DAY 7

Threads, Exceptions, and Assertions

Today, you complete your weeklong journey through the Java language by learning about three of its most useful elements: threads, exceptions, and assertions.

Threads are objects that implement the `Runnable` interface, which indicates that they can run simultaneously with other parts of a Java program. *Exceptions* are objects that represent errors that may occur in a Java program. *Assertions* are conditional statements and Boolean values that indicate a program is running correctly, providing another means of detecting errors.

Threads enable your programs to make more efficient use of resources by isolating the computing-intensive parts of a program so that they don't slow down the rest of the program. Exceptions and assertions enable your programs to recognize errors and respond to them. Exceptions even assist your programs to correct the conditions if possible.

You start with exceptions because they're one of the things that you use when working with both assertions and threads.

Exceptions

Programmers in any language endeavor to write bug-free programs, programs that never crash, programs that can handle any circumstance with grace and recover from unusual situations without causing a user any undue stress. Good intentions aside, programs like this don't exist.

In real programs, errors occur because programmers didn't anticipate possible problems, didn't test enough, or encountered situations out of their control—bad data from users, corrupt files that don't have the right data in them, network connections that don't connect, hardware devices that don't respond, sun spots, gremlins, and so on.

In Java, the strange events that might cause a program to fail are called exceptions. Java defines a number of language features that deal with exceptions:

- How to handle exceptions in your code and recover gracefully from potential problems
- How to tell Java and users of your classes that you're expecting a potential exception
- How to create an exception if you detect one
- How your code is limited, yet made more robust by exceptions

With most programming languages, handling error conditions requires much more work than handling a program that is running properly. It can require a confusing structure of conditional statements to deal with errors that might occur.

As an example, consider the following statements that could be used to load a file from disk. File input and output can be problematic because of a number of different circumstances such as disk errors, file-not-found errors, and the like. If the program must have the data from the file to operate properly, it must deal with all these circumstances before continuing.

Here's the structure of one possible solution:

```
int status = loadTextFile();
if (status != 1) {
    // something unusual happened, describe it
    switch (status) {
        case 2:
            System.out.println("File not found");
            break;
        case 3:
            System.out.println("Disk error");
            break;
        case 4:
            System.out.println("File corrupted");
            break;
        default:
            System.out.println("Error");
    }
} else {
    // file loaded OK, continue with program
}
```

This code tries to load a file with a method call to `loadTextFile()`, which presumably has been defined elsewhere in the class. The method returns an integer that indicates whether the file loaded properly (a value of 1) or an error occurred (anything other than 1).

Depending on the error that occurs, the program uses a `switch` statement to address it. The end result is an elaborate block of code in which the most common circumstance—a successful file load—can be lost amid the error-handling code. This is the result of handling only one possible error. If other errors take place later in the program, you might end up with more nested `if-else` and `switch-case` blocks.

As you can see, error management would become a major problem after you start creating larger programs. Different programmers designate special values for handling errors, and they might not document them well if at all.

Code to manage these kinds of errors can often obscure the program's original intent, making the class difficult to read and maintain.

Finally, if you try to deal with errors in this manner, there's no easy way for the compiler to check for consistency the way it can check to make sure that you called a method with the right arguments.

Although the previous example uses Java syntax, you don't ever have to deal with errors that way with the Java language. There's a better technique to deal with exceptional circumstances in a program: the use of a group of classes called exceptions.

Exceptions include errors that could be fatal to your program as well as other unusual situations. By managing exceptions, you can manage errors and possibly work around them.

Errors and other conditions in Java programs can be much more easily managed through a combination of special language features, consistency checking at compile time, and a set of extensible exception classes.

With these features, you can add a whole new dimension to the behavior and design of your classes, your class hierarchy, and your overall system. Your class and interface definitions describe how your program is supposed to behave given the best circumstances.

By integrating exception handling into your program design, you can consistently describe how the program will behave when circumstances are not ideal, and allow people who use your classes to know what to expect in those cases.

Exception Classes

At this point, it's likely that you've run into at least one Java exception—perhaps you mistyped a method name or made a mistake in your code that caused a problem. Maybe you tried to run a Java application without providing the command-line arguments that were needed and saw an `ArrayIndexOutOfBoundsException` message.

Chances are, when an exception occurred, the application quit and spewed a bunch of mysterious errors to the screen. Those errors are exceptions. When your program stops without successfully finishing its work, an exception is thrown. Exceptions can be thrown by the virtual machine, thrown by classes you use, or intentionally thrown in your own programs.

The term "thrown" is fitting because exceptions also can be "caught." Catching an exception involves dealing with the exceptional circumstance so that your program doesn't crash—you learn more about this later today.

The heart of the Java exception system is the exception itself. Exceptions in Java are instances of classes that inherit from the `Throwable` class. An instance of a `Throwable` class is created when an exception is thrown.

`Throwable` has two subclasses: `Error` and `Exception`. Instances of `Error` are internal errors involving the Java virtual machine (the runtime environment). These errors are rare and usually fatal to the program; there's not much that you can do about them (either to catch them or to throw them yourself).

The class `Exception` is more relevant to your own programming. Subclasses of `Exception` fall into two general groups:

- Runtime exceptions (subclasses of the class `RuntimeException`) such as `ArrayIndexOutofBoundsException`, `SecurityException`, and `NullPointerException`
- Other exceptions such as `EOFException` and `MalformedURLException`

Runtime exceptions usually occur because of code that isn't very robust. An `ArrayIndexOutofBounds` exception, for example, should never be thrown if you're properly checking to make sure that your code stays within the bounds of an array. `NullPointerException` exceptions happen when you try to use a variable that doesn't refer to an object yet.

CAUTION

If your program is causing runtime exceptions under any circumstances, you should fix those problems before you even begin dealing with exception management.

The final group of exceptions indicate something strange and out of control is happening. An `EOFException`, for example, happens when you're reading from a file and the file ends before you expected it to end. A `MalformedURLException` happens when a URL isn't in the right format (perhaps a user typed it incorrectly). This group includes exceptions that you create to signal unusual cases that might occur in your own programs.

