9 Exception-Handling

Overview

The situation described here illustrates the full process for developing software of significant complexity. It also illustrates the need for technical knowledge of a subject matter area in order to develop software that solves a problem in that area. However, it is not so difficult that you cannot follow it with moderate effort.

- Section 9.1 describes the problem to be solved by the Investor software.
- Sections 9.2-9.4 present the throwing and catching of Exceptions and apply those new language features to parsing numbers and using text files. These are the only new language features needed elsewhere in this book.
- Section 9.5 gives the Analysis, Test Plan, and Design of the Investor software, which uses five classes: IO, Investor, InvestmentManager, Portfolio, and Asset.
- Sections 9.6-9.7 use arrays to develop the Portfolio class.
- Section 9.8 uses statistics and Math methods for the Asset class.
- Section 9.9 is an independent section that tell about some kinds of Java statements not otherwise used in this book.

9.1 Problem Description For The Investor Software

A group of investors comes to your consulting firm to ask for help in choosing their investments. Each of them has a substantial amount of money in a tax-sheltered 401(k) plan at work, ranging from $100,000 to $500,000. The investors want to know what investment strategy will give them the best growth in value until retirement. Your job is to develop software that will help them.

The investors are in their early to mid forties. They have children who are beginning college, so they will not be able to add to their retirement plans for the next 20 years or so (even after the progeny graduate, the loans have to be paid off). On the other hand, federal law prevents them from withdrawing from the 401(k) plan until they are around age 60, except with a hefty penalty. The government makes this restriction because of the great tax benefits a 401(k), 403(b), and IRA offer: The investments grow free of taxes for as long as they remain in the retirement plan. For these reasons, none of the investors plans to add to or subtract from the retirement plans for the next 15 to 20 years.

The money in this particular 401(k) plan is invested in five different mutual funds, each with a diverse holding in a single class of financial assets. The five asset classes available are money market, short-term bonds, domestic large-cap stocks, domestic small-cap stocks, and foreign stocks. The money market fund, for instance, purchases U.S. Treasury bonds and other high-quality bonds with 3-month maturities. A bond purchased for $9,880 will repay $10,000 at the end of 3 months, which is $120 interest, which is 1.21% for the 3 months, or 4.95% annualized. Since the mutual fund holds bonds of various maturities coming due every week at least, and since repayment of the bonds is never in doubt, the value of the investment never drops in value. This is what investors mean when they talk about "cash."

The company that these investors work for only allows them to change the allocation of money among the mutual funds quarterly. That is, each three months an investor can move part or all of the money from one or more mutual funds in to other mutual funds.

Each investor currently has a certain amount of the 401(k) money in each asset class. The investors want to know how to adjust the allocation of money to the various mutual funds each quarter so they have the best results at the end of the 15 to 20 years.

The background required

Your consulting firm is able to take on this assignment only because:

(a) it has an expert on call who knows very well how 401(k) plans, mutual funds, and the financial markets work (the expert's name is Bill), and
(b) it has software designers who have a moderate understanding of those things.

The investors had previously gone to another software development firm, but the people there, though quite good at developing software, knew almost nothing about investing. Your firm knows how important it is that software designers be reasonably comfortable with the subject matter area that the software is to model. That is why your firm hired job applicants with courses in a wide range of technical and scientific fields, even if they had only moderately good grades in computer science courses. The firm ignored applicants who excelled at their computer science courses but had little breadth of knowledge.

After all, no one pays good money for just any old program, no matter how efficient and well-designed. The program has to solve a problem that people need to have solved, and that generally requires knowledge of a subject matter area such as biology, physics, chemistry, or economics. And more often than not, it requires a moderately sophisticated understanding of mathematics.

The development for this problem will become a little technical at times. That is unavoidable in much of software development. You need to see how technical knowledge is applied to produce a good software design. And this particular technical knowledge is something you should have for your own personal needs anyway. If you do not understand how long-term financial investments work, you need to learn about them. After all, what good will it do to make a lot of money as a computer scientist if you do not know anything about how to invest it for long-term growth? You do not want to have to live on peanut butter and baked beans during the several decades of your retirement.

9.2 Handling RuntimeExceptions; The try/catch Statement

The investors will enter decimal numbers in response to requests from the investor program. From time to time the user may mis-type a numeral, accidentally hitting a letter or two decimal points or some such. That makes the string of characters "ill-formed". Some coding may have the following statements:

```
String s = JOptionPane.showInputDialog (prompt);
return Double.parseDouble (s);
```

The method call `Double.parseDouble(s)` can produce the exceptional situation. If the string of characters is ill-formed, this `parseDouble` method call executes the following statement, which throws a NumberFormatException at the `askDouble` method:

```
throw new NumberFormatException ("some error message");
```

An Exception is an object that signals a problem that should be handled. As written, the `askDouble` method cannot handle the problem, so it crashes the program. The `askDouble` method should be modified to ask again for a well-formed numeric value. As another example of a throw statement, the following statement might be appropriate in a method whose parameter `par` is supposed to be non-negative:

```
if (par < 0)
   throw new IllegalArgumentException ("negative parameter");
```

Thrown Exceptions

Certain methods or operations throw an Exception when given information that does not fit their requirements. Exceptions come in two groups: RuntimeExceptions and other Exceptions. The following lists the kinds of **RuntimeExceptions** that you are most likely to see with your programs:

- **ArithmeticException** is thrown when e.g. you evaluate `x / y` and `y` is zero. This only applies to integers; with doubles, you get infinity as the answer.
- **IndexOutOfBoundsException** is thrown when e.g. you refer to `s.charAt(k)` but `k >= s.length()`. The two subtypes are ArrayIndexOutOfBoundsException for an array and StringIndexOutOfBoundsException for a String value.
- **NegativeArraySizeException** is thrown when e.g. you execute `new int[n]` to create an array and the value of `n` is a negative number.
- **NullPointerException** is thrown when e.g. you refer to `b[k]` and `b` takes on the null value, or when you put a dot after an object variable that takes on the null value. But the latter only applies when the object variable is followed by an instance variable or instance method; it does not apply for a class variable or class method.
- **ClassCastException** is thrown when e.g. you use `(Worker) x` for an object variable `x` but `x` does not refer to a Worker object.
- **NumberFormatException** is thrown by e.g. `parseDouble` or `parseInt` for an ill-formed numeral.

