CHAPTER 7

Exceptions, Assertions, and Logging

In this chapter

In a perfect world, users would never enter data in the wrong form, files they choose to open would always exist, and code would never have bugs. So far, we have mostly presented code as if we lived in this kind of perfect world. It is now time to turn to the mechanisms the Java programming language has for dealing with the real world of bad data and buggy code.

Encountering errors is unpleasant. If a user loses all the work he or she did during a program session because of a programming mistake or some external circumstance, that user may forever turn away from your program. At the very least, you must:

- Notify the user of an error;
- Save all work; and
- Allow users to gracefully exit the program.

For exceptional situations, such as bad input data with the potential to bomb the program, Java uses a form of error trapping called, naturally enough, *exception handling.* Exception handling in Java is similar to that in C++ or Delphi. The first part of this chapter covers Java's exceptions.

During testing, you need to run lots of checks to make sure your program does the right thing. But those checks can be time consuming and unnecessary after testing has completed. You could just remove the checks and stick them back in when additional testing is required—but that is tedious. The second part of this chapter shows you how to use the assertion facility for selectively activating checks.

When your program does the wrong thing, you can't always communicate with the user or terminate. Instead, you may want to record the problem for later analysis. The third part of this chapter discusses the standard Java logging framework.

7.1 Dealing with Errors

Suppose an error occurs while a Java program is running. The error might be caused by a file containing wrong information, a flaky network connection, or (we hate to mention it) use of an invalid array index or an attempt to use an object reference that hasn't yet been assigned to an object. Users expect that programs will act sensibly when errors happen. If an operation cannot be completed because of an error, the program ought to either

- Return to a safe state and enable the user to execute other commands; or
- Allow the user to save all work and terminate the program gracefully.

This may not be easy to do, because the code that detects (or even causes) the error condition is usually far removed from the code that can roll back the data to a safe state or the code that can save the user's work and exit cheerfully. The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation. To handle exceptional situations in your program, you must take into account the errors and problems that may occur. What sorts of problems do you need to consider?

- *User input errors.* In addition to the inevitable typos, some users like to blaze their own trail instead of following directions. Suppose, for example, that a user asks to connect to a URL that is syntactically wrong. Your code should check the syntax, but suppose it does not. Then the network layer will complain.

- *Device errors.* Hardware does not always do what you want it to. The printer may be turned off. A web page may be temporarily unavailable. Devices will often fail in the middle of a task. For example, a printer may run out of paper during printing.
- *Physical limitations.* Disks can fill up; you can run out of available memory.
- *Code errors.* A method may not perform correctly. For example, it could deliver wrong answers or use other methods incorrectly. Computing an invalid array index, trying to find a nonexistent entry in a hash table, or trying to pop an empty stack are all examples of a code error.

The traditional reaction to an error in a method is to return a special error code that the calling method analyzes. For example, methods that read information back from files often return a `-1` end-of-file value marker rather than a standard character. This can be an efficient method for dealing with many exceptional conditions. Another common return value to denote an error condition is the `null` reference.

Unfortunately, it is not always possible to return an error code. There may be no obvious way of distinguishing valid and invalid data. A method returning an integer cannot simply return `-1` to denote the error; the value `-1` might be a perfectly valid result.

Instead, as we mentioned back in Chapter 5, Java allows every method an alternative exit path if it is unable to complete its task in the normal way. In this situation, the method does not return a value. Instead, it *throws* an object that encapsulates the error information. Note that the method exits immediately; it does not return its normal (or any) value. Moreover, execution does not resume at the code that called the method; instead, the exception-handling mechanism begins its search for an *exception handler* that can deal with this particular error condition.

Exceptions have their own syntax and are part of a special inheritance hierarchy. We'll take up the syntax first and then give a few hints on how to use this language feature effectively.

7.1.1 The Classification of Exceptions

In the Java programming language, an exception object is always an instance of a class derived from `Throwable`. As you will soon see, you can create your own exception classes if the ones built into Java do not suit your needs.

Figure 7.1 is a simplified diagram of the exception hierarchy in Java.

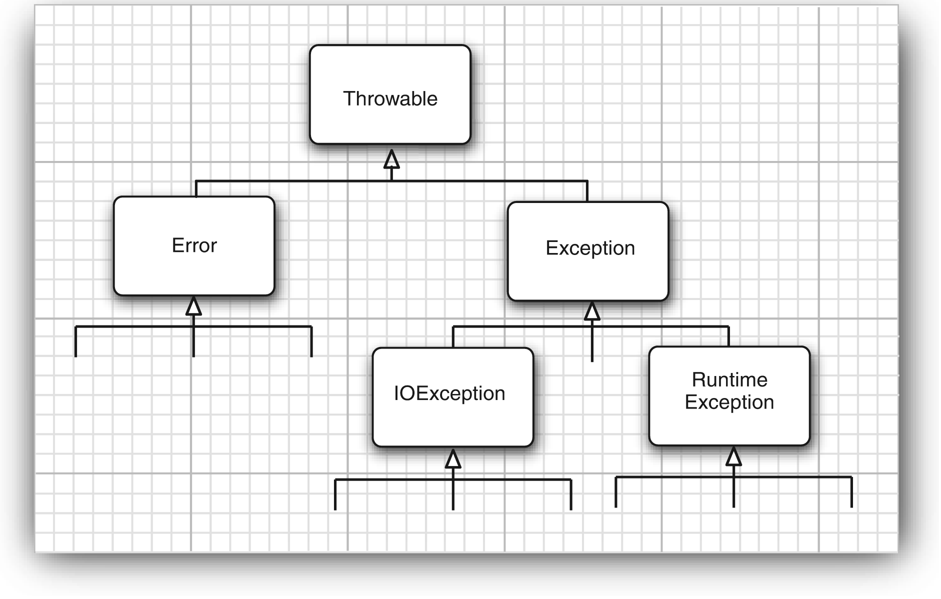


Figure 7.1 Exception hierarchy in Java

Notice that all exceptions descend from `Throwable`, but the hierarchy immediately splits into two branches: `Error` and `Exception`.

The `Error` hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system. You should not throw an object of this type. There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully. These situations are quite rare.

When doing Java programming, focus on the `Exception` hierarchy. The `Exception` hierarchy also splits into two branches: exceptions that derive from `RuntimeException` and those that do not. The general rule is this: A `RuntimeException` happens because you made a programming error. Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.

Exceptions that inherit from `RuntimeException` include such problems as

- A bad cast
- An out-of-bounds array access
- A null pointer access

Exceptions that do not inherit from `RuntimeException` include

- Trying to read past the end of a file
- Trying to open a file that doesn't exist
- Trying to find a `Class` object for a string that does not denote an existing class

The rule "If it is a `RuntimeException`, it was your fault" works pretty well. You could have avoided that `ArrayIndexOutOfBoundsException` by testing the array index against the array bounds. The `NullPointerException` would not have happened had you checked whether the variable was `null` before using it.

How about a file that doesn't exist? Can't you first check whether the file exists, and then open it? Well, the file might be deleted right after you checked for its existence. Thus, the notion of "existence" depends on the environment, not just on your code.

The Java Language Specification calls any exception that derives from the class `Error` or the class `RuntimeException` an *unchecked* exception. All other exceptions are called *checked* exceptions. This is useful terminology that we also adopt. The compiler checks that you provide exception handlers for all checked exceptions.

NOTE: The name `RuntimeException` is somewhat confusing. Of course, all of the errors we are discussing occur at runtime.

C++ NOTE: If you are familiar with the (much more limited) exception hierarchy of the standard C++ library, you may be really confused at this point. C++ has two fundamental exception classes, `runtime_error` and `logic_error`. The `logic_error` class is the equivalent of Java's `RuntimeException` and also denotes logical errors in the program. The `runtime_error` class is the base class for exceptions caused by unpredictable problems. It is equivalent to those exceptions in Java that are not of type `RuntimeException`.

7.1.2 Declaring Checked Exceptions

A Java method can throw an exception if it encounters a situation it cannot handle. The idea is simple: A method will not only tell the Java compiler what values it can return, *it is also going to tell the compiler what can go wrong.* For example, code that attempts to read from a file knows that the file might not exist or that it might be empty. The code that tries to process the information in a file therefore will need to notify the compiler that it can throw some sort of `IOException`.

The place in which you advertise that your method can throw an exception is the header of the method; the header changes to reflect the checked exceptions the method can throw. For example, here is the declaration of one of the constructors of the FileInputStream class from the standard library. (See Chapter 2 of Volume II for more on input and output.)

```
public FileInputStream(String name) throws FileNotFoundException
```

The declaration says that this constructor produces a FileInputStream object from a String parameter but that it *also* can go wrong in a special way—by throwing a FileNotFoundException. If this sad state should come to pass, the constructor call will not initialize a new FileInputStream object but instead will throw an object of the FileNotFoundException class. If it does, the runtime system will begin to search for an exception handler that knows how to deal with FileNotFoundException objects.

When you write your own methods, you don't have to advertise every possible throwable object that your method might actually throw. To understand when (and what) you have to advertise in the throws clause of the methods you write, keep in mind that an exception is thrown in any of the following four situations:

- You call a method that throws a checked exception—for example, the FileInputStream constructor.
- You detect an error and throw a checked exception with the throw statement (we cover the throw statement in the next section).
- You make a programming error, such as a[-1] = 0 that gives rise to an unchecked exception (in this case, an ArrayIndexOutOfBoundsException).
- An internal error occurs in the virtual machine or runtime library.

If either of the first two scenarios occurs, you must tell the programmers who will use your method about the possibility of an exception. Why? Any method that throws an exception is a potential death trap. If no handler catches the exception, the current thread of execution terminates.

As with Java methods that are part of the supplied classes, you declare that your method may throw an exception with an *exception specification* in the method header.

```
class MyAnimation
{
   . . .
   public Image loadImage(String s) throws IOException
   {
      . . .
   }
}
```

If a method might throw more than one checked exception type, you must list all exception classes in the header. Separate them by commas, as in the following example:

```
class MyAnimation
{
   . . .
   public Image loadImage(String s) throws FileNotFoundException, EOFException
   {
      . . .
   }
}
```

However, you do not need to advertise internal Java errors—that is, exceptions inheriting from `Error`. Any code could potentially throw those exceptions, and they are entirely beyond your control.