Exceptions are arranged in a hierarchy just as other classes are, where the `Exception` superclasses are more general errors and the subclasses are more specific errors. This organization becomes more important to you as you deal with exceptions in your own code.

The primary exception classes are part of the `java.lang` package (including `Throwable`, `Exception`, and `RuntimeException`). Many of the other packages define other exceptions, and those exceptions are used throughout the class library. For example, the `java.io` package defines a general exception class called `IOException`, which is subclassed not only in the `java.io` package for input and output exceptions (`EOFException` and `FileNotFoundException`) but also in the `java.net` classes for networking exceptions such as `MalformedURLException`.

Managing Exceptions

Now that you know what an exception is, how do you deal with one in your own code? In many cases, the Java compiler enforces exception management when you try to use methods that throw exceptions; you need to deal with those exceptions in your own code, or it simply won't compile. In this section, you learn about consistency checking and how to use three new keywords—`try`, `catch`, and `finally`—to deal with exceptions that might occur.

Exception Consistency Checking

The more you work with the Java class libraries, the more likely you'll run into a compiler error (an exception!) such as this one:

```
XMLParser.java:32: Exception java.lang.InterruptedException
must be caught or it must be declared in the throws clause
of this method.
```

In Java, a method can indicate the kinds of errors it might potentially throw. For example, methods that read from files can throw `IOException` errors, so those methods are declared with a special modifier that indicates potential errors. When you use those methods in your own Java programs, you have to protect your code against those exceptions. This rule is enforced by the compiler itself, in the same way that it checks to make sure that you're using methods with the correct number of arguments and that all your variable types match what you're assigning to them.

Why is this check in place? It makes your programs less likely to crash with fatal errors because you know upfront the kind of exceptions that can be thrown by the methods a program uses.

You no longer have to pore over the documentation or the code of an object you're going to use to ensure that you've dealt with all the potential problems—Java does the checking for you. On the other side, if you define your methods so that they indicate the exceptions they can throw, Java can tell your objects' users to handle those errors.

Protecting Code and Catching Exceptions

Assume that you've been happily coding and the compiler screeches to a halt with an exception as a class is compiled. According to the message, you have to either catch the error or declare that your method throws it.

First, we'll deal with catching potential exceptions, which requires two things:

- You protect the code that contains the method that might throw an exception inside a `try` block.
- You deal with an exception inside a `catch` block.

What `try` and `catch` effectively mean is, "Try this bit of code that might cause an exception. If it executes okay, go on with the program. If the code doesn't execute, catch the exception and deal with it."

You've seen `try` and `catch` before. On Day 6, "Packages, Interfaces, and Other Class Features," you used code when using a `String` value to create a floating-point number:

```
try {
    float in = Float.parseFloat(input);
} catch (NumberFormatException nfe) {
    System.out.println(input + " is not a valid number.");
}
```

Here's what's happening in these statements: The `Float.parseFloat()` class method could potentially throw an exception of the class `NumberFormatException`, which signifies that the thread has been interrupted for some reason.

To handle this exception, the call to `parseFloat()` is placed inside a `try` block and an associated `catch` block has been set up. This `catch` block receives any `NumberFormatException` objects thrown within the `try` block.

The part of the `catch` clause inside the parentheses is similar to a method definition's argument list. It contains the class of exception to be caught and a variable name. You can use the variable to refer to that exception object inside the `catch` block.

One common use for this object is to call its `getMessage()` method. This method is present in all exceptions, and it displays a detailed error message describing what happened.

Another useful method is `printStackTrace()`, which displays the sequence of method calls that led to the statement that generated the exception.

The following example is a revised version of the `try-catch` block used on Day 6:

```
try {
    float in = Float.parseFloat(input);
} catch (NumberFormatException nfe) {
    System.out.println("Oops: " + nfe.getMessage());
}
```

The examples you have seen thus far catch a specific type of exception. Because exception classes are organized into a hierarchy and you can use a subclass anywhere that a superclass is expected, you can catch groups of exceptions within the same `catch` statement.

As an example, when you start writing programs that handle input and output from files, Internet servers, and other places, you deal with several different types of `IOException` exceptions (the *IO* stands for *input/output*). These exceptions include two of its subclasses, `EOFException` and `FileNotFoundException`. By catching `IOException`, you also catch instances of any `IOException` subclass.

To catch several different exceptions that aren't related by inheritance, you can use multiple `catch` blocks for a single `try`, like this:

```
try {
    // code that might generate exceptions
} catch (IOException ioe) {
    System.out.println("Input/output error");
    System.out.println(ioe.getMessage());
} catch (ClassNotFoundException cnfe) {
    System.out.println("Class not found");
    System.out.println(cnfe.getMessage());
} catch (InterruptedException ie) {
    System.out.println("Program interrupted");
    System.out.println(ie.getMessage());
}
```

In a multiple `catch` block, the first `catch` block that matches is executed and the rest ignored.

CAUTION

You can run into unexpected problems by using an `Exception` superclass in a `catch` block followed by one or more of its subclasses in their own `catch` blocks. For example, the input/output exception `IOException` is the superclass of the end-of-file exception `EOFException`. If you put an `IOException` block above an `EOFException` block, the subclass never catches any exceptions.

The `finally` Clause

Suppose that there is some action in your code that you absolutely must do, no matter what happens, whether an exception is thrown or not. This is usually to free some external resource after acquiring it, to close a file after opening it, or something similar.

Although you could put that action both inside a `catch` block and outside it, that would be duplicating the same code in two different places, which is a situation you should avoid as much as possible in your programming.

Instead, put one copy of that code inside a special optional block of the `try-catch` statement that uses the keyword `finally`:

```
try {
    readTextFile();
} catch (IOException ioe) {
    // deal with IO errors
} finally {
   closeTextFile();
}
```

Today's first project shows how a `finally` statement can be used inside a method.

The `HexRead` application in Listing 7.1 reads sequences of two-digit hexadecimal numbers and displays their decimal values. There are three sequences to read:

- `000A110D1D260219`
- `78700F1318141E0C`
- `6A197D45B0FFFFFF`

As you learned on Day 2, “The ABCs of Programming,” hexadecimal is a base-16 numbering system where the single-digit numbers range from `00` (decimal 0) to `0F` (decimal 15), and double-digit numbers range from `10` (decimal 16) to `FF` (decimal 255).

