutoCloseable** interface defined by **java.lang**. This interface defines the **close( )** method. **AutoCloseable** is inherited by the **Closeable** interface in **java.io**. Both interfaces are implemented by the stream classes. Thus, **try**-with-resources can be used when working with streams, including file streams.

As a first example of automatically closing a file, here is a reworked version of the **ShowFile** program that uses it:

```
/* This version of the ShowFile program uses a try-with-resources
   statement to automatically close a file after it is no longer needed.

   Note: This code requires JDK 7 or later.
*/

import java.io.*;

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

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

    // The following code uses a try-with-resources statement to open
    // a file and then automatically close it when the try block is left.
    try(FileInputStream 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");
    }

  }
}
```

In the program, pay special attention to how the file is opened within the **try** statement:

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

Notice how the resource-specification portion of the **try** declares a **FileInputStream** called **fin**, which is then assigned a reference to the file opened by its constructor. Thus, in this version of the program, the variable **fin** is local to the **try** block, being created when the **try** is entered. When the **try** is left, the stream associated with **fin** is automatically closed by an implicit call to **close( )**. You don't need to call **close( )** explicitly, which means that you can't forget to close the file. This is a key advantage of using **try**-with-resources.

It is important to understand that the resource declared in the **try** statement is implicitly **final**. This means that you can't assign to the resource after it has been created. Also, the scope of the resource is limited to the **try**-with-resources statement.

You can manage more than one resource within a single **try** statement. To do so, simply separate each resource specification with a semicolon. The following program shows an example. It reworks the **CopyFile** program shown earlier so that it uses a single **try**-with-resources statement to manage both **fin** and **fout**.

```
/* A version of CopyFile that uses try-with-resources.
   It demonstrates two resources (in this case files) being
   managed by a single try statement.
*/

import java.io.*;

class CopyFile {
  public static void main(String args[]) throws IOException
  {
    int i;

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

    // Open and manage two files via the try statement.
    try (FileInputStream fin = new FileInputStream(args[0]);
         FileOutputStream 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);
    }
  }
}
```

In this program, notice how the input and output files are opened within the **try** block:

```
try (FileInputStream fin = new FileInputStream(args[0]);
     FileOutputStream fout = new FileOutputStream(args[1]))
{
  // ...
```

After this **try** block ends, both **fin** and **fout** will have been closed. If you compare this version of the program to the previous version, you will see that it is much shorter. The ability to streamline source code is a side-benefit of automatic resource management.

There is one other aspect to **try**-with-resources that needs to be mentioned. In general, when a **try** block executes, it is possible that an exception inside the **try** block will lead to another exception that occurs when the resource is closed in a **finally** clause. In the case of a "normal" **try** statement, the original exception is lost, being preempted by the second exception. However, when using **try**-with-resources, the second exception is *suppressed*. It is not, however, lost. Instead, it is added to the list of suppressed exceptions associated with the first exception. The list of suppressed exceptions can be obtained by using the **getSuppressed( )** method defined by **Throwable**.

Because of the benefits that the **try**-with-resources statement offers, it will be used by many, but not all, of the example programs in this edition of this book. Some of the examples will still use the traditional approach to closing a resource. There are several reasons for this. First, there is legacy code that still relies on the traditional approach. It is important that all Java programmers be fully versed in, and comfortable with, the traditional approach when maintaining this older code. Second, because not all project development will immediately switch to a new version of the JDK, it is likely that some programmers will continue to work in a pre-JDK 7 environment for a period of time. In such situations, the expanded form of **try** is not available. Finally, there may be cases in which explicitly closing a resource is more appropriate than the automated approach. For these reasons, some of the examples in this book will continue to use the traditional approach, explicitly calling **close( )**. In addition to illustrating the traditional technique, these examples can also be compiled and run by all readers in all environments.

REMEMBER A few examples in this book use the traditional approach to closing files as a means of illustrating this technique, which is widely used in legacy code. However, for new code, you will usually want to use the new automated approach supported by the **try**-with-resources statement just described.

Applet Fundamentals

All of the preceding examples in this book have been Java console-based applications. However, these types of applications constitute only one class of Java programs. Another type of program is the applet. As mentioned in Chapter 1, *applets* are small applications that

are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run various computations without introducing the risk of viruses or breaching data integrity.

Many of the issues connected with the creation and use of applets are found in Part II, when the **applet** package is examined, and also when Swing is described in Part III. However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas.

Let's begin with the simple applet shown here:

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("A Simple Applet", 20, 20);
  }
}
```

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through a GUI framework, not through the console-based I/O classes. One of these frameworks is the AWT, and that is the framework used here to introduce applet programming. The AWT contains very basic support for a window-based, graphical user interface. As you might expect, the AWT is quite large, and a detailed discussion of it is found in Part II of this book. Fortunately, this simple applet makes very limited use of the AWT. (Another commonly used GUI for applets is Swing, but this approach is described later in this book.) The second **import** statement imports the **applet** package, which contains the class **Applet**. Every AWT-based applet that you create must be a subclass (either directly or indirectly) of **Applet**.

The next line in the program declares the class **SimpleApplet**. This class must be declared as **public**, because it will be accessed by code that is outside the program.

Inside **SimpleApplet**, **paint( )** is declared. This method is defined by the AWT and must be overridden by the applet. **paint( )** is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x*,*y*. In a Java window, the upper-left corner is location 0,0. The call to **drawString( )** in the applet causes the message "A Simple Applet" to be displayed beginning at location 20,20.

Notice that the applet does not have a **main( )** method. Unlike Java programs, applets do not begin execution at **main( )**. In fact, most applets don't even have a **main( )** method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

After you enter the source code for **SimpleApplet**, compile in the same way that you have been compiling programs. However, running **SimpleApplet** involves a different process. In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.
- Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

One way to execute an applet in a web browser is to write a short HTML text file that contains a tag that loads the applet. At the time of this writing, Oracle recommends using the APPLET tag for this purpose. (The OBJECT tag can also be used. See Chapter 23 for further information regarding applet deployment strategies.) Using APPLET, here is the HTML file that executes **SimpleApplet**:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The **width** and **height** statements specify the dimensions of the display area used by the applet. (The APPLET tag contains several other options that are examined more closely in Part II.) After you create this file, you can use it to execute the applet.

NOTE Beginning with the release of Java 7, update 21, Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running. Applets stored in the local file system, such as you would create when compiling the examples in this book, are especially sensitive to this change. You may need to adjust the security settings in the Java Control Panel to run a local applet in a browser. At the time of this writing, Oracle recommends against the use of local applets, recommending instead that applets be executed through a web server. Furthermore, it is expected that unsigned local applets will be blocked from execution in the future. In general, for applets that will be distributed via the Internet, such as commercial applications, signing is a virtual necessity. The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this book. However, extensive information is found on Oracle's website. Finally, as mentioned, the easiest way to try the applet examples is to use **appletviewer**.

To execute **SimpleApplet** with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called **RunApp.html**, then the following command line will run **SimpleApplet**:

```
C:\>appletviewer RunApp.html
```

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the

APPLET tag. By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the **SimpleApplet** source file looks like this:

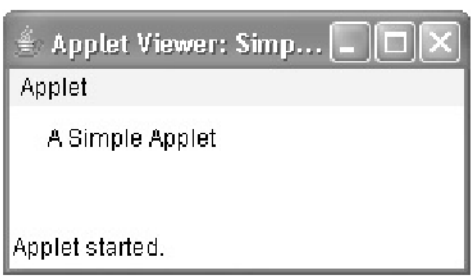
```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
  public void paint(Graphics g) {
    g.drawString("A Simple Applet", 20, 20);
  }
}
```

With this approach, you can quickly iterate through applet development by using these three steps:

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

The window produced by **SimpleApplet**, as displayed by the applet viewer, is shown in the following illustration. Of course, the precise appearance of the applet viewer frame may differ based on your environment. For this reason, the screen captures in this book reflect a number of different environments.



While the subject of applets is more fully discussed later in this book, here are the key points that you should remember now:

- Applets do not need a **main( )** method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java's stream I/O classes. Instead, applets use the interface provided by a GUI framework.

The transient and volatile Modifiers

Java defines two interesting type modifiers: **transient** and **volatile**. These modifiers are used to handle somewhat specialized situations.

When an instance variable is declared as **transient**, then its value need not persist when an object is stored. For example:

```
class T {
  transient int a; // will not persist
  int b; // will persist
}
```

Here, if an object of type **T** is written to a persistent storage area, the contents of **a** would not be saved, but the contents of **b** would.

The **volatile** modifier tells the compiler that the variable modified by **volatile** can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or *master*) copy of the variable is updated at various times, such as when a **synchronized** method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as **volatile**, which tells the compiler that it must always use the master copy of a **volatile** variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy.

Using instanceof

Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object's type at run time is important involves casting. In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can be detected only at run time. For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn't legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator **instanceof** to answer this question.

The **instanceof** operator has this general form:

objref instanceof *type*

Here, *objref* is a reference to an instance of a class, and *type* is a class type. If *objref* is of the specified type or can be cast into the specified type, then the **instanceof** operator evaluates to **true**. Otherwise, its result is **false**. Thus, **instanceof** is the means by which your program can obtain run-time type information about an object.

The following program demonstrates **instanceof**:

```
// Demonstrate instanceof operator.
class A {
  int i, j;
}

class B {
  int i, j;
}

class C extends A {
  int k;
}

class D extends A {
  int k;
}

class InstanceOf {
  public static void main(String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
    D d = new D();
    if(a instanceof A)
      System.out.println("a is instance of A");
    if(b instanceof B)
      System.out.println("b is instance of B");
    if(c instanceof C)
      System.out.println("c is instance of C");
    if(c instanceof A)
      System.out.println("c can be cast to A");

    if(a instanceof C)
      System.out.println("a can be cast to C");

    System.out.println();

    // compare types of derived types
    A ob;

    ob = d; // A reference to d
    System.out.println("ob now refers to d");
    if(ob instanceof D)
      System.out.println("ob is instance of D");

    System.out.println();

    ob = c; // A reference to c
    System.out.println("ob now refers to c");

    if(ob instanceof D)
      System.out.println("ob can be cast to D");
    else
      System.out.println("ob cannot be cast to D");

    if(ob instanceof A)
      System.out.println("ob can be cast to A");

    System.out.println();
```

```
    // all objects can be cast to Object
    if(a instanceof Object)
      System.out.println("a may be cast to Object");
    if(b instanceof Object)
      System.out.println("b may be cast to Object");
    if(c instanceof Object)
      System.out.println("c may be cast to Object");
    if(d instanceof Object)
      System.out.println("d may be cast to Object");
  }
}
```

The output from this program is shown here:

```
a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

The **instanceof** operator isn't needed by most programs, because, generally, you know the type of object with which you are working. However, it can be very useful when you're writing generalized routines that operate on objects of a complex class hierarchy.

strictfp

With the creation of Java 2, the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases. By modifying a class, a method, or interface with **strictfp**, you ensure that floating-point calculations (and thus all truncations) take place precisely as they did in earlier versions of Java. When a class is modified by **strictfp**, all the methods in the class are also modified by **strictfp** automatically.

For example, the following fragment tells Java to use the original floating-point model for calculations in all methods defined within **MyClass**:

```
strictfp class MyClass { //...
```

Frankly, most programmers never need to use **strictfp**, because it affects only a very small class of problems.

Native Methods

Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on-the-fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the **native** modifier, but do not define any body for the method. For example:

```
public native int meth() ;
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code.

Most native methods are written in C. The mechanism used to integrate C code with a Java program is called the *Java Native Interface (JNI)*. A detailed description of the JNI is beyond the scope of this book, but the approach described here provides sufficient information for simple applications.

NOTE The precise steps that you need to follow will vary between different Java environments. They also depend on the language that you are using to implement the native method. The following discussion assumes a Windows environment. The language used to implement the native method is C. Also, the approach shown here uses a dynamically linked library, but beginning with JDK 8, it is possible to create a statically linked library.

The easiest way to understand the process is to work through an example. To begin, enter the following short program, which uses a **native** method called **test( )**:

```
// A simple example that uses a native method.
public class NativeDemo {
  int i;
  public static void main(String args[]) {
    NativeDemo ob = new NativeDemo();

    ob.i = 10;
    System.out.println("This is ob.i before the native method:" +
                       ob.i);
    ob.test(); // call a native method
    System.out.println("This is ob.i after the native method:" +
                       ob.i);
  }

  // declare native method
  public native void test() ;

  // load DLL that contains static method
```

```
  static {
    System.loadLibrary("NativeDemo");
  }
}
```

Notice that the **test( )** method is declared as **native** and has no body. This is the method that we will implement in C shortly. Also notice the **static** block. As explained earlier in this book, a **static** block is executed only once, when your program begins execution (or, more precisely, when its class is first loaded). In this case, it is used to load the dynamic link library that contains the native implementation of **test( )**. (You will see how to create this library soon.)

The library is loaded by the **loadLibrary( )** method, which is part of the **System** class. This is its general form:

static void loadLibrary(String *filename*)

Here, *filename* is a string that specifies the name of the file that holds the library. For the Windows environment, this file is assumed to have the .DLL extension.

After you enter the program, compile it to produce **NativeDemo.class**. Next, you must use **javah.exe** to produce one file: **NativeDemo.h**. (**javah.exe** is included in the JDK.) You will include **NativeDemo.h** in your implementation of **test( )**. To produce **NativeDemo.h**, use the following command:

javah -jni NativeDemo

This command produces a header file called **NativeDemo.h**. This file must be included in the C file that implements **test( )**. The output produced by this command is shown here:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef _ _cplusplus
extern "C" {
#endif
/*
 * Class:     NativeDemo
 * Method:    test
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeDemo_test
   (JNIEnv *, jobject);

#ifdef _ _cplusplus
}
#endif
#endif
```

Pay special attention to the following line, which defines the prototype for the **test( )** function that you will create:

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Notice that the name of the function is **Java_NativeDemo_test( )**. You must use this as the name of the native function that you implement. That is, instead of creating a C function called **test( )**, you will create one called **Java_NativeDemo_test( )**. The **NativeDemo** component of the prefix is added because it identifies the **test( )** method as being part of the **NativeDemo** class. Remember, another class may define its own native **test( )** method that is completely different from the one declared by **NativeDemo**. Including the class name in the prefix provides a way to differentiate between differing versions. As a general rule, native functions will be given a name whose prefix includes the name of the class in which they are declared.

After producing the necessary header file, you can write your implementation of **test( )** and store it in a file named **NativeDemo.c**:

```
/* This file contains the C version of the
   test() method.
*/

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
  jclass cls;
  jfieldID fid;
  jint i;

  printf("Starting the native method.\n");
  cls = (*env)->GetObjectClass(env, obj);
  fid = (*env)->GetFieldID(env, cls, "i", "I");

  if(fid == 0) {
    printf("Could not get field id.\n");
    return;
  }
  i = (*env)->GetIntField(env, obj, fid);
  printf("i = %d\n", i);
  (*env)->SetIntField(env, obj, fid, 2*i);
  printf("Ending the native method.\n");
}
```

Notice that this file includes **jni.h**, which contains interfacing information. This file is provided by your Java compiler. The header file **NativeDemo.h** was created by **javah** earlier.

In this function, the **GetObjectClass( )** method is used to obtain a C structure that has information about the class **NativeDemo**. The **GetFieldID( )** method returns a C structure with information about the field named "i" for the class. **GetIntField( )** retrieves the original value of that field. **SetIntField( )** stores an updated value in that field. (See the file **jni.h** for additional methods that handle other types of data.)

After creating **NativeDemo.c**, you must compile it and create a DLL. To do this by using the Microsoft C/C++ compiler, use the following command line. (You might need to specify the path to **jni.h** and its subordinate file **jni_md.h**.)

```
Cl /LD NativeDemo.c
```

This produces a file called **NativeDemo.dll**. Once this is done, you can execute the Java program, which will produce the following output:

```
This is ob.i before the native method: 10
Starting the native method.
i = 10
Ending the native method.
This is ob.i after the native method: 20
```

Problems with Native Methods

Native methods seem to offer great promise, because they enable you to gain access to an existing base of library routines, and they offer the possibility of faster run-time execution. But native methods also introduce two significant problems:

- **Potential security risk** Because a native method executes actual machine code, it can gain access to any part of the host system. That is, native code is not confined to the Java execution environment. This could allow a virus infection, for example. For this reason, unsigned applets cannot use native methods. Also, the loading of DLLs can be restricted, and their loading is subject to the approval of the security manager.
- **Loss of portability** Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program. Further, because each native method is CPU- and operating system–dependent, each DLL is inherently nonportable. Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed.