The RuntimeException class is a subclass of Exception. The kinds of Exceptions listed above are all subclasses of RuntimeException. All of them are in the `java.lang` package that is automatically imported into every class.

Crash-guarding an expression

When you write a method that executes a statement that can throw a RuntimeException, you should normally handle the situation in your method. The best way to handle it is to make sure that the Exception can never be thrown: Test a value before using it in a way that could throw an Exception object. For instance:

- The phrase `if (x != 0) y = 3 / x` cannot throw an ArithmeticException.
- The phrase `for (k = 0;  k < b.length && b[k] < given;  k++)` cannot throw an IndexOutOfBoundsException when `b[k]` is evaluated.
- The phrase `if (p != null && p.getAge() > 5)` cannot throw a NullPointerException.
- Nor can `if (p == null || p.getAge() <= 5)`.

These phrases use crash-guard conditions, so-called because the conditions guard against crashing the program due to an uncaught Exception. The last three listed depend on the short-circuit evaluation of boolean operators.

The try/catch statement

You can avoid all RuntimeExceptions by proper programming. But the NumberFormatException and a few others cannot be avoided easily. In those cases, it is generally easier to handle the Exception by writing a **try/catch statement** as follows:

```
try
{  // normal action assuming the Exception does not arise
}catch (Exception e)  // specify the kind of Exception
{  // special action to take if and when the Exception arises
}
```

The try/catch statement requires the beginning and ending braces for both blocks even if you have only one statement inside a block. If an Exception is thrown by any statement within the **try-block**, that block terminates abruptly and the statements within the **catch-block** are executed. In Figure 9.1, for instance, when the `result` method is called with the argument 0, no value is assigned to `quotient` and no value is returned by the method call. Instead, `y = 7` is the very next operation carried out. Note: This book puts the right brace on the same line with the `catch` keyword to remind you that `catch` is not the first word of a statement, as it does with the `while` in a do-while statement.

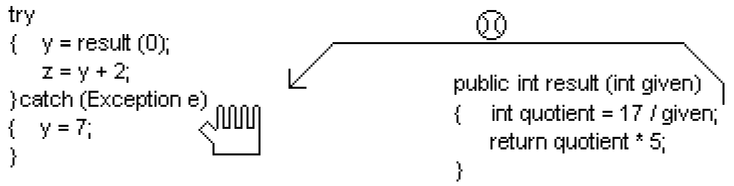

Figure 9.1 Method throwing an exception caught by a catch-block

Insisting on valid numeric input

In many cases when the user enters an ill-formed numeral, the software should repeatedly try for good input. This is done by the following logic. It has an **infinite loop**, which is normally indicated in Java by `for(;;)`. The runtime system executes the logic in the try-block. If no Exception arises, the correct value is returned. In general, the runtime system totally ignores a catch-block unless some action in a try-block throws an Exception:

```
public double getUserInput (String prompt)
{  for (;;)
      try
      {  String s = JOptionPane.showInputDialog (prompt);
         return Double.parseDouble (s);
      }catch (NumberFormatException e)
      {  prompt = "That was a badly-typed number. Try again.";
      }
}
```

The Exception is thrown if the String parameter s that is to be converted to a decimal number contains letters or other defects. In that case, the runtime system goes immediately to the catch-block and executes its logic. This prints a message and goes through another iteration of the loop. The `for(;;)` idiom is special in that the compiler does not insist you have a `return` statement in both the try and the catch blocks.

If the user clicks the CANCEL button in response to the coding above, `s` is null and so `parseDouble` throws a NullPointerException. Since no catch-block is available for a NullPointerException, the Exception will be thrown to the method that called the coding.

Multiple catch-blocks

If you know there are several kinds of Exceptions that a logic sequence could produce, and each should be handled differently, you may have more than one catch-block, as shown in Listing 9.1 (see next page). The runtime system checks each of the Exception types in order until it finds the first one that fits (the same class or a superclass of the kind of Exception that was thrown). So the assignment `x = s.charAt(5)` is not made (in the third catch-block) if 5 is outside the range of possible indexes (i.e., `b.length() <=` 5). The last catch-block in the statement catches all remaining kinds of Exception that occur.

Listing 9.1 A try/catch statement with multiple catch-blocks

```
try
{  x = s.charAt (5) / Integer.parseInt (s);
}catch (NumberFormatException e)
{  return 0;
}catch (IndexOutOfBoundsException e)
{  System.out.println (e.getMessage() + " for index 5");
}catch (ArithmeticException e)
{  x = s.charAt (5);
}catch (Exception e)
{  System.exit (0);
}
```

Note that each catch-block has what is basically a parameter of Exception type, which makes it different from any other kind of Java statement. A catch-block is "called" by the runtime system when any statement within the try-block throws its kind of Exception. All Exception objects have a method named `getMessage` that returns an informative String value, as used in the second catch-block.

The `getUserInput` method given earlier in this section could have the following second catch-block added at the end of its try-catch statement. That would cause the segment to continue asking for numeric input:

```
catch (NullPointerException e)
{  say ("I really must insist you enter a number.");
}
```

Exception-handling in the IO class

Listing 9.2 (see next page) shows a more sophisticated version of `askDouble`, `askInt`, and `askLine` than the one in Chapter Six. It uses exception-handling. This will be much better for the investor software. The `askLine` method substitutes an empty String for the null value you get when the user cancels the input window. The `askInt` and `askDouble` methods rely on this to avoid a NullPointerException. They both allow the user to press the ENTER key to indicate an entry of zero. They use the standard `trim()` method from the String class to discard any leading or trailing blanks.