LISTING 7.1 The Full Text of `HexRead.java`

```
 1: class HexRead {
 2:     String[] input = { "000A110D1D260219 ",
 3:         "78700F1318141E0C ",
 4:         "6A197D45B0FFFFFF " };
 5:
 6:     public static void main(String[] arguments) {
 7:         HexRead hex = new HexRead();
 8:         for (int i = 0; i < hex.input.length; i++)
 9:             hex.readLine(hex.input[i]);
10:     }
11:
12:     void readLine(String code) {
13:         try {
14:             for (int j = 0; j + 1 < code.length(); j += 2) {
15:                 String sub = code.substring(j, j+2);
16:                 int num = Integer.parseInt(sub, 16);
17:                 if (num == 255)
18:                     return;
19:                 System.out.print(num + " ");
20:             }
21:         } finally {
22:             System.out.println("**");
23:         }
24:         return;
25:     }
26: }
```

The output of this program is as follows:

```
0 10 17 13 29 38 2 25 **
120 112 15 19 24 20 30 12 **
106 25 125 69 176 **
```

Line 15 of the program reads two characters from `code`, the string that was sent to the `readLine()` method, by calling the string’s `substring(`*`int`*`, `*`int`*`)` method.

NOTE

In the substring() method of the String class, you select a substring in a somewhat counterintuitive way. The first argument specifies the index of the first character to include in the substring, but the second argument does not specify the last character. Instead, the second argument indicates the index of the last character plus 1. A call to substring(2, 5) for a string would return the characters from index position 2 to index position 4.

The two-character substring contains a hexadecimal number stored as a String. The Integer class method parseInt can be used with a second argument to convert this number into an integer. Use 16 as the argument for a hexadecimal (base 16) conversion, 8 for an octal (base 8) conversion, and so on.

In the HexRead application, the hexadecimal FF is used to fill out the end of a sequence and should not be displayed as a decimal value. This is accomplished by using a try-finally block in lines 13–23 of Listing 7.1.

The try-finally block causes an unusual thing to happen when the return statement is encountered at line 18. You would expect return to cause the readLine() method to be exited immediately.

Because it is within a try-finally block, the statement within the finally block is executed no matter how the try block is exited. The text "**" is displayed at the end of a line of decimal values.

NOTE

The finally statement is useful outside exceptions—it can execute cleanup code after a return, break, or continue statement inside loops. For the latter cases, you use a try statement with a finally but without a catch statement.

Declaring Methods That Might Throw Exceptions

In previous examples, you learned how to deal with methods that might throw exceptions by protecting code and catching any exceptions that occur. The Java compiler checks to make sure that you've dealt with a method's exceptions—but how did it know which exceptions to tell you about in the first place?

The answer is that the original method indicated the exceptions that it might possibly throw as part of its definition. You can use this mechanism in your own methods—in fact, it's good style to do so to make sure that users of your classes are alerted to the errors your methods may experience.

To indicate that a method may possibly throw an exception, you use a special clause in the method definition called `throws`.

The `throws` Clause

If some code in your method's body may throw an exception, add the `throws` keyword after the closing parenthesis of the method followed by the name or names of the exception that your method throws, as in this example:

```
public boolean getFormula(int x, int y) throws NumberFormatException {
    // body of method
}
```

If your method may throw multiple kinds of exceptions, you can declare them all in the `throws` clause separated by commas:

```
public boolean storeFormula(int x, int y)
    throws NumberFormatException, EOFException {
        // body of method
}
```

Note that as with `catch`, you can use a superclass of a group of exception to indicate that your method may throw any subclass of that exception. For instance:

```
public void loadFormula() throws IOException {
    // ...
}
```

Keep in mind that adding a `throws` method to your method definition simply means that the method might throw an exception if something goes wrong, not that it actually will. The `throws` clause provides extra information to your method definition about potential exceptions and allows Java to make sure that your method is being used correctly by other people.

Think of a method's overall description as a contract between the designer of that method and the caller of the method. (You can be on either side of that contract, of course.)

Usually the description indicates the types of a method's arguments, what it returns, and the particulars of what it normally does. By using `throws`, you are adding information about the abnormal things the method can do. This new part of the contract helps separate and make explicit all the places where exceptional conditions should be handled in your program, and that makes large-scale design easier.

Which Exceptions Should You Throw?

After you decide to declare that your method might throw an exception, you must decide which exceptions it might throw and actually throw them or call a method that will throw them (you learn about throwing your own exceptions in the next section).

In many instances, this is apparent from the operation of the method itself. Perhaps you're already creating and throwing your own exceptions, in which case, you'll know exactly which exceptions to throw.

You don't really have to list all the possible exceptions that your method could throw; some exceptions are handled by the runtime itself and are so common that you don't have to deal with them.

In particular, exceptions of either the `Error` or `RuntimeException` class or any of their subclasses do not have to be listed in your `throws` clause.

They get special treatment because they can occur anywhere within a Java program and are usually conditions that you, as the programmer, did not directly cause.

One good example is `OutOfMemoryError`, which can happen anywhere, at any time, and for any number of reasons. These two types of exceptions are called *unchecked exceptions*.

Unchecked exceptions are subclasses of the `RuntimeException` and `Error` classes and are usually thrown by the Java runtime itself. You do not have to declare that your method throws them and usually need not deal with them in any other way.

NOTE

You can, of course, choose to list these errors and runtime exceptions in your `throws` clause if you want, but your method's callers will not be forced to handle them; only non-runtime exceptions must be handled.

All other exceptions are called *checked exceptions* and are potential candidates for a `throws` clause in your method.

Passing On Exceptions

There are times when it doesn't make sense for your method to deal with an exception. It might be better for the method that calls your method to deal with that exception. There's nothing wrong with this; it's a fairly common occurrence that you pass an exception back to the method that calls your method.

