6 Assertion facility

This chapter covers

- Using assertions
- Controlling assertions from the command line
- Controlling assertions from code
- Knowing when to use assertions

The assertion facility provides a mechanism for adding optional "sanity checks" to your code. These checks are used during the development and testing phases, but are turned off when the software is deployed. This allows the programmer to insert debugging checks that might be too slow or memory-intensive to use in a real context, but that help during development. In a sense, assertions are a lot like error checks, except that they are turned off for deployment.

Assertions generally are implemented in such a way that they can be compiled out; in languages like C and C++, this means using the preprocessor. Since Java doesn't have macro facilities, features that otherwise might be created by the programmer must be built into the language themselves. As a result, assertions have not been used widely in Java. The 1.4 release of the JDK corrects this.

One of the most important features of this facility is that these checks can be turned on and off at runtime, which means that you don't have to decide during development whether or not these checks should remain in the code or be removed before deployment.

6.1 Assertion basics

An *assertion* is a conditional expression that should evaluate to `true` if and only if your code is working correctly. If the expression evaluates to `false`, an error is signaled.

Here is an example of an assertion (shown in bold):

```
public class aClass {
  public void aMethod( int argument ) {
    Foo foo = null;

    // ... somehow get a Foo object

    // Now check to make sure we've managed to get one:
    assert foo != null;
  }
}
```

This asserts that `foo` is not `null`. If `foo` is in fact `null`, an `AssertionError` is thrown. Any code that executes after this line can safely assume that `foo` is not `null`.

Assertions are very simple, but we'll be looking at their usage in detail because assertions are very important in the quest for robust code. In this chapter, we'll learn not only how to use assertions, but when and where to use them.

6.1.1 Why use assertions?

It is widely acknowledged in programming circles that software isn't stable enough. We all know we're not doing enough error-checking. It is also acknowledged that

error-handling comprises a substantial amount of programming effort and a substantial portion of the resulting code. Error-handling code is also relatively dull to write, especially compared with the main algorithm whose errors are being handled.

Furthermore, dealing with errors can sometimes force you to consider design questions that you may be trying to avoid. In such situations, programmers generally just ignore the possibility of error, mostly because they don't want to lose their train of thought.

As programmers, we need to do more error-checking, and assertions are an important step in this direction. Assertions can be used to catch conditions that we don't expect to happen. This may sound paradoxical, but given that we rarely check for enough errors, it makes a certain sense. For every error we think of while programming, there are a whole bunch more that never occur to us. It's not possible to eliminate all errors, but we can plan ahead for the unexpected.

6.1.2 Assertions vs. other error code

The programmer's decision to use an assertion instead of other error-handling code is often based on a general rule of programming psychology: the less likely a programmer thinks an error is, the less code she will write to deal with it. An assertion is easier to write than a `RuntimeException`; a `RuntimeException` is easier to write than a regular `Exception`. Since assertions are easy and quick to use, getting into the habit of using them means you will catch more errors before they cause you trouble.

The choice between these different methods of dealing with an exceptional case really depends on how "exceptional" the exception really is. Is it something that should *never* happen? Something that should only happen due to the error of another programmer? Of the end user? Of the system administrator who configured the server? Of the person who configured the client machine? Is it something that will never happen after the software is released? Is it something that can be tolerated in the field? What damage can result if this exceptional case happens even *once* in a real-life situation?

All of these questions are relevant. A good rule of thumb is that you should use an assertion for exceptional cases that you would like to forget about. An assertion is the quickest way to deal with, and forget, a condition or state that you don't expect to have to deal with. For example, an application might have a hidden configuration file that it never deletes. Since it's *possible, but unlikely*, that the user will ever delete this hidden configuration file, it might be a good idea, after trying to open the file, to assert that the open worked. It almost certainly will, but it's a good idea to check.

Just about any computer programmer—or any computer user—can tell you that software isn't stable enough. Some software bugs are routine, and most of them

come without a hint of explanation. You have likely encountered some, if not all, of these problems:

- Printer drivers not printing, and not giving a warning
- Files not showing up on an FTP site after upload
- Web servers returning empty pages
- Hardware devices not being recognized
- System configuration differing after each reboot
- Programs crashing because of corrupted input
- Programs simply not running, or failing unexpectedly

Buried under each of these bugs, many levels down the chain of inter-program communication, is an exceptional condition that some programmer, somewhere, forgot to handle. Handling these errors gracefully is the best possible approach, but it's not possible to handle everything in a complete fashion. Think of an assertion as the smallest (and easiest) way to handle an exceptional case.

6.1.3 Designing by contract

If you are familiar with the *design-by-contract* programming methodology, then you can think of assertions as a good way of ensuring *preconditions*, *postconditions*, and *invariants*. Preconditions are contractual guarantees that must be true at the start of a method, and postconditions are the same, except they are in effect at the end of a method. Assertions can be good for ensuring preconditions if, and only if, the method is not a public method. (Public methods should make an explicit check and throw an exception—see section 6.4.2 for more on this.) Assertions are always good for postconditions.

Invariants, broadly defined, are conditions that should always be true. They are often checked before and after a computation. Since they should always be true, assertions are an excellent way to implement them.

6.2 Working with assertions

An assertion is a convenient syntax for checking for an error. In a sense, it's really just a shorthand for a full error check. In this section, we'll examine the syntax used for assertions and look at the equivalent expression from pre-assertion Java. We'll also examine the command-line and programmatic interfaces that can be used to enable and disable assertions at runtime.

6.2.1 Assertion syntax

An assertion is a conditional error; operationally, assertions are very simple. There are two distinct flavors of assertion: simple and complex.

Flavor 1 (simple)

Using the simpler syntax, an assertion consists of the keyword `assert`, followed by an expression:

```
assert expression;
```

This should be read as, "if expression isn't true, that's very bad, so throw an error immediately."