Perhaps the only time you should not handle a potential RuntimeException within a method you write is when you do not know which of several possible ways to handle it, because it depends on what the calling method wants to do. For instance, a method that is to compare two objects to see which comes first, using `x.compareTo(y)`, throws a ClassCastException if the two objects are not the same type (such as one Person object and one Time object). The method cannot know what the appropriate action is. So it lets the calling method handle the Exception. The calling method should either avoid the possibility of a ClassCastException with a crash-guard or other protection, or it should have a try-catch statement taking whatever action is appropriate.

Technical Note The phrase `finally {...}` is allowed at the end of the try-catch structure. It is sometimes used to release system resources that were earlier allocated. You will almost surely have no need for this feature in your first few computer science courses.

Language elements
A Statement can be: throw SomeObjectReferenceOfSubclassOfException
A Statement can be: try {StatementGroup} catch (ClassName VariableName) {StatementGroup}
The ClassName must be a subclass of Exception. You may have more than one catch-block.

Listing 9.2 Alternative to the IO class in Listing 6.2

```
import javax.swing.JOptionPane;

public class IO
{
   private static String itsTitle = "";

   /** Change the title displayed on message dialogs. */

   public static void setTitle (String givenTitle)
   {  itsTitle = givenTitle;
   }  //======================

   /** Display a message to the user of the software. */

   public static void say (Object message)
   {  JOptionPane.showMessageDialog (null, message,
                            itsTitle, JOptionPane.PLAIN_MESSAGE);
   }  //======================

   /** Display prompt to the user; wait for the user to enter:
    *     (a) for askLine, a string of characters;
    *     (b) for askInt, a whole number;
    *     (c) for askDouble, a decimal number;
    *   Return that value; but return "" or zero on null input. */

   public static String askLine (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return s == null  ?  ""  :  s;
   }  //======================

   public static double askDouble (String prompt)
   {  for (;;)
         try
         {  String s = askLine (prompt).trim();
            return s.length() == 0 ? 0 : Double.parseDouble (s);
         }catch (NumberFormatException e)
         {  prompt = "Ill-formed number: " + prompt;
         }
   }  //======================

   public static int askInt (String prompt)
   {  for (;;)
         try
         {  String s = askLine (prompt).trim();
            return s.length() == 0  ?  0 : Integer.parseInt (s);
         }catch (NumberFormatException e)
         {  prompt = "Ill-formed integer: " + prompt;
         }
   }  //======================
}
```

Exercise 9.1 Write an example of an expression using a crash-guard if-statement for ArrayIndexOutOfBoundsException and one for StringIndexOutOfBoundsException.
Exercise 9.2 Write an example of an expression using a crash-guard for NegativeArraySizeException.
Exercise 9.3 Listing 6.1 could throw a RuntimeException in some cases. Revise it with an appropriate try/catch statement.

Exercise 9.4 Write a statement that throws a NullPointerException with the message "the parameter is null".
Exercise 9.5 If the user makes two bad entries for `askInt("Entry?")` from Listing 9.2, what are the three prompts?
Exercise 9.6* Write a test program that contains `for(;;)` and a statement after the body of that for-statement. What does the compiler say when you try to compile it? What about `while(true)`? What about `while(2+2==4)`?
Exercise 9.7* Write a version of `askInt` that has two additional int parameters named `min` and `max`. The method is to require the user to repeatedly enter a whole number `n` until it satisfies `min <= n <= max`.

9.3 Basic Text File Input; Handling Checked Exceptions

Several programs developed in this book get their input from a file, not from the user at the keyboard. First you need a Reader object. You can get one by either of these two constructor calls, which create an object of a subclass of the Reader class:

- `new FileReader ("name.txt")` connects to a disk file named "name.txt".
- `new InputStreamReader (System.in)` connects to the keyboard.

Next you need a method to read one line at a time from the input source. The BufferedReader class in the standard library provides a `readLine` method that does this; it returns null when it reaches the end of the file. The BufferedReader constructor takes a Reader object as the parameter. For instance, the following is an example of coding that counts the number of lines in a file with the name `filename`:

```
BufferedReader fi = new BufferedReader
                       (new FileReader (filename));
int lines = 0;
while (fi.readLine() != null)
   lines++;
```

The BufferedReader and InputStreamReader classes are subclasses of the Reader class, and FileReader is a subclass of InputStreamReader. All of these classes are in the `java.io` package, so you should have `import java.io.*;` above a class definition that uses these Reader subclasses.

The FileReader constructor can throw an **IOException** if the file is not there or if the file is protected against the program reading from it. And the `readLine` method can throw an IOException if the file is corrupted. This is an Exception that you have to explicitly handle it in your coding. We next develop the Buffin class, designed to isolate the exception-handling and to give an easy-to-remember way of constructing file objects. For instance, you can replace the first statement in the code segment just shown by the following, to get the same result without needing explicit exception-handling:

```
Buffin fi = new Buffin (filename);
```

The logic of the Buffin constructor

The Buffin constructor is in Listing 9.3 (see next page), together with the private field variables it needs. When you call the Buffin constructor with a String value, it tries to connect to the physical hard-disk file of the given name. If that try fails (e.g., if the String value is the empty string), the FileReader constructor throws an IOException (specifically, a FileNotFoundException). But the try/catch statement in the private `openFile` method deftly catches the throw and connects to the keyboard instead. Either way, the `openFile` method returns to the Buffin constructor an object from a subclass of Reader.