For example, consider the hypothetical example of `WebRetriever`, a class that loads a Web page using its Web address and stores it in a file. As you learn on Day 17, "Communicating Across the Internet," you can't work with Web addresses without dealing with `MalformedURLException`, the exception thrown when an address isn't in the right format.

To use `WebRetriever`, another class calls its constructor method with the address as an argument. If the address specified by the other class isn't in the right format, a `MalformedURLException` is thrown. Instead of dealing with this, the constructor of the `WebRetriever` class could have the following definition:

```
public WebRetriever() throws MalformedURLException {
    // ...
}
```

This would force any class that works with `WebRetriever` objects to deal with `MalformedURLException` errors (or pass the buck with their own `throws` clause, of course).

One thing is true at all times: It's better to pass on exceptions to calling methods than to catch them and do nothing in response.

In addition to declaring methods that throw exceptions, there's one other instance in which your method definition may include a `throws` clause: Within that method, you want to call a method that throws an exception, but you don't want to catch or deal with that exception.

Rather than using the `try` and `catch` clauses in your method's body, you can declare your method with a `throws` clause so that it, too, might possibly throw the appropriate exception. It's then the responsibility of the method that calls your method to deal with that exception. This is the other case that tells the Java compiler that you have done something with a given exception.

Using this technique, you could create a method that deals with number format exceptions without a `try-catch` block:

```
public void readFloat(String input) throws NumberFormatException {
    float in = Float.parseFloat(input);
}
```

After you declare your method to throw an exception, you can use other methods that also throw those exceptions inside the body of this method, without needing to protect the code or catch the exception.

NOTE

You can, of course, deal with other exceptions using `try` and `catch` in the body of your method in addition to passing on the exceptions you listed in the `throws` clause. You also can both deal with the exception in some way and then rethrow it so that your method's calling method has to deal with it anyhow. You learn how to throw methods in the next section.

`throws` and Inheritance

If your method definition overrides a method in a superclass that includes a `throws` clause, there are special rules for how your overridden method deals with `throws`. Unlike other parts of the method signature that must mimic those of the method it is overriding, your new method does not require the same set of exceptions listed in the `throws` clause.

Because there's a possibility that your new method might deal better with exceptions instead of just throwing them, your method can potentially throw fewer types of exceptions. It could even throw no exceptions at all. That means that you can have the following two class definitions and things will work just fine:

```
public class RadioPlayer {
    public void startPlaying() throws SoundException {
        // body of method
    }
}

public class StereoPlayer extends RadioPlayer {
    public void startPlaying() {
        // body of method
    }
}
```

The converse of this rule is not true: A subclass method cannot throw more exceptions (either exceptions of different types or more general exception classes) than its superclass method.

Creating and Throwing Your Own Exceptions

There are two sides to every exception: the side that throws the exception and the side that catches it. An exception can be tossed around a number of times to a number of methods before it's caught, but eventually it will be caught and dealt with.

Who does the actual throwing? Where do exceptions come from? Many exceptions are thrown by the Java runtime or by methods inside the Java classes themselves. You also can throw any of the standard exceptions that the Java class libraries define, or you can create and throw your own exceptions.

Throwing Exceptions

Declaring that your method throws an exception is useful only to your method's users and to the Java compiler, which checks to make sure that all your exceptions are being handled. The declaration itself doesn't do anything to actually throw that exception should it occur; you must do that yourself as needed in the body of the method.

You need to create a new instance of an exception class to throw an exception. After you have that instance, use the `throw` statement to throw it.

Here's an example using a hypothetical `NotInServiceException` class that is a subclass of the `Exception` class:

```
NotInServiceException nise = new NotInServiceException();
throw nise;
```

You only can throw objects that implement the `Throwable` interface.

Depending on the exception class you're using, the exception also may have arguments to its constructor that you can use. The most common of these is a string argument, which enables you to describe the problem in greater detail (which can be useful for debugging purposes). Here's an example:

```
NotInServiceException nise = new
    NotInServiceException("Exception: Database Not in Service");
throw nise;
```

After an exception is thrown, the method exits immediately without executing any other code, other than the code inside a `finally` block if one exists. The method won't return a value either. If the calling method does not have a `try` or `catch` surrounding the call to your method, the program might exit based on the exception you threw.

Creating Your Own Exceptions

Although there are a fair number of exceptions in the Java class library that you can use in your own methods, you might need to create your own exceptions to handle the different kinds of errors that your programs run into. Creating new exceptions is easy.

Your new exception should inherit from some other exception in the Java hierarchy. All user-created exceptions should be part of the `Exception` hierarchy rather than the `Error` hierarchy, which is reserved for errors involving the Java virtual machine. Look for an exception that's close to the one you're creating; for example, an exception for a bad file format would logically be an `IOException`. If you can't find a closely related exception for your new exception, consider inheriting from `Exception`, which forms the “top” of the exception hierarchy for checked exceptions (unchecked exceptions should inherit from `RuntimeException`).

Exception classes typically have two constructors: The first takes no arguments, and the second takes a single string as an argument.

Exception classes are like other classes. You can put them in their own source files and compile them just as you would other classes:

```
public class SunSpotException extends Exception {
    public SunSpotException() {}
    public SunSpotException(String msg) {
        super(msg);
    }
}
```

Combining `throws`, `try`, and `throw`

What if you want to combine all the approaches shown so far? You want to handle incoming exceptions yourself in your method, but also you want the option to pass the exception on to your method’s caller. Simply using `try` and `catch` doesn’t pass on the exception, and adding a `throws` clause doesn’t give you a chance to deal with the exception.

If you want to both manage the exception and pass it on to the caller, use all three mechanisms: the `throws` clause, the `try` statement, and a `throw` statement to explicitly rethrow the exception.

Here’s a method that uses this technique:

```
public void readMessage() throws IOException {
    MessageReader mr = new MessageReader();

    try {
        mr.loadHeader();
    } catch (IOException e) {
        // do something to handle the
        // IO exception and then rethrow
        // the exception ...
        throw e;
    }
}
```

This works because exception handlers can be nested. You handle the exception by doing something responsible with it but decide that it is important enough to give the method’s caller a chance to handle it as well.