Here is the assertion example from section 6.1. This uses the simple syntax:

```
public class aClass {
  public void aMethod( int argument ) {
    Foo foo = null;

    // ... somehow get a Foo object

    // Now check to make sure we've managed to get one:
    assert foo != null;
  }
}
```

Again, operationally, this is roughly equivalent to the following:

```
public class aClass {
  public void aMethod( int argument ) {
    Foo foo = null;

    // ... somehow get a Foo object

    // Now check to make sure we've managed to get one:
    if (!(foo != null)) {
      throw new AssertionError();
    }
  }
}
```

Flavor 2 (complex)

The more complex syntax goes as follows:

```
assert expression_1 : expression_2;
```

This should be read as, "if expression_1 isn't true, throw an error containing the value of expression_2."

NOTE The second expression must be a valid argument to the constructor of the `AssertionError` object.

Here's the example from the previous section, but this time using the complex syntax:

```
public class aClass {
  public void aMethod( int argument ) {
    Foo foo = null;

    // ... somehow get a Foo object

    // Now check to make sure we've managed to get one:
    assert foo != null : "Can't get a Foo, argument="+argument;
  }
}
```

This is roughly equivalent to the following:

```
public class aClass {
  public void aMethod( int argument ) {
    Foo foo = null;

    // ... somehow get a Foo object

    // Now check to make sure we've managed to get one:
    if (!(foo != null)) {
      throw new AssertionError(
        "Can't get a Foo, argument="+argument );
    }
  }
}
```

As mentioned in the previous note, the second expression in the complex version of `assert` must be a valid argument to the constructor of the `AssertionError` object. `AssertionError` has constructors that take any of the following types:

- object
- boolean
- char
- int
- long
- float
- double

This allows the second expression of an assertion to have any kind of data type as an argument, making assertions as easy to use as `System.out` and `System.err`. This is intended to encourage the use of assertions over print statements.

Choosing a flavor

The choice of whether to use the simple syntax or the complex syntax comes down to how much information you want to provide the person running the program. In some cases, it's enough to tell the person where the error occurred; in others, it's important to print out certain values so that the bug can be repaired. If you can't decide, a good rule of thumb is to use the simple syntax. If, at some later point, you want the assertion to provide more information, you can easily change it to use the complex syntax.

6.2.2 Compiling with assertions

Assertions require a change to Java's *syntax*, so there is a slight issue with backward-compatibility. Once `assert` is a keyword, it can no longer be a variable or method name, and code like this is not compatible with the new syntax:

```
public void method() {
  int assert = getAssert();
}
```

Because of the dangerous possibility of breaking seven years' worth of Java code, the JDK 1.4 from Sun Microsystems allows you to select whether you want the new syntax or not.

To use the old syntax, and thus allow the word "assert" to be used as a keyword, you must execute the compiler using the `–source 1.3` option. At the command line, you would type this:

```
javac -source 1.3
```

To use the new syntax, and thus enable assertions, you would use this command on the command line:

```
javac -source 1.4
```

If unspecified, `-source 1.3` is assumed, so that existing code will compile normally even if it uses `assert` as a regular identifier. It is expected that all code will eventually compile under the new syntax; the older syntax is provided for those cases where the keyword `assert` was used as a variable or class name.

If you use assertions in your code, it will be incompatible with versions of the JRE prior to 1.4 because assertions need methods and fields from the `Class` and

`ClassLoader` classes. This is true even if you don't use the programmatic enable and disable methods mentioned in section 6.2.4.

This shouldn't be cause for alarm—using any new feature of a new release of Java will make the resulting class files incompatible with earlier versions of the JRE. Note that this incompatibility is purely a *library* incompatibility—there is no compatibility problem at the JVM level.

6.2.3 Controlling assertions from the command line

One of the most useful features of assertions is that they can be turned off during normal usage, so that they don't incur any speed penalty. By the same token, they need to be turned on when a problem arises. Assertions are off by default.

Although the assertion specification does not require a particular technique for enabling or disabling assertions, it does strongly recommend that such a technique exist for any implementation of the Java language. The implementation described here is that of the release of JDK 1.4 from Sun Microsystems. It is likely that most other implementations will closely follow this model.

Assertions are enabled on the command line via the `-ea` switch, which is an abbreviation for the `-enableassertions` switch. The following two commands are equivalent:

```
java -ea myPackage.myProgram
java -enableassertions myPackage.myProgram
```

Assertions are similarly disabled with either the `-da` or `-disableassertions` commands:

```
java -da myPackage.myProgram
java -disableassertions myPackage.myProgram
```

Assertions can be enabled or disabled for specific packages or classes. To specify a class, use the class name. To specify a package, use the package name followed by "...":

```
java -ea:<class> myPackage.myProgram
java -da:<package>... myPackage.myProgram
```

Note that each enable or disable modifies the one before it, so that you can, for example, enable assertions in general, but disable them in a particular package.

```
java -ea -da:<package>... myPackage.myProgram
```

Finally, you can enable or disable assertions in the unnamed root package (the one in the current directory) using the following commands:

```
java -ea:... myPackage.myProgram
java -da:... myPackage.myProgram
```

Note that assertions within system classes that come installed with your JVM can be enabled and disabled separately using the `-esa` and `-dsa` switches, which are abbreviations for `-enablesystemassertions` and `-disablesystemassertions`, respectively. The various command-line switches for using assertions are listed in table 6.1.

Table 6.1 Command-line switches for enabling and disabling assertions. These options are taken from JDK 1.4 from Sun Microsystems; other implementations may have other techniques for turning assertions on and off.