Similarly, you should not advertise unchecked exceptions inheriting from `RuntimeException`.

```
class MyAnimation
{
   . . .
   void drawImage(int i) throws ArrayIndexOutOfBoundsException // bad style
   {
      . . .
   }
}
```

These runtime errors are completely under your control. If you are so concerned about array index errors, you should spend your time fixing them instead of advertising the possibility that they can happen.

In summary, a method must declare all the *checked* exceptions that it might throw. Unchecked exceptions are either beyond your control (`Error`) or result from conditions that you should not have allowed in the first place (`RuntimeException`). If your method fails to faithfully declare all checked exceptions, the compiler will issue an error message.

Of course, as you have already seen in quite a few examples, instead of declaring the exception, you can also catch it. Then the exception won't be thrown out of the method, and no `throws` specification is necessary. You will see later in this chapter how to decide whether to catch an exception or to enable someone else to catch it.

CAUTION: If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method. (It is OK to throw more specific exceptions, or not to throw any exceptions in the subclass method.) In particular, if the superclass method throws no checked exception at all, neither can the subclass. For example, if you override `JComponent.paintComponent`, your `paintComponent` method must not throw any checked exceptions, because the superclass method doesn't throw any.

When a method in a class declares that it throws an exception that is an instance of a particular class, it may throw an exception of that class or of any of its subclasses. For example, the `FileInputStream` constructor could have declared that it throws an `IOException`. In that case, you would not have known what kind of `IOException` it is; it could be a plain `IOException` or an object of one of the various subclasses, such as `FileNotFoundException`.

C++ NOTE: The `throws` specifier is the same as the `throw` specifier in C++, with one important difference. In C++, `throw` specifiers are enforced at runtime, not at compile time. That is, the C++ compiler pays no attention to exception specifications. But if an exception is thrown in a function that is not part of the `throw` list, the `unexpected` function is called, and, by default, the program terminates.

Also, in C++, a function may throw any exception if no `throw` specification is given. In Java, a method without a `throws` specifier may not throw any checked exceptions at all.

7.1.3 How to Throw an Exception

Now, suppose something terrible has happened in your code. You have a method, `readData`, that is reading in a file whose header promised

```
Content-length: 1024
```

but you got an end of file after 733 characters. You may decide this situation is so abnormal that you want to throw an exception.

You need to decide what exception type to throw. Some kind of `IOException` would be a good choice. Perusing the Java API documentation, you find an `EOFException` with the description "Signals that an EOF has been reached unexpectedly during input." Perfect. Here is how you throw it:

```
throw new EOFException();
```

or, if you prefer,

```
EOFException e = new EOFException();
throw e;
```

Here is how it all fits together:

```
String readData(Scanner in) throws EOFException
{
   . . .
   while (. . .)
   {
      if (!in.hasNext()) // EOF encountered
      {
         if (n < len)
            throw new EOFException();
      }
      . . .
   }
   return s;
}
```

The `EOFException` has a second constructor that takes a string argument. You can put this to good use by describing the exceptional condition more carefully.

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

As you can see, throwing an exception is easy if one of the existing exception classes works for you. In this case:

1. Find an appropriate exception class.
2. Make an object of that class.
3. Throw it.

Once a method throws an exception, it does not return to its caller. This means you do not have to worry about cooking up a default return value or an error code.

C++ **C++ NOTE:** Throwing an exception is the same in C++ and in Java, with one small difference. In Java, you can throw only objects of subclasses of `Throwable`. In C++, you can throw values of any type.

7.1.4 Creating Exception Classes

Your code may run into a problem which is not adequately described by any of the standard exception classes. In this case, it is easy enough to create your own exception class. Just derive it from `Exception`, or from a child class of `Exception` such as `IOException`. It is customary to give both a default constructor and a constructor that contains a detailed message. (The `toString` method of the `Throwable` superclass returns a string containing that detailed message, which is handy for debugging.)

```
class FileFormatException extends IOException
{
   public FileFormatException() {}
   public FileFormatException(String gripe)
   {
      super(gripe);
   }
}
```

Now you are ready to throw your very own exception type.

```
String readData(BufferedReader in) throws FileFormatException
{
   . . .
   while (. . .)
   {
      if (ch == -1) // EOF encountered

      {
         if (n < len)
            throw new FileFormatException();
      }
      . . .
   }
   return s;
}
```

`java.lang.Throwable` 1.0

- `Throwable()`

  constructs a new `Throwable` object with no detailed message.
- `Throwable(String message)`

  constructs a new `Throwable` object with the specified detailed message. By convention, all derived exception classes support both a default constructor and a constructor with a detailed message.
- `String getMessage()`

  gets the detailed message of the `Throwable` object.

7.2 Catching Exceptions

You now know how to throw an exception. It is pretty easy: You throw it and you forget it. Of course, some code has to catch the exception. Catching exceptions requires more planning. That's what the next sections will cover.

7.2.1 Catching an Exception

If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace. GUI programs (both applets and applications) catch exceptions, print stack trace messages, and then go back to the user interface processing loop. (When you are debugging a GUI program, it is a good idea to keep the console on the screen and not minimized.)

To catch an exception, set up a `try/catch` block. The simplest form of the `try` block is as follows:

```
try
{
   code
   more code
   more code
}
catch (ExceptionType e)
{
   handler for this type
}
```

If any code inside the `try` block throws an exception of the class specified in the `catch` clause, then

1. The program skips the remainder of the code in the `try` block.
2. The program executes the handler code inside the `catch` clause.

If none of the code inside the `try` block throws an exception, then the program skips the `catch` clause.

If any of the code in a method throws an exception of a type other than the one named in the `catch` clause, this method exits immediately. (Hopefully, one of its callers has already provided a `catch` clause for that type.)

To show this at work, here's some fairly typical code for reading in data:

```
public void read(String filename)
{
   try
   {
         InputStream in = new FileInputStream(filename);
         int b;
         while ((b = in.read()) != -1)
         {
            process input
         }
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }
```

Notice that most of the code in the `try` clause is straightforward: It reads and processes bytes until we encounter the end of the file. As you can see by looking at the Java API, there is the possibility that the `read` method will throw an `IOException`. In that case, we skip out of the entire `while` loop, enter the `catch` clause, and generate a stack trace. For a toy program, that seems like a reasonable way to deal with this exception. What other choice do you have?

Often, the best choice is to do nothing at all and simply pass the exception on to the caller. If an error occurs in the `read` method, let the caller of the `read` method worry about it! If we take that approach, then we have to advertise the fact that the method may throw an `IOException`.

```
   public void read(String filename) throws IOException
   {
      InputStream in = new FileInputStream(filename);
      int b;
      while ((b = in.read()) != -1)
      {
         process input
      }
   }
```

Remember, the compiler strictly enforces the `throws` specifiers. If you call a method that throws a checked exception, you must either handle it or pass it on.

Which of the two is better? As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle.

When you propagate an exception, you must add a `throws` specifier to alert the caller that an exception may be thrown.

Look at the Java API documentation to see what methods throw which exceptions. Then decide whether you should handle them or add them to the `throws` list. There is nothing embarrassing about the latter choice. It is better to direct an exception to a competent handler than to squelch it.

Please keep in mind that there is, as we mentioned earlier, one exception to this rule. If you are writing a method that overrides a superclass method which throws no exceptions (such as `paintComponent` in `JComponent`), then you *must* catch each checked exception in the method's code. You are not allowed to add more `throws` specifiers to a subclass method than are present in the superclass method.

C++ **C++ NOTE:** Catching exceptions is almost the same in Java and in C++. Strictly speaking, the analog of

```
catch (Exception e) // Java
```

is

```
catch (Exception& e) // C++
```

There is no analog to the C++ `catch (. . .)`. This is not needed in Java because all exceptions derive from a common superclass.

7.2.2 Catching Multiple Exceptions

You can catch multiple exception types in a `try` block and handle each type differently. Use a separate `catch` clause for each type as in the following example:

```
try
{
   code that might throw exceptions
}
catch (FileNotFoundException e)
{
   emergency action for missing files
}
catch (UnknownHostException e)
{
   emergency action for unknown hosts
}
catch (IOException e)
{
   emergency action for all other I/O problems
}
```

The exception object may contain information about the nature of the exception. To find out more about the object, try

```
e.getMessage()
```

to get the detailed error message (if there is one), or

```
e.getClass().getName()
```

to get the actual type of the exception object.

As of Java SE7, you can catch multiple exception types in the same `catch` clause. For example, suppose that the action for missing files and unknown hosts is the same. Then you can combine the `catch` clauses:

```
try
{
   code that might throw exceptions
}
catch (FileNotFoundException | UnknownHostException e)
{
   emergency action for missing files and unknown hosts
}
catch (IOException e)
{
   emergency action for all other I/O problems
}
```

This feature is only needed when catching exception types that are not subclasses of one another.

NOTE: When you catch multiple exceptions, the exception variable is implicitly `final`. For example, you cannot assign a different value to `e` in the body of the clause

```
catch (FileNotFoundException | UnknownHostException e) { . . . }
```

NOTE: Catching multiple exceptions doesn't just make your code look simpler but also more efficient. The generated bytecodes contain a single block for the shared `catch` clause.

7.2.3 Rethrowing and Chaining Exceptions

You can throw an exception in a `catch` clause. Typically, you do this when you want to change the exception type. If you build a subsystem that other programmers use, it makes a lot of sense to use an exception type that indicates a failure of the subsystem. An example of such an exception type is the `ServletException`. The code that executes a servlet may not want to know in minute detail what went wrong, but it definitely wants to know that the servlet was at fault.

Here is how you can catch an exception and rethrow it:

```
try
{
   access the database
}
catch (SQLException e)
{
   throw new ServletException("database error: " + e.getMessage());
}
```

Here, the `ServletException` is constructed with the message text of the exception.