Exceptions can float all the way up the chain of method callers this way (usually not being handled by most of them) until, at last, the system itself handles any uncaught exceptions by aborting your program and printing an error message.

If it's possible for you to catch an exception and do something intelligent with it, you should.

When and When Not to Use Exceptions

Because throwing, catching, and declaring exceptions are related concepts and can be confusing, here's a quick summary of when to do what.

When to Use Exceptions

You can do one of three things if your method calls another method that has a `throws` clause:

- Deal with the exception by using `try` and `catch` statements
- Pass the exception up the calling chain by adding your own `throws` clause to your method definition
- Perform both of the preceding methods by catching the exception using `catch` and then explicitly rethrowing it using `throw`

In cases where a method throws more than one exception, you can handle each of those exceptions differently. For example, you might catch some of those exceptions while allowing others to pass up the calling chain.

If your method throws its own exceptions, you should declare that it throws those methods using the `throws` statement. If your method overrides a superclass method that has a `throws` statement, you can throw the same types of exceptions or subclasses of those exceptions; you cannot throw any different types of exceptions.

Finally, if your method has been declared with a `throws` clause, don't forget to actually throw the exception in the body of your method using the `throw` statement.

When Not to Use Exceptions

Although they might seem appropriate at the time, there are several cases in which you should not use exceptions.

First, you should not use exceptions for circumstances you expect and could avoid easily. For example, although you can rely on an `ArrayIndexOutofBounds` exception to indicate when you've gone past the end of an array, it's easy to use the array's `length` variable to prevent you from going beyond the bounds.

In addition, if your users will enter data that must be an integer, testing to make sure that the data is an integer is a much better idea than throwing an exception and dealing with it somewhere else.

Exceptions take up a lot of processing time for your Java program. A simple test or series of tests will run much faster than exception handling and make your program more efficient. Exceptions should be used only for truly exceptional cases that are out of your control.

It's also easy to get carried away with exceptions and to try to make sure that all your methods have been declared to throw all the possible exceptions that they can possibly throw. This makes your code more complex; in addition, if other people will be using your code, they'll have to deal with handling all the exceptions that your methods might throw.

You're making more work for everyone involved when you get carried away with exceptions. Declaring a method to throw either few or many exceptions is a trade-off; the more exceptions your method can throw, the more complex that method is to use. Declare only the exceptions that have a reasonably fair chance of happening and that make sense for the overall design of your classes.

Bad Style Using Exceptions

When you first start using exceptions, it might be appealing to work around the compiler errors that result when you use a method that declares a `throws` statement. Although it is legal to add an empty `catch` clause or to add a `throws` statement to your own method (and there are appropriate reasons for doing so), intentionally dropping exceptions without dealing with them subverts the checks that the Java compiler does for you.

The Java exception system was designed so that if an error can occur, you're warned about it. Ignoring those warnings and working around them makes it possible for fatal errors to occur in your program—errors that you could have avoided with a few lines of code. Even worse, adding `throws` clauses to your methods to avoid exceptions means that the users of your methods (objects further up in the calling chain) will have to deal with them. You've just made your methods more difficult to use.

Compiler errors regarding exceptions are there to remind you to reflect on these issues. Take the time to deal with the exceptions that might affect your code. This extra care richly rewards you as you reuse your classes in later projects and in larger and larger programs. Of course, the Java 2 class library has been written with exactly this degree of care, and that's one of the reasons it's robust enough to be used in constructing all your Java projects.

Assertions

Exceptions are one way to improve the reliability of your Java programs. Assertions are expressions that represent a condition that a programmer believes to be true at a specific place in a program. If an assertion isn't true, an error results.

The `assert` keyword is followed by a conditional expression or Boolean value, as in this example:

```
assert price > 0;
```

In this example, the `assert` statement claims that a variable named `price` has a value greater than 0. Assertions are a way to assure yourself that a program is running correctly by putting it to the test, writing conditional expressions that identify correct behavior.

The assert keyword must be followed by one of three things: an expression that is true or false, a `boolean` variable, or a method that returns a `boolean`.

If the assertion that follows the `assert` keyword is not true, an `AssertionError` exception is thrown. To make the error message associated with an assertion more meaningful, you can specify a string in an `assert` statement, as in the following example:

```
assert price > 0 : "Price less than 0.";
```

In this example, if `price` is less than 0 when the `assert` statement is executed, an `AssertionError` exception is thrown with the error message "Price less than 0".

You can catch these exceptions or leave them for the Java interpreter to deal with. Here's an example of how the SDK's interpreter responds when an `assert` statement is false:

```
Exception in thread "main" java.lang.AssertionError
    at AssertTest.main(AssertTest.java:14)
```

Here's an example when an `assert` statement with a descriptive error message is false:

```
Exception in thread "main" java.lang.AssertionError: Price less than 0.
    at AssertTest.main(AssertTest.java:14)
```

Although assertions are an official part of the Java language, they are not supported by default by the tools included with the SDK, and the same may be true with other Java development tools.

To enable assertions with the SDK, you must use command-line arguments when running the interpreter.

A class that contains assert statements can be compiled normally, as long as you're using a current version of the SDK.

The compiler includes support for assertions in the class file (or files) that it produces. (In Java 2 version 1.4, the compiler required the `-source 1.4` flag to support assertions.)

There are several ways to turn on assertions in the SDK's Java interpreter.

To enable assertions in all classes except those in the Java class library, use the `-ea` argument, as in this example:

```
java -ea PriceChecker
```

To enable assertions only in one class, follow `-ea` with a colon (":") and the name of the class, like this:

```
java -ea:PriceChecker PriceChecker
```

You also can enable assertions for a specific package by following `-ea:` with the name of the package (or `...` for the default package).

TIP

There's also an `-esa` flag that enables assertions in the Java class library. There isn't much reason for you to do this because you're probably not testing the reliability of that code.

When a class that contains assertions is run without an `-ea` or `-esa` flag, all `assert` statements are ignored.

Because Java has added the `assert` keyword, you must not use it as the name of a variable in your programs, even if they are not compiled with support for assertions enabled.