Switch	Example	Meaning
`-ea`	`Java -ea`	Enable assertions by default
`-da`	`Java -da`	Disable assertions by default
`-ea:<classname>`	`Java -ea:AssertPackageTest`	Enable assertions in class `AssertPackageTest`
`-da:<classname>`	`Java -da:AssertPackageTest`	Disable assertions in class `AssertPackageTest`
`-ea:<packagename>...`	`Java –ea:pkg0...`	Enable assertions in package `pkg0`
`-da:<packagename>...`	`Java –da:pkg0...`	Disable assertions in package `pkg0`
`-esa`	`Java –esa`	Enable assertions in system classes
`-dsa`	`Java -dsa`	Disable assertions in system classes

Command-line examples

Let's take a look at some examples of these options in action. In these examples, we have an application called `AssertPackageTest` that creates an instance of each of three classes, each one in a different package. These instances will print a message if assertions are turned on for them:

```
import pkg0.Class0;
import pkg0.subpkg0.Class2;
import pkg1.Class1;

public class AssertPackageTest
{
  static public void main( String args[] ) {
    new Class0();
    new Class1();
    new Class2();
```

```
    }
}
```

The following examples of running `AssertPackageTest` first state what is being done, and then show the command that runs the program and the output it produces (if any).

Leave assertions off by default:

```
java AssertPackageTest
```

(No output)

Turn assertions on for all non-system classes:

```
java -ea AssertPackageTest
Assertions enabled in AssertPackageTest
Assertions enabled in pkg0.Class0
Assertions enabled in pkg1.Class1
Assertions enabled in pkg0.subpkg0.Class2
```

Turn assertions on for a single package:

```
java -ea:pkg0... AssertPackageTest
Assertions enabled in pkg0.Class0
Assertions enabled in pkg0.subpkg0.Class2
```

Forget the "..." after a package name:

```
java -ea:pkg0 AssertPackageTest
```

(No output)

Turn assertions on for a single class:

```
java -ea:pkg0.Class0 AssertPackageTest
Assertions enabled in pkg0.Class0
```

Turn assertions on for a different class:

```
java -ea:pkg0.subpkg0.Class2 AssertPackageTest
Assertions enabled in pkg0.subpkg0.Class2
```

Turn assertions on for a subpackage:

```
java -ea:pkg0.subpkg0... AssertPackageTest
Assertions enabled in pkg0.subpkg0.Class2
```

Turn assertions on in general, but off for a package:

```
java -ea -da:pkg1... AssertPackageTest
Assertions enabled in AssertPackageTest
Assertions enabled in pkg0.Class0
Assertions enabled in pkg0.subpkg0.Class2
```

Turn assertions on for a package, but off for a subpackage of that package:

```
java -ea:pkg0... -da:pkg0.subpkg0... AssertPackageTest
Assertions enabled in pkg0.Class0
```

Turn assertions on only in the unnamed default package:

```
java -ea:... -da:pkg0... -da:pgk1... AssertPackageTest
```

```
Assertions enabled in AssertPackageTest
```

Turn assertions on in general, but off in the unnamed default package:

```
java -ea -da:... AssertPackageTest
Assertions enabled in pkg0.Class0
Assertions enabled in pkg1.Class1
Assertions enabled in pkg0.subpkg0.Class2
```

6.2.4 Controlling assertions programmatically

It is also possible to enable or disable assertions from the program itself. In general, this is something you won't need to do unless you are writing a debugger or some other kind of program whose purpose is to manage a Java program running in the same JVM.

Each class contains an "assertion status" flag that tells the system whether assertions are enabled for that class. Each time an `assert` line is reached, the containing class is checked for the value of this flag, to see if the assertion should be processed or skipped.

This flag can be set via the class's `ClassLoader`, using the following approach:

```
public void setClassAssertionStatus(String className,
                                     boolean enabled);
```

The arguments are as follows:

- `className`—The name of the class whose assertion status is to be set
- `enabled`—Whether assertions should be on or off

This flag can also be turned on for an entire package using another method of `ClassLoader`:

```
public void setPackageAssertionStatus(String packageName,
                                       boolean enabled);
```

The arguments are as follows:

- `packageName`—The name of the package whose classes are to have their assertion status set
- `enabled`—Whether assertions should be on or off

Note that this method applies not just to the specified package, but to all subpackages within it.

A `ClassLoader` also has a default assertion status that is passed to any class loaded through it. The default can be set with the following method of `ClassLoader`:

```
public void setDefaultAssertionStatus(boolean enabled);
```

The argument is as follows:

- `enabled`—Whether assertions should be on or off by default

Finally, `ClassLoader` has a method that lets you clear all the assertion settings that have gone before. This not only clears the default assertion status (thus turning assertions off by default), it also removes any per-class and per-package settings that have been made against this `ClassLoader`:

```
public void clearAssertionStatus();
```

Another method, `Class.desiredAssertionStatus()`, will be discussed in section 6.2.8.

NOTE The assertion status flags set by these methods do not affect classes already loaded and initialized by the `ClassLoader`—they only affect classes that are loaded and initialized subsequently. Remember to set these flags before loading the classes that you want to be affected by them.

6.2.5 Removing assertions completely

Even if you run your code with assertions disabled, they are still in the class files. Although this depends completely on the particular implementation of the Java platform you are using, it is quite likely that the assertions will be taking up some space, as well as some time, in your running program.

If this is a problem, you can apply the standard technique for removing code without actually removing it:

```
static final boolean doAsserts = false;

public void method() {
  if (doAsserts) assert expression;
}
```

Because `doAsserts` is final, the Java compiler is required to remove this line of code from the execution, resulting in savings in both time and space.

WARNING Removing assertions is strongly discouraged unless there is good reason, such as the need to run with a very small memory footprint. Assertions are most useful if they can be turned on at any time, even long after the release of the software.

6.2.6 Determining if assertions are enabled

There are times when you might need to determine whether assertions are enabled. For example, your assertions might need to do extra calculations in order to properly check your code, and you might want to avoid doing those calculations if assertions are disabled.