However, it is a better idea to set the original exception as the "cause" of the new exception:

```
try
{
   access the database
}
catch (SQLException e)
{
   Throwable se = new ServletException("database error");
   se.initCause(e);
   throw se;
}
```

When the exception is caught, the original exception can be retrieved:

```
Throwable e = se.getCause();
```

This wrapping technique is highly recommended. It allows you to throw high-level exceptions in subsystems without losing the details of the original failure.

✓ **TIP:** The wrapping technique is also useful if a checked exception occurs in a method that is not allowed to throw a checked exception. You can catch the checked exception and wrap it into a runtime exception.

Sometimes, you just want to log an exception and rethrow it without any change:

```
try
{
   access the database
}
catch (Exception e)
{
   logger.log(level, message, e);
   throw e;
}
```

Before Java SE 7, there was a problem with this approach. Suppose the code is inside a method

```
public void updateRecord() throws SQLException
```

The Java compiler looked at the `throw` statement inside the `catch` block, then at the type of `e`, and complained that this method might throw any `Exception`, not just a `SQLException`. This has now been improved. The compiler now tracks the fact that `e` originates from the `try` block. Provided that the only checked exceptions in that block are `SQLException` instances, and provided that `e` is not changed in the `catch` block, it is valid to declare the enclosing method as `throws SQLException`.

7.2.4 The `finally` Clause

When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up. One solution is to catch and rethrow all exceptions. But this solution is tedious because you need to clean up the resource allocation in two places—in the normal code and in the exception code.

Java has a better solution: the `finally` clause. Here we show you how to properly close a file in Java. If you do any database programming, you will need to use the same technique to close connections to the database. As you will see in Chapter 4 of Volume II, it is very important to close all database connections properly, even when exceptions occur.

The code in the `finally` clause executes whether or not an exception was caught. In the following example, the program will dispose of the graphics context *under all circumstances*:

```
InputStream in = new FileInputStream(. . .);
try
{
   // 1
   code that might throw exceptions
   // 2
}
catch (IOException e)
{
   // 3
   show error message
   // 4
}
finally
{
   // 5
   in.close();
}
// 6
```

Let us look at the three possible situations in which the program will execute the `finally` clause.

1. The code throws no exceptions. In this case, the program first executes all the code in the `try` block. Then, it executes the code in the `finally` clause. Afterwards, execution continues with the first statement after the `finally` clause. In other words, execution passes through points 1, 2, 5, and 6.
2. The code throws an exception that is caught in a `catch` clause—in our case, an `IOException`. For this, the program executes all code in the `try` block, up to the point at which the exception was thrown. The remaining code in the `try` block is skipped. The program then executes the code in the matching `catch` clause, and then the code in the `finally` clause.

   If the `catch` clause does not throw an exception, the program executes the first line after the `finally` clause. In this scenario, execution passes through points 1, 3, 4, 5, and 6.

   If the `catch` clause throws an exception, then the exception is thrown back to the caller of this method, and execution passes through points 1, 3, and 5 only.
3. The code throws an exception that is not caught in any `catch` clause. Here, the program executes all code in the `try` block until the exception is thrown. The remaining code in the `try` block is skipped. Then, the code in the `finally` clause is executed, and the exception is thrown back to the caller of this method. Execution passes through points 1 and 5 only.

You can use the `finally` clause without a `catch` clause. For example, consider the following `try` statement:

```
InputStream in = . . .;
try
{
   code that might throw exceptions
}
finally
{
   in.close();
}
```

The `in.close()` statement in the `finally` clause is executed whether or not an exception is encountered in the `try` block. Of course, if an exception is encountered, it is rethrown and must be caught in another `catch` clause.

In fact, as explained in the following tip, we think it is a very good idea to use the `finally` clause in this way whenever you need to close a resource.

TIP: We strongly suggest that you *decouple* `try/catch` and `try/finally` blocks. This makes your code far less confusing. For example:

```
InputStream in = . . .;
try
{
   try
   {
      code that might throw exceptions
   }
   finally
   {
      in.close();
   }
}
catch (IOException e)
{
   show error message
}
```

The inner `try` block has a single responsibility: to make sure that the input stream is closed. The outer `try` block has a single responsibility: to ensure that errors are reported. Not only is this solution clearer, it is also more functional: Errors in the `finally` clause are reported.

CAUTION: A `finally` clause can yield unexpected results when it contains `return` statements. Suppose you exit the middle of a `try` block with a `return` statement. Before the method returns, the `finally` block is executed. If the `finally` block also contains a `return` statement, then it masks the original return value. Consider this contrived example:

```
public static int f(int n)
{
   try
   {
      int r = n * n;
      return r;
   }
   finally
   {
      if (n == 2) return 0;
   }
}
```

If you call `f(2)`, then the `try` block computes `r = 4` and executes the `return` statement. However, the `finally` clause is executed before the method actually returns and causes the method to return `0`, ignoring the original return value of `4`.

Sometimes the finally clause gives you grief—namely, if the cleanup method can also throw an exception. Suppose you want to make sure that you close a stream when an exception hits in the stream processing code.

```
InputStream in = . . .;
try
{
   code that might throw exceptions
}
finally
{
   in.close();
}
```

Now suppose that the code in the try block throws some exception *other than* an IOException which is of interest to the caller of the code. The finally block executes, and the close method is called. That method can itself throw an IOException! When it does, the original exception is lost and the exception of the close method is thrown instead.

This is a problem because the first exception is likely to be more interesting. If you want to do the right thing and rethrow the original exception, the code becomes incredibly tedious. Here is one way of setting it up:

```
InputStream in = . . .;
Exception ex = null;
try
{
   try
   {
      code that might throw exceptions
   }
   catch (Exception e)
   {
      ex = e;
      throw e;
   }
}
finally
{
   try
   {
      in.close();
   }
   catch (Exception e)
   {
      if (ex == null) throw e;
   }
}
```

Fortunately, Java SE 7 has made it much easier to deal with closing resources, as you will see in the next section.

7.2.5 The Try-with-Resources Statement

Java SE 7 provides a useful shortcut to the code pattern

```
open a resource
try
{
   work with the resource
}
finally
{
   close the resource
}
```

provided the resource belongs to a class that implements the `AutoCloseable` interface. That interface has a single method

```
void close() throws Exception
```

NOTE: There is also a `Closeable` interface. It is a subinterface of `AutoCloseable`, also with a single `close` method. However, that method is declared to throw an `IOException`.

In its simplest variant, the try-with-resources statement has the form

```
try (Resource res = . . .)
{
   work with res
}
```

When the `try` block exits, then `res.close()` is called automatically. Here is a typical example—reading all words of a file:

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words"), "UTF-8"))
{
   while (in.hasNext())
      System.out.println(in.next());
}
```

When the block exits normally, or when there was an exception, the `in.close()` method is called, exactly as if you had used a `finally` block.

You can specify multiple resources. For example,

```
try (Scanner in = new Scanner(new FileInputStream("/usr/share/dict/words"), "UTF-8");
   PrintWriter out = new PrintWriter("out.txt"))
{
   while (in.hasNext())
      out.println(in.next().toUpperCase());
}
```

No matter how the block exits, both `in` and `out` are closed. If you programmed this by hand, you would need two nested `try`/`finally` statements.

As you have seen in the preceding section, a difficulty arises when the `try` block throws an exception and the `close` method also throws an exception. The try-with-resources statement handles this situation quite elegantly. The original exception is rethrown, and any exceptions thrown by `close` methods are considered "suppressed." They are automatically caught and added to the original exception with the `addSuppressed` method. If you are interested in them, call the `getSuppressed` method which yields an array of the suppressed expressions from `close` methods.

You don't want to program this by hand. Use the try-with-resources statement whenever you need to close a resource.

NOTE: A try-with-resources statement can itself have `catch` clauses and a `finally` clause. These are executed after closing the resources. In practice, it's probably not a good idea to pile so much onto a single `try` statement.

7.2.6 Analyzing Stack Trace Elements

A *stack trace* is a listing of all pending method calls at a particular point in the execution of a program. You have almost certainly seen stack trace listings—they are displayed whenever a Java program terminates with an uncaught exception.

You can access the text description of a stack trace by calling the `printStackTrace` method of the `Throwable` class.

```
Throwable t = new Throwable();
StringWriter out = new StringWriter();
t.printStackTrace(new PrintWriter(out));
String description = out.toString();
```

A more flexible approach is the `getStackTrace` method that yields an array of `StackTraceElement` objects, which you can analyze in your program. For example:

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
   analyze frame
```

The `StackTraceElement` class has methods to obtain the file name and line number, as well as the class and method name, of the executing line of code. The `toString` method yields a formatted string containing all of this information.