The use of native methods should be restricted, because they render your Java programs nonportable and pose significant security risks.

Using assert

Another relatively new addition to Java is the keyword **assert**. It is used during program development to create an *assertion*, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an **assert** statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an **AssertionError** is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The **assert** keyword has two forms. The first is shown here:

assert *condition*;

Here, *condition* is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default **AssertionError** object is thrown.

The second form of **assert** is shown here:

assert *condition: expr*;

In this version, *expr* is a value that is passed to the **AssertionError** constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for *expr*, but any non-**void** expression is allowed as long as it defines a reasonable string conversion.

Here is an example that uses **assert**. It verifies that the return value of **getnum( )** is positive.

```
// Demonstrate assert.
class AssertDemo {
  static int val = 3;

  // Return an integer.
  static int getnum() {
    return val--;
  }

  public static void main(String args[])
  {
    int n;

    for(int i=0; i < 10; i++) {
      n = getnum();

      assert n > 0; // will fail when n is 0

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

To enable assertion checking at run time, you must specify the **-ea** option. For example, to enable assertions for **AssertDemo**, execute it using this line:

```
java -ea AssertDemo
```

After compiling and running as just described, the program creates the following output:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError
        at AssertDemo.main(AssertDemo.java:17)
```

In **main( )**, repeated calls are made to the method **getnum( )**, which returns an integer value. The return value of **getnum( )** is assigned to **n** and then tested using this **assert** statement:

```
assert n > 0; // will fail when n is 0
```

This statement will fail when **n** equals 0, which it will after the fourth call. When this happens, an exception is thrown.

As explained, you can specify the message displayed when an assertion fails. For example, if you substitute

```
assert n > 0 : "n is negative!";
```

for the assertion in the preceding program, then the following output will be generated:

```
n is 3
n is 2
n is 1
Exception in thread "main" java.lang.AssertionError: n is
negative!
        at AssertDemo.main(AssertDemo.java:17)
```

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled. For example, consider this variation of the preceding program:

```
// A poor way to use assert!!!
class AssertDemo {
  // get a random number generator
  static int val = 3;

  // Return an integer.
  static int getnum() {
    return val--;
  }

  public static void main(String args[])
  {
    int n = 0;

    for(int i=0; i < 10; i++) {

      assert (n = getnum()) > 0; // This is not a good idea!

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

In this version of the program, the call to **getnum( )** is moved inside the **assert** statement. Although this works fine if assertions are enabled, it will cause a malfunction when assertions are disabled, because the call to **getnum( )** will never be executed! In fact, **n** must now be

initialized, because the compiler will recognize that it might not be assigned a value by the **assert** statement.

Assertions are a good addition to Java because they streamline the type of error checking that is common during development. For example, prior to **assert**, if you wanted to verify that **n** was positive in the preceding program, you had to use a sequence of code similar to this:

```
if(n < 0) {
   System.out.println("n is negative!");
   return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

Assertion Enabling and Disabling Options

When executing code, you can disable all assertions by using the **-da** option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the **-ea** or **-da** option. For example, to enable assertions in a package called **MyPack**, use

```
-ea:MyPack...
```

To disable assertions in **MyPack**, use

```
-da:MyPack...
```

You can also specify a class with the **-ea** or **-da** option. For example, this enables **AssertDemo** individually:

```
-ea:AssertDemo
```

Static Import

Java includes a feature called *static import* that expands the capabilities of the **import** keyword. By following **import** with the keyword **static**, an **import** statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member.

To understand the usefulness of static import, let's begin with an example that does *not* use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java's built-in math class **Math**, which is part of **java.lang**. The first is **Math.pow( )**, which returns a value raised to a specified power. The second is **Math.sqrt( )**, which returns the square root of its argument.

```
// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
```

```
    double hypot;
    side1 = 3.0;
    side2 = 4.0;

    // Notice how sqrt() and pow() must be qualified by
    // their class name, which is Math.
    hypot = Math.sqrt(Math.pow(side1, 2) +
                      Math.pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

Because **pow( )** and **sqrt( )** are static methods, they must be called through the use of their class' name, **Math**. This results in a somewhat unwieldy hypotenuse calculation:

```
hypot = Math.sqrt(Math.pow(side1, 2) +
                  Math.pow(side2, 2));
```

As this simple example illustrates, having to specify the class name each time **pow( )** or **sqrt( )** (or any of Java's other math methods, such as **sin( )**, **cos( )**, and **tan( )**) is used can grow tedious.

You can eliminate the tedium of specifying the class name through the use of static import, as shown in the following version of the preceding program:

```
// Use static import to bring sqrt() and pow() into view.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Here, sqrt() and pow() can be called by themselves,
    // without their class name.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

In this version, the names **sqrt** and **pow** are brought into view by these static import statements:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

After these statements, it is no longer necessary to qualify **sqrt( )** or **pow( )** with their class name. Therefore, the hypotenuse calculation can more conveniently be specified, as shown here:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

As you can see, this form is considerably more readable.

There are two general forms of the **import static** statement. The first, which is used by the preceding example, brings into view a single name. Its general form is shown here:

import static *pkg.type-name.static-member-name*;

Here, *type-name* is the name of a class or interface that contains the desired static member. Its full package name is specified by *pkg*. The name of the member is specified by *static-member-name*.

The second form of static import imports all static members of a given class or interface. Its general form is shown here:

import static *pkg.type-name.**;

If you will be using many static methods or fields defined by a class, then this form lets you bring them into view without having to specify each individually. Therefore, the preceding program could have used this single **import** statement to bring both **pow( )** and **sqrt( )** (and *all other* static members of **Math**) into view:

```
import static java.lang.Math.*;
```

Of course, static import is not limited just to the **Math** class or just to methods. For example, this brings the static field **System.out** into view:

```
import static java.lang.System.out;
```

After this statement, you can output to the console without having to qualify **out** with **System**, as shown here:

```
out.println("After importing System.out, you can use out directly.");
```

Whether importing **System.out** as just shown is a good idea is subject to debate. Although it does shorten the statement, it is no longer instantly clear to anyone reading the program that the **out** being referred to is **System.out**.

One other point: in addition to importing the static members of classes and interfaces defined by the Java API, you can also use static import to import the static members of classes and interfaces that you create.

As convenient as static import can be, it is important not to abuse it. Remember, the reason that Java organizes its libraries into packages is to avoid namespace collisions. When you import static members, you are bringing those members into the global namespace. Thus, you are increasing the potential for namespace conflicts and for the inadvertent hiding of other names. If you are using a static member once or twice in the program, it's best not to import it. Also, some static names, such as **System.out**, are so recognizable that you might not want to import them. Static import is designed for those situations in which you are using a static member repeatedly, such as when performing a series of mathematical computations. In essence, you should use, but not abuse, this feature.

Invoking Overloaded Constructors Through this( )

When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the **this** keyword. The general form is shown here:

this(*arg-list*)

When **this( )** is executed, the overloaded constructor that matches the parameter list specified by *arg-list* is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to **this( )** must be the first statement within the constructor.

To understand how **this( )** can be used, let's work through a short example. First, consider the following class that *does not* use **this( )**:

```
class MyClass {
  int a;
  int b;

  // initialize a and b individually
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // initialize a and b to the same value
  MyClass(int i) {
    a = i;
    b = i;
  }

  // give a and b default values of 0
  MyClass( ) {
    a = 0;
    b = 0;
  }
}
```

This class contains three constructors, each of which initializes the values of **a** and **b**. The first is passed individual values for **a** and **b**. The second is passed just one value, which is assigned to both **a** and **b**. The third gives **a** and **b** default values of zero.

By using **this( )**, it is possible to rewrite **MyClass** as shown here:

```
class MyClass {
  int a;
  int b;

  // initialize a and b individually
  MyClass(int i, int j) {
    a = i;
    b = j;
  }

  // initialize a and b to the same value
  MyClass(int i) {
    this(i, i); // invokes MyClass(i, i)
  }

  // give a and b default values of 0
  MyClass( ) {
    this(0); // invokes MyClass(0)
  }
}
```

In this version of **MyClass**, the only constructor that actually assigns values to the **a** and **b** fields is **MyClass(int, int)**. The other two constructors simply invoke that constructor (either directly or indirectly) through **this( )**. For example, consider what happens when this statement executes:

```
MyClass mc = new MyClass(8);
```

The call to **MyClass(8)** causes **this(8, 8)** to be executed, which translates into a call to **MyClass(8, 8)**, because this is the version of the **MyClass** constructor whose parameter list matches the arguments passed via **this( )**. Now, consider the following statement, which uses the default constructor:

```
MyClass mc2 = new MyClass();
```

In this case, **this(0)** is called. This causes **MyClass(0)** to be invoked because it is the constructor with the matching parameter list. Of course, **MyClass(0)** then calls **MyClass(0,0)** as just described.