The following fragment of code tests to see whether assertions are enabled or not:

```
public void method() {
  boolean assertionsAreEnabled = false;

  assert (assertionsAreEnabled = true);

  if (assertionsAreEnabled) {
    System.out.println( "Assertions are enabled!" );
  } else {
    System.out.println( "Assertions are disabled!" );
  }
}
```

Note the trickiness here—that's a *single* equals sign inside the assertion expression, so it's an assignment rather than a comparison. Instead of checking whether `assertionsAreEnabled` is `true`, it actually sets `assertionsAreEnabled` to be `true`.

It is, in general, a bad idea to put any kind of side effects inside an `assert` expression, because you don't know if the expression will even get executed—that depends on whether assertions are enabled. This case is an exception, though—not only is the side effect localized to this fragment, but it is in fact the *point* of the construction. We allow the assertion status to have a side effect because we want to know if assertions are enabled.

One thing you might want to do in very critical applications is to refuse to run without assertions. This is somewhat nonstandard, since assertions, by their very nature, are supposed to be enabled at the whim of the person running the program, rather than at the whim of the programmer. However, if it is important to ensure that they are on, the following can be done:

```
public void method() {
  boolean assertionsAreEnabled = false;

  assert (assertionsAreEnabled = true);

  if (!assertionsAreEnabled) {
    throw new RuntimeException( "Assertions must be enabled!" );
  }
}
```

Be careful not to fall into the trap of using an assertion to do this check. The following code won't work if assertions are turned off, and thus misses the whole point:

```
public void method() {
  boolean assertionsAreEnabled = false;

  assert (assertionsAreEnabled = true);

  assert assertionsAreEnabled;
}
```

6.2.7 Catching an assertion failure

Since assertions fail by throwing an error, it's possible to catch an assertion failure. Under normal circumstances, you should rethrow the error, because it is crucial that assertion failures come to the attention of the operator of the program as soon as possible.

However, there are times when you might want to catch an `AssertionError`, do something, and then rethrow the error. If your application has a network console, you might want to send the assertion failure across the network to the console before quitting.

If you do catch an assertion failure, make sure to rethrow it! It's okay to catch an exception, because exceptions are *designed* to be caught. But an assertion failure generally implies a really unexpected failure—something that deserves immediate attention.

In the example in listing 6.1, we trap the `AssertionError` in order to get stack trace information. We then rethrow the `AssertionError` from within the `catch` block.

Listing 6.1

```
public void method() {
    AssertionError ae = null;

    try {
      int a = anotherMethod();
      // ...
      assert i==10;     ● The assertion
      // ...
    } catch( AssertionError ae2 ) {     ● Trapping the assertion failure
      ae = ae2;
      StackTraceElement stes[] = ae.getStackTrace();
```

```
      if (stes.length>0) {
        StackTraceElement first = stes[0];
        System.out.println( "NOTE: Assertion failure in "+
          first.getFileName()+" at line "+first.getLineNumber() );
      } else {
        System.out.println( "NOTE: No info available." );
      }

      throw ae;        ● Rethrowing the assertion failure
    }
}
```

6.2.8 Assertions and class initialization

According to the assertion specifications, whether or not assertions are turned on for a class is determined during the initialization process. In most circumstances, a class cannot be used before it has been initialized. But there are some cases in which this is not true, which presents an ambiguity: if a class is not yet initialized, are assertions on or off?

Listing 6.2 presents an example of how code in a class can be run before it is finished initializing. It makes use of a certain paradoxical relationship between two classes, `CircularA` and `CircularB`.

Listing 6.2

```
public class CircularA
{
  static {
    CircularB.report();
  }
}

public class CircularB extends CircularA
{
  static public void report() {
    boolean assertionsOn = false;
    assert assertionsOn=true;
    System.out.println( "Assertions are "+
      (assertionsOn?"on":"off") );
  }

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

Here's the problem: `CircularB` must be initialized before its `main()` method can run, and `CircularB` is a subclass of `CircularA`, so `CircularA` must be initialized before `CircularB` can be initialized. However, `CircularA` has a static initializer that makes a call to `CircularB`, so `CircularB.report()` gets called before `CircularB` is fully initialized.

By running `CircularA` with assertions fully disabled, you can see that assertions are nevertheless enabled during initialization:

```
java -da -dsa CircularB
Assertions are on
Assertions are off
```

The assertion specification requires that assertions be enabled within a class during its initialization period, regardless of any other command-line settings or `ClassLoader` settings that have been (or will be) put into effect. This is the reason that the first call to `report()` states that assertions are on. Because of this, if a class checks, using the usual methods, to see if assertions are on, it will get a false positive during the initialization process.

The following method allows you to find out whether assertions will be enabled or not after initialization is complete. This is a method of `Class`.

```
public boolean desiredAssertionStatus();
```

In the few cases where you might validly make an execution decision based on whether assertions are enabled or not, this method can help you find this out during the initialization process.

Let's take a look at `CircularA` again. It was shown originally in listing 6.2, but in listing 6.3 it has been modified to check the real assertion status using `desiredAssertionStatus()`:

Listing 6.3

```
public class CircularA
{
  static {
    CircularB.report();
  }
}

public class CircularB extends CircularA
{
  static public void report() {
    boolean assertionsOn = false;
    assert assertionsOn=true;
    boolean assertionsWillBeOn =
      new CircularA().getClass().desiredAssertionStatus();
```

```
    System.out.println(
      "Assertions in CircularA: current="+assertionsOn+
      " desired="+assertionsWillBeOn );
  }

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

Here's the output:

```
java -da -dsa CircularB
Assertions in CircularA: current=true desired=false
Assertions in CircularA: current=false desired=false
```

As you can see, the desired assertion status is always `false`, even though assertions are temporarily on during initialization.

6.3 Assertion examples

This section presents examples of the kinds of conditions you might check for inside an assertion. We'll use both flavors of assertions so that you get a feel for each one.