The static `Thread.getAllStackTraces` method yields the stack traces of all threads. Here is how you use that method:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
   StackTraceElement[] frames = map.get(t);
   analyze frames
}
```

See Chapters 9 and 14 for more information on the `Map` interface and threads.

Listing 7.1 prints the stack trace of a recursive factorial function. For example, if you compute `factorial(3)`, the printout is

```
factorial(3):
stackTrace.StackTraceTest.factorial(StackTraceTest.java:20)
stackTrace.StackTraceTest.main(StackTraceTest.java:36)
factorial(2):
stackTrace.StackTraceTest.factorial(StackTraceTest.java:20)
stackTrace.StackTraceTest.factorial(StackTraceTest.java:26)
stackTrace.StackTraceTest.main(StackTraceTest.java:36)
factorial(1):
stackTrace.StackTraceTest.factorial(StackTraceTest.java:20)
stackTrace.StackTraceTest.factorial(StackTraceTest.java:26)
stackTrace.StackTraceTest.factorial(StackTraceTest.java:26)
stackTrace.StackTraceTest.main(StackTraceTest.java:36)
return 1
return 2
return 6
```

Listing 7.1 `stackTrace/StackTraceTest.java`

```
package stackTrace;

import java.util.*;

/**
 * A program that displays a trace feature of a recursive method call.
 * @version 1.01 2004-05-10
 * @author Cay Horstmann
 */
public class StackTraceTest
{
   /**
    * Computes the factorial of a number
    * @param n a non-negative integer
    * @return n! = 1 * 2 * . . . * n
    */
   public static int factorial(int n)
   {
      System.out.println("factorial(" + n + "):");
      Throwable t = new Throwable();
      StackTraceElement[] frames = t.getStackTrace();
      for (StackTraceElement f : frames)
         System.out.println(f);
      int r;
      if (n <= 1) r = 1;
      else r = n * factorial(n - 1);
      System.out.println("return " + r);
      return r;
   }

   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();
      factorial(n);
   }
}
```

`java.lang.Throwable` 1.0

- `Throwable(Throwable cause)` 1.4
- `Throwable(String message, Throwable cause)` 1.4

  constructs a `Throwable` with a given cause.

- `Throwable initCause(Throwable cause)` 1.4

  sets the cause for this object or throws an exception if this object already has a cause. Returns `this`.

- `Throwable getCause()` 1.4

  gets the exception object that was set as the cause for this object, or `null` if no cause was set.

- `StackTraceElement[] getStackTrace()` 1.4

  gets the trace of the call stack at the time this object was constructed.

(Continues)

`java.lang.Throwable` 1.0 *(Continued)*

- `void addSuppressed(Throwable t)` 7

  adds a "suppressed" exception to this exception. This happens in a try-with-resources statement where `t` is an exception thrown by a `close` method.
- `Throwable[] getSuppressed()` 7

  gets all "suppressed" exceptions of this exception. Typically, these are exceptions thrown by a `close` method in a try-with-resources statement.

`java.lang.Exception` 1.0

- `Exception(Throwable cause)` 1.4
- `Exception(String message, Throwable cause)`

  constructs an `Exception` with a given cause.

`java.lang.RuntimeException` 1.0

- `RuntimeException(Throwable cause)` 1.4
- `RuntimeException(String message, Throwable cause)` 1.4

  constructs a `RuntimeException` with a given cause.

`java.lang.StackTraceElement` 1.4

- `String getFileName()`

  gets the name of the source file containing the execution point of this element, or `null` if the information is not available.
- `int getLineNumber()`

  gets the line number of the source file containing the execution point of this element, or `-1` if the information is not available.
- `String getClassName()`

  gets the fully qualified name of the class containing the execution point of this element.
- `String getMethodName()`

  gets the name of the method containing the execution point of this element. The name of a constructor is `<init>`. The name of a static initializer is `<clinit>`. You can't distinguish between overloaded methods with the same name.

(Continues)

java.lang.StackTraceElement 1.4 *(Continued)*

- `boolean isNativeMethod()`

  returns `true` if the execution point of this element is inside a native method.
- `String toString()`

  returns a formatted string containing the class and method name and the file name and line number, if available.

7.3 Tips for Using Exceptions

There is a certain amount of controversy about the proper use of exceptions. Some programmers believe that all checked exceptions are a nuisance, others can't seem to throw enough of them. We think that exceptions (even checked exceptions) have their place, and offer you these tips for their proper use.

1. *Exception handling is not supposed to replace a simple test.*

   As an example of this, we wrote some code that tries 10,000,000 times to pop an empty stack. It first does this by finding out whether the stack is empty.

   ```
   if (!s.empty()) s.pop();
   ```

   Next, we force it to pop the stack no matter what and then catch the `EmptyStackException` that tells us we should not have done that.

   ```
   try
   {
      s.pop();
   }
   catch (EmptyStackException e)
   {
   }
   ```

   On our test machine, the version that calls `isEmpty` ran in 646 milliseconds. The version that catches the `EmptyStackException` ran in 21,739 milliseconds.

   As you can see, it took far longer to catch an exception than to perform a simple test. The moral is: Use exceptions for exceptional circumstances only.

2. *Do not micromanage exceptions.*

   Many programmers wrap every statement in a separate `try` block.

   ```
   PrintStream out;
   Stack s;
   ```

```
for (i = 0; i < 100; i++)
{
   try
   {
      n = s.pop();
   }
   catch (EmptyStackException e)
   {
      // stack was empty
   }
   try
   {
      out.writeInt(n);
   }
   catch (IOException e)
   {
      // problem writing to file
   }
}
```

This approach blows up your code dramatically. Think about the task that you want the code to accomplish. Here, we want to pop 100 numbers off a stack and save them to a file. (Never mind why—it is just a toy example.) There is nothing we can do if a problem rears its ugly head. If the stack is empty, it will not become occupied. If the file contains an error, the error will not magically go away. It therefore makes sense to wrap the *entire task* in a `try` block. If any one operation fails, you can then abandon the task.

```
try
{
   for (i = 0; i < 100; i++)
   {
      n = s.pop();
      out.writeInt(n);
   }
}
catch (IOException e)
{
   // problem writing to file
}
catch (EmptyStackException e)
{
   // stack was empty
}
```

This code looks much cleaner. It fulfills one of the promises of exception handling: to *separate* normal processing from error handling.

3. *Make good use of the exception hierarchy.*

   Don't just throw a `RuntimeException`. Find an appropriate subclass or create your own.

   Don't just catch `Throwable`. It makes your code hard to read and maintain.

   Respect the difference between checked and unchecked exceptions. Checked exceptions are inherently burdensome—don't throw them for logic errors. (For example, the reflection library gets this wrong. Callers often need to catch exceptions that they know can never happen.)

   Do not hesitate to turn an exception into another exception that is more appropriate. For example, when you parse an integer in a file, catch the `NumberFormatException` and turn it into a subclass of `IOException` or `MySubsystemException`.

4. *Do not squelch exceptions.*

   In Java, there is a tremendous temptation to shut up exceptions. If you're writing a method that calls a method that might throw an exception once a century, the compiler whines because you have not declared the exception in the `throws` list of your method. You do not want to put it in the `throws` list because then the compiler will whine about all the methods that call your method. So you just shut it up:

   ```
   public Image loadImage(String s)
   {
      try
      {
         // code that threatens to throw checked exceptions
      }
      catch (Exception e)
      {} // so there
   }
   ```

   Now your code will compile without a hitch. It will run fine, except when an exception occurs. Then, the exception will be silently ignored. If you believe that exceptions are at all important, you should make some effort to handle them right.

5. *When you detect an error, "tough love" works better than indulgence.*

   Some programmers worry about throwing exceptions when they detect errors. Maybe it would be better to return a dummy value rather than throw an exception when a method is called with invalid parameters? For example, should `Stack.pop` return `null`, or throw an exception when a stack is empty? We think it is better to throw a `EmptyStackException` at the point of failure than to have a `NullPointerException` occur at later time.

6. *Propagating exceptions is not a sign of shame.*

   Many programmers feel compelled to catch all exceptions that are thrown. If they call a method that throws an exception, such as the `FileInputStream` constructor or the `readLine` method, they instinctively catch the exception that may be generated. Often, it is actually better to *propagate* the exception instead of catching it:

   ```
   public void readStuff(String filename) throws IOException // not a sign of shame!
   {
      InputStream in = new FileInputStream(filename);
      . . .
   }
   ```

   Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.

NOTE: Rules 5 and 6 can be summarized as “throw early, catch late.”

7.4 Using Assertions

Assertions are a commonly used idiom of defensive programming. In the following sections, you will learn how to use them effectively.

7.4.1 The Assertion Concept

Suppose you are convinced that a particular property is fulfilled, and you rely on that property in your code. For example, you may be computing

```
double y = Math.sqrt(x);
```

You are certain that `x` is not negative. Perhaps it is the result of another computation that can’t have a negative result, or it is a parameter of a method that requires its callers to supply only positive inputs. Still, you want to double-check rather than allow confusing “not a number” floating-point values creep into your computation. You could, of course, throw an exception:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

But this code stays in the program, even after testing is complete. If you have lots of checks of this kind, the program may run quite a bit slower than it should.

The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.

The Java language has a keyword `assert`. There are two forms:

```
assert condition;
```

and

```
assert condition : expression;
```

Both statements evaluate the condition and throw an `AssertionError` if it is `false`. In the second statement, the expression is passed to the constructor of the `AssertionError` object and turned into a message string.

NOTE: The sole purpose of the *expression* part is to produce a message string. The `AssertionError` object does not store the actual expression value, so you can't query it later. As the JDK documentation states with paternalistic charm, doing so "would encourage programmers to attempt to recover from assertion failure, which defeats the purpose of the facility."

To assert that `x` is non-negative, you can simply use the statement

```
assert x >= 0;
```

Or you can pass the actual value of `x` into the `AssertionError` object, so that it gets displayed later.

```
assert x >= 0 : x;
```

C++ NOTE: The `assert` macro of the C language turns the assertion condition into a string that is printed if the assertion fails. For example, if `assert(x >= 0)` fails, it prints that `"x >= 0"` is the failing condition. In Java, the condition is not automatically part of the error report. If you want to see it, you have to pass it as a string into the `AssertionError` object: `assert x >= 0 : "x >= 0"`.

7.4.2 Assertion Enabling and Disabling

By default, assertions are disabled. Enable them by running the program with the `-enableassertions` or `-ea` option:

```
java -enableassertions MyApp
```

Note that you do not have to recompile your program to enable or disable assertions. Enabling or disabling assertions is a function of the *class loader*. When assertions are disabled, the class loader strips out the assertion code so that it won't slow execution.

You can even turn on assertions in specific classes or in entire packages. For example:

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

This command turns on assertions for the class `MyClass` and all classes in the `com.mycompany.mylib` package *and its subpackages*. The option `-ea...` turns on assertions in all classes of the default package.

You can also disable assertions in certain classes and packages with the `-disableassertions` or `-da` option:

```
java -ea:... -da:MyClass MyApp
```

Some classes are not loaded by a class loader but directly by the virtual machine. You can use these switches to selectively enable or disable assertions in those classes.

However, the `-ea` and `-da` switches that enable or disable all assertions do not apply to the "system classes" without class loaders. Use the `-enablesystemassertions/-esa` switch to enable assertions in system classes.

It is also possible to programmatically control the assertion status of class loaders. See the API notes at the end of this section.

7.4.3 Using Assertions for Parameter Checking

The Java language gives you three mechanisms to deal with system failures:

- Throwing an exception
- Logging
- Using assertions

When should you choose assertions? Keep these points in mind:

- Assertion failures are intended to be fatal, unrecoverable errors.
- Assertion checks are turned on only during development and testing. (This is sometimes jokingly described as "wearing a life jacket when you are close to shore, and throwing it overboard once you are in the middle of the ocean.")

Therefore, you would not use assertions for signaling recoverable conditions to another part of the program or for communicating problems to the program user. Assertions should only be used to locate internal program errors during testing.

Let's look at a common scenario—the checking of method parameters. Should you use assertions to check for illegal index values or `null` references? To answer that question, you have to look at the documentation of the method. Suppose you implement a sorting method.