Listing 9.3 The Buffin class of objects

```
import java.io.*;

public class Buffin extends BufferedReader
{
   private static boolean temp;
   /////////////////////////////////
   private boolean isKeyboard;


   /** Connect to the disk file with the given name.  If this
    *  cannot be done, connect to the keyboard instead. */

   public Buffin (String filename)
   {  super (openFile (filename));
      isKeyboard = temp;
   }  //======================

   private static Reader openFile (String filename)
   {  try
      {  temp = false;
         return new FileReader (filename);  // IOException here
      }catch (IOException e)
      {  temp = true;
         return new InputStreamReader (System.in);
      }
   }  //======================

   /** Read one line from the file and return it.
    *  Return null if at the end of the file. */

   public String readLine()
   {  if (isKeyboard)
      {  System.out.print (" input> ");
         System.out.flush();  // flush the output buffer
      }
      try
      {  return super.readLine(); // in BufferedReader
      }catch (IOException e)
      {  System.out.println ("Cannot read from the file");
         return null;
      }
   }  //======================
}
```

The call of `super` then passes that Reader object to the BufferedReader constructor that accepts a Reader object; so the Buffin constructor in effect calls `new BufferedReader (someReader)`. The `isKeyboard` attribute of the Buffin object records whether the Reader object is the keyboard or a hard disk file.

The shenanigans with `isKeyboard` and `temp` in the Buffin class are probably puzzling. Why not assign to `isKeyboard` directly? The problem is, the `openFile` method cannot do so. The `this` object being constructed does not exist until after the return from the `super` call, therefore its instance variable `this.isKeyboard` does not exist either. So instead we store the value in `temp` until such time as `this.isKeyboard` exists.

You could legitimately wonder why a separate `openFile` method is needed. Why not have the body of the constructor be `try {super (new FileReader (filename));}...`? The reason is that the very first statement in a constructor has to be the `super` call; it cannot be a try/catch statement.

The logic of the readLine method in Buffin

The Buffin `readLine` method overrides the BufferedReader `readLine` method. It first checks whether input is coming from the keyboard. If so, it prompts with the > symbol, so the user knows the system is waiting for something to be typed. The `flush` method call is used to coordinate between input and output; it causes the prompt to appear on the screen before the user enters the input.

The Buffin `readLine` method then calls BufferedReader's `readLine` method to read one line of characters from the stream and returns it. If the input is null, that means that the end of the file has been reached. If reading throws an IOException, the try/catch statement in the `readLine` method deftly catches the throw, prints an error message on the screen, and returns null.

Technical Note If you want to write information to a disk file, the simplest way is to use `System.out.println` messages and re-direct the output. For instance, the output to `System.out` from a program named X will be re-directed to a file named `answer.txt` if you start the program running with the following command in the terminal window:

```
java X > answer.txt
```

The Exception class

The Exception class is the superclass of all the various Exception types. The standard Sun library has the following definition:

```
public class Exception extends Throwable
{  public Exception()
   {  super();
   }  //======================
   public Exception (String s)
   {  super(s);
   }  //======================
}
```

The string of characters in the second constructor is a message that describes the kind of error that occurred. The **Throwable** class, the superclass of the Exception class, has a `getMessage` method that returns that String value. So if `e` is the Exception parameter of a catch-block, you can use `e.getMessage()` within a catch-block to access the message.

The subclasses of the Exception class that you are most likely to encounter during execution of a program are RuntimeException, InterruptedException, and IOException. An **IOException** can occur when you attempt to open a file of a given name and no such file exists, or when you try to write a value to a file that is locked against changes, or when you try to read something more from a file after you have already read everything in it. An **InterruptedException** can occur when a thread is put to sleep (discussed later in this chapter).

Checked Exceptions

All subclasses of Exception that are not RuntimeExceptions are called **checked Exceptions**. That includes the IOException class. The general rule is that the compiler requires a method to have coding that does something about an Exception if:

- That method calls a method that can throw a checked Exception, or
- That method contains a statement that can throw a checked Exception.

That method must either have a try/catch statement that handles the Exception that might be thrown or have the **throws clause** `throws SomeException` at the end of its heading. The SomeException specified in the heading must be the kind of checked Exception you are to handle or else a superclass of it.

When you write a method with a throws clause, you are passing the buck, i.e., throwing the checked Exception at whatever method called your method. This is often called "propagating the Exception." Then that calling method has to have logic that does something about it, again one of the two options just mentioned. If your method is the main method, however, the program crashes after printing the error message.

An example of throwing an Exception further is in the following independent class method, assuming that a call of `printOneItem` can throw a PrinterAbortException (when the user cancels the print job prematurely):

```
public static void printStuff (Object[] toPrint)
            throws java.awt.print.PrinterAbortException
{  for (int k = 0;  k < toPrint.length;  k++)
      printOneItem (toPrint[k]);
}  //=====================
```

You may put a `throws` clause in the heading of a method that throws a RuntimeException if you think it is clearer. But a method that calls it is still not required to handle it, since it is not a checked Exception. General principle: A RuntimeException should never occur in a correct program (except NumberFormatException), but a checked Exception can arise for reasons other than a bug in the program (e.g., no such file). So you rarely try/catch a RuntimeException other than NumberFormatException.

Language elements
A MethodHeading can have this phrase at the end: throws ClassName
The ClassName must be a subclass of Exception.
You may list several ClassNames in the throws clause, separated by commas.