6.3.1 Avoiding inconsistent states

The most common application of assertions is to ensure that the program remains in a consistent state.

JARGON A *consistent state* is any configuration of your program that makes sense according to the logic you've defined for it. An *inconsistent state* is any configuration that should never be reached.

Here's an example that checks for an inconsistent state. Let's say your program makes use of a `PipedInputStream/PipedOutputStream` pair.

```
import java.io.*;

public class Example
{
  private PipedInputStream pin;
  private PipedOutputStream pout;

  private void initializePipe() throws IOException {
    pin = new PipedInputStream();
    pout = new PipedOutputStream( pin );
  }
}
```

When the program starts, it has not yet created these objects, so it is in the state shown in table 6.2. Although the pipe cannot be used (because it is not there), this is nevertheless a *consistent state*, because it is intentional: before we've created our pipe, these variables must necessarily be `null`.

Table 6.2 Before the pipe is created, the variables must be `null`.

Variable	Value
`pin`	`null`
`pout`	`null`

Later on, we create the pipe by calling `initializePipe()`, at which point the state of our variables has changed, as shown in table 6.3. We're still in a consistent state, because here we define consistency to mean a state in which we have both ends of the pipe available to us.

Table 6.3 After the pipe is created, the state of the variables changes.

Variable	Value
`pin`	`java.io.PipedInputStream@3fbdb0`
`pout`	`java.io.PipedOutputStream@3e86d0`

Still later, our program closes the pipe and sets both variables to `null`, as shown in table 6.4. This is again a consistent state, representing the fact that we're done with the pipe.

Table 6.4 When the pipe is closed, the state of the variables is set to `null`.

Variable	Value
`pin`	`null`
`pout`	`null`

When our code has become much more complicated, however, we find that we have a bug in which, for some complicated reason, `pin` was set to `null` while `pout` was still pointing to an object (see table 6.5). This is an inconsistent state. As the designers of the code, we know that this state should never be entered. It represents no *conceptual* state that we could name. It does not represent the state in which we haven't started using the pipe, nor the state in which we are in the middle of using the pipe.

Table 6.5 A bug causes one variable to be non-`null`.

Variable	Value
`pin`	`null`
`pout`	`java.io.PipedOutputStream@3e86d0`

Realizing this, we decide to add assertions in various places to check for this inconsistent state. We want to make sure that both objects are `null` at the same time.

```
public void someMethod() {
  assert (pin==null) == (pout==null) :
    "Warning: pipe is inconsistent" );
}
```

We can sprinkle this assertion all over the code, as necessary.

6.3.2 Narrowing the range of states

There are times when we need to restrict the set of states that our program can be in, but we wouldn't really call it an issue of *consistency*. Generally this is as simple as trying to ensure that a particular variable contains a value within a certain subrange of possible values.

Let's say that our program is a little physics simulation, and our math should ensure that the velocities of our objects don't get out of control. Let's make sure:

```
public void runSimulation() {
  // ...

  assert
    Math.abs( velocity ) < 2000 :
    "Object way too fast! velocity="+velocity;
}
```

This example ensures that `velocity` never gets to be 2,000 or greater.

6.3.3 Ensuring consistency between container objects and contained objects

We'll use a somewhat more complex program to give a sense of what it means for a program to be in an inconsistent state and demonstrate the use of assertions. Our example will be an excerpt from a hypothetical chat server.

The `ChatServer` object maintains two lists of `Connection` objects. Each `Connection` object represents the connection to a `Client` and can be in one of two states: active or suspended. A suspended `Client` is one whose user has left his computer for a while.

The `ChatServer` object also has two methods for sending out messages to connected clients:

- `sendMessage()` sends a message to a particular connection
- `sendMessageToAll()` iterates through all of the active connections and calls `sendMessage()` on each one to send it a message

The problem is to make sure that the active list only contains active clients, and that the suspended list only contains suspended clients. To this end, we create a couple of methods, `setActive()` and `setSuspended()`. These set the state of a connection and also move it to the correct list:

```
import java.io.*;
import java.net.*;

public class ChatServer
{
  private List activeConnections;
  private List suspendedConnections;

  // ...

  synchronized void setActive( Connection connection ) {
    connection.setActive();
    suspendedConnections.remove( connection );
    activeConnections.add( connection );
  }

  synchronized void setSuspended( Connection connection ) {
    connection.setSuspended();
    passiveConnections.remove( connection );
    suspendedConnections.add( connection );
  }

  synchronized private void sendMessageToAll( Message message ) {
    for (Iterator iter=activeConnections.iterator();
                  iter.hasNext();) {
      Connection connection = (Connection)iter.next();
      sendMessage( connection, message );
    }
  }

  synchronized private void sendMessage( Connection connection,
                                         Message message ) {
    // ... send the message out to a particular connection
  }
}
```

Creating methods to perform list maintenance doesn't solve the problem entirely. Other code in our class, not seen here, might modify the `activeConnections` and `suspendedConnections` lists. Sometimes this happens on behalf of code we don't

have control over (or even have source code for). Even when we do control the code entirely, we still might make a mistake and fail to maintain the lists properly.

This is a perfect job for assertions:

```
import java.io.*;
import java.net.*;

public class ChatServer
{
  private List activeConnections;
  private List suspendedConnections;

  // ...

  synchronized void setActive( Connection connection ) {
    connection.setActive();
    suspendedConnections.remove( connection );
    activeConnections.add( connection );
  }

  synchronized void setSuspended( Connection connection ) {
    connection.setSuspended();
    passiveConnections.remove( connection );
    suspendedConnections.add( connection );
  }

  synchronized private void sendMessageToAll( Message message ) {
    for (Iterator iter=activeConnections.iterator();
                  iter.hasNext();) {
      Connection connection = (Connection)iter.next();
      assert connection.isActive();
      sendMessage( connection, message );
    }
  }

  synchronized private void sendMessage( Connection connection,
                                         Message message ) {
    assert activeConnections.contains( connection );
    // ... send the message out to a particular connection
  }
}
```

You might have noticed that there is some redundancy to the assertions. `sendMessageToAll()` iterates through the connections, and then calls `sendMessage()` for each connection. Both methods use assertions to make sure messages are never sent to a suspended connection. We're really getting two assertions per connection, which isn't necessary unless we suspect that the status of a connection can be changing at any time, and it probably can't because we're using proper synchronization.