```
/**
   Sorts the specified range of the specified array in ascending numerical order.
   The range to be sorted extends from fromIndex, inclusive, to toIndex, exclusive.
   @param a the array to be sorted.
   @param fromIndex the index of the first element (inclusive) to be sorted.
   @param toIndex the index of the last element (exclusive) to be sorted.
   @throws IllegalArgumentException if fromIndex > toIndex
   @throws ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length
*/
static void sort(int[] a, int fromIndex, int toIndex)
```

The documentation states that the method throws an exception if the index values are incorrect. That behavior is part of the contract that the method makes with its callers. If you implement the method, you have to respect that contract and throw the indicated exceptions. It would not be appropriate to use assertions instead.

Should you assert that `a` is not `null`? That is not appropriate either. The method documentation is silent on the behavior of the method when `a` is `null`. The callers have the right to assume that the method will return successfully in that case and not throw an assertion error.

However, suppose the method contract had been slightly different:

```
@param a the array to be sorted (must not be null).
```

Now the callers of the method have been put on notice that it is illegal to call the method with a `null` array. Then the method may start with the assertion

```
assert a != null;
```

Computer scientists call this kind of contract a *precondition*. The original method had no preconditions on its parameters—it promised a well-defined behavior in all cases. The revised method has a single precondition: that `a` is not `null`. If the caller fails to fulfill the precondition, then all bets are off and the method can do anything it wants. In fact, with the assertion in place, the method has a rather unpredictable behavior when it is called illegally. It sometimes throws an assertion error, and sometimes a null pointer exception, depending on how its class loader is configured.

7.4.4 Using Assertions for Documenting Assumptions

Many programmers use comments to document their underlying assumptions. Consider this example from `http://docs.oracle.com/javase/6/docs/technotes/guides/language/assert.html`:

```
if (i % 3 == 0)
   . . .
else if (i % 3 == 1)
   . . .
else // (i % 3 == 2)
   . . .
```

In this case, it makes a lot of sense to use an assertion instead.

```
if (i % 3 == 0)
   . . .
else if (i % 3 == 1)
   . . .
else
{
   assert i % 3 == 2;
   . . .
}
```

Of course, it would make even more sense to think through the issue thoroughly. What are the possible values of `i % 3`? If `i` is positive, the remainders must be `0`, `1`, or `2`. If `i` is negative, then the remainders can be `-1` or `-2`. Thus, the real assumption is that `i` is not negative. A better assertion would be

```
assert i >= 0;
```

before the `if` statement.

At any rate, this example shows a good use of assertions as a self-check for the programmer. As you can see, assertions are a tactical tool for testing and debugging. In contrast, logging is a strategic tool for the entire lifecycle of a program. We will examine logging in the next section.

`java.lang.ClassLoader` 1.0

- `void setDefaultAssertionStatus(boolean b)` 1.4

  enables or disables assertions for all classes loaded by this class loader that don't have an explicit class or package assertion status.
- `void setClassAssertionStatus(String className, boolean b)` 1.4

  enables or disables assertions for the given class and its inner classes.
- `void setPackageAssertionStatus(String packageName, boolean b)` 1.4

  enables or disables assertions for all classes in the given package and its subpackages.
- `void clearAssertionStatus()` 1.4

  removes all explicit class and package assertion status settings and disables assertions for all classes loaded by this class loader.

7.5 Logging

Every Java programmer is familiar with the process of inserting calls to `System.out.println` into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements, only to put them back in when the next problem surfaces. The logging API is designed to overcome this problem. Here are the principal advantages of the API:

- It is easy to suppress all log records or just those below a certain level, and just as easy to turn them back on.
- Suppressed logs are very cheap, so that there is only a minimal penalty for leaving the logging code in your application.
- Log records can be directed to different handlers—for displaying in the console, writing to a file, and so on.
- Both loggers and handlers can filter records. Filters can discard boring log entries, using any criteria supplied by the filter implementor.
- Log records can be formatted in different ways—for example, in plain text or XML.
- Applications can use multiple loggers, with hierarchical names such as `com.mycompany.myapp`, similar to package names.
- By default, the logging configuration is controlled by a configuration file. Applications can replace this mechanism if desired.

7.5.1 Basic Logging

For simple logging, use the global logger and call its `info` method:

```
Logger.getGlobal().info("File->Open menu item selected");
```

By default, the record is printed like this:

```
May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen
INFO: File->Open menu item selected
```

But if you call

```
Logger.getGlobal().setLevel(Level.OFF);
```

at an appropriate place (such as the beginning of `main`), all logging is suppressed.

7.5.2 Advanced Logging

Now that you have seen “logging for dummies,” let’s go on to industrial-strength logging. In a professional application, you wouldn’t want to log all records to a single global logger. Instead, you can define your own loggers.

Call the `getLogger` method to create or retrieve a logger:

```
private static final Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

TIP: A logger that is not referenced by any variable can be garbage collected. To prevent this, save a reference to the logger with a static variable, as in the example above.

Similar to package names, logger names are hierarchical. In fact, they are *more* hierarchical than packages. There is no semantic relationship between a package and its parent, but logger parents and children share certain properties. For example, if you set the log level on the logger `"com.mycompany"`, then the child loggers inherit that level.

There are seven logging levels:

- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

By default, the top three levels are actually logged. You can set a different level—for example,

```
logger.setLevel(Level.FINE);
```

Now `FINE` and all levels above it are logged.

You can also use `Level.ALL` to turn on logging for all levels or `Level.OFF` to turn all logging off.

There are logging methods for all levels, such as

```
logger.warning(message);
logger.fine(message);
```

and so on. Alternatively, you can use the `log` method and supply the level, such as

```
logger.log(Level.FINE, message);
```

TIP: The default logging configuration logs all records with the level of `INFO` or higher. Therefore, you should use the levels `CONFIG`, `FINE`, `FINER`, and `FINEST` for debugging messages that are useful for diagnostics but meaningless to the user.

CAUTION: If you set the logging level to a value finer than `INFO`, you also need to change the log handler configuration. The default log handler suppresses messages below `INFO`. See the next section for details.

The default log record shows the name of the class and method that contain the logging call, as inferred from the call stack. However, if the virtual machine optimizes execution, accurate call information may not be available. You can use the `logp` method to give the precise location of the calling class and method. The method signature is

```
void logp(Level l, String className, String methodName, String message)
```

There are convenience methods for tracing execution flow:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

For example:

```
int read(String file, String pattern)
{
   logger.entering("com.mycompany.mylib.Reader", "read",
      new Object[] { file, pattern });
   . . .
   logger.exiting("com.mycompany.mylib.Reader", "read", count);
   return count;
}
```

These calls generate log records of level `FINER` that start with the strings `ENTRY` and `RETURN`.

NOTE: At some point in the future, the logging methods with an `Object[]` parameter will be rewritten to support variable parameter lists ("varargs"). Then, you will be able to make calls such as `logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)`.

A common use for logging is to log unexpected exceptions. Two convenience methods include a description of the exception in the log record.

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Typical uses are

```
if (. . .)
{
   IOException exception = new IOException(". . .");
   logger.throwing("com.mycompany.mylib.Reader", "read", exception);
   throw exception;
}
```

and

```
try
{
   . . .
}
catch (IOException e)
{
   Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

The `throwing` call logs a record with level `FINER` and a message that starts with `THROW`.

7.5.3 Changing the Log Manager Configuration

You can change various properties of the logging system by editing a configuration file. The default configuration file is located at

```
jre/lib/logging.properties
```

To use another file, set the `java.util.logging.config.file` property to the file location by starting your application with

```
java -Djava.util.logging.config.file=configFile MainClass
```

CAUTION: The log manager is initialized during VM startup, before `main` executes. If you call `System.setProperty("java.util.logging.config.file", file)` in `main`, also call `LogManager.readConfiguration()` to reinitialize the log manager.

To change the default logging level, edit the configuration file and modify the line

```
.level=INFO
```

You can specify the logging levels for your own loggers by adding lines such as

```
com.mycompany.myapp.level=FINE
```

That is, append the `.level` suffix to the logger name.

As you will see later in this section, the loggers don't actually send the messages to the console—that is the job of the handlers. Handlers also have levels. To see `FINE` messages on the console, you also need to set

```
java.util.logging.ConsoleHandler.level=FINE
```

CAUTION: The settings in the log manager configuration are *not* system properties. Starting a program with `-Dcom.mycompany.myapp.level=FINE` does not have any effect on the logger.

CAUTION: Up to Java SE 7, the API documentation of the `LogManager` class claims that you can set the `java.util.logging.config.class` and `java.util.logging.config.file` properties via the Preferences API. This is false—see bug 4691587 in the Java bug database (`http://bugs.sun.com/bugdatabase`).

NOTE: The logging properties file is processed by the `java.util.logging.LogManager` class. It is possible to specify a different log manager by setting the `java.util.logging.manager` system property to the name of a subclass. Alternatively, you can keep the standard log manager and still bypass the initialization from the logging properties file. Set the `java.util.logging.config.class` system property to the name of a class that sets log manager properties in some other way. See the API documentation for the `LogManager` class for more information.

It is also possible to change logging levels in a running program by using the `jconsole` program. See `www.oracle.com/technetwork/articles/java/jconsole-1564139.html#LoggingControl` for information.

7.5.4 Localization

You may want to localize logging messages so that they are readable for international users. Internationalization of applications is the topic of Chapter 5 of Volume II. Briefly, here are the points to keep in mind when localizing logging messages.

Localized applications contain locale-specific information in *resource bundles*. A resource bundle consists of a set of mappings for various locales (such as United

States or Germany). For example, a resource bundle may map the string `"readingFile"` into strings `"Reading file"` in English or `"Achtung! Datei wird eingelesen"` in German.

A program may contain multiple resource bundles—for example, one for menus and another for log messages. Each resource bundle has a name (such as `"com.mycompany.logmessages"`). To add mappings to a resource bundle, supply a file for each locale. English message mappings are in a file `com/mycompany/logmessages_en.properties`, and German message mappings are in a file `com/mycompany/logmessages_de.properties`. (The `en` and `de` are the language codes.) You place the files together with the class files of your application, so that the `ResourceBundle` class will automatically locate them. These files are plain text files, consisting of entries such as

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
. . .
```