One reason why invoking overloaded constructors through **this( )** can be useful is that it can prevent the unnecessary duplication of code. In many cases, reducing duplicate code decreases the time it takes to load your class because often the object code is smaller. This is especially important for programs delivered via the Internet in which load times are an issue. Using **this( )** can also help structure your code when constructors contain a large amount of duplicate code.

However, you need to be careful. Constructors that call **this( )** will execute a bit slower than those that contain all of their initialization code inline. This is because the call and return mechanism used when the second constructor is invoked adds overhead. If your class will be used to create only a handful of objects, or if the constructors in the class that call **this( )** will be seldom used, then this decrease in run-time performance is probably

insignificant. However, if your class will be used to create a large number of objects (on the order of thousands) during program execution, then the negative impact of the increased overhead could be meaningful. Because object creation affects all users of your class, there will be cases in which you must carefully weigh the benefits of faster load time against the increased time it takes to create an object.

Here is another consideration: for very short constructors, such as those used by **MyClass**, there is often little difference in the size of the object code whether **this( )** is used or not. (Actually, there are cases in which no reduction in the size of the object code is achieved.) This is because the bytecode that sets up and returns from the call to **this( )** adds instructions to the object file. Therefore, in these types of situations, even though duplicate code is eliminated, using **this( )** will not obtain significant savings in terms of load time. However, the added cost in terms of overhead to each object's construction will still be incurred. Therefore, **this( )** is most applicable to constructors that contain large amounts of initialization code, not those that simply set the value of a handful of fields.

There are two restrictions you need to keep in mind when using **this( )**. First, you cannot use any instance variable of the constructor's class in a call to **this( )**. Second, you cannot use **super( )** and **this( )** in the same constructor because each must be the first statement in the constructor.

Compact API Profiles

JDK 8 adds a feature that organizes subsets of the API library into what are called *compact profiles*. These are called **compact1**, **compact2**, and **compact3**. Each profile contains a subset of the library. Furthermore, **compact2** includes all of **compact1**, and **compact3** includes all of **compact2**. Thus, each profile builds on the previous one. The advantage of the compact profiles is that an application that does not require the full library need not download it. Using a compact profile reduces the size of the library, thus enabling some types of Java applications to run on devices that could not otherwise support the entire Java API. The use of a compact profile can also reduce the time it takes to load a program. The Java API documentation indicates to which (if any) profile each API element belongs.

When compiling a program, you can determine if a program uses only APIs defined by a compact profile by using the **-profile** option. Here is its general form:

javac -profile *profileName programName*

Here, *profileName* specifies the profile, which must be **compact1**, **compact2**, or **compact3**. For example:

```
javac -profile compact2 Test.java
```

Here, the **compact2** profile is specified. If **Test.java** contains an API that is not part of **compact2**, then a compilation error will result.

CHAPTER

14 Generics

Since the original 1.0 release in 1995, many new features have been added to Java. One that has had a profound impact is *generics*. Introduced by JDK 5, generics changed Java in two important ways. First, it added a new syntactical element to the language. Second, it caused changes to many of the classes and methods in the core API. Today, generics are an integral part of Java programming, and a solid understanding of this important feature is required. It is examined here in detail.

Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**, or **Thread**. With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort. The expressive power generics added to the language fundamentally changed the way that Java code is written.

Perhaps the one feature of Java that has been most significantly affected by generics is the *Collections Framework*. The Collections Framework is part of the Java API and is described in detail in Chapter 18, but a brief mention is useful now. A *collection* is a group of objects. The Collections Framework defines several classes, such as lists and maps, that manage collections. The collection classes have always been able to work with any type of object. The benefit that generics added is that the collection classes can now be used with complete type safety. Thus, in addition to being a powerful language element on its own, generics also enabled an existing feature to be substantially improved. This is another reason why generics were such an important addition to Java.

This chapter describes the syntax, theory, and use of generics. It also shows how generics provide type safety for some previously difficult cases. Once you have completed this chapter, you will want to examine Chapter 18, which covers the Collections Framework. There you will find many examples of generics at work.

What Are Generics?

At its core, the term *generics* means *parameterized types*. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic*, as in *generic class* or *generic method*.

It is important to understand that Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**. Because **Object** is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects. The problem was that they could not do so with type safety.

Generics added the type safety that was lacking. They also streamlined the process, because it is no longer necessary to explicitly employ casts to translate between **Object** and the type of data that is actually being operated upon. With generics, all casts are automatic and implicit. Thus, generics expanded your ability to reuse code and let you do so safely and easily.

NOTE A Warning to C++ Programmers: Although generics are similar to templates in C++, they are not the same. There are some fundamental differences between the two approaches to generic types. If you have a background in C++, it is important not to jump to conclusions about how generics work in Java.

A Simple Generics Example

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }

  // Show type of T.
```

```
  void showType() {
    System.out.println("Type of T is " +
                       ob.getClass().getName());
  }
}

// Demonstrate the generic class.
class GenDemo {
  public static void main(String args[]) {
    // Create a Gen reference for Integers.
    Gen<Integer> iOb;

    // Create a Gen<Integer> object and assign its
    // reference to iOb. Notice the use of autoboxing
    // to encapsulate the value 88 within an Integer object.
    iOb = new Gen<Integer>(88);

    // Show the type of data used by iOb.
    iOb.showType();

    // Get the value in iOb. Notice that
    // no cast is needed.
    int v = iOb.getob();
    System.out.println("value: " + v);

    System.out.println();

    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String> ("Generics Test");

    // Show the type of data used by strOb.
    strOb.showType();

    // Get the value of strOb. Again, notice
    // that no cast is needed.
    String str = strOb.getob();
    System.out.println("value: " + str);
  }
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

Let's examine this program carefully.

First, notice how **Gen** is declared by the following line:

```
class Gen<T> {
```

Here, **T** is the name of a *type parameter*. This name is used as a placeholder for the actual type that will be passed to **Gen** when an object is created. Thus, **T** is used within **Gen** whenever the type parameter is needed. Notice that **T** is contained within **< >**. This syntax can be generalized. Whenever a type parameter is being declared, it is specified within angle brackets. Because **Gen** uses a type parameter, **Gen** is a generic class, which is also called a *parameterized type*.

Next, **T** is used to declare an object called **ob**, as shown here:

```
T ob; // declare an object of type T
```

As explained, **T** is a placeholder for the actual type that will be specified when a **Gen** object is created. Thus, **ob** will be an object of the type passed to **T**. For example, if type **String** is passed to **T**, then in that instance, **ob** will be of type **String**.

Now consider **Gen**'s constructor:

```
Gen(T o) {
  ob = o;
}
```

Notice that its parameter, **o**, is of type **T**. This means that the actual type of **o** is determined by the type passed to **T** when a **Gen** object is created. Also, because both the parameter **o** and the member variable **ob** are of type **T**, they will both be of the same actual type when a **Gen** object is created.

The type parameter **T** can also be used to specify the return type of a method, as is the case with the **getob( )** method, shown here:

```
T getob() {
  return ob;
}
```

Because **ob** is also of type **T**, its type is compatible with the return type specified by **getob( )**.

The **showType( )** method displays the type of **T** by calling **getName( )** on the **Class** object returned by the call to **getClass( )** on **ob**. The **getClass( )** method is defined by **Object** and is thus a member of all class types. It returns a **Class** object that corresponds to the type of the class of the object on which it is called. **Class** defines the **getName( )** method, which returns a string representation of the class name.

The **GenDemo** class demonstrates the generic **Gen** class. It first creates a version of **Gen** for integers, as shown here:

```
Gen<Integer> iOb;
```

Look closely at this declaration. First, notice that the type **Integer** is specified within the angle brackets after **Gen**. In this case, **Integer** is a *type argument* that is passed to **Gen**'s type parameter, **T**. This effectively creates a version of **Gen** in which all references to **T** are translated into references to **Integer**. Thus, for this declaration, **ob** is of type **Integer**, and the return type of **getob( )** is of type **Integer**.

Before moving on, it's necessary to state that the Java compiler does not actually create different versions of **Gen**, or of any other generic class. Although it's helpful to think in these terms, it is not what actually happens. Instead, the compiler removes all generic type information, substituting the necessary casts, to make your code *behave as if* a specific version of **Gen** were created. Thus, there is really only one version of **Gen** that actually exists in your program. The process of removing generic type information is called *erasure*, and we will return to this topic later in this chapter.

The next line assigns to **iOb** a reference to an instance of an **Integer** version of the **Gen** class:

```
iOb = new Gen<Integer>(88);
```

Notice that when the **Gen** constructor is called, the type argument **Integer** is also specified. This is because the type of the object (in this case **iOb**) to which the reference is being assigned is of type **Gen<Integer>**. Thus, the reference returned by **new** must also be of type **Gen<Integer>**. If it isn't, a compile-time error will result. For example, the following assignment will cause a compile-time error:

```
iOb = new Gen<Double>(88.0); // Error!
```

Because **iOb** is of type **Gen<Integer>**, it can't be used to refer to an object of **Gen<Double>**. This type checking is one of the main benefits of generics because it ensures type safety.

NOTE As you will see later in this chapter, beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic class. In the interest of clarity, we will use the full syntax at this time.

As the comments in the program state, the assignment

```
iOb = new Gen<Integer>(88);
```

makes use of autoboxing to encapsulate the value 88, which is an **int**, into an **Integer**. This works because **Gen<Integer>** creates a constructor that takes an **Integer** argument. Because an **Integer** is expected, Java will automatically box 88 inside one. Of course, the assignment could also have been written explicitly, like this:

```
iOb = new Gen<Integer>(new Integer(88));
```

However, there would be no benefit to using this version.

The program then displays the type of **ob** within **iOb**, which is **Integer**. Next, the program obtains the value of **ob** by use of the following line:

```
int v = iOb.getob();
```

Because the return type of **getob( )** is **T**, which was replaced by **Integer** when **iOb** was declared, the return type of **getob( )** is also **Integer**, which unboxes into **int** when assigned to **v** (which is an **int**). Thus, there is no need to cast the return type of **getob( )** to **Integer**.

Of course, it's not necessary to use the auto-unboxing feature. The preceding line could have been written like this, too:

```
int v = iOb.getob().intValue();
```

However, the auto-unboxing feature makes the code more compact.

Next, **GenDemo** declares an object of type **Gen<String>**:

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Because the type argument is **String**, **String** is substituted for **T** inside **Gen**. This creates (conceptually) a **String** version of **Gen**, as the remaining lines in the program demonstrate.

Generics Work Only with Reference Types

When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter. Therefore, the following declaration is illegal:

```
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

Of course, not being able to specify a primitive type is not a serious restriction because you can use the type wrappers (as the preceding example did) to encapsulate a primitive type. Further, Java's autoboxing and auto-unboxing mechanism makes the use of the type wrapper transparent.

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

Even though both **iOb** and **strOb** are of type **Gen<T>**, they are references to different types because their type parameters differ. This is part of the way that generics add type safety and prevent errors.

How Generics Improve Type Safety

At this point, you might be asking yourself the following question: Given that the same functionality found in the generic **Gen** class can be achieved without generics, by simply specifying **Object** as the data type and employing the proper casts, what is the benefit of making **Gen** generic? The answer is that generics automatically ensure the type safety of all operations involving **Gen**. In the process, they eliminate the need for you to enter casts and to type-check code by hand.

To understand the benefits of generics, first consider the following program that creates a non-generic equivalent of **Gen**:

```
// NonGen is functionally equivalent to Gen
// but does not use generics.
class NonGen {
  Object ob; // ob is now of type Object

  // Pass the constructor a reference to
  // an object of type Object
  NonGen(Object o) {
    ob = o;
  }

  // Return type Object.
  Object getob() {
    return ob;
  }

  // Show type of ob.
  void showType() {
    System.out.println("Type of ob is " +
                       ob.getClass().getName());
  }
}

// Demonstrate the non-generic class.
class NonGenDemo {
  public static void main(String args[]) {
    NonGen iOb;

    // Create NonGen Object and store
    // an Integer in it. Autoboxing still occurs.
    iOb = new NonGen(88);

    // Show the type of data used by iOb.
    iOb.showType();

    // Get the value of iOb.
    // This time, a cast is necessary.
    int v = (Integer) iOb.getob();
    System.out.println("value: " + v);

    System.out.println();

    // Create another NonGen object and
    // store a String in it.
    NonGen strOb = new NonGen("Non-Generics Test");

    // Show the type of data used by strOb.
    strOb.showType();

    // Get the value of strOb.
    // Again, notice that a cast is necessary.
```

```
    String str = (String) strOb.getob();
    System.out.println("value: " + str);

    // This compiles, but is conceptually wrong!
    iOb = strOb;
    v = (Integer) iOb.getob(); // run-time error!
  }
}
```

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **Object**. This makes **NonGen** able to store any type of object, as can the generic version. However, it also prevents the Java compiler from having any real knowledge about the type of data actually stored in **NonGen**, which is bad for two reasons. First, explicit casts must be employed to retrieve the stored data. Second, many kinds of type mismatch errors cannot be found until run time. Let's look closely at each problem.

Notice this line:

```
int v = (Integer) iOb.getob();
```

Because the return type of **getob( )** is **Object**, the cast to **Integer** is necessary to enable that value to be auto-unboxed and stored in **v**. If you remove the cast, the program will not compile. With the generic version, this cast was implicit. In the non-generic version, the cast must be explicit. This is not only an inconvenience, but also a potential source of error.

Now, consider the following sequence from near the end of the program:

```
// This compiles, but is conceptually wrong!
iOb = strOb;
v = (Integer) iOb.getob(); // run-time error!
```

Here, **strOb** is assigned to **iOb**. However, **strOb** refers to an object that contains a string, not an integer. This assignment is syntactically valid because all **NonGen** references are the same, and any **NonGen** reference can refer to any other **NonGen** object. However, the statement is semantically wrong, as the next line shows. Here, the return type of **getob( )** is cast to **Integer**, and then an attempt is made to assign this value to **v**. The trouble is that **iOb** now refers to an object that stores a **String**, not an **Integer**. Unfortunately, without the use of generics, the Java compiler has no way to know this. Instead, a run-time exception occurs when the cast to **Integer** is attempted. As you know, it is extremely bad to have run-time exceptions occur in your code!

The preceding sequence can't occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a run-time exception. The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **Object** references to create "generic" code has always been possible, that code was not type safe, and its misuse could result in run-time exceptions. Generics prevent this from occurring. In essence, through generics, run-time errors are converted into compile-time errors. This is a major advantage.

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Pass the constructor a reference to
  // an object of type T and an object of type V.
  TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }

  // Show types of T and V.
  void showTypes() {
    System.out.println("Type of T is " +
                       ob1.getClass().getName());

    System.out.println("Type of V is " +
                       ob2.getClass().getName());
  }

  T getob1() {
    return ob1;
  }

  V getob2() {
    return ob2;
  }
}

// Demonstrate TwoGen.
class SimpGen {
  public static void main(String args[]) {

    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Generics");

    // Show the types.
    tgObj.showTypes();

    // Obtain and show values.
    int v = tgObj.getob1();
    System.out.println("value: " + v);

    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

The output from this program is shown here:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

Notice how **TwoGen** is declared:

```
class TwoGen<T, V> {
```

It specifies two type parameters: **T** and **V**, separated by a comma. Because it has two type parameters, two type arguments must be passed to **TwoGen** when an object is created, as shown next:

```
TwoGen<Integer, String> tgObj =
  new TwoGen<Integer, String>(88, "Generics");
```

In this case, **Integer** is substituted for **T**, and **String** is substituted for **V**.

Although the two type arguments differ in this example, it is possible for both types to be the same. For example, the following line of code is valid:

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

In this case, both **T** and **V** would be of type **String**. Of course, if the type arguments were always the same, then two type parameters would be unnecessary.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { // …

Here is the full syntax for declaring a reference to a generic class and instance creation:

class-name<*type-arg-list*> *var-name* =
new *class-name*<*type-arg-list*>(*cons-arg-list*);

Bounded Types

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, **float**s, and **double**s. Thus, you want to specify the type of the numbers generically, using a type parameter. To create such a class, you might try something like this:

```
// Stats attempts (unsuccessfully) to
// create a generic class that can compute
```

```
// the average of an array of numbers of
// any given type.
//
// The class contains an error!
class Stats<T> {
  T[] nums; // nums is an array of type T

  // Pass the constructor a reference to
  // an array of type T.
  Stats(T[] o) {
    nums = o;
  }

  // Return type double in all cases.
  double average() {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue(); // Error!!!

    return sum / nums.length;
  }
}
```

In **Stats**, the **average( )** method attempts to obtain the **double** version of each number in the **nums** array by calling **doubleValue( )**. Because all numeric classes, such as **Integer** and **Double**, are subclasses of **Number**, and **Number** defines the **doubleValue( )** method, this method is available to all numeric wrapper classes. The trouble is that the compiler has no way to know that you are intending to create **Stats** objects using only numeric types. Thus, when you try to compile **Stats**, an error is reported that indicates that the **doubleValue( )** method is unknown. To solve this problem, you need some way to tell the compiler that you intend to pass only numeric types to **T**. Furthermore, you need some way to *ensure* that *only* numeric types are actually passed.

To handle such situations, Java provides *bounded types*. When specifying a type parameter, you can create an upper bound that declares the superclass from which all type arguments must be derived. This is accomplished through the use of an **extends** clause when specifying the type parameter, as shown here:

<T extends *superclass>*

This specifies that *T* can only be replaced by *superclass*, or subclasses of *superclass*. Thus, *superclass* defines an inclusive, upper limit.

You can use an upper bound to fix the **Stats** class shown earlier by specifying **Number** as an upper bound, as shown here:

```
// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
  T[] nums; // array of Number or subclass
```

```
  // Pass the constructor a reference to
  // an array of type Number or subclass.
  Stats(T[] o) {
    nums = o;
  }

  // Return type double in all cases.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;
  }
}

// Demonstrate Stats.
class BoundsDemo {
  public static void main(String args[]) {

    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);

    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average();
    System.out.println("dob average is " + w);

    // This won't compile because String is not a
    // subclass of Number.
//    String strs[] = { "1", "2", "3", "4", "5" };
//    Stats<String> strob = new Stats<String>(strs);

//    double x = strob.average();
//    System.out.println("strob average is " + v);

  }
}
```

The output is shown here:

```
Average is 3.0
Average is 3.3
```

Notice how **Stats** is now declared by this line:

```
class Stats<T extends Number> {
```

Because the type **T** is now bounded by **Number**, the Java compiler knows that all objects of type **T** can call **doubleValue( )** because it is a method declared by **Number**. This is, by itself, a major advantage. However, as an added bonus, the bounding of **T** also prevents nonnumeric **Stats** objects from being created. For example, if you try removing the comments from the lines at the end of the program, and then try recompiling, you will receive compile-time errors because **String** is not a subclass of **Number**.

In addition to using a class type as a bound, you can also use an interface type. In fact, you can specify multiple interfaces as bounds. Furthermore, a bound can include both a class type and one or more interfaces. In this case, the class type must be specified first. When a bound includes an interface type, only type arguments that implement that interface are legal. When specifying a bound that has a class and an interface, or multiple interfaces, use the **&** operator to connect them. For example,

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Here, **T** is bounded by a class called **MyClass** and an interface called **MyInterface**. Thus, any type argument passed to **T** must be a subclass of **MyClass** and implement **MyInterface**.

Using Wildcard Arguments

As useful as type safety is, sometimes it can get in the way of perfectly acceptable constructs. For example, given the **Stats** class shown at the end of the preceding section, assume that you want to add a method called **sameAvg( )** that determines if two **Stats** objects contain arrays that yield the same average, no matter what type of numeric data each object holds. For example, if one object contains the **double** values 1.0, 2.0, and 3.0, and the other object contains the integer values 2, 1, and 3, then the averages will be the same. One way to implement **sameAvg( )** is to pass it a **Stats** argument, and then compare the average of that argument against the invoking object, returning true only if the averages are the same. For example, you want to be able to call **sameAvg( )**, as shown here:

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
  System.out.println("Averages are the same.");
else
  System.out.println("Averages differ.");
```

At first, creating **sameAvg( )** seems like an easy problem. Because **Stats** is generic and its **average( )** method can work on any type of **Stats** object, it seems that creating **sameAvg( )** would be straightforward. Unfortunately, trouble starts as soon as you try to declare a parameter of type **Stats**. Because **Stats** is a parameterized type, what do you specify for **Stats**' type parameter when you declare a parameter of that type?

At first, you might think of a solution like this, in which **T** is used as the type parameter:

```
// This won't work!
// Determine if two averages are the same.
boolean sameAvg(Stats<T> ob) {
  if(average() == ob.average())
    return true;

  return false;
}
```

The trouble with this attempt is that it will work only with other **Stats** objects whose type is the same as the invoking object. For example, if the invoking object is of type **Stats<Integer>**, then the parameter **ob** must also be of type **Stats<Integer>**. It can't be used to compare the average of an object of type **Stats<Double>** with the average of an object of type **Stats<Short>**, for example. Therefore, this approach won't work except in a very narrow context and does not yield a general (that is, generic) solution.

To create a generic **sameAvg( )** method, you must use another feature of Java generics: the *wildcard* argument. The wildcard argument is specified by the **?**, and it represents an unknown type. Using a wildcard, here is one way to write the **sameAvg( )** method:

```
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob) {
  if(average() == ob.average())
    return true;

  return false;
}
```

Here, **Stats<?>** matches any **Stats** object, allowing any two **Stats** objects to have their averages compared. The following program demonstrates this:

```
// Use a wildcard.
class Stats<T extends Number> {
  T[] nums; // array of Number or subclass

  // Pass the constructor a reference to
  // an array of type Number or subclass.
  Stats(T[] o) {
    nums = o;
  }

  // Return type double in all cases.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();
```

```
    return sum / nums.length;
  }

  // Determine if two averages are the same.
  // Notice the use of the wildcard.
  boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
      return true;

    return false;
  }
}

// Demonstrate wildcard.
class WildcardDemo {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);

    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.average();
    System.out.println("dob average is " + w);

    Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
    Stats<Float> fob = new Stats<Float>(fnums);
    double x = fob.average();
    System.out.println("fob average is " + x);

    // See which arrays have same average.
    System.out.print("Averages of iob and dob ");
    if(iob.sameAvg(dob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");

    System.out.print("Averages of iob and fob ");
    if(iob.sameAvg(fob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");
  }
}
```

The output is shown here:

```
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

One last point: It is important to understand that the wildcard does not affect what type of **Stats** objects can be created. This is governed by the **extends** clause in the **Stats** declaration. The wildcard simply matches any *valid* **Stats** object.

Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy. To understand why, let's work through an example. Consider the following hierarchy of classes that encapsulate coordinates:

```
// Two-dimensional coordinates.
class TwoD {
  int x, y;

  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
  int z;

  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
  int t;

  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}
```

At the top of the hierarchy is **TwoD**, which encapsulates a two-dimensional, XY coordinate. **TwoD** is inherited by **ThreeD**, which adds a third dimension, creating an XYZ coordinate. **ThreeD** is inherited by **FourD**, which adds a fourth dimension (time), yielding a four-dimensional coordinate.

Shown next is a generic class called **Coords**, which stores an array of coordinates:

```
// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}
```

Notice that **Coords** specifies a type parameter bounded by **TwoD**. This means that any array stored in a **Coords** object will contain objects of type **TwoD** or one of its subclasses.

Now, assume that you want to write a method that displays the X and Y coordinates for each element in the **coords** array of a **Coords** object. Because all types of **Coords** objects have at least two coordinates (X and Y), this is easy to do using a wildcard, as shown here:

```
static void showXY(Coords<?> c) {
  System.out.println("X Y Coordinates:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y);
   System.out.println();
}
```

Because **Coords** is a bounded generic type that specifies **TwoD** as an upper bound, all objects that can be used to create a **Coords** object will be arrays of type **TwoD**, or of classes derived from **TwoD**. Thus, **showXY( )** can display the contents of any **Coords** object.

However, what if you want to create a method that displays the X, Y, and Z coordinates of a **ThreeD** or **FourD** object? The trouble is that not all **Coords** objects will have three coordinates, because a **Coords<TwoD>** object will only have X and Y. Therefore, how do you write a method that displays the X, Y, and Z coordinates for **Coords<ThreeD>** and **Coords<FourD>** objects, while preventing that method from being used with **Coords<TwoD>** objects? The answer is the *bounded wildcard argument*.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Using a bounded wildcard, it is easy to create a method that displays the X, Y, and Z coordinates of a **Coords** object, if that object actually has those three coordinates. For example, the following **showXYZ( )** method shows the X, Y, and Z coordinates of the elements stored in a **Coords** object, if those elements are actually of type **ThreeD** (or are derived from **ThreeD**):

```
static void showXYZ(Coords<? extends ThreeD> c) {
  System.out.println("X Y Z Coordinates:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " +
                       c.coords[i].y + " " +
                       c.coords[i].z);
  System.out.println();
}
```

Notice that an **extends** clause has been added to the wildcard in the declaration of parameter **c**. It states that the **?** can match any type as long as it is **ThreeD**, or a class derived from **ThreeD**. Thus, the **extends** clause establishes an upper bound that the **?** can match. Because of this bound, **showXYZ( )** can be called with references to objects of type **Coords<ThreeD>** or **Coords<FourD>**, but not with a reference of type **Coords<TwoD>**. Attempting to call **showXZY( )** with a **Coords<TwoD>** reference results in a compile-time error, thus ensuring type safety.

Here is an entire program that demonstrates the actions of a bounded wildcard argument:

```
// Bounded Wildcard arguments.