If the preceding code were the only code in the class, we might consider taking out the second assertion, because it isn't necessary. But we might have other places

in our code that call `sendMessage()` directly, so we want to make sure we have an assertion happening in that case.

TIP We don't need to worry so much about the efficiency of assertions, because, in real usage, the assertions don't even get run. They only get run in a developmental context, where we don't mind wasting a few CPU cycles if it means our code is more stable.

6.3.4 More complicated consistency checks

Sometimes, there are consistency checks that are just too complicated to easily fit inside a single expression on a single line. In such cases, we can make a helper method that does the consistency check, and use an assertion to call it.

For example, suppose we have a class called `EmployeeDatabase` that contains a number of interlocking maps, lists, and sets. If you're not careful, it's easy to get some of the relationships between these objects into an inconsistent state.

At the same time, checking for consistency involves traversing the lists, keeping track of which things are on which lists, comparing sets of membership states, and so on. Often, consistency checking for such data structures involves building up a subset of the relationship from scratch while making sure it is consistent.

Since an assertion is an assertion of a single expression, it can be awkward, or even impossible, to put a complicated calculation right inside the `assert` expression, so we move it out to a helper method:

```
public class EmployeeDatabase()
{
  private Set employees;
  private Map employeeGroups;
  private SortedMap employeeTitles;
  private Set groups;
  private Map groupMemberships;
  private Set projects;
  private List groupDeadlines;

  public void doSomething() {

    // ...
    assert isConsistent() : "Error: inconsistent state!";
  }

  private boolean isConsistent() {

    // check lots and lots of stuff here
    // ...
  }
}
```

Setting things up this way can make code a lot easier to read, since the safety checks are all in one place. It also helps avoid having multiple copies of the safety checks in several places in the code, thus reducing code size and eliminating redundancy.

6.4 Knowing when to use assertions

The trickiest thing about assertions isn't knowing how to use them—it's knowing *when* and *where* to use them. This section outlines a number of guidelines, summarized in table 6.6, which should help you understand what assertions are appropriate for, and what they are not appropriate for.

Table 6.6 Assertions are often confused with regular conditionals. Follow these rules to distinguish what your particular situation calls for.

Assertion do's	Assertion don'ts
Do use to enforce internal assumptions about aspects of data structures	Don't use to enforce command-line usage
Do use to enforce constraints on arguments to private methods	Don't use to enforce constraints on arguments to public methods
Do use to check conditions at the end of any kind of method	Don't use to enforce public usage patterns or protocols
Do use to check for conditional cases that should never happen	Don't use to enforce a property of a piece of user-supplied information
Do use to check for conditional cases that should never happen, even if you're really sure they can never happen	Don't use as a shorthand for `if (something) error();`
Do use to check related conditions at the start of any method	Don't use as an externally controllable conditional
Do use to check things in the middle of a long-lived loop	Don't use as a check on the correctness of your compiler, operating system, or hardware, unless you have a specific reason to believe there is something wrong with it and are in the process of debugging it
Do use in lieu of nothing	

6.4.1 Rules of use

An assertion is not just a concise way to say `if (expression) then`. Rather, it is the basis of a discipline for making programs more robust.

It is very important to distinguish between situations where an assertion is needed and situations where a regular conditional is needed. The following rules should give you an idea of when assertions are appropriate, and when they are not.

Rule: do not use assertions to enforce command-line usage

Programs that use command-line arguments should always check the validity of the arguments, but this should be done with a regular conditional. The following example is an inappropriate use of an assertion:

```
public class Application
{
  static public void main( String args[] ) {
    // BAD!!
    assert args.length == 3;

    int a = Integer.parseInt( args[0] );
    int b = Integer.parseInt( args[1] );
    int c = Integer.parseInt( args[2] );
  }
}
```

It may be true that your program simply cannot run unless it is supplied with three arguments on the command line, but in this case it would be better to throw a proper `RuntimeException`:

```
public class App
{
  static public void main( String args[] ) {
    if (args.length != 3)
      throw new RuntimeException( "Usage: <progname> a b c" );

    int a = Integer.parseInt( args[0] );
    int b = Integer.parseInt( args[1] );
    int c = Integer.parseInt( args[2] );
  }
}
```

Assertions are meant to require that the program be consistent with itself, not that the user be consistent with the program.

Rule: use assertions to enforce constraints on arguments to private methods

The following private method takes two arguments; one is required, and one is optional:

```
private void method( Object required, Object optional ) {
  assert( required != null ) : "method(): required=null";
}
```

In general, `private` methods are probably being called by code we have control over, and which we expect are written correctly and consistently. As a result, we would like to think that all calls to our method are correct. We enforce this assumption with an assertion.

The same reasoning may apply to `protected` and `package-protected` methods.

Rule: do not use assertions to enforce constraints on arguments to public methods

The following public method takes two arguments: a source and a sink that are connected. Before disconnecting them, we'd like to ensure that they are connected to begin with:

```
public void disconnect( Source source, sink sink ) {
  // BAD!!
  assert source.isConnected( sink ) :
    "disconnect(): not connected "+source+","+sink;
}
```

In this example, `disconnect()` can only remove a connection between a `Source` and a `Sink` if they are in fact connected. However, because this method is `public`, the code that calls it might not be under your control.