Exercise 9.8 If you call the Buffin constructor to open a file that cannot be opened, you get the keyboard instead. You cannot easily tell whether this happened. Write a Buffin method `public boolean checkError()` that returns `true` if that happened and `false` if it did not (this is the way the PrintWriter class handles exceptional cases).
Exercise 9.9 Revise the `printStuff` method to catch a PrinterAbortException, print the message it contains, and immediately return from the method.
Exercise 9.10 Revise the `printStuff` method to catch a PrinterAbortException, print the message it contains, and continue printing the rest of the values.
Exercise 9.11* (a) Could the Buffin class be redesigned so that temp is an instance variable and it still accomplishes what it should? (b) If it can, would that be better or worse than the current design? (c) Same two questions, except for temp as a local variable.
Exercise 9.12* Revise Listing 9.3 so that the instance variables are `isKeyboard` and a BufferedReader object, and Buffin does not extend anything but the Object class.
Exercise 9.13* Search the java documentation to find two new kinds of checked Exceptions. Describe the circumstances in which they are thrown.

Part B Enrichment And Reinforcement

9.4 Throwing Your Own Exceptions

Suppose you are to develop a method that is to process its parameters and its executor in a certain way. However, certain input values can make proper processing impossible. This method is called from several different places, and the way the exceptional situation should be handled depends on the place from which your method is called.

This is often the situation when you are writing a method as part of a library class that will be used by several programs that others have written or will write in the future. In this kind of situation, you should usually throw an Exception. That definitely solves your problem of how to handle the special situation, because throwing an Exception terminates abruptly all processing of the method that throws it. So all further processing within your method can be done with the knowledge that the exceptional case has disappeared.

Your first consideration is whether the troublesome situation could have been easily guarded against by the method that called your method. If so, you should normally throw a RuntimeException; if not, you should normally throw a checked Exception. The difference is this: If you throw a checked Exception, any method that calls your method must have a try/catch statement to handle it (or pass it higher), even if the calling method guarded against the exceptional case and so could not possibly cause an Exception.

The Thread.sleep command

For example, Java has a statement that pauses the current thread of execution for a given number of milliseconds (Threads are discussed in detail at the end of Chapter Eleven):

```
Thread.sleep (300)
```

This causes a pause of three-tenths of a second. If you are running several programs at once on your computer, this `sleep` command frees up the computer chip for those other threads of execution. By contrast, a busywait using a looping statement that executes while doing nothing ties up the computer chip. Two examples of busywait are the following loops:

```
while (System.currentTimeMillis() < longTime)
{  }
for (int k = 0;  k < 5000000; k++)
{  }
```

The `Thread.sleep` command can throw a checked Exception, namely, an **InterruptedException**. This happens when some other thread of execution within your program "wakes up" your thread before the specified time is up. You are forced to have a try/catch statement for this Exception even when your program does not have any other threads of execution:

```
try
{  Thread.sleep (someMilliseconds);
}catch (InterruptedException e)
{  // no need for any statements here
}
```

It would be much pleasanter to be able to write just one line here instead of six. But you cannot because InterruptedException is not a RuntimeException. It was set up this way because, if you have a program with several threads of execution, and you put one of them to sleep, there is usually no way for you to guard against one of those other threads interrupting your sleeping thread. By contrast, you can easily guard against a NullPointerException with a reasonable amount of care.

```
java.lang.Exception
     java.lang.InterruptedException
     java.io.IOException
          java.io.EOFException
          java.io.FileNotFoundException
     java.io.print.PrinterException
          java.io.print.PrinterAbortException
     java.lang.RuntimeException
```

Figure 9.2 Partial listing of the Exception hierarchy

How to throw an Exception

The simplest way to throw an Exception and avoid a troublesome situation is to use one of the following two statements (the first when you want a checked Exception):

```
throw new Exception (someExplanatoryMessage);
throw new RuntimeException (someExplanatoryMessage);
```

The explanatory message is a String value. Almost all subclasses of Exception have two constructors, one with a String parameter and one not. You use the one without the parameter if you feel that the name of the Exception is enough information. In a catch-block, you may call `e.getMessage()` to retrieve the message that `e` has, assuming `e` is the name you choose for the Exception parameter of the catch-block.

For example, the Worker constructor in Listing 7.6 returns a worker with a null name if the input String `s` is null. It could instead throw a checked Exception as follows:

```
if (s == null)
   throw new Exception ("The string does not exist");
```

The constructor would therefore have to have the phrase `throws Exception` at the end of its heading as follows. Then any method that calls it must handle the Exception:

```
public Worker (String s) throws Exception
```

The main methods in Listing 7.4 and Listing 7.7 call that constructor, so they would also have to have revised headings as follows, unless you rewrite the logic with a try/catch statement:

```
public static void main (String[] args) throws Exception
```

If you wanted a WorkerList to have a method that returns the last Worker in the list, it could be written to throw an unchecked Exception for an empty list as follows:

```
public Worker getLastWorker() throws RuntimeException
{  if (itsSize > 0)
      return itsItem [itsSize - 1];
   else
      throw new RuntimeException ("The list is empty!");
}  //======================
```

Note that this logic does not execute a return statement in both branches. You are only required to execute either a `return` or a `throw` statement in all branches of an if-statement at the end of a method that is to return a value. The phrase `throws RuntimeException` is optional in the heading of this revised `getLastWorker` method, because it is an unchecked Exception.

Creating a subclass of Exception

You could create your own class of Exceptions if you wish. You should subclass either Exception or RuntimeException, depending on whether you want it checked. And you should have the two standard constructors that almost all Exception subclasses have. For instance, the last statement of the `getLastWorker` method just given could be either `throw new BadPersonException("The list is empty!")` or `throw new BadPersonException()` if you have the following class definition:

```
public class BadPersonException extends RuntimeException
{  public BadPersonException()
   {  super();
   }  //======================
   public BadPersonException (String message)
   {  super (message);
   }  //======================
}
```

You are allowed to extend other subclasses of Exception when you define your own Exception class. But it is simplest to make all the Exception classes you define just like the BadPersonException class just shown, with two possible differences: (a) replace "BadPerson" by whatever you want in three places, and (b) either keep or omit "Runtime" in the heading.