When requesting a logger, you can specify a resource bundle:

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

Then you specify the resource bundle key, not the actual message string, for the log message.

```
logger.info("readingFile");
```

You often need to include arguments into localized messages. A message may contain placeholders: `{0}`, `{1}`, and so on. For example, to include the file name with a log message, use the placeholder like this:

```
Reading file {0}.
Achtung! Datei {0} wird eingelesen.
```

Then, to pass values into the placeholders, call one of the following methods:

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

7.5.5 Handlers

By default, loggers send records to a `ConsoleHandler` that prints them to the `System.err` stream. Specifically, the logger sends the record to the parent handler, and the ultimate ancestor (with name `""`) has a `ConsoleHandler`.

Like loggers, handlers have a logging level. For a record to be logged, its logging level must be above the threshold of *both* the logger and the handler. The log manager configuration file sets the logging level of the default console handler as

```
java.util.logging.ConsoleHandler.level=INFO
```

To log records with level `FINE`, change both the default logger level and the handler level in the configuration. Alternatively, you can bypass the configuration file altogether and install your own handler.

```
Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);
```

By default, a logger sends records both to its own handlers and the handlers of the parent. Our logger is a child of the primordial logger (with name `""`) that sends all records with level `INFO` and above to the console. We don't want to see those records twice, however, so we set the `useParentHandlers` property to `false`.

To send log records elsewhere, add another handler. The logging API provides two useful handlers for this purpose: a `FileHandler` and a `SocketHandler`. The `SocketHandler` sends records to a specified host and port. Of greater interest is the `FileHandler` that collects records in a file.

You can simply send records to a default file handler, like this:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

The records are sent to a file `java`*n*`.log` in the user's home directory, where *n* is a number to make the file unique. If a user's system has no concept of the user's home directory (for example, in Windows 95/98/Me), then the file is stored in a default location such as `C:\Windows`. By default, the records are formatted in XML. A typical log record has the form

```
<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>
```

You can modify the default behavior of the file handler by setting various parameters in the log manager configuration (see Table 7.1) or by using another constructor (see the API notes at the end of this section).

You probably don't want to use the default log file name. Therefore, you should use another pattern, such as `%h/myapp.log`. (See Table 7.2 for an explanation of the pattern variables.)

Table 7.1 File Handler Configuration Parameters

Configuration Property	Description	Default
`java.util.logging.FileHandler.level`	The handler level	`Level.ALL`
`java.util.logging.FileHandler.append`	Controls whether the handler should append to an existing file, or open a new file for each program run	`false`
`java.util.logging.FileHandler.limit`	The approximate maximum number of bytes to write in a file before opening another (`0` = no limit)	`0` (no limit) in the `FileHandler` class, `50000` in the default log manager configuration
`java.util.logging.FileHandler.pattern`	The pattern for the log file name. See Table 7.2 for pattern variables.	`%h/java%u.log`
`java.util.logging.FileHandler.count`	The number of logs in a rotation sequence	`1` (no rotation)
`java.util.logging.FileHandler.filter`	The filter class to use	No filtering
`java.util.logging.FileHandler.encoding`	The character encoding to use	The platform encoding
`java.util.logging.FileHandler.formatter`	The record formatter	`java.util.logging.XMLFormatter`

Table 7.2 Log File Pattern Variables

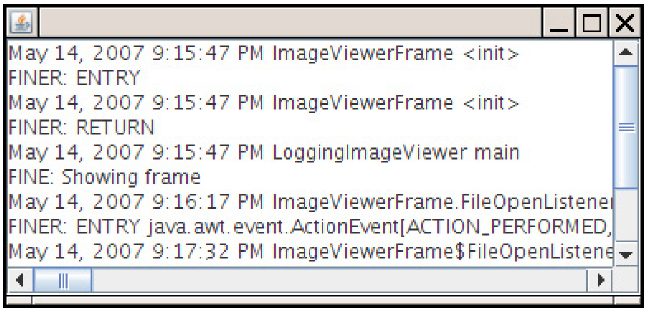
Variable	Description
`%h`	The value of the `user.home` system property
`%t`	The system temporary directory
`%u`	A unique number to resolve conflicts
`%g`	The generation number for rotated logs. (A `.%g` suffix is used if rotation is specified and the pattern doesn't contain `%g`.)
`%%`	The `%` character

If multiple applications (or multiple copies of the same application) use the same log file, you should turn the `append` flag on. Alternatively, use `%u` in the file name pattern so that each application creates a unique copy of the log.

It is also a good idea to turn file rotation on. Log files are kept in a rotation sequence, such as `myapp.log.0`, `myapp.log.1`, `myapp.log.2`, and so on. Whenever a file exceeds the size limit, the oldest log is deleted, the other files are renamed, and a new file with generation number `0` is created.

TIP: Many programmers use logging as an aid for the technical support staff. If a program misbehaves in the field, the user can send back the log files for inspection. In that case, you should turn the `append` flag on, use rotating logs, or both.

You can also define your own handlers by extending the `Handler` or the `StreamHandler` class. We define such a handler in the example program at the end of this section. That handler displays the records in a window (see Figure 7.2).



```
May 14, 2007 9:15:47 PM ImageViewerFrame <init>
FINER: ENTRY
May 14, 2007 9:15:47 PM ImageViewerFrame <init>
FINER: RETURN
May 14, 2007 9:15:47 PM LoggingImageViewer main
FINE: Showing frame
May 14, 2007 9:16:17 PM ImageViewerFrame.FileOpenListener
FINER: ENTRY java.awt.event.ActionEvent[ACTION_PERFORMED,
May 14, 2007 9:17:32 PM ImageViewerFrame$FileOpenListene
```

Figure 7.2 A log handler that displays records in a window

The handler extends the `StreamHandler` class and installs a stream whose `write` methods display the stream output in a text area.

```
class WindowHandler extends StreamHandler
{
   public WindowHandler()
   {
      . . .
      final JTextArea output = new JTextArea();
```

```
         setOutputStream(new
            OutputStream()
            {
               public void write(int b) {} // not called
               public void write(byte[] b, int off, int len)
               {
                  output.append(new String(b, off, len));
               }
            });
      }
      . . .
   }
```

There is just one problem with this approach—the handler buffers the records and only writes them to the stream when the buffer is full. Therefore, we override the `publish` method to flush the buffer after each record:

```
class WindowHandler extends StreamHandler
{
   . . .
   public void publish(LogRecord record)
   {
      super.publish(record);
      flush();
   }
}
```

If you want to write more exotic stream handlers, extend the `Handler` class and define the `publish`, `flush`, and `close` methods.

7.5.6 Filters

By default, records are filtered according to their logging levels. Each logger and handler can have an optional filter to perform additional filtering. To define a filter, implement the `Filter` interface and define the method

```
boolean isLoggable(LogRecord record)
```

Analyze the log record, using any criteria that you desire, and return `true` for those records that should be included in the log. For example, a particular filter may only be interested in the messages generated by the `entering` and `exiting` methods. The filter should then call `record.getMessage()` and check whether it starts with `ENTRY` or `RETURN`.

To install a filter into a logger or handler, simply call the `setFilter` method. Note that you can have at most one filter at a time.

7.5.7 Formatters

The ConsoleHandler and FileHandler classes emit the log records in text and XML formats. However, you can define your own formats as well. You need to extend the Formatter class and override the method

```
String format(LogRecord record)
```

Format the information in the record in any way you like and return the resulting string. In your format method, you may want to call the method

```
String formatMessage(LogRecord record)
```

That method formats the message part of the record, substituting parameters and applying localization.

Many file formats (such as XML) require a head and tail parts that surround the formatted records. To achieve this, override the methods

```
String getHead(Handler h)
String getTail(Handler h)
```

Finally, call the setFormatter method to install the formatter into the handler.

7.5.8 A Logging Recipe

With so many options for logging, it is easy to lose track of the fundamentals. The following recipe summarizes the most common operations.

1. For a simple application, choose a single logger. It is a good idea to give the logger the same name as your main application package, such as com.mycompany.myprog. You can always get the logger by calling

   ```
   Logger logger = Logger.getLogger("com.mycompany.myprog");
   ```

   For convenience, you may want to add static fields

   ```
   private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
   ```

   to classes with a lot of logging activity.

2. The default logging configuration logs all messages of level INFO or higher to the console. Users can override the default configuration, but as you have seen, the process is a bit involved. Therefore, it is a good idea to install a more reasonable default in your application.

   The following code ensures that all messages are logged to an application-specific file. Place the code into the main method of your application.

```
if (System.getProperty("java.util.logging.config.class") == null
      && System.getProperty("java.util.logging.config.file") == null)
{
   try
   {
      Logger.getLogger("").setLevel(Level.ALL);
      final int LOG_ROTATION_COUNT = 10;
      Handler handler = new FileHandler("%h/myapp.log", 0, LOG_ROTATION_COUNT);
      Logger.getLogger("").addHandler(handler);
   }
   catch (IOException e)
   {
      logger.log(Level.SEVERE, "Can't create log file handler", e);
   }
}
```

3. Now you are ready to log to your heart's content. Keep in mind that all messages with level `INFO`, `WARNING`, and `SEVERE` show up on the console. Therefore, reserve these levels for messages that are meaningful to the users of your program. The level `FINE` is a good choice for logging messages that are intended for programmers.