More importantly, a `public` method guarantees that it will enforce the requirements of its specified interface in all situations. Assertions, on the other hand, are not guaranteed to run—they will only enforce their constraints if assertions are enabled in the runtime environment. This violates the promises made by the `public` method.

In this case, you should assume that the calling code might be in error, and throw a proper exception:

```
public void disconnect( Source source, sink sink ) throws IOException
{
  if (!source.isConnected( sink )) {
    throw new IOException(
      "disconnect(): not connected "+source+","+sink );
  }
}
```

This exception will be thrown regardless of whether assertions are on or off.

Rule: do not use assertions to enforce public usage patterns or protocols

The following `public` class can be in one of two states: open or closed. It is an error to open a `Connection` that is already open, or to close one that is already closed. However, we would not use an assertion to ensure that these mistakes are not made:

```
public class Connection
{
  private boolean isOpen = false;

  public void open() {
    // ...
    isOpen = true;
  }

  public void close() {
    // BAD!!
    assert isOpen : "Cannot close a connection that is not open!";
    // ...
  }
}
```

The programmer has attempted to enforce the requirement that a `Connection` can only be closed if it is already open.

This usage is valid *if and only if* the `Connection` class were a `private` class, or were otherwise guaranteed to be invisible to the outside, *and* if we were willing to ensure and assume that any code that uses this class is written correctly. In this case, it would be legitimate to enforce this assumption with an assertion.

However, if the `Connection` class is used publicly, it would not be surprising to find a bug in which someone tried to close a `Connection` that wasn't open in the first place. In this case, a regular exception would be better:

```
public class Connection
{
  private boolean isOpen = false;

  public void open() {
    // ...
    isOpen = true;
  }

  public void close() throws ConnectionException {
    if (!isOpen) {
      throw new ConnectionException(
        "Cannot close a connection that is not open!" );
    }

    // ...
  }
}
```

If you go the other route, and attempt to ensure that this code is only called from call sites you control, think twice. Any code you write now may be used or reused later in a different configuration. Anything can happen after the initial revision, so it's best to be on the safe side. Using an explicit exception provides the most information to a frustrated programmer down the line.

Rule: do not use assertions to enforce a property of a piece of user-supplied information

In the following code fragment, the programmer has used an assertion to make sure that a ZIP code has either five or nine digits:

```
public void processZipCode( String zipCode ) {
  if (zipCode.length() == 5) {
    // ...
  } else if (zipCode.length() == 9) {
    // ...
  } else {
    // BAD!!
    assert false : "Only 5- and 9-digit zip codes supported";
  }
}
```

Assertions should be used to enforce internal consistency, not correct input. The preceding code would be better served by using an explicit exception:

```
public void processZipCode( String zipCode )
    throws ZipCodeException {
  if (zipCode.length() == 5) {
    // ...
  } else if (zipCode.length() == 9) {
    // ...
  } else {
    throw new ZipCodeException(
      "Only 5- and 9-digit zip codes supported" );
  }
}
```

6.4.2 What to check for

Once you know where assertions should be used, you have to decide what to check and what not to check. Assertions are often used to check for things that are usually neglected, so keep the following rules in mind when you are deciding where to use assertions.

Rule: use assertions to enforce internal assumptions about aspects of data structures

The following private method takes an array of three integers. We use an assertion to make sure that the array is of the correct length:

```
private void showDate( int array[] ) {
  assert( array.length==3 );
}
```

We expect calls to this code to be written properly, and thus to only supply arrays of length three. This assertion merely enforces this assumption.

Java does have bounds-checked arrays, which means that this assertion isn't quite as crucial as it would be in a language like C or C++. However, this does not mean that using an assertion here isn't a good idea.

Rule: use assertions to check conditions at the end of any kind of method

Let's enhance the previous example with a few *postconditions*—that is, conditions checked after the body of a method, just before returning:

```
public class Connection
{
  private boolean isOpen = false;

  public void open() {
    // ...

    isOpen = true;

    // ...

    assert isOpen;
  }

  public void close() throws ConnectionException {
    if (!isOpen) {
      throw new ConnectionException(
        "Cannot close a connection that is not open!" );
    }

    // ...

    isOpen = false;

    // ...

    assert !isOpen;
  }
}
```

These assertions might seem redundant, but there's no telling what might come between the line where `isOpen` is set to `true`, and the line where `isOpen` is `asserted` to be `true`.

The assignment might eventually be put inside a conditional, removing the direct redundancy. Someone might forget to throw an exception, causing the assertion to be reached when it should have been skipped. The method might grow to be much larger, and get factored into several methods for readability. There's no telling what might happen to your code.

Rule: use assertions to check for conditional cases that should never happen

In the following code, the assertion checks for a conditional case that can't happen:

```
private int getValue() {
  if (/* something */) {
    return 0;
  } else if (/* something else */) {
    return 1;
  } else {
    return 2;
  }
}

public void method() {
  int a = getValue(); // returns 0, 1, or 2

  if (a==0) {
    // deal with 0 ...
  } else if (a==1) {
    // deal with 1 ...
  } else if (a==2) {
    // deal with 2 ...
  } else {
    assert false : "Impossible: a is out of range";
  }
}
```

In this example, we are receiving a value that we believe can only be in a certain range. It is valid in this case to use an assertion because `method()` makes no promises about handling values other than 0, 1, or 2.

Here is another way to write the code for `method()`:

```
public void method() {
  int a = getValue(); // returns 0, 1, or 2

  if (a==0) {
    // deal with 0 ...
  } else if (a==1) {
    // deal with 1 ...
  } else {
    assert a==2 : "Impossible: a is out of range";
    // deal with 2 ...
  }
}
```

This code is semantically equivalent to the original version.

