[Previous Section | Next Section | Chapter Index | Main Index]

Section 8.3

Exceptions and try..catch

Subsections

GETTING A PROGRAM TO WORK under ideal circumstances is usually a lot easier than making the program robust. A robust program can survive unusual or "exceptional" circumstances without crashing. One approach to writing robust programs is to anticipate the problems that might arise and to include tests in the program for each possible problem. For example, a program will crash if it tries to use an array element `A[i]`, when `i` is not within the declared range of indices for the array `A`. A robust program must anticipate the possibility of a bad index and guard against it. One way to do this is to write the program in a way that ensures (as a postcondition of the code that precedes the array reference) that the index is in the legal range. Another way is to test whether the index value is legal before using it in the array. This could be done with an `if` statement:

```
if (i < 0 || i >= A.length) {
   ...  // Do something to handle the out-of-range index, i
}
else {
   ...  // Process the array element, A[i]
}
```

There are some problems with this approach. It is difficult and sometimes impossible to anticipate all the possible things that might go wrong. It's not always clear what to do when an error is detected. Furthermore, trying to anticipate all the possible problems can turn what would otherwise be a straightforward program into a messy tangle of `if` statements.

8.3.1 Exceptions and Exception Classes

We have already seen in Section 3.7 that Java provides a neater, more structured alternative technique for dealing with errors that can occur while a program is running. The technique is referred to as exception handling. The word "exception" is meant to be more general than "error." It includes any circumstance that arises as the program is executed which is meant to be treated as an exception to the normal flow of control of the program. An exception might be an error, or it might just be a special case that you would rather not have clutter up your elegant algorithm.

When an exception occurs during the execution of a program, we say that the exception is thrown. When this happens, the normal flow of the program is thrown off-track, and the program is in danger of crashing. However, the crash can be avoided if the exception is caught and handled in some way. An exception can be thrown in one part of a program and caught in a different part. An exception that is not caught will generally cause the program to crash. (More exactly, the thread that throws the exception will crash. In a multithreaded program, it is possible for other threads to continue even after one crashes. We will cover threads in Chapter 12. In particular, GUI programs are multithreaded, and parts of the program might continue to function even while other parts are non-functional because of exceptions.)

By the way, since Java programs are executed by a Java interpreter, having a program crash simply means that it terminates abnormally and prematurely. It doesn't mean that the Java interpreter will crash. In effect, the interpreter catches any exceptions that are not caught by the program. The interpreter responds by terminating the program. In many other programming languages, a crashed

program will sometimes crash the entire system and freeze the computer until it is restarted. With Java, such system crashes should be impossible -- which means that when they happen, you have the satisfaction of blaming the system rather than your own program.

Exceptions were introduced in Section 3.7, along with the `try..catch` statement, which is used to catch and handle exceptions. However, that section did not cover the complete syntax of `try..catch` or the full complexity of exceptions. In this section, we cover these topics in full detail.