The next project, `CalorieCounter`, is a calculator application that uses an assertion. Listing 7.2 contains the source.

LISTING 7.2 The Full Source of `CalorieCounter.java`

```
public class CalorieCounter {
    float count;

    public CalorieCounter(float calories, float fat, float fiber) {
        if (fiber > 4) {
            fiber = 4;
        }
        count = (calories / 50) + (fat / 12) - (fiber / 5);
```

LISTING 7.2 continued

```
 9:         assert count > 0 : "Adjusted calories < 0";
10:     }
11:
12:     public static void main(String[] arguments) {
13:         if (arguments.length < 2) {
14:             System.out.println("Usage: java CalorieCounter cal fat fiber");
15:             System.exit(-1);
16:         }
17:         try {
18:             int calories = Integer.parseInt(arguments[0]);
19:             int fat = Integer.parseInt(arguments[1]);
20:             int fiber = Integer.parseInt(arguments[2]);
21:             CalorieCounter diet = new CalorieCounter(calories, fat, fiber);
22:             System.out.println("Adjusted calories: " + diet.count);
23:         } catch (NumberFormatException nfe) {
24:             System.out.println("All arguments must be numeric.");
25:             System.exit(-1);
26:         }
27:     }
28: }
```

The `CalorieCounter` application calculates an adjusted calories total for a food item using its calories, fat grams, and fiber grams as input. Programs such as this are common in weight management programs, enabling dieters to monitor their daily food intake.

The application takes three command-line arguments: `calories`, `fat`, and `fiber`, which are received as strings and converted to integer values in lines 18–20.

The `CalorieCounter` constructor takes the three values and plugs them into a formula in line 8 to produce an adjusted calorie count.

One of the assumptions of the constructor is that the adjusted count always will be a positive value. This is challenged with the following `assert` statement:

```
assert count > 0 : "Adjusted calories < 0";
```

The compiled class should be run with the `-ea` flag to employ assertions, as in this example:

```
java -ea CalorieCounter 150 3 0
```

Those values produce an adjusted calorie count of 3.25. To see the assertion proven false, use 30 calories, 0 grams of fat, and 6 grams of fiber as input.

Assertions are an unusual feature of the Java language—under most circumstances they cause absolutely nothing to happen. They're a means of expressing in a class the conditions under which it is running correctly (and the things you assume to be true as it runs). If you make liberal use of them in a class, it will either be more reliable or you'll learn that some of your assumptions are incorrect, which is useful knowledge in its own right.

CAUTION

Some Java programmers believe that because assertions can be turned off at runtime, they're an unreliable means of improving the reliability of a class.

Threads

One thing to consider in Java programming is how system resources are being used. Graphics, complex mathematical computations, and other intensive tasks can take up a lot of processor time.

This is especially true of programs that have a graphical user interface, which is a style of software that you'll be learning about next week.

If you write a graphical Java program that is doing something that consumes a lot of the computer's time, you might find that the program's graphical user interface responds slowly—drop-down lists take a second or more to appear, button clicks are recognized slowly, and so on.

To solve this problem, you can segregate the processor-hogging functions in a Java class so that they run separately from the rest of the program.

This is possible through the use of a feature of the Java language called threads.

Threads are parts of a program set up to run on their own while the rest of the program does something else. This also is called *multitasking* because the program can handle more than one task simultaneously.

Threads are ideal for anything that takes up a lot of processing time and runs continuously.

By putting the workload of the program into a thread, you are freeing up the rest of the program to handle other things. You also make handling the program easier for the virtual machine because all the intensive work is isolated into its own thread.

Writing a Threaded Program

Threads are implemented in Java with the `Thread` class in the `java.lang` package.

The simplest use of threads is to make a program pause in execution and stay idle during that time. To do this, call the `Thread` class method `sleep(long)` with the number of milliseconds to pause as the only argument.

This method throws an exception, `InterruptedException`, whenever the paused thread has been interrupted for some reason. (One possible reason: The user closes the program while it is sleeping.)

The following statements stop a program in its tracks for 3 seconds:

```
try {
    Thread.sleep(3000);
catch (InterruptedException ie) {
    // do nothing
}
```

The `catch` block does nothing, which is typical when you're using `sleep()`.

One way to use threads is to put all the time-consuming behavior into its own class.

A thread can be created in two ways: By subclassing the `Thread` class or implementing the `Runnable` interface in another class. Both belong to the `java.lang` package, so no import statement is necessary to refer to them.

Because the `Thread` class implements `Runnable`, both techniques result in objects that start and stop threads in the same manner.

To implement the `Runnable` interface, add the keyword `implements` to the class declaration followed by the name of the interface, as in the following example:

```
public class StockTicker implements Runnable {
    public void run() {
       // ...
    }
}
```

When a class implements an interface, it must include all methods of that interface. The `Runnable` interface contains only one method, `run()`.

The first step in creating a thread is to create a reference to an object of the `Thread` class:

```
Thread runner;
```

This statement creates a reference to a thread, but no `Thread` object has been assigned to it yet. Threads are created by calling the constructor `Thread(Object)` with the threaded object as an argument. You could create a threaded `StockTicker` object with the following statement:

```
StockTicker tix = new StockTicker();
Thread tickerThread = new Thread(tix);
```

Two good places to create threads are the constructor for an application and the constructor for a component (such as a panel).

A thread is begun by calling its `start()` method, as in the following statement:

```
tickerThread.start();
```

The following statements can be used in a thread class to start the thread:

```
Thread runner;
if (runner == null) {
    runner = new Thread(this);
    runner.start();
}
```

The `this` keyword used in the `Thread()` constructor refers to the object in which these statements are contained. The `runner` variable has a value of `null` before any object is assigned to it, so the `if` statement is used to make sure that the thread is not started more than once.

To run a thread, its `start()` method is called, as in this statement from the preceding example:

```
runner.start();
```

Calling a thread's `start()` method causes another method to be called—namely, the `run()` method that must be present in all threaded objects.

The `run()` method is the engine of a threaded class. In the introduction to threads, they were described as a means of segregating processor-intensive work so that it ran separately from the rest of a class. This kind of behavior would be contained within a thread's `run()` method and the methods that it calls.

A Threaded Application

Threaded programming requires a lot of interaction among different objects, so it should become clearer when you see it in action.

Listing 7.3 contains a class that finds a specific prime number in a sequence, such as the 10th prime, 100th prime, or 1,000th prime. This can take some time, especially for numbers beyond 100,000, so the search for the right prime takes place in its own thread.

Enter the text of Listing 7.3 in your Java editor and save it as `PrimeFinder.java`.

LISTING 7.3 The Full Text of `PrimeFinder.java`

```java
public class PrimeFinder implements Runnable {
    public long target;
    public long prime;
    public boolean finished = false;
    private Thread runner;

    PrimeFinder(long inTarget) {
        target = inTarget;
        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void run() {
        long numPrimes = 0;
        long candidate = 2;
        while (numPrimes < target) {
            if (isPrime(candidate)) {
                numPrimes++;
                prime = candidate;
            }
            candidate++;
        }
        finished = true;
    }

    boolean isPrime(long checkNumber) {
        double root = Math.sqrt(checkNumber);
        for (int i = 2; i <= root; i++) {
            if (checkNumber % i == 0)
                return false;
        }
        return true;
    }
}
```

Compile the `PrimeFinder` class when you're finished. This class doesn't have a `main()` method, so you can't run it as an application. You create a program that uses this class next.

The `PrimeFinder` class implements the `Runnable` interface, so it can be run as a thread.

There are three public instance variables:

- `target`—A Long that indicates when the specified prime in the sequence has been found. If you're looking for the 5,000th prime, `target` equals 5000.

- `prime`—A `long` that holds the last prime number found by this class.
- `finished`—A Boolean that indicates when the target has been reached.

There is also a private instance variable called `runner` that holds the `Thread` object that this class runs in. This object should be equal to `null` before the thread has been started.

The `PrimeFinder` constructor method in lines 7–13 sets the `target` instance variable and starts the thread if it hasn't already been started. When the thread's `start()` method is called, it in turn calls the `run()` method of the threaded class.

The `run()` method is in lines 15–26. This method does most of the work of the thread, which is typical of threaded classes. You want to put the most computing-intensive tasks in their own thread so that they don't bog down the rest of the program.

This method uses two new variables: `numPrimes`, the number of primes that have been found, and `candidate`, the number that might possibly be prime. The `candidate` variable begins at the first possible prime number, which is 2.

The `while` loop in lines 18–24 continues until the right number of primes has been found.

First, it checks whether the current `candidate` is prime by calling the `isPrime(`*`long`*`)` method, which returns `true` if the number is prime and `false` otherwise.

If the `candidate` is prime, `numPrimes` increases by one, and the `prime` instance variable is set to this prime number.

The `candidate` variable is then incremented by one, and the loop continues.

After the right number of primes has been found, the `while` loop ends, and the `finished` instance variable is set to `true`. This indicates that the `PrimeFinder` object has found the right prime number and is finished searching.

The end of the `run()` method is reached in line 26, and the thread is no longer doing any work.

The `isPrime()` method is contained in lines 28–35. This method determines whether a number is prime by using the `%` operator, which returns the remainder of a division operation. If a number is evenly divisible by 2 or any higher number (leaving a remainder of 0), it is not a prime number.

Listing 7.4 contains an application that uses the `PrimeFinder` class. Enter the text of Listing 7.4 and save the file as `PrimeThreads.java`.

LISTING 7.4 The Full Text of `PrimeThreads.java`

```
public class PrimeThreads {
    public static void main(String[] arguments) {
        PrimeThreads pt = new PrimeThreads(arguments);
    }

    public PrimeThreads(String[] arguments) {
        PrimeFinder[] finder = new PrimeFinder[arguments.length];
        for (int i = 0; i < arguments.length; i++) {
            try {
                long count = Long.parseLong(arguments[i]);
                finder[i] = new PrimeFinder(count);
                System.out.println("Looking for prime " + count);
            } catch (NumberFormatException nfe) {
                System.out.println("Error: " + nfe.getMessage());
            }
        }
        boolean complete = false;
        while (!complete) {
            complete = true;
            for (int j = 0; j < finder.length; j++) {
                if (finder[j] == null) continue;
                if (!finder[j].finished) {
                    complete = false;
                } else {
                    displayResult(finder[j]);
                    finder[j] = null;
                }
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                // do nothing
            }
        }
    }

    private void displayResult(PrimeFinder finder) {
        System.out.println("Prime " + finder.target
            + " is " + finder.prime);
    }
}
```

Save and compile the file when you're finished.

The `PrimeThreads` application can be used to find one or more prime numbers in sequence. Specify the prime numbers that you're looking for as command-line arguments and include as many as you want.

If you're using the SDK, here's an example of how you can run the application:

```
java PrimeThreads 1 10 100 1000
```

This produces the following output:

```
Looking for prime 1
Looking for prime 10
Looking for prime 100
Looking for prime 1000
Prime 1 is 2
Prime 10 is 29
Prime 100 is 541
Prime 1000 is 7919
```

The `for` loop in lines 8–16 of the `PrimeThreads` application creates one `PrimeFinder` object for each command-line argument specified when the program is run.

Because arguments are `Strings` and the `PrimeFinder` constructor requires `long` values, the `Long.parseLong(`*`String`*`)` class method is used to handle the conversion. All the number-parsing methods throw `NumberFormatException` exceptions, so they are enclosed in `try-catch` blocks to deal with arguments that are not numeric.

When a `PrimeFinder` object is created, the object starts running in its own thread (as specified in the `PrimeFinder` constructor).

The `while` loop in lines 18–34 checks to see whether any `PrimeFinder` thread has completed, which is indicated by its `finished` instance variable equaling `true`. When a thread has completed, the `displayResult()` method is called in line 25 to display the prime number that was found. The thread then is set to `null`, freeing the object for garbage collection (and preventing its result from being displayed more than once).

The call to `Thread.sleep(1000)` in line 30 causes the `while` loop to pause for 1 second during each pass through the loop. A slowdown in loops helps keep the Java interpreter from executing statements at such a furious pace that it becomes bogged down.

Stopping a Thread

Stopping a thread is a little more complicated than starting one. The `Thread` class includes a `stop()` method that can be called to stop a thread, but it creates instabilities in Java's runtime environment and can introduce hard-to-detect errors into a program. For this reason, the method has been deprecated, indicating that it should not be used in favor of another technique.

A better way to stop a thread is to place a loop in the thread's `run()` method that ends when a variable changes in value, as in the following example:

```
public void run() {
    while (okToRun == true) {
        // ...
    }
}
```

The okToRun variable could be an instance variable of the thread's class. If it is changed to false, the loop inside the run() method ends.

Another option you can use to stop a thread is to only loop in the run() method while the currently running thread has a variable that references it.

In previous examples, a Thread object called runner has been used to hold the current thread.

A class method, Thread.currentThread(), returns a reference to the current thread (in other words, the thread in which the object is running).

The following run() method loops as long as runner and currentThread() refer to the same object:

```
public void run() {
    Thread thisThread = Thread.currentThread();
    while (runner == thisThread) {
        // ...
    }
}
```

If you use a loop like this, you can stop the thread anywhere in the class with the following statement:

```
runner = null;
```

Summary

Exceptions, assertions, and threads aid your program's design and robustness.

Exceptions enable you to manage potential errors. By using try, catch, and finally, you can protect code that might result in exceptions by handling those exceptions as they occur.

Handling exceptions is only half the equation; the other half is generating and throwing exceptions. A throws clause tells a method's users that the method might throw an exception. It also can be used to pass on an exception from a method call in the body of your method.

You can create and throw exceptions with the `throw` keyword and even define your own new exceptions and subclasses of `Exception`.

Assertions enable you to use conditional statements and booleans to indicate that a program is running correctly. When this isn't the case, an assertion exception is thrown.

Threads enable you to run the most processor-intensive parts of a Java class separately from the rest of the class. This is especially useful when the class is doing something computing-intensive such as animation, complex mathematics, or looping through a large amount of data quickly.

You also can use threads to do several things at once and to start and stop threads externally.

Threads implement the `Runnable` interface, which contains one method: `run()`. When you start a thread by calling its `start()` method, the thread's `run()` method is called automatically.

Q&A

Q I'm still not sure I understand the differences between exceptions, errors, and runtime exceptions. Is there another way of looking at them?

A Errors are caused by dynamic linking or virtual machine problems, and are thus too low-level for most programs to care about—or be able to handle even if they did care about them.

Runtime exceptions are generated by the normal execution of Java code, and although they occasionally reflect a condition you will want to handle explicitly, more often they reflect a coding mistake made by the programmer, and thus simply need to print an error to help flag that mistake.

Exceptions that are non-runtime exceptions (`IOException` exceptions, for example) are conditions that, because of their nature, should be explicitly handled by any robust and well-thought-out code. The Java class library has been written using only a few of these, but those few are important to using the system safely and correctly. The compiler helps you handle these exceptions properly via its `throws` clause checks and restrictions.

Q Is there any way to get around the strict restrictions placed on methods by the `throws` clause?

A Yes. Suppose that you have thought long and hard and have decided that you need to circumvent this restriction. This is almost never the case because the right solution is to go back and redesign your methods to reflect the exceptions that you need to throw. Imagine, however, that for some reason a system class has you in a bind. Your first solution is to subclass `RuntimeException` to make up a new, unchecked exception of your own. Now you can throw it to your heart's content because the `throws` clause that was annoying you does not need to include this new exception. If you need a lot of such exceptions, an elegant approach is to mix in some novel exception interfaces with your new `Runtime` classes. You're free to choose whatever subset of these new interfaces you want to `catch` (none of the normal `Runtime` exceptions need to be caught), whereas any leftover `Runtime` exceptions are allowed to go through that otherwise annoying standard method in the library.

Quiz

Review today's material by taking this three-question quiz.

Questions

1. What keyword is used to jump out of a `try` block and into a `finally` block?
   a. `catch`
   b. `return`
   c. `while`
2. What class should be the superclass of any exceptions you create in Java?
   a. `Throwable`
   b. `Error`
   c. `Exception`
3. If a class implements the `Runnable` interface, what methods must the class contain?
   a. `start()`, `stop()`, and `run()`
   b. `actionPerformed()`
   c. `run()`

Answers

1. b.
2. c. `Throwable` and `Error` are of use primarily by Java. The kinds of errors you'll want to note in your programs belong in the `Exception` hierarchy.
3. c. The `Runnable` interface requires only the `run()` method.

Certification Practice

The following question is the kind of thing you could expect to be asked on a Java programming certification test. Answer it without looking at today's material or using the Java compiler to test the code.

The `AverageValue` application is supposed to take up to 10 floating-point numbers as command-line arguments and display their average.

Given:

```
public class AverageValue {
    public static void main(String[] arguments) {
        float[] temps = new float[10];
        float sum = 0;
        int count = 0;
        int i;
        for (i = 0; i < arguments.length & i < 10; i++) {
            try {
                temps[i] = Float.parseFloat(arguments[i]);
                count++;
            } catch (NumberFormatException nfe) {
                System.out.println("Invalid input: " + arguments[i]);
            }
            sum += temps[i];
        }
        System.out.println("Average: " + (sum / i));
    }
}
```

Which statement contains an error?

a. `for (i = 0; i < arguments.length & i < 10; i++) {`

b. `sum += temps[i];`

c. `System.out.println("Average: " + (sum / i));`

d. None; the program is correct.

The answer is available on the book's Web site at `http://www.java21days.com`. Visit the Day 7 page and click the Certification Practice link.

Exercises

To extend your knowledge of the subjects covered today, try the following exercises:

1. Modify the `PrimeFinder` class so that it throws a new exception, `NegativeNumberException`, if a negative number is sent to the constructor.

2. Modify the `PrimeThreads` application so that it can handle the new `NegativeNumberException` error.

Where applicable, exercise solutions are offered on the book's Web site at `http://www.java21days.com`.