Here is yet another equivalent implementation of the same method:

```
public void method() {
  int a = getValue(); // returns 0, 1, or 2

  switch( a ) {
    case 0:
      // deal with 0 ...
      break;
    case 1:
      // deal with 1 ...
      break;
    case 2:
      // deal with 2 ...
      break;
    default:
      assert false : "Impossible: a is out of range";
      break;
  }
}
```

Rule: use assertions to check for conditional cases that should never happen, even if you're really sure they can never happen

This next example might seem overly cautious:

```
public void method() {
  int a = getValue(); // returns 0, 1, or 2

  assert a>=0 && a<=2 : "Impossible: a is out of range";

  // ...

  if (a==0) {
    // deal with 0 ...
  } else if (a==1) {
    // deal with 1 ...
  } else {
    assert a==2;
    // deal with 2 ...
  }
}
```

In the preceding code fragment, it looks like we've checked twice for the same exact condition. As written, the assertions are redundant, because the value doesn't change between the first and second assertions. But let's take a look at the same code two releases later:

```
public void method() {
  int a = getValue(); // returns 0, 1, or 2
```

```
  assert a>=0 && a<=2 : "Impossible: a is out of range";

  // ...

  boolean shouldPromote = shouldPromote( b, c, d );
  if (shouldPromote && somethingElse)
    a++;
  a = modifyMaybe( a );

  // ...

  if (a==0) {
    // deal with 0 ...
  } else if (a==1) {
    // deal with 1 ...
  } else {
    assert a==2;
    // deal with 2 ...
  }
}
```

See? Without intending to complicate the invariants of your code, you or someone else has made the situation rather more complicated. Someone has inserted code between the first assertion and the second assertion that changes the value of the variable being checked. The second assertion is no longer redundant.

TIP It is good programming discipline to always have a final `else {}` case for any conditional. If you add one, but you know that it should never be reached, add an `assert false;`.

Rule: use assertions to check related conditions at the start of any method

In this example, the method `processZipCode()` wants to make sure that the program has already loaded a valid ZIP code map before it can process a ZIP code:

```
public void processZipCode( String zipCode ) {
  assert zipCodeMapIsValid();

  // ...
}
```

It is a fine idea to check related data structures in select locations, just in case. The idea here is to catch bugs early and often.

Rule: use assertions to check things in the middle of a long-lived loop

The `Server` class shown in listing 6.4 contains an inner loop that listens for network connections. This code might run for hours or days.

Listing 6.4

```
public class Server
{
  private ServerSocket serverSocket;

  public void acceptConnections() {
    while (true) {
      Socket socket = serverSocket.accept();      ● Long pause here

      assert socketListIsValid();      ● It's good to check some stuff

      // deal with new connection ...
    }
  }
}
```

A check placed in the middle of the long-lived loop ensures that assumptions that were made at the start of the loop continue to hold after time has passed.

6.4.3 Miscellaneous rules

This section highlights a few rules that don't fit into either of the previous two categories. Many of them serve to prevent the use of assertions where a stronger form of error-checking is preferable.

Rule: do not use an assertion as a shorthand for "if (something) error();"

Here, we incorrectly use an assertion to make sure that a port number is 1024 or greater:

```
public class Server
{
  private int port;

  public void listen() {
    // BAD!!
    assert port >= 1024 : "No permission to listen on port "+port;

    // ...
  }
}
```

In the preceding code, the programmer has been a little bit lazy, using an assertion for a valid, exceptional condition. Apparently he has forgotten that assertions generally do not run outside the development process. This should be an exception:

```
public class Server
{
  private int port;
```

```
  public void listen() {
    if (port < 1024) {
      throw new RuntimeException(
        "No permission to listen on port "+port );
    }

    // ...
  }
}
```

This version of the code ensures that the port number will be checked even when assertions are not enabled.

Rule: do not use an assertion as an externally controllable conditional

Here's a clever trick you might be tempted to do:

```
public class Application
{

  static private boolean turnLoggingOn() {
    // Turn logging on
    // ...

    return true;
  }

  static public void main( String args[] ) throws Exception {
    // ...

    // BAD!!
    assert turnLoggingOn();
  }
}
```

```
java -da Application
```

```
java -ea Application
```

As we saw in section 6.2.3, you can enable or disable assertions from the command line. You could use this ability to enable or disable something else by invoking that something else from within an assertion.

As clever as this is, it's bad idea, because it changes the semantics of the `-ea` and `-da` switches, and hijacks this facility for another purpose. End users should be able to enable or disable assertions purely on the basis of how vigilant they want the software to be during its execution; they should not have to worry about otherwise changing the semantics of the program.

WARNING Assertions should never have side effects, because the semantics of your program would be subject to whether assertions are enabled are not.

Rule: do not use assertions to check the correctness of your compiler, operating system, or hardware, unless you are debugging it

This code is clearly redundant:

```
public void method() {
  int a = 10;

  // REDUNDANT!!
  assert a==10;
}
```

As written, this assertion cannot possibly be triggered without a serious problem with your system at some level. A compiler bug might cause this assertion to trigger, and if you suspect that your compiler has a bug, this is a perfectly valid thing to do. However, putting such things in your code as a matter of course can confuse someone who might read it down the line, causing them to spend a good deal of time trying to figure out why the assertion was added in the first place.

Final rule: any assertion is better than nothing

As we saw at the beginning of the chapter, assertions are meant, above all, to be convenient enough to be added as an afterthought. Any time you suspect that you might be making an assumption that might not be true, and it makes you at all nervous, add an assertion. An assertion that is never triggered is far better than one that would have been triggered but isn't there.

6.5 Summary

In the computer science community, assertions are widely understood to be a powerful and flexible way to allow code to check itself for errors. It is designed to be as convenient as possible so that programmers can use it during development without distracting themselves from the task at hand.

In this sense, assertions have a *psychological* design as much as they have a technological design. Assertions are relatively cheap to implement, and while the implementation can differ from platform to platform, it is safe to assume that assertions can be used freely without a loss of program speed. The result will be better, more reliable code.