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

Section 3.7

Introduction to Exceptions and try..catch

Subsections

IN ADDITION TO THE CONTROL structures that determine the normal flow of control in a program, Java has a way to deal with "exceptional" cases that throw the flow of control off its normal track. When an error occurs during the execution of a program, the default behavior is to terminate the program and to print an error message. However, Java makes it possible to "catch" such errors and program a response different from simply letting the program crash. This is done with the try..catch statement. In this section, we will take a preliminary and incomplete look the `try..catch` statement, leaving out a lot of the rather complex syntax of this statement. Error handling is a complex topic, which we will return to in Section 8.3, and we will cover the full syntax of `try..catch` at that time.

3.7.1 Exceptions

The term exception is used to refer to the type of event that one might want to handle with a `try..catch`. An exception is an exception to the normal flow of control in the program. The term is used in preference to "error" because in some cases, an exception might not be considered to be an error at all. You can sometimes think of an exception as just another way to organize a program.

Exceptions in Java are represented as objects of type *Exception*. Actual exceptions are usually defined by subclasses of *Exception*. Different subclasses represent different types of exceptions. We will look at only two types of exception in this section: *NumberFormatException* and *IllegalArgumentException*.

A *NumberFormatException* can occur when an attempt is made to convert a string into a number. Such conversions are done by the functions `Integer.parseInt` and `Double.parseDouble`. (See Subsection 2.5.7.) Consider the function call `Integer.parseInt(str)` where `str` is a variable of type *String*. If the value of `str` is the string `"42"`, then the function call will correctly convert the string into the int 42. However, if the value of `str` is, say, `"fred"`, the function call will fail because `"fred"` is not a legal string representation of an int value. In this case, an exception of type *NumberFormatException* occurs. If nothing is done to handle the exception, the program will crash.

An *IllegalArgumentException* can occur when an illegal value is passed as a parameter to a subroutine. For example, if a subroutine requires that a parameter be greater than or equal to zero, an *IllegalArgumentException* might occur when a negative value is passed to the subroutine. How to respond to the illegal value is up to the person who wrote the subroutine, so we can't simply say that every illegal parameter value will result in an *IllegalArgumentException*. However, it is a common response.

3.7.2 try..catch

When an exception occurs, we say that the exception is "thrown." For example, we say that

`Integer.parseInt(str)` throws an exception of type *NumberFormatException* when the value of `str` is illegal. When an exception is thrown, it is possible to "catch" the exception and prevent it from crashing the program. This is done with a try..catch statement. In simplified form, the syntax for a `try..catch` statement can be:

```
try {
   statements-1
}
catch ( exception-class-name  variable-name ) {
   statements-2
}
```

The **exception-class-name** could be *NumberFormatException*, *IllegalArgumentException*, or some other exception class. When the computer executes this `try..catch` statement, it executes **statements-1**, the statements inside the `try` part. If no exception occurs during the execution of **statements-1**, then the computer just skips over the `catch` part and proceeds with the rest of the program. However, if an exception of type **exception-class-name** occurs during the execution of **statements-1**, the computer immediately jumps from the point where the exception occurs to the `catch` part and executes **statements-2**, skipping any remaining statements in **statements-1**. Note that only one type of exception is caught; if some other type of exception occurs during the execution of **statements-1**, it will crash the program as usual.

During the execution of **statements-2**, the **variable-name** represents the exception object, so that you can, for example, print it out. The exception object contains information about the cause of the exception. This includes an error message, which will be displayed if you print out the exception object.

After the end of the `catch` part, the computer proceeds with the rest of the program; the exception has been caught and handled and does not crash the program.

By the way, note that the braces, { and }, are part of the syntax of the `try..catch` statement. They are required even if there is only one statement between the braces. This is different from the other statements we have seen, where the braces around a single statement are optional.

As an example, suppose that `str` is a variable of type *String* whose value might or might not represent a legal real number. Then we could say:

```
double x;
try {
   x = Double.parseDouble(str);
   System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "Not a legal number." );
   x = Double.NaN;
}
```

If an error is thrown by the call to `Double.parseDouble(str)`, then the output statement in the `try` part is skipped, and the statement in the `catch` part is executed. (In this example, I set `x` to be the value `Double.NaN` when an exception occurs. `Double.NaN` is the special "not-a-number" value for type double.)

It's **not** always a good idea to catch exceptions and continue with the program. Often that can just lead to an even bigger mess later on, and it might be better just to let the exception crash the program at the point where it occurs. However, sometimes it's possible to recover from an error.

Suppose, for example, we want a program that will find the average of a sequence of real numbers entered by the user, and we want the user to signal the end of the sequence by entering a blank line. (This is similar to the sample program *ComputeAverage.java* from Section 3.3, but in that program the user entered a zero to signal end-of-input.) If we use `TextIO.getlnInt()` to read the user's input, we will have no way of detecting the blank line, since that function simply skips over blank lines. A solution is to use `TextIO.getln()` to read the user's input. This allows us to detect a blank input line, and we can convert non-blank inputs to numbers using `Double.parseDouble`. And we can use `try..catch` to avoid crashing the program when the user's input is not a legal number. Here's the program:

```
import textio.TextIO;

public class ComputeAverage2 {

   public static void main(String[] args) {
       String str;     // The user's input.
       double number;  // The input converted into a number.
       double total;   // The total of all numbers entered.
       double avg;     // The average of the numbers.
       int count;      // The number of numbers entered.
       total = 0;
       count = 0;
       System.out.println("Enter your numbers, press return to end.");
       while (true) {
          System.out.print("? ");
          str = TextIO.getln();
          if (str.equals("")) {
             break; // Exit the loop, since the input line was blank.
          }
          try {
              number = Double.parseDouble(str);
              // If an error occurs, the next 2 lines are skipped!
              total = total + number;
              count = count + 1;
          }
          catch (NumberFormatException e) {
              System.out.println("Not a legal number!  Try again.");
          }
       }
       avg = total/count;
       System.out.printf("The average of %d numbers is %1.6g%n", count, avg);
   }

}
```

3.7.3 Exceptions in TextIO

When `TextIO` reads a numeric value from the user, it makes sure that the user's response is legal, using a technique similar to the `while` loop and `try..catch` in the previous example. However, `TextIO` can read data from other sources besides the user. (See Subsection 2.4.4.) When it is reading from a file, there is no reasonable way for `TextIO` to recover from an illegal value in the input, so it responds by throwing an exception. To keep things simple, `TextIO` only throws exceptions of type *IllegalArgumentException*, no matter what type of error it encounters. For example, an exception will occur if an attempt is made to read from a file after all the data in the file has already been read. In `TextIO`, the exception is of type *IllegalArgumentException*. If you have a better response to file errors than to let the program crash, you can use a `try..catch` to catch exceptions of type

IllegalArgumentException.

As an example, we will look at yet another number-averaging program. In this case, we will read the numbers from a file. Assume that the file contains nothing but real numbers, and we want a program that will read the numbers and find their sum and their average. Since it is unknown how many numbers are in the file, there is the question of when to stop reading. One approach is simply to try to keep reading indefinitely. When the end of the file is reached, an exception occurs. This exception is not really an error—it's just a way of detecting the end of the data, so we can catch the exception and finish up the program. We can read the data in a `while (true)` loop and break out of the loop when an exception occurs. This is an example of the somewhat unusual technique of using an exception as part of the expected flow of control in a program.

To read from the file, we need to know the file's name. To make the program more general, we can let the user enter the file name, instead of hard-coding a fixed file name in the program. However, it is possible that the user will enter the name of a file that does not exist. When we use `TextIO.readfile` to open a file that does not exist, an exception of type *IllegalArgumentException* occurs. We can catch this exception and ask the user to enter a different file name. Here is a complete program that uses all these ideas:

```
import textio.TextIO;

/**
 * This program reads numbers from a file.  It computes the sum and
 * the average of the numbers that it reads.  The file should contain
 * nothing but numbers of type double; if this is not the case, the
 * output will be the sum and average of however many numbers were
 * successfully read from the file.  The name of the file will be
 * input by the user.
 */
public class AverageNumbersFromFile {

   public static void main(String[] args) {

      while (true) {
         String fileName;  // The name of the file, to be input by the user.
         System.out.print("Enter the name of the file: ");
         fileName = TextIO.getln();
         try {
            TextIO.readFile( fileName );  // Try to open the file for input.
            break;  // If that succeeds, break out of the loop.
         }
         catch ( IllegalArgumentException e ) {
            System.out.println("Can't read from the file \"" + fileName + "\".");
            System.out.println("Please try again.\n");
         }
      }

      /* At this point, TextIO is reading from the file. */

      double number;  // A number read from the data file.
      double sum;     // The sum of all the numbers read so far.
      int count;      // The number of numbers that were read.

      sum = 0;
      count = 0;

      try {
         while (true) { // Loop ends when an exception occurs.
             number = TextIO.getDouble();
```

```
                    count++;  // This is skipped when the exception occurs
                    sum += number;
                }
            }
            catch ( IllegalArgumentException e ) {
                // We expect this to occur when the end-of-file is encountered.
                // We don't consider this to be an error, so there is nothing to do
                // in this catch clause.  Just proceed with the rest of the program.
            }

            // At this point, we've read the entire file.

            System.out.println();
            System.out.println("Number of data values read: " + count);
            System.out.println("The sum of the data values: " + sum);
            if ( count == 0 )
                System.out.println("Can't compute an average of 0 values.");
            else
                System.out.println("The average of the values:  " + (sum/count));

        }

    }
```

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