   Whenever you are tempted to call `System.out.println`, emit a log message instead:

   ```
   logger.fine("File open dialog canceled");
   ```

   It is also a good idea to log unexpected exceptions. For example:

   ```
   try
   {
      . . .
   }
   catch (SomeException e)
   {
      logger.log(Level.FINE, "explanation", e);
   }
   ```

   Listing 7.2 puts this recipe to use with an added twist: Logging messages are also displayed in a log window.

Listing 7.2 `logging/LoggingImageViewer.java`

```
package logging;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.logging.*;
import javax.swing.*;

```

```
/**
 * A modification of the image viewer program that logs various events.
 * @version 1.03 2015-08-20
 * @author Cay Horstmann
 */
public class LoggingImageViewer
{
   public static void main(String[] args)
   {
      if (System.getProperty("java.util.logging.config.class") == null
            && System.getProperty("java.util.logging.config.file") == null)
      {
         try
         {
            Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
            final int LOG_ROTATION_COUNT = 10;
            Handler handler = new FileHandler("%h/LoggingImageViewer.log", 0, LOG_ROTATION_COUNT);
            Logger.getLogger("com.horstmann.corejava").addHandler(handler);
         }
         catch (IOException e)
         {
            Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
                  "Can't create log file handler", e);
         }
      }

      EventQueue.invokeLater(() ->
            {
               Handler windowHandler = new WindowHandler();
               windowHandler.setLevel(Level.ALL);
               Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);

               JFrame frame = new ImageViewerFrame();
               frame.setTitle("LoggingImageViewer");
               frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

               Logger.getLogger("com.horstmann.corejava").fine("Showing frame");
               frame.setVisible(true);
            });
   }
}

/**
 * The frame that shows the image.
 */
class ImageViewerFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 300;
   private static final int DEFAULT_HEIGHT = 400;
```

(Continues)

Listing 7.2 *(Continued)*

```

   private JLabel label;
   private static Logger logger = Logger.getLogger("com.horstmann.corejava");

   public ImageViewerFrame()
   {
      logger.entering("ImageViewerFrame", "<init>");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // set up menu bar
      JMenuBar menuBar = new JMenuBar();
      setJMenuBar(menuBar);

      JMenu menu = new JMenu("File");
      menuBar.add(menu);

      JMenuItem openItem = new JMenuItem("Open");
      menu.add(openItem);
      openItem.addActionListener(new FileOpenListener());

      JMenuItem exitItem = new JMenuItem("Exit");
      menu.add(exitItem);
      exitItem.addActionListener(new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               logger.fine("Exiting.");
               System.exit(0);
            }
         });

      // use a label to display the images
      label = new JLabel();
      add(label);
      logger.exiting("ImageViewerFrame", "<init>");
   }

   private class FileOpenListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         logger.entering("ImageViewerFrame.FileOpenListener", "actionPerformed", event);

         // set up file chooser
         JFileChooser chooser = new JFileChooser();
         chooser.setCurrentDirectory(new File("."));

         // accept all files ending with .gif
         chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
            {
               public boolean accept(File f)
               {
                  return f.getName().toLowerCase().endsWith(".gif") || f.isDirectory();
               }

               public String getDescription()
               {
                  return "GIF Images";
               }
            });

         // show file chooser dialog
         int r = chooser.showOpenDialog(ImageViewerFrame.this);

         // if image file accepted, set it as icon of the label
         if (r == JFileChooser.APPROVE_OPTION)
         {
            String name = chooser.getSelectedFile().getPath();
            logger.log(Level.FINE, "Reading file {0}", name);
            label.setIcon(new ImageIcon(name));
         }
         else logger.fine("File open dialog canceled.");
         logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
      }
   }
}

/**
 * A handler for displaying log records in a window.
 */
class WindowHandler extends StreamHandler
{
   private JFrame frame;

   public WindowHandler()
   {
      frame = new JFrame();
      final JTextArea output = new JTextArea();
      output.setEditable(false);
      frame.setSize(200, 200);
      frame.add(new JScrollPane(output));
      frame.setFocusableWindowState(false);
      frame.setVisible(true);
```

(Continues)

Listing 7.2 *(Continued)*

```
      setOutputStream(new OutputStream()
         {
            public void write(int b)
            {
            } // not called

            public void write(byte[] b, int off, int len)
            {
               output.append(new String(b, off, len));
            }
         });
   }

   public void publish(LogRecord record)
   {
      if (!frame.isVisible()) return;
      super.publish(record);
      flush();
   }
}
```

`java.util.logging.Logger` 1.4

- `Logger getLogger(String loggerName)`
- `Logger getLogger(String loggerName, String bundleName)`

  gets the logger with the given name. If the logger doesn't exist, it is created.

Parameters:	`loggerName`	The hierarchical logger name, such as `com.mycompany.myapp`
	`bundleName`	The name of the resource bundle for looking up localized messages

- `void severe(String message)`
- `void warning(String message)`
- `void info(String message)`
- `void config(String message)`
- `void fine(String message)`
- `void finer(String message)`
- `void finest(String message)`

  logs a record with the level indicated by the method name and the given message.

(Continues)

`java.util.logging.Logger` 1.4 *(Continued)*

- `void entering(String className, String methodName)`
- `void entering(String className, String methodName, Object param)`
- `void entering(String className, String methodName, Object[] param)`
- `void exiting(String className, String methodName)`
- `void exiting(String className, String methodName, Object result)`

  logs a record that describes entering or exiting a method with the given parameter(s) or return value.
- `void throwing(String className, String methodName, Throwable t)`

  logs a record that describes throwing of the given exception object.
- `void log(Level level, String message)`
- `void log(Level level, String message, Object obj)`
- `void log(Level level, String message, Object[] objs)`
- `void log(Level level, String message, Throwable t)`

  logs a record with the given level and message, optionally including objects or a throwable. To include objects, the message must contain formatting placeholders (`{0}`, `{1}`, and so on).
- `void logp(Level level, String className, String methodName, String message)`
- `void logp(Level level, String className, String methodName, String message, Object obj)`
- `void logp(Level level, String className, String methodName, String message, Object[] objs)`
- `void logp(Level level, String className, String methodName, String message, Throwable t)`

  logs a record with the given level, precise caller information, and message, optionally including objects or a throwable.
- `void logrb(Level level, String className, String methodName, String bundleName, String message)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`

  logs a record with the given level, precise caller information, resource bundle name, and message, optionally including objects or a throwable.
- `Level getLevel()`
- `void setLevel(Level l)`

  gets and sets the level of this logger.

(Continues)

`java.util.logging.Logger` 1.4 *(Continued)*

- `Logger getParent()`
- `void setParent(Logger l)`

  gets and sets the parent logger of this logger.

- `Handler[] getHandlers()`

  gets all handlers of this logger.

- `void addHandler(Handler h)`
- `void removeHandler(Handler h)`

  adds or removes a handler for this logger.

- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`

  gets and sets the "use parent handler" property. If this property is `true`, the logger forwards all logged records to the handlers of its parent.

- `Filter getFilter()`
- `void setFilter(Filter f)`

  gets and sets the filter of this logger.

`java.util.logging.Handler` 1.4

- `abstract void publish(LogRecord record)`

  sends the record to the intended destination.

- `abstract void flush()`

  flushes any buffered data.

- `abstract void close()`

  flushes any buffered data and releases all associated resources.

- `Filter getFilter()`
- `void setFilter(Filter f)`

  gets and sets the filter of this handler.

- `Formatter getFormatter()`
- `void setFormatter(Formatter f)`

  gets and sets the formatter of this handler.

- `Level getLevel()`
- `void setLevel(Level l)`

  gets and sets the level of this handler.

`java.util.logging.ConsoleHandler` 1.4

- `ConsoleHandler()`

  constructs a new console handler.

`java.util.logging.FileHandler` 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`

  constructs a file handler.

Parameters:		
	`pattern`	The pattern for constructing the log file name. See Table 7.2 for pattern variables.
	`limit`	The approximate maximum number of bytes before a new log file is opened.
	`count`	The number of files in a rotation sequence.
	`append`	`true` if a newly constructed file handler object should append to an existing log file.

`java.util.logging.LogRecord` 1.4

- `Level getLevel()`

  gets the logging level of this record.
- `String getLoggerName()`

  gets the name of the logger that is logging this record.
- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`

  gets the resource bundle, or its name, to be used for localizing the message, or `null` if none is provided.
- `String getMessage()`

  gets the “raw” message before localization or formatting.
- `Object[] getParameters()`

  gets the parameter objects, or `null` if none is provided.

(Continues)

`java.util.logging.LogRecord` 1.4 *(Continued)*

- `Throwable getThrown()`

  gets the thrown object, or `null` if none is provided.
- `String getSourceClassName()`
- `String getSourceMethodName()`

  gets the location of the code that logged this record. This information may be supplied by the logging code or automatically inferred from the runtime stack. It might be inaccurate if the logging code supplied the wrong value or if the running code was optimized so that the exact location cannot be inferred.
- `long getMillis()`

  gets the creation time, in milliseconds, since 1970.
- `long getSequenceNumber()`

  gets the unique sequence number of this record.
- `int getThreadID()`

  gets the unique ID for the thread in which this record was created. These IDs are assigned by the `LogRecord` class and have no relationship to other thread IDs.

***`java.util.logging.Filter`* 1.4**

- `boolean isLoggable(LogRecord record)`

  returns `true` if the given log record should be logged.

`java.util.logging.Formatter` 1.4

- `abstract String format(LogRecord record)`

  returns the string that results from formatting the given log record.
- `String getHead(Handler h)`
- `String getTail(Handler h)`

  returns the strings that should appear at the head and tail of the document containing the log records. The `Formatter` superclass defines these methods to return the empty string; override them if necessary.
- `String formatMessage(LogRecord record)`

  returns the localized and formatted message part of the log record.

7.6 Debugging Tips

Suppose you wrote your program and made it bulletproof by catching and properly handling all exceptions. Then you run it, and it does not work right. Now what? (If you never have this problem, you can skip the remainder of this chapter.)

Of course, it is best if you have a convenient and powerful debugger. Debuggers are available as a part of professional development environments such as Eclipse and NetBeans. In this section, we offer you a number of tips that may be worth trying before you launch the debugger.

1. You can print or log the value of any variable with code like this:

   ```
   System.out.println("x=" + x);
   ```

   or

   ```
   Logger.getGlobal().info("x=" + x);
   ```

   If `x` is a number, it is converted to its string equivalent. If `x` is an object, Java calls its `toString` method. To get the state of the implicit parameter object, print the state of the `this` object.

   ```
   Logger.getGlobal().info("this=" + this);
   ```

   Most of the classes in the Java library are very conscientious about overriding the `toString` method to give you useful information about the class. This is a real boon for debugging. You should make the same effort in your classes.

2. One seemingly little-known but very useful trick is putting a separate `main` method in each class. Inside it, you can put a unit test stub that lets you test the class in isolation.

   ```
   public class MyClass
   {
      methods and fields
      . . .
      public static void main(String[] args)
      {
         test code
      }
   }
   ```

   Make a few objects, call all methods, and check that each of them does the right thing. You can leave all these `main` methods in place and launch the Java virtual machine separately on each of the files to run the tests. When you run an applet, none of these `main` methods are ever called. When you run an application, the Java virtual machine calls only the `main` method of the startup class.

3. If you liked the preceding tip, you should check out JUnit from `http://junit.org`. JUnit is a very popular unit testing framework that makes it easy to organize suites of test cases. Run the tests whenever you make changes to a class, and add another test case whenever you find a bug.
4. A *logging proxy* is an object of a subclass that intercepts method calls, logs them, and then calls the superclass. For example, if you have trouble with the `nextDouble` method of the `Random` class, you can create a proxy object as an instance of an anonymous subclass:

   ```
   Random generator = new
      Random()
      {
         public double nextDouble()
         {
            double result = super.nextDouble();
            Logger.getGlobal().info("nextDouble: " + result);
            return result;
         }
      };
   ```

   Whenever the `nextDouble` method is called, a log message is generated.

   To find out who called the method, generate a stack trace.
5. You can get a stack trace from any exception object with the `printStackTrace` method in the `Throwable` class. The following code catches any exception, prints the exception object and the stack trace, and rethrows the exception so it can find its intended handler.

   ```
   try
   {
      . . .
   }
   catch (Throwable t)
   {
      t.printStackTrace();
      throw t;
   }
   ```

   You don't even need to catch an exception to generate a stack trace. Simply insert the statement

   ```
   Thread.dumpStack();
   ```

   anywhere into your code to get a stack trace.

6. Normally, the stack trace is displayed on `System.err`. If you want to log or display the stack trace, here is how you can capture it into a string:

```
StringWriter out = new StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String description = out.toString();
```

7. It is often handy to trap program errors in a file. However, errors are sent to `System.err`, not `System.out`. Therefore, you cannot simply trap them by running

```
java MyProgram > errors.txt
```

   Instead, capture the error stream as

```
java MyProgram 2> errors.txt
```

   To capture both `System.err` and `System.out` in the same file, use

```
java MyProgram 1> errors.txt 2>&1
```

   This works in `bash` and the Windows shell.

8. Having the stack traces of uncaught exceptions show up in `System.err` is not ideal. These messages are confusing to end users if they happen to see them, and they are not available for diagnostic purposes when you need them. A better approach is to log them to a file. You can change the handler for uncaught exceptions with the static `Thread.setDefaultUncaughtExceptionHandler` method:

```
Thread.setDefaultUncaughtExceptionHandler(
      new Thread.UncaughtExceptionHandler()
      {
         public void uncaughtException(Thread t, Throwable e)
         {
            save information in log file
         };
      });
```

9. To watch class loading, launch the Java virtual machine with the `-verbose` flag. You will get a printout such as the following:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
```

```
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
. . .
```

This can occasionally be helpful to diagnose class path problems.

10. The `-Xlint` option tells the compiler to spot common code problems. For example, if you compile with the command

    ```
    javac -Xlint:fallthrough
    ```

    the compiler will report missing `break` statements in `switch` statements. (The term "lint" originally described a tool for locating potential problems in C programs, but is now generically applied to any tools that flag constructs that are questionable but not illegal.)

    The following options are available:

`-Xlint` or `-Xlint:all`	Carries out all checks
`-Xlint:deprecation`	Same as `-deprecation`, checks for deprecated methods
`-Xlint:fallthrough`	Checks for missing `break` statements in `switch` statements
`-Xlint:finally`	Warns about `finally` clauses that cannot complete normally
`-Xlint:none`	Carries out none of the checks
`-Xlint:path`	Checks that all directories on the class path and source path exist
`-Xlint:serial`	Warns about serializable classes without `serialVersionUID` (see Chapter 1 of Volume II)
`-Xlint:unchecked`	Warns of unsafe conversions between generic and raw types (see Chapter 8)

11. The Java VM has support for *monitoring and management* of Java applications, allowing the installation of agents in the virtual machine that track memory consumption, thread usage, class loading, and so on. This feature is particularly important for large and long-running Java programs, such as application servers. As a demonstration of these capabilities, the JDK ships with a graphical tool called `jconsole` that displays statistics about the performance of a virtual machine (see Figure 7.3). Find out the ID of the operating system process that runs the virtual machine. In UNIX/Linux, run the `ps` utility; in Windows, use the task manager. Then launch the `jconsole` program:

    ```
    jconsole processID
    ```

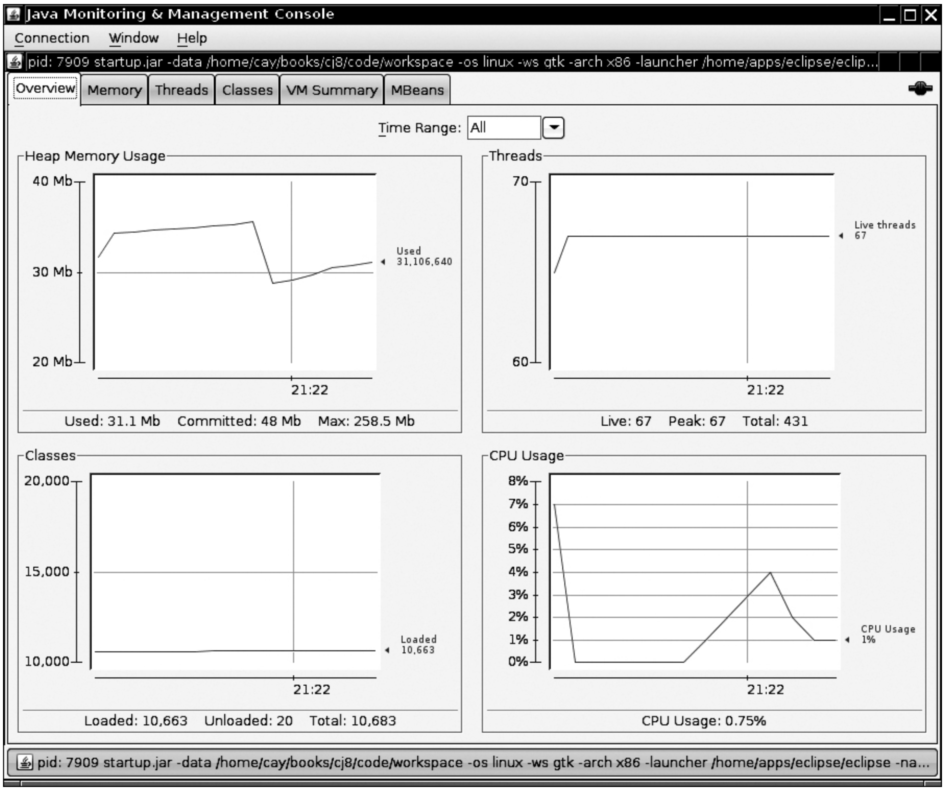

Java Monitoring & Management Console
Connection Window Help
pid: 7909 startup.jar -data /home/cay/books/cj8/code/workspace -os linux -ws gtk -arch x86 -launcher /home/apps/eclipse/eclip...
Overview Memory Threads Classes VM Summary MBeans
Time Range: All
Heap Memory Usage
40 Mb
30 Mb
20 Mb
Used 31,106,640
21:22
Used: 31.1 Mb Committed: 48 Mb Max: 258.5 Mb
Threads
70
60
Live threads 67
21:22
Live: 67 Peak: 67 Total: 431
Classes
20,000
15,000
10,000
Loaded 10,663
21:22
Loaded: 10,663 Unloaded: 20 Total: 10,683
CPU Usage
8%
7%
6%
5%
4%
3%
2%
1%
0%
CPU Usage 1%
21:22
CPU Usage: 0.75%
pid: 7909 startup.jar -data /home/cay/books/cj8/code/workspace -os linux -ws gtk -arch x86 -launcher /home/apps/eclipse/eclipse -na...

Figure 7.3 The jconsole program

The console gives you a wealth of information about your running program. See `www.oracle.com/technetwork/articles/java/jconsole-1564139.html` for more information.

12. You can use the `jmap` utility to get a heap dump that shows you every object on the heap. Use these commands:

    ```
    jmap -dump:format=b,file=dumpFileName processID
    jhat dumpFileName
    ```

    Then, point your browser to `localhost:7000`. You will get a web application that lets you drill down into the contents of the heap at the time of the dump.

13. If you launch the Java virtual machine with the `-Xprof` flag, it runs a rudimentary *profiler* that keeps track of the methods in your code that were executed most often. The profiling information is sent to `System.out`. The output also tells you which methods were compiled by the just-in-time compiler.

CAUTION: The `-X` options of the compiler are not officially supported and may not be present in all versions of the JDK. Run `java -X` to get a listing of all nonstandard options.

This chapter introduced you to exception handling and logging. You also saw useful hints for testing and debugging. The next two chapters cover generic programming and its most important application: the Java collections framework.