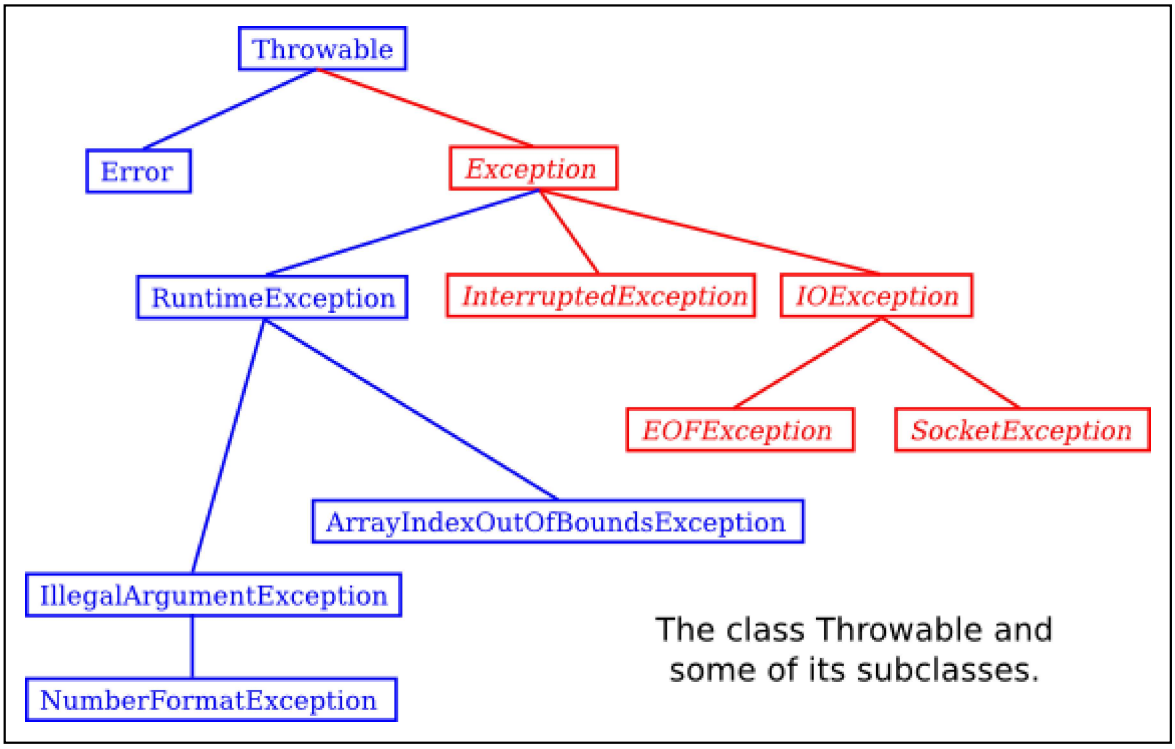
When an exception occurs, the thing that is actually "thrown" is an object. This object can carry information (in its instance variables) from the point where the exception occurs to the point where it is caught and handled. This information always includes the subroutine call stack, which is a list of the subroutines that were being executed when the exception was thrown. (Since one subroutine can call another, several subroutines can be active at the same time.) Typically, an exception object also includes an error message describing what happened to cause the exception, and it can contain other data as well. All exception objects must belong to a subclass of the standard class `java.lang.Throwable`. In general, each different type of exception is represented by its own subclass of Throwable, and these subclasses are arranged in a fairly complex class hierarchy that shows the relationship among various types of exception. Throwable has two direct subclasses, Error and Exception. These two subclasses in turn have many other predefined subclasses. In addition, a programmer can create new exception classes to represent new types of exception.

Most of the subclasses of the class Error represent serious errors within the Java virtual machine that should ordinarily cause program termination because there is no reasonable way to handle them. In general, you should not try to catch and handle such errors. An example is a ClassFormatError, which occurs when the Java virtual machine finds some kind of illegal data in a file that is supposed to contain a compiled Java class. If that class was being loaded as part of the program, then there is really no way for the program to proceed.

On the other hand, subclasses of the class Exception represent exceptions that are meant to be caught. In many cases, these are exceptions that might naturally be called "errors," but they are errors in the program or in input data that a programmer can anticipate and possibly respond to in some reasonable way. (However, you should avoid the temptation of saying, "Well, I'll just put a thing here to catch all the errors that might occur, so my program won't crash." If you don't have a reasonable way to respond to the error, it's best just to let the program crash, because trying to go on will probably only lead to worse things down the road -- in the worst case, a program that gives an incorrect answer without giving you any indication that the answer might be wrong!)

The class Exception has its own subclass, RuntimeException. This class groups together many common exceptions, including all those that have been covered in previous sections. For example, IllegalArgumentException and NullPointerException are subclasses of RuntimeException. A RuntimeException generally indicates a bug in the program, which the programmer should fix. RuntimeExceptions and Errors share the property that a program can simply ignore the possibility that they might occur. ("Ignoring" here means that you are content to let your program crash if the exception occurs.) For example, a program does this every time it uses an array reference like `A[i]` without making arrangements to catch a possible ArrayIndexOutOfBoundsException. For all other exception classes besides Error, RuntimeException, and their subclasses, exception-handling is "mandatory" in a sense that I'll discuss below.

The following diagram is a class hierarchy showing the class `Throwable` and just a few of its subclasses. Classes that require mandatory exception-handling are shown in red:



The class Throwable includes several instance methods that can be used with any exception object. If `e` is of type Throwable (or one of its subclasses), then `e.getMessage()` is a function that returns a String that describes the exception. The function `e.toString()`, which is used by the system whenever it needs a string representation of the object, returns a String that contains the name of the class to which the exception belongs as well as the same string that would be returned by `e.getMessage()`. And the method `e.printStackTrace()` writes a stack trace to standard output that tells which subroutines were active when the exception occurred. A stack trace can be very useful when you are trying to determine the cause of the problem. (Note that if an exception is **not** caught by the program, then the default response to the exception prints the stack trace to standard output.)

8.3.2 The try Statement

To catch exceptions in a Java program, you need a `try` statement. We have been using such statements since Section 3.7, but the full syntax of the `try` statement is more complicated than what was presented there. The `try` statements that we have used so far had a syntax similar to the following example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
   e.printStackTrace();
}
```

Here, the computer tries to execute the block of statements following the word "`try`". If no exception occurs during the execution of this block, then the "`catch`" part of the statement is simply ignored. However, if an exception of type ArrayIndexOutOfBoundsException occurs, then the computer jumps immediately to the `catch` clause of the `try` statement. This block of statements is said to be an exception handler for ArrayIndexOutOfBoundsException. By handling the exception in this way, you prevent it from crashing the program. Before the body of the `catch` clause is executed, the object that

represents the exception is assigned to the variable `e`, which is used in this example to print a stack trace.

However, the full syntax of the `try` statement has many options. It will take a while to go through them. For one thing, a `try..catch` statement can have more than one `catch` clause. This makes it possible to catch several different types of exception with one `try` statement. In the above example, in addition to the possible ArrayIndexOutOfBoundsException, there is a possible NullPointerException which will occur if the value of `M` is `null`. We can handle both possible exceptions by adding a second `catch` clause to the `try` statement:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( ArrayIndexOutOfBoundsException e ) {
   System.out.println("M is the wrong size to have a determinant.");
}
catch ( NullPointerException e ) {
   System.out.print("Programming error!  M doesn't exist." + );
}
```

Here, the computer tries to execute the statements in the `try` clause. If no error occurs, both of the `catch` clauses are skipped. If an ArrayIndexOutOfBoundsException occurs, the computer executes the body of the first `catch` clause and skips the second one. If a NullPointerException occurs, it jumps to the second `catch` clause and executes that.

Note that both ArrayIndexOutOfBoundsException and NullPointerException are subclasses of RuntimeException. It's possible to catch all RuntimeExceptions with a single `catch` clause. For example:

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( RuntimeException err ) {
   System.out.println("Sorry, an error has occurred.");
   System.out.println("The error was: " + err);
}
```

The `catch` clause in this `try` statement will catch any exception belonging to class RuntimeException or to any of its subclasses. This shows why exception classes are organized into a class hierarchy. It allows you the option of casting your net narrowly to catch only a specific type of exception. Or you can cast your net widely to catch a wide class of exceptions. Because of subclassing, when there are multiple `catch` clauses in a `try` statement, it is possible that a given exception might match several of those `catch` clauses. For example, an exception of type NullPointerException would match `catch` clauses for NullPointerException, RuntimeException, Exception, or Throwable. In this case, only the **first** `catch` clause that matches the exception is executed.

Of course, catching RuntimeException would catch many more types of exception than the two that we are interested in. It is possible to combine several specific exception types in a single `catch` clause. For example,

```
try {
    double determinant = M[0][0]*M[1][1] - M[0][1]*M[1][0];
    System.out.println("The determinant of M is " + determinant);
}
catch ( NullPointerException | ArrayIndexOutOfBoundsException err ) {
   System.out.println("Sorry, an error has occurred.");
```

```
   System.out.println("The error was: " + err);
}
```

Here, the two exception types are combined with a "|", the vertical line character that is also used in the boolean **or** operator. This example will catch errors of type NullPointerException or ArrayIndexOutOfBoundsException, and no other types.

The example I've been using here is not realistic, because you are not very likely to use exception-handling to guard against null pointers and bad array indices. This is a case where careful programming is better than exception handling: Just be sure that your program assigns a reasonable, non-`null` value to the array `M`. You would certainly resent it if the designers of Java forced you to set up a `try..catch` statement every time you wanted to use an array! This is why handling of potential RuntimeExceptions is not mandatory. There are just too many things that might go wrong! (This also shows that exception-handling does not solve the problem of program robustness. It just gives you a tool that will in many cases let you approach the problem in a more organized way.)

I have still not completely specified the syntax of the `try` statement. The next variation is the possibility of a finally clause at the end of a `try` statement. With this addition, syntax of the `try` statement can be described as:

```
try {
   statements
}
optional-catch-clauses
optional-finally-clause
```

Note that the `catch` clauses are also listed as optional. The `try` statement can include zero or more `catch` clauses and, optionally, a `finally` clause. The `try` statement **must** include one or the other. That is, a `try` statement can have either a `finally` clause, or one or more `catch` clauses, or both. The syntax for a `catch` clause is

```
catch ( exception-class-names variable-name ) {
   statements
}
```

where **exception-class-names** can be a single exception class or several classes separated by "|". The syntax for a `finally` clause is

```
finally {
   statements
}
```

The semantics of the `finally` clause is that the block of statements in the `finally` clause is guaranteed to be executed as the last step in the execution of the try statement, whether or not any exception occurs and whether or not any exception that does occur is caught and handled. The `finally` clause is meant for doing essential cleanup that under no circumstances should be omitted. One example of this type of cleanup is closing a network connection. Although you don't yet know enough about networking to look at the actual programming in this case, we can consider some pseudocode:

```
try {
   open a network connection
   communicate over the connection
}
catch ( IOException e ) {
   report the error
}
finally {
   if the connection was successfully opened
```

```
            close the connection
    }
```

The `finally` clause ensures that the network connection will definitely be closed, whether or not an error occurs during the communication. The pseudocode in this example follows a general pattern that can be used to robustly obtain a resource, use the resource, and then release the resource.

The pattern of obtaining a resource, then using the resource, and then releasing the resource is very common. Note that the resource can only be released if no error occurred while obtaining it. And, if it was successfully obtained, then it should be closed whether or not an error occurs while using it. This pattern is so common that it leads to one last option in the `try` statement syntax. With this option, you only need code to obtain the resource, and you don't need to worry about releasing it. That will happen automatically at the end of the `try` statement.

In order for this to work, the resource must be represented by an object that implements an interface named AutoCloseable, which defines a single method named `close()`, with no parameters. Standard Java classes that represent things like files and network connections already implement AutoClosable. So does the Scanner class, which was introduced in Subsection 2.4.6. In that section, I showed how to use a Scanner to read from `System.in`. Although I didn't do it in that section, it's considered good form to close a Scanner after using it. Here is an example that uses the resource pattern in a `try` statement to make sure that the Scanner is closed automatically:

```
try( Scanner in = new Scanner(System.in) ) {
    // Use the Scanner to read from standard input
}
catch (Exception e) {
    // ... some error occurred while using the Scanner
}
```

The statement that allocates the resource goes in parentheses after the word "try". The statement must have the form of a variable declaration that includes an initialization of the variable. The variable is local to the `try` statement. (You can actually declare several variables in the parentheses, separated by semicolons.) In this example, we can be sure that `in.close()` will definitely be called by the system at the end of the `try` statement, as long as the Scanner was successfully initialized.

This is all getting quite complicated, and I won't continue the discussion here. The sample program *TryStatementDemo.java* demonstrates a `try` statement with all its options, and it includes a lot of comments to help you understand what can happen when you run the program.

8.3.3 Throwing Exceptions

There are times when it makes sense for a program to deliberately throw an exception. This is the case when the program discovers some sort of exceptional or error condition, but there is no reasonable way to handle the error at the point where the problem is discovered. The program can throw an exception in the hope that some other part of the program will catch and handle the exception. This can be done with a throw statement. You have already seen an example of this in Subsection 4.3.8. In this section, we cover the `throw` statement more fully. The syntax of the `throw` statement is:

```
throw  exception-object ;
```

The **exception-object** must be an object belonging to one of the subclasses of `Throwable`. Usually, it will in fact belong to one of the subclasses of `Exception`. In most cases, it will be a newly constructed object created with the `new` operator. For example:

```
throw new ArithmeticException("Division by zero");
```

The parameter in the constructor becomes the error message in the exception object; if `e` refers to the object, the error message can be retrieved by calling `e.getMessage()`. (You might find this example a bit odd, because you might expect the system itself to throw an ArithmeticException when an attempt is made to divide by zero. So why should a programmer bother to throw the exception? Recall that if the numbers that are being divided are of type int, then division by zero will indeed throw an ArithmeticException. However, no arithmetic operations with floating-point numbers will ever produce an exception. Instead, the special value `Double.NaN` is used to represent the result of an illegal operation. In some situations, you might prefer to throw an ArithmeticException when a real number is divided by zero.)

An exception can be thrown either by the system or by a `throw` statement. The exception is processed in exactly the same way in either case. Suppose that the exception is thrown inside a `try` statement. If that `try` statement has a `catch` clause that handles that type of exception, then the computer jumps to the `catch` clause and executes it. The exception has been handled. After handling the exception, the computer executes the `finally` clause of the `try` statement, if there is one. It then continues normally with the rest of the program, which follows the `try` statement. If the exception is not immediately caught and handled, the processing of the exception will continue.

When an exception is thrown during the execution of a subroutine and the exception is not handled in the same subroutine, then that subroutine is terminated (after the execution of any pending `finally` clauses). Then the routine that called that subroutine gets a chance to handle the exception. That is, if the subroutine was called inside a `try` statement that has an appropriate `catch` clause, then **that** `catch` clause will be executed and the program will continue on normally from there. Again, if the second routine does not handle the exception, then it also is terminated and the routine that called **it** (if any) gets the next shot at the exception. The exception will crash the program only if it passes up through the entire chain of subroutine calls without being handled. (In fact, even this is not quite true: In a multithreaded program, only the thread in which the exception occurred is terminated.)

A subroutine that might generate an exception can announce this fact by adding a clause "`throws` **exception-class-name**" to the header of the routine. For example:

```
/**
 * Returns the larger of the two roots of the quadratic equation
 * A*x*x + B*x + C = 0, provided it has any roots.  If A == 0 or
 * if the discriminant, B*B - 4*A*C, is negative, then an exception
 * of type IllegalArgumentException is thrown.
 */
static public double root( double A, double B, double C )
                                throws IllegalArgumentException {
    if (A == 0) {
      throw new IllegalArgumentException("A can't be zero.");
    }
    else {
       double disc = B*B - 4*A*C;
       if (disc < 0)
          throw new IllegalArgumentException("Discriminant < zero.");
       return  (-B + Math.sqrt(disc)) / (2*A);
    }
}
```

As discussed in the previous section, the computation in this subroutine has the preconditions that `A != 0` and `B*B-4*A*C >= 0`. The subroutine throws an exception of type IllegalArgumentException when either of these preconditions is violated. When an illegal condition is found in a subroutine, throwing an exception is often a reasonable response. If the program that called the subroutine knows some good way to handle the error, it can catch the exception. If not, the program will crash -- and the programmer will know that the program needs to be fixed.

A `throws` clause in a subroutine heading can declare several different types of exception, separated by commas. For example:

```
void processArray(int[] A) throws NullPointerException,
                                  ArrayIndexOutOfBoundsException { ...
```

8.3.4 Mandatory Exception Handling

In the preceding example, declaring that the subroutine `root()` can throw an IllegalArgumentException is just a courtesy to potential readers of this routine. This is because handling of IllegalArgumentExceptions is not "mandatory." A routine can throw an IllegalArgumentException without announcing the possibility. And a program that calls that routine is free either to catch or to ignore the exception, just as a programmer can choose either to catch or to ignore an exception of type NullPointerException.

For those exception classes that require mandatory handling, the situation is different. If a subroutine can throw such an exception, that fact **must** be announced in a `throws` clause in the routine definition. Failing to do so is a syntax error that will be reported by the compiler. Exceptions that require mandatory handling are called checked exceptions. The compiler will check that such exceptions are handled by the program.

Suppose that some statement in the body of a subroutine can generate a checked exception, one that requires mandatory handling. The statement could be a `throw` statement, which throws the exception directly, or it could be a call to a subroutine that can throw the exception. In either case, the exception **must** be handled. This can be done in one of two ways: The first way is to place the statement in a `try` statement that has a `catch` clause that handles the exception; in this case, the exception is handled within the subroutine, so that no caller of the subroutine can ever see the exception. The second way is to declare that the subroutine can throw the exception. This is done by adding a "`throws`" clause to the subroutine heading, which alerts any callers to the possibility that the exception might be generated when the subroutine is executed. The caller will, in turn, be forced either to handle the exception in a `try` statement or to declare the exception in a `throws` clause in its own header.

Exception-handling is mandatory for any exception class that is **not** a subclass of either Error or RuntimeException. These checked exceptions generally represent conditions that are outside the control of the programmer. For example, they might represent bad input or an illegal action taken by the user. There is no way to **avoid** such errors, so a robust program has to be prepared to handle them. The design of Java makes it impossible for programmers to ignore the possibility of such errors.

Among the checked exceptions are several that can occur when using Java's input/output routines. This means that you can't even use these routines unless you understand something about exception-handling. Chapter 11 deals with input/output and uses checked exceptions extensively.

8.3.5 Programming with Exceptions

Exceptions can be used to help write robust programs. They provide an organized and structured approach to robustness. Without exceptions, a program can become cluttered with `if` statements that test for various possible error conditions. With exceptions, it becomes possible to write a clean implementation of an algorithm that will handle all the normal cases. The exceptional cases can be handled elsewhere, in a `catch` clause of a `try` statement.

When a program encounters an exceptional condition and has no way of handling it immediately, the program can throw an exception. In some cases, it makes sense to throw an exception belonging to

one of Java's predefined classes, such as IllegalArgumentException or IOException. However, if there is no standard class that adequately represents the exceptional condition, the programmer can define a new exception class. The new class must extend the standard class Throwable or one of its subclasses. In general, if the programmer does **not** want to require mandatory exception handling, the new class will extend RuntimeException (or one of its subclasses). To create a new checked exception class, which **does** require mandatory handling, the programmer can extend one of the other subclasses of Exception or can extend Exception itself.

Here, for example, is a class that extends Exception, and therefore requires mandatory exception handling when it is used:

```
public class ParseError extends Exception {
   public ParseError(String message) {
         // Create a ParseError object containing
         // the given message as its error message.
      super(message);
   }
}
```

The class contains only a constructor that makes it possible to create a ParseError object containing a given error message. (The statement "`super(message)`" calls a constructor in the superclass, Exception. See Subsection 5.6.3.) Of course the class inherits the `getMessage()` and `printStackTrace()` routines from its superclass. If `e` refers to an object of type ParseError, then the function call `e.getMessage()` will retrieve the error message that was specified in the constructor. But the main point of the ParseError class is simply to exist. When an object of type ParseError is thrown, it indicates that a certain type of error has occurred. (Parsing, by the way, refers to figuring out the syntax of a string. A ParseError would indicate, presumably, that some string that is being processed by the program does not have the expected form.)

A `throw` statement can be used in a program to throw an error of type ParseError. The constructor for the ParseError object must specify an error message. For example:

```
throw new ParseError("Encountered an illegal negative number.");
```

or

```
throw new ParseError("The word '" + word
                                + "' is not a valid file name.");
```

Since ParseError is defined as a subclass of Exception, it is a checked exception. If the `throw` statement does not occur in a `try` statement that catches the error, then the subroutine that contains the `throw` must declare that it can throw a ParseError by adding the clause "`throws ParseError`" to the subroutine heading. For example,

```
void getUserData() throws ParseError {
   . . .
}
```

This would not be required if ParseError were defined as a subclass of RuntimeException instead of Exception, since in that case ParseErrors would not be checked exceptions.

A routine that wants to handle ParseErrors can use a `try` statement with a `catch` clause that catches ParseErrors. For example:

```
try {
   getUserData();
   processUserData();
}
```

```
catch (ParseError pe) {
   . . .  // Handle the error
}
```

Note that since ParseError is a subclass of Exception, a `catch` clause of the form "`catch (Exception e)`" would also catch ParseErrors, along with any other object of type Exception.

Sometimes, it's useful to store extra data in an exception object. For example,

```
class ShipDestroyed extends RuntimeException {
   Ship ship;  // Which ship was destroyed.
   int where_x, where_y;  // Location where ship was destroyed.
   ShipDestroyed(String message, Ship s, int x, int y) {
         // Constructor creates a ShipDestroyed object
         // carrying an error message plus the information
         // that the ship s was destroyed at location (x,y)
         // on the screen.
       super(message);
       ship = s;
       where_x = x;
       where_y = y;
   }
}
```

Here, a ShipDestroyed object contains an error message and some information about a ship that was destroyed. This could be used, for example, in a statement:

```
if ( userShip.isHit() )
   throw new ShipDestroyed("You've been hit!", userShip, xPos, yPos);
```

Note that the condition represented by a ShipDestroyed object might not even be considered an error. It could be just an expected interruption to the normal flow of a game. Exceptions can sometimes be used to handle such interruptions neatly.

The ability to throw exceptions is particularly useful in writing general-purpose methods and classes that are meant to be used in more than one program. In this case, the person writing the method or class often has no reasonable way of handling the error, since that person has no way of knowing exactly how the method or class will be used. In such circumstances, a novice programmer is often tempted to print an error message and forge ahead, but this is almost never satisfactory since it can lead to unpredictable results down the line. Printing an error message and terminating the program is almost as bad, since it gives the program no chance to handle the error.

The program that calls the method or uses the class needs to know that the error has occurred. In languages that do not support exceptions, the only alternative is to return some special value or to set the value of some variable to indicate that an error has occurred. For example, the `readMeasurement()` function in Subsection 8.2.2 returns the value `-1` if the user's input is illegal. However, this only does any good if the main program bothers to test the return value. It is very easy to be lazy about checking for special return values every time a subroutine is called. And in this case, using `-1` as a signal that an error has occurred makes it impossible to allow negative measurements. Exceptions are a cleaner way for a subroutine to react when it encounters an error.

It is easy to modify the `readMeasurement()` function to use exceptions instead of a special return value to signal an error. My modified subroutine throws a ParseError when the user's input is illegal, where ParseError is the subclass of Exception that was defined above. (Arguably, it might be reasonable to avoid defining a new class by using the standard exception class IllegalArgumentException instead.) The changes from the original version are shown in red:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:   The input line is not empty.
 * Postcondition:  If the user's input is legal, the measurement
 *                 is converted to inches and returned.
 * @throws ParseError if the user's input is not legal.
 */
static double readMeasurement() throws ParseError {

   double inches;  // Total number of inches in user's measurement.

   double measurement;  // One measurement,
                        //   such as the 12 in "12 miles."
   String units;        // The units specified for the measurement,
                        //   such as "miles."

   char ch;  // Used to peek at next character in the user's input.

   inches = 0;  // No inches have yet been read.

   skipBlanks();
   ch = TextIO.peek();

   /* As long as there is more input on the line, read a measurement and
      add the equivalent number of inches to the variable, inches.  If an
      error is detected during the loop, end the subroutine immediately
      by throwing a ParseError. */

   while (ch != '\n') {

       /* Get the next measurement and the units.  Before reading
          anything, make sure that a legal value is there to read. */

       if ( ! Character.isDigit(ch) ) {
           throw new ParseError("Expected to find a number, but found " + ch);
       }
       measurement = TextIO.getDouble();

       skipBlanks();
       if (TextIO.peek() == '\n') {
          throw new ParseError("Missing unit of measure at end of line.");
       }
       units = TextIO.getWord();
       units = units.toLowerCase();

       /* Convert the measurement to inches and add it to the total. */

       if (units.equals("inch")
               || units.equals("inches") || units.equals("in")) {
           inches += measurement;
       }
       else if (units.equals("foot")
                   || units.equals("feet") || units.equals("ft")) {
           inches += measurement * 12;
       }
       else if (units.equals("yard")
                   || units.equals("yards") || units.equals("yd")) {
           inches += measurement * 36;
       }
       else if (units.equals("mile")
                   || units.equals("miles") || units.equals("mi")) {
           inches += measurement * 12 * 5280;
       }
       else {
           throw new ParseError("\"" + units
```

```
                                        + "\" is not a legal unit of measure.");
                }

                /* Look ahead to see whether the next thing on the line is
                   the end-of-line. */

                skipBlanks();
                ch = TextIO.peek();

             }  // end while

             return inches;

          } // end readMeasurement()
```

In the main program, this subroutine is called in a `try` statement of the form

```
try {
   inches = readMeasurement();
}
catch (ParseError e) {
   . . .  // Handle the error.
}
```

The complete program can be found in the file *LengthConverter3.java*. From the user's point of view, this program has exactly the same behavior as the program *LengthConverter2* from the previous section. Internally, however, the programs are significantly different, since `LengthConverter3` uses exception handling.

[Previous Section | Next Section | Chapter Index | Main Index]