// Two-dimensional coordinates.
class TwoD {
  int x, y;

  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}

// Three-dimensional coordinates.
class ThreeD extends TwoD {
  int z;

  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}

// Four-dimensional coordinates.
class FourD extends ThreeD {
  int t;

  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}

// This class holds an array of coordinate objects.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}

// Demonstrate a bounded wildcard.
class BoundedWildcard {
  static void showXY(Coords<?> c) {
    System.out.println("X Y Coordinates:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
                         c.coords[i].y);
    System.out.println();
  }

  static void showXYZ(Coords<? extends ThreeD> c) {
    System.out.println("X Y Z Coordinates:");
    for(int i=0; i < c.coords.length; i++)
```

```
      System.out.println(c.coords[i].x + " " +
                         c.coords[i].y + " " +
                         c.coords[i].z);
    System.out.println();
  }

  static void showAll(Coords<? extends FourD> c) {
    System.out.println("X Y Z T Coordinates:");
    for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " +
                         c.coords[i].y + " " +
                         c.coords[i].z + " " +
                         c.coords[i].t);
    System.out.println();
  }

  public static void main(String args[]) {
    TwoD td[] = {
      new TwoD(0, 0),
      new TwoD(7, 9),
      new TwoD(18, 4),
      new TwoD(-1, -23)
    };

    Coords<TwoD> tdlocs = new Coords<TwoD>(td);

    System.out.println("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD
//  showXYZ(tdlocs); // Error, not a ThreeD
//  showAll(tdlocs); // Error, not a FourD

    // Now, create some FourD objects.
    FourD fd[] = {
      new FourD(1, 2, 3, 4),
      new FourD(6, 8, 14, 8),
      new FourD(22, 9, 4, 9),
      new FourD(3, -2, -23, 17)
    };

    Coords<FourD> fdlocs = new Coords<FourD>(fd);

    System.out.println("Contents of fdlocs.");
    // These are all OK.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
  }
}
```

The output from the program is shown here:

```
Contents of tdlocs.
X Y Coordinates:
0 0
```

```
7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Notice these commented-out lines:

```
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
```

Because **tdlocs** is a **Coords(TwoD)** object, it cannot be used to call **showXYZ( )** or **showAll( )** because bounded wildcard arguments in their declarations prevent it. To prove this to yourself, try removing the comment symbols, and then attempt to compile the program. You will receive compilation errors because of the type mismatches.

In general, to establish an upper bound for a wildcard, use the following type of wildcard expression:

<? extends *superclass*>

where *superclass* is the name of the class that serves as the upper bound. Remember, this is an inclusive clause because the class forming the upper bound (that is, specified by *superclass*) is also within bounds.

You can also specify a lower bound for a wildcard by adding a **super** clause to a wildcard declaration. Here is its general form:

<? super *subclass*>

In this case, only classes that are superclasses of *subclass* are acceptable arguments. This is an inclusive clause.

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter. However, it is possible to declare a generic method that uses one or more type parameters of

its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let's begin with an example. The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn( )**. The **isIn( )** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {

  // Determine if an object is in an array.
  static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
    for(int i=0; i < y.length; i++)
      if(x.equals(y[i])) return true;

    return false;
  }

  public static void main(String args[]) {

    // Use isIn() on Integers.
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if(isIn(2, nums))
      System.out.println("2 is in nums");

    if(!isIn(7, nums))
      System.out.println("7 is not in nums");

    System.out.println();

    // Use isIn() on Strings.
    String strs[] = { "one", "two", "three",
                      "four", "five" };

    if(isIn("two", strs))
      System.out.println("two is in strs");

    if(!isIn("seven", strs))
      System.out.println("seven is not in strs");

    // Oops! Won't compile! Types must be compatible.
//    if(isIn("two", nums))
//      System.out.println("two is in strs");
  }
}
```

The output from the program is shown here:

```
2 is in nums
7 is not in nums
```

```
two is in strs
seven is not in strs
```

Let's examine **isIn( )** closely. First, notice how it is declared by this line:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
```

The type parameters are declared *before* the return type of the method. Also note that **T** extends **Comparable<T>**. **Comparable** is an interface declared in **java.lang.** A class that implements **Comparable** defines objects that can be ordered. Thus, requiring an upper bound of **Comparable** ensures that **isIn**( ) can be used only with objects that are capable of being compared. **Comparable** is generic, and its type parameter specifies the type of objects that it compares. (Shortly, you will see how to create a generic interface.) Next, notice that the type **V** is upper-bounded by **T**. Thus, **V** must either be the same as type **T**, or a subclass of **T**. This relationship enforces that **isIn( )** can be called only with arguments that are compatible with each other. Also notice that **isIn( )** is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

Now, notice how **isIn( )** is called within **main( )** by use of the normal call syntax, without the need to specify type arguments. This is because the types of the arguments are automatically discerned, and the types of **T** and **V** are adjusted accordingly. For example, in the first call:

```
if(isIn(2, nums))
```

the type of the first argument is **Integer** (due to autoboxing), which causes **Integer** to be substituted for **T**. The base type of the second argument is also **Integer**, which makes **Integer** a substitute for **V**, too. In the second call, **String** types are used, and the types of **T** and **V** are replaced by **String**.

Although type inference will be sufficient for most generic method calls, you can explicitly specify the type argument if needed. For example, here is how the first call to **isIn( )** looks when the type arguments are specified:

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

Of course, in this case, there is nothing gained by specifying the type arguments. Furthermore, JDK 8 has improved type inference as it relates to methods. As a result, there are fewer cases in which explicit type arguments are needed.

Now, notice the commented-out code, shown here:

```
//    if(isIn("two", nums))
//      System.out.println("two is in strs");
```

If you remove the comments and then try to compile the program, you will receive an error. The reason is that the type parameter **V** is bounded by **T** in the **extends** clause in **V**'s declaration. This means that **V** must be either type **T**, or a subclass of **T**. In this case, the first argument is of type **String**, making **T** into **String**, but the second argument is of type

Integer, which is not a subclass of **String**. This causes a compile-time type-mismatch error. This ability to enforce type safety is one of the most important advantages of generic methods.

The syntax used to create **isIn( )** can be generalized. Here is the syntax for a generic method:

<type-param-list> ret-type meth-name (*param-list*) { // …

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Generic Constructors

It is possible for constructors to be generic, even if their class is not. For example, consider the following short program:

```
// Use a generic constructor.
class GenCons {
  private double val;

  <T extends Number> GenCons(T arg) {
    val = arg.doubleValue();
  }

  void showval() {
    System.out.println("val: " + val);
  }
}

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

    GenCons test = new GenCons(100);
    GenCons test2 = new GenCons(123.5F);

    test.showval();
    test2.showval();
  }
}
```

The output is shown here:

```
val: 100.0
val: 123.5
```

Because **GenCons( )** specifies a parameter of a generic type, which must be a subclass of **Number**, **GenCons( )** can be called with any numeric type, including **Integer**, **Float**, or **Double**. Therefore, even though **GenCons** is not a generic class, its constructor is generic.

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces. Generic interfaces are specified just like generic classes. Here is an example. It creates an interface called **MinMax** that declares the methods **min( )** and **max( )**, which are expected to return the minimum and maximum value of some set of objects.

```
// A generic interface example.

// A Min/Max interface.
interface MinMax<T extends Comparable<T>> {
  T min();
  T max();
}

// Now, implement MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
  T[] vals;

  MyClass(T[] o) { vals = o; }

  // Return the minimum value in vals.
  public T min() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) < 0) v = vals[i];

    return v;
  }

  // Return the maximum value in vals.
  public T max() {
    T v = vals[0];

    for(int i=1; i < vals.length; i++)
      if(vals[i].compareTo(v) > 0) v = vals[i];

    return v;
  }
}

class GenIFDemo {
  public static void main(String args[]) {
    Integer inums[] = {3, 6, 2, 8, 6 };
    Character chs[] = {'b', 'r', 'p', 'w' };

    MyClass<Integer> iob = new MyClass<Integer>(inums);
    MyClass<Character> cob = new MyClass<Character>(chs);

    System.out.println("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());
```

```
    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
  }
}
```

The output is shown here:

```
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

Although most aspects of this program should be easy to understand, a couple of key points need to be made. First, notice that **MinMax** is declared like this:

```
interface MinMax<T extends Comparable<T>> {
```

In general, a generic interface is declared in the same way as is a generic class. In this case, the type parameter is **T**, and its upper bound is **Comparable**. As explained earlier, **Comparable** is an interface defined by **java.lang** that specifies how objects are compared. Its type parameter specifies the type of the objects being compared.

Next, **MinMax** is implemented by **MyClass**. Notice the declaration of **MyClass**, shown here:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Pay special attention to the way that the type parameter **T** is declared by **MyClass** and then passed to **MinMax**. Because **MinMax** requires a type that implements **Comparable**, the implementing class (**MyClass** in this case) must specify the same bound. Furthermore, once this bound has been established, there is no need to specify it again in the **implements** clause. In fact, it would be wrong to do so. For example, this line is incorrect and won't compile:

```
// This is wrong!
class MyClass<T extends Comparable<T>>
        implements MinMax<T extends Comparable<T>> {
```

Once the type parameter has been established, it is simply passed to the interface without further modification.

In general, if a class implements a generic interface, then that class must also be generic, at least to the extent that it takes a type parameter that is passed to the interface. For example, the following attempt to declare **MyClass** is in error:

```
class MyClass implements MinMax<T> { // Wrong!
```

Because **MyClass** does not declare a type parameter, there is no way to pass one to **MinMax**. In this case, the identifier **T** is simply unknown, and the compiler reports an error. Of course, if a class implements a *specific type* of generic interface, such as shown here:

```
class MyClass implements MinMax<Integer> { // OK
```

then the implementing class does not need to be generic.

The generic interface offers two benefits. First, it can be implemented for different types of data. Second, it allows you to put constraints (that is, bounds) on the types of data for which the interface can be implemented. In the **MinMax** example, only types that implement the **Comparable** interface can be passed to **T**.

Here is the generalized syntax for a generic interface:

interface *interface-name*<*type-param-list*> { // …

Here, *type-param-list* is a comma-separated list of type parameters. When a generic interface is implemented, you must specify the type arguments, as shown here:

class *class-name*<*type-param-list*>
    implements *interface-name*<*type-arg-list*> {

Raw Types and Legacy Code

Because support for generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code. At the time of this writing, there is still pre-generics legacy code that must remain both functional and compatible with generics. Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code.

To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a *raw type* for the class. This raw type is compatible with legacy code, which has no knowledge of generics. The main drawback to using the raw type is that the type safety of generics is lost.

Here is an example that shows a raw type in action:

```
// Demonstrate a raw type.
class Gen<T> {

  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Demonstrate raw type.
class RawDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);
```

```
    // Create a Gen object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");

    // Create a raw-type Gen object and give it
    // a Double value.
    Gen raw = new Gen(new Double(98.6));

    // Cast here is necessary because type is unknown.
    double d = (Double) raw.getob();
    System.out.println("value: " + d);

    // The use of a raw type can lead to run-time
    // exceptions. Here are some examples.

    // The following cast causes a run-time error!
//    int i = (Integer) raw.getob(); // run-time error

    // This assignment overrides type safety.
    strOb = raw; // OK, but potentially wrong
//    String str = strOb.getob(); // run-time error

    // This assignment also overrides type safety.
    raw = iOb; // OK, but potentially wrong
//    d = (Double) raw.getob(); // run-time error
  }
}
```

This program contains several interesting things. First, a raw type of the generic **Gen** class is created by the following declaration:

```
Gen raw = new Gen(new Double(98.6));
```

Notice that no type arguments are specified. In essence, this creates a **Gen** object whose type **T** is replaced by **Object**.

A raw type is not type safe. Thus, a variable of a raw type can be assigned a reference to any type of **Gen** object. The reverse is also allowed; a variable of a specific **Gen** type can be assigned a reference to a raw **Gen** object. However, both operations are potentially unsafe because the type checking mechanism of generics is circumvented.

This lack of type safety is illustrated by the commented-out lines at the end of the program. Let's examine each case. First, consider the following situation:

```
//    int i = (Integer) raw.getob(); // run-time error
```

In this statement, the value of **ob** inside **raw** is obtained, and this value is cast to **Integer**. The trouble is that **raw** contains a **Double** value, not an integer value. However, this cannot be detected at compile time because the type of **raw** is unknown. Thus, this statement fails at run time.

The next sequence assigns to a **strOb** (a reference of type **Gen<String>**) a reference to a raw **Gen** object:

```
    strOb = raw; // OK, but potentially wrong
//    String str = strOb.getob(); // run-time error
```

The assignment, itself, is syntactically correct, but questionable. Because **strOb** is of type **Gen<String>**, it is assumed to contain a **String**. However, after the assignment, the object referred to by **strOb** contains a **Double**. Thus, at run time, when an attempt is made to assign the contents of **strOb** to **str**, a run-time error results because **strOb** now contains a **Double**. Thus, the assignment of a raw reference to a generic reference bypasses the type-safety mechanism.

The following sequence inverts the preceding case:

```
    raw = iOb; // OK, but potentially wrong
//    d = (Double) raw.getob(); // run-time error
```

Here, a generic reference is assigned to a raw reference variable. Although this is syntactically correct, it can lead to problems, as illustrated by the second line. In this case, **raw** now refers to an object that contains an **Integer** object, but the cast assumes that it contains a **Double**. This error cannot be prevented at compile time. Rather, it causes a run-time error.

Because of the potential for danger inherent in raw types, **javac** displays *unchecked warnings* when a raw type is used in a way that might jeopardize type safety. In the preceding program, these lines generate unchecked warnings:

```
Gen raw = new Gen(new Double(98.6));

strOb = raw; // OK, but potentially wrong
```

In the first line, it is the call to the **Gen** constructor without a type argument that causes the warning. In the second line, it is the assignment of a raw reference to a generic variable that generates the warning.

At first, you might think that this line should also generate an unchecked warning, but it does not:

```
raw = iOb; // OK, but potentially wrong
```

No compiler warning is issued because the assignment does not cause any *further* loss of type safety than had already occurred when **raw** was created.

One final point: You should limit the use of raw types to those cases in which you must mix legacy code with newer, generic code. Raw types are simply a transitional feature and not something that should be used for new code.

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a non-generic class. Thus, a generic class can act as a superclass or be a subclass. The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses. This is similar to the way that constructor arguments must be passed up a hierarchy.

Using a Generic Superclass

Here is a simple example of a hierarchy that uses a generic superclass:

```
// A simple generic class hierarchy.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}
```

In this hierarchy, **Gen2** extends the generic class **Gen**. Notice how **Gen2** is declared by the following line:

```
class Gen2<T> extends Gen<T> {
```

The type parameter **T** is specified by **Gen2** and is also passed to **Gen** in the **extends** clause. This means that whatever type is passed to **Gen2** will also be passed to **Gen**. For example, this declaration,

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

passes **Integer** as the type parameter to **Gen**. Thus, the **ob** inside the **Gen** portion of **Gen2** will be of type **Integer**.

Notice also that **Gen2** does not use the type parameter **T** except to support the **Gen** superclass. Thus, even if a subclass of a generic superclass would otherwise not need to be generic, it still must specify the type parameter(s) required by its generic superclass.

Of course, a subclass is free to add its own type parameters, if needed. For example, here is a variation on the preceding hierarchy in which **Gen2** adds a type parameter of its own:

```
// A subclass can add its own type parameters.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
```