Exercise 9.14 Define a standard BadStringException class that is a checked Exception.
Exercise 9.15 Revise Listing 6.4 to have the `trimFront` and `firstWord` methods throw a BadStringException when the String value has zero characters.
Exercise 9.16 Revise the `yearsToDouble` class method in Listing 6.1 to throw an ArithmeticException when the interest rate is not positive. Modify the method heading as needed.
Exercise 9.17* Revise the `parseDouble` method of Listing 6.5 to throw a BadStringException when the string does not contain a digit or the first non-whitespace character other than '-' or '-.' or '.' is not a digit.

9.5 Analysis, Test Plan, And Design For The Investor Software

Begin the analysis by studying the statement of what the clients want, to see where the description is incomplete or self-contradictory. Then ask the clients questions to find out what they really want. Some such questions are as follows, with the clients' answers:

Question Will the software be used for any investments other than the 401(k) money?
Answer The investors reveal that they have IRA accounts in addition to their 401(k) plans. The IRAs do not have enough in the accounts to make it worthwhile to buy individual stock holdings, so they are also invested only in mutual funds. The IRA mutual funds allow trades once each day. So allow for daily price changes of assets.

Question Will the software have to allow for taking money out of the accounts?
Answer Once they retire, the investors expect to have regular monthly withdrawals for living expenses. Also, they may start regular monthly additions to the 401(k) accounts and IRA accounts in perhaps 15 years, to build up their retirement funds in the five years or so before they retire. So allow for a fixed-amount addition to or subtraction from a portfolio of mutual funds each month.

Question How are the mutual fund holdings affected by these changes?
Answer Each deposit is to be added to the mutual funds in proportion to the current holdings. Similarly, each withdrawal is to be taken from the mutual funds proportionally.

Question How exactly does a mutual fund works?
Answer The mutual fund buys or sells some securities every day as investors move money into or out of the fund. For instance, if the fund has $500 million at the end of one day, and an investor has $50,000 invested in that fund, then that investor owns 0.01% of the assets of that fund as of the close of business that day. If the fund's assets have a value of $505 million at the end of the next day, the fund managers take out say $10,000 for their expenses and salaries and profits, leaving $504,990,000 net value in the mutual fund. The $4,990,000 of profit belongs to the individual investors in proportion to what each owned. In particular, the person who had $50,000 in the fund the day before sees the value of that investment go up by $499.

Mutual funds change in price once each trading day, conventionally at 4 p.m. when the normal markets close, and a year has roughly 252 trading days, allowing for Wall Street holidays. You will need an algorithm for simulating the change in value for the day, which will give a different result for each of the mutual funds. A simulation is needed because no one knows what the markets will do, in terms of prices of investments rising and falling. "Simulation" in this case means educated guesswork.

Question How should the change in price each day be modeled?
Answer The clients have no idea. Bill the expert will research historical trends, develop good formulas for this calculation with random numbers, then check back with the clients with proposals they can accept or modify. He will give you a very rough approximation that you can use until he finishes working this out with the clients. So until Bill determines acceptable algorithms, you are essentially developing a prototype of the program.

Average returns

If an investment for a 3-year period has e.g. annual returns of 7%, 31%, and -13%, respectively, then the total value at the end of the three years is calculated by first converting to the annual multipliers of 1.07, 1.31, and 0.87, then multiplying them together to get 1.22. So the total 3-year return is 22%. In general, we use these annual multipliers in calculations rather than the vernacular percentages (note that adding the percentages and getting the total of 25% does not give us any meaningful value).

An **average return** is not found by dividing the 22% (let alone the 25%) by 3. Instead, you figure out what multiplier you could have used every year to get the same result. Since 106.85% to the third power is 122%, the average return is 6.85%. This way of calculating an average is called the **geometric average**.

Tax-free growth

An IRA or 401(k) plan offers investment growth of your after-tax investment essentially free of income taxes (except for a non-deductible IRA). They require that the money invested come from earned income (rather than e.g. interest earnings) and that you do not withdraw it until you are close to retirement age. There are two basic kinds of good IRAs, Roth IRA and deductible IRA, depending on whether the taxes on the money invested are paid when you deposit the money or when you withdraw it.

Suppose you can spare $700 to invest in after-tax money, and you invest it in stocks using an IRA when your combined federal and state income tax rate is 30%. If you let the money grow for 25 years before you spend it, it could reasonably be expected to increase by a factor of perhaps 10. The two main kinds of tax shelters are:

- You can put the $700 in a Roth IRA. It grows to about $7,000, all of which you get to spend tax-free.
- You can put $1000 in a deductible IRA or a 401(k) or a 403(b) and get $300 back immediately on your income taxes. It grows to about $10,000, of which $3,000 is due to the $300 deferred taxes and $7,000 is due to your after-tax cost of $700. When you cash it in and pay taxes at the 30% combined tax rate, you simply give the government its $3,000 and you keep the $7,000 that your $700 became.

Either way, the out-of-pocket cost to you is the same, namely, $700. And the end result is the same, namely, $7,000. Your $700 out-of-pocket cost grows tax free either way. The only difference is when you give the government its due.

There are other considerations that indicate what blend of Roth and deductible tax-shelters you should have. A crude approximation is that money on which you expect to pay tax at a substantially lower rate when you retire should be in a deductible tax-shelter, and all other money should be in a Roth IRA. The reasons are (a) if the tax rate is lower, you get to keep part of the government's $3,000 (e.g., if your combined rate in retirement is only 20%, your taxes are $2,000, so you get an extra $1,000); (b) the Roth IRA has better features than a deductible IRA in most other respects.

Planning the tests

Bill the expert tells you that a particular mutual fund will have two multipliers to describe its long-term behavior. One is its daily average and the other is its daily volatility. The daily volatility measures how much fluctuation there is in the day-to-day returns. These two values for the money market could be 1.0002 (averaging somewhat more than 5% annually) and 1.0001 (very low volatility).

The simplest test of the software is to put $10,000 in just one mutual fund and let it ride for 20 years. Calculate by hand what the **expected result** would be, which is the daily average multiplier to the power 252*20 ("by hand" means with a spreadsheet or calculator, of course). Run the test 10 or 20 times, then check that the software gives you values distributed somewhat evenly around the expected result. Do this for at least three of the available mutual funds.

A second test is to split $10,000 evenly among the five mutual funds, $2,000 each. At the end of each month (which is 21 trading days), **rebalance** whatever money you have as 20% in each of the five mutual funds. Repeat this for 20 years. Run this test 10 or 20 times, then check that the software gives you values distributed somewhat evenly around the expected result. The **expected result** is found as follows:

1. Calculate each of the five daily average multipliers to the power 21 to get the five monthly averages;
2. Find the arithmetic average of those five monthly averages;
3. Raise it to the power 12*20 (the total number of months).

Each of those tests can be redone with the regular addition of say $200 per month to the account. In the case of the rebalancing test, the $200 would be split evenly among the five mutual funds when the deposit is made. Then the tests can be done again with the regular subtraction of say $50 per month.

Main logic design of the Investor software

The overall logic of the Investor program is fairly simple:

1. Create a portfolio of five mutual funds with the initial allocations that the user specifies.
2. Allow many days to pass, updating as needed for market changes and the user's decisions.

You need to break this logic down into separate steps. When the program begins operation, it should print out a summary of what each of the five asset classes is. Then it should find out how much the user has in the account and how it is initially distributed. It is probably simplest to first add the entire amount to the money market account and then ask the user to reallocate it among the five asset classes. The program should also ask how much the user is adding to the account each month.

The program could ask the user each day for the desired allocation among the five asset classes, then report the value in each asset class one day later, then repeat the process. But that means `252*20` inputs from the user for a 20-year period. This is surely unacceptable to the user. At a very optimistic 3 seconds per input, that is still over 4 hours for one run of the program.

It would be far better to let the user say how many days to let the money ride before a new decision about allocations is made. Then the program can calculate the change in value of each asset class over that many days, report those values, then repeat the process. That way, if a speculator wants to make allocation decisions once a week, the waiting period can be 5 trading days. If an investor wants to make allocation decisions once each quarter, the waiting period can be 63 days (one-fourth of the 252 trading days per year).

The program should make it very easy to give a default allocation of leaving the money ride by just pressing the ENTER key, and a default number of trading days to wait also by pressing the ENTER key. Probably the most useful default number of trading days is whatever the investor chose the previous time. You go back to the client to see if this is acceptable (sometimes revisions in the analysis come during the test-plan and design steps). You rework this preliminary design as shown in the accompanying design block (see next page).

The object design

You see that there are three kinds of objects mentioned in the top-level design: a single asset class, a portfolio consisting of five asset classes, and the user. You look carefully at what each object has to be able to do -- its operations. The user will give you the input and receive the output, which is what `IO.askDouble` and `IO.say` do. That is, the IO class from Listing 9.2 can be in essence the virtual user.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

```
1.  Create a portfolio of the 5 asset classes with no money in it.
2.  Describe each one of the 5 asset classes (name, bad year, average year, good year).
3.  Ask the user for the initial balance and the monthly deposit.
4.  Repeat the following until the user says to stop...
            4a.  Ask the user whether assets are to be reallocated; if yes, then...
                     Find out what the reallocations are and make them.
            4b.  Ask the user how many days to wait until the next reallocation decision.
            4c.  Change the amounts in the five asset classes for that many days.
            4d.  Display the amounts in each asset class at the end of that period.
            4e.  Ask if the user wants to stop this processing loop.
5.  Terminate the processing.
```

What if the user enters a negative number when asked for the number of days? That must be a mistake, and it is not mentioned in the design. This a defect has to be corrected at this point. You would find it easiest to just pretend that the user pressed the ENTER key in that case. You check with the clients and they say that is acceptable.

Preliminary designs of the other two kinds of objects, derived from the main logic design, are in Listing 9.4 (see next page) as classes with stubbed methods. The methods for the Portfolio class are designed to match up with the steps in the main design fairly closely. The methods in the Asset class follow from the Portfolio methods:

- To construct a Portfolio object you must construct several Asset objects.
- To describe a whole Portfolio object you must describe the individual Asset objects.
- To let a number of days pass, you must let one day pass at a time for each Asset object.

The Asset class is to be used only with a Portfolio class, so the Asset class is not declared as a public class. This allows the convenience of putting it in the same `Portfolio.java` file. A text file to be compiled in Java can contain several separate classes, as long as only one is public (other than classes inside other classes); the name of that public class has to be the name of the text file.

The constant values for one month of trading days and one year of trading days will be used in the Asset class (to convert between daily and annual multipliers), and in the Portfolio class (for calculations involving the monthly deposit), as well as probably the program itself. Since the program uses the Portfolio class which uses the Asset class, it makes sense to put these constants in the Asset class.

Exercise 9.18 If you let an investment ride for three years, and it makes 50% each of the first two years and loses 50% the third year, what is your average return?
Exercise 9.19 How much would $700 grow to after 20 years if it earned 10% every year in a Roth IRA? (use a spreadsheet or calculator for this exercise and the next).
Exercise 9.20 How much would $700 grow to after 20 years if it earned 10% every year in an investment subject to 30% taxes each year? What is the ratio of your answer to that of the preceding exercise?
Exercise 9.21 If you invest your IRA in a way that earns 7% every year for 13 years, and Jo makes a taxable investment on which tax is paid each year at a 30% combined tax rate, at what rate does Jo have to earn in order to keep up with you in after-tax value? (You can do this in your head).

Listing 9.4 Description of the Portfolio and Asset classes

```
public class Portfolio     // stubbed version
{
   /** Construct portfolio with five standard asset classes. */
   public Portfolio()                                          { }

   /** Return description of all available investment choices. */
   public String describeInvestmentChoices()     { return null; }

   /** Add the given amount to the Money Market fund.  */
   public void addToMoneyMarket (double amountAdded)           { }

   /** Accept a reallocation of investments by the user.  */
   public void reallocateAssets()                              { }

   /** Make the given amount the monthly deposit.  */
   public void setMonthlyDeposit (double deposit)              { }

   /** Update for changes in price levels for several days.  */
   public void letTimePass (int numDays)                       { }

   /** Return the values for each holding.  */
   public String getCurrentValues()              { return null; }
}
//##############################################################


class Asset                // stubbed version
{
   /** Number of trading days in one month and in one year. */
   public static final int MONTH = 21;
   public static final int YEAR  = 12 * MONTH;
   ///////////////////////////

   /** Create 1 Asset object with the given daily multipliers. */
   public Asset (String name, double avg, double volatility)  { }

   /** Describe the asset to the user in one line of output, in
    *  terms of a good year, an average year, and a bad year. */
   public String description()                   { return null; }

   /** Return the multiplier for the next day's market activity,
    *  e.g., 1.02 for a 2% increase. */
   public double oneDaysMultiplier()                { return 0; }
}
```

Exercise 9.22* For the preceding problem, call the answer X%. If Jo invested in something with a fluctuating annual return averaging X%, would Jo keep up with you? How do you know? If Jo's annual return were fixed at X% and yours fluctuated but still averaged 7%, would you keep up with Jo?
Exercise 9.23* Do some research to find out why non-deductible IRAs and annuities are far worse than either deductible IRAs or Roth IRAs.

9.6 Version 1 Of Iterative Development

The main logic begins by creating a Portfolio object. Then it prints out a description of the five asset classes. This tells the user what kind of results can be expected in good years and bad. The main logic is overly complex, so you should have a separate InvestmentManager object to carry out some of the subtasks. The initialization of this portfolio for the amount invested, and the determination of the amount of the monthly deposit or withdrawal, is a separate well-defined task, so it makes sense to have an InvestmentManager `agent` get this initial setup. This cuts down on the length of the main method, which makes it clearer.

In fact, the whole process of getting user input and updating the portfolio should be kept in a place other than the `main` method, since `main` is not reusable (because of the `System.exit(0)` command required when you use the JOptionPane methods). The InvestmentManager's jobs are tailored for the Investor class, so we can make it a non-public class to be compiled in the same file as the Investor class. The coding for the Investor class containing the `main` method is in Listing 9.5.

Listing 9.5 The Investor class application

```
public class Investor
{
   /** Ask the user how much money to invest and how much to
    *  add or subtract monthly.  Then process a sequence of
    *  investment decisions for as long as the user wants.
    *  Developed by Dr. William C. Jones       January 2001 */

   public static void main (String[] args)
   {  IO.say ("This program tests your success at investing.");
      Portfolio wealth = new Portfolio();
      System.out.println (wealth.describeInvestmentChoices());

      InvestmentManager agent = new InvestmentManager();
      agent.askInitialSetup (wealth);
      agent.processInvestmentDecisions (wealth);
      System.exit (0);
   }  //======================
}
```

The InvestmentManager methods

The `askInitialSetup` method is sequential logic: (a) Ask for the total assets invested; (b) Put it (for now) in the money market ("cash"); (c) Ask for the amount to be deposited each month; (d) Record that deposit amount. This will be a negative number if the user is making monthly withdrawals. It will be zero if the user is making no changes.

Asking the user for one of three choices -- reallocate/stop/continue -- and checking the value to make sure it is acceptable, is a little complex. That task can be relegated to a separate method. For the main logic, you use `IO.askInt` to ask for the number of days to wait for the next decision. `IO.askInt` returns zero if the user just presses the ENTER key. So you need to get the result of `IO.askInt` and see if it is positive. If so, you may store it in the variable that keeps track of the number of days to wait, otherwise you leave that variable with its earlier value. The initial value for this variable should be 21, the number of days in one month. A reasonable implementation is in Listing 9.6 (see next page).

Listing 9.6 The InvestmentManager class

```
class InvestmentManager  // Helper class for Investor application
{
   /** Get the starting values for the portfolio.  */

   public void askInitialSetup (Portfolio wealth)
   {  double value = IO.askDouble ("Invest how much initially?");
      wealth.addToMoneyMarket (value);
      IO.say ("Initially it is all in the money market.");
      value = IO.askDouble ("How much do you add each month "
               + "(Use a negative number for a withdrawal)? ");
      wealth.setMonthlyDeposit (value);
   }  //======================

   /** Repeatedly get user input on re-allocating investments
    *  and update the portfolio's status correspondingly. */

   public void processInvestmentDecisions (Portfolio wealth)
   {  int days = Asset.MONTH;  // trading days in one month
      char choice = this.askUsersMenuChoice();
      while (choice != 'S' && choice != 's')  // S means STOP
      {  if (choice == 'Y' || choice == 'y')
            wealth.reallocateAssets();
         int answer = IO.askInt ("How many days before the "
                 + "next decision \n(ENTER if no change)? ");
         if (answer > 0)   // ENTER returns zero
            days = answer;
         wealth.letTimePass (days);
         System.out.println (wealth.getCurrentValues());
         choice = this.askUsersMenuChoice();
      }
   }  //======================

   /** Get the first letter of user's choice from the menu.  */

   public char askUsersMenuChoice()
   {  String input = IO.askLine ("Enter YES (or anything "
               + "starting with Y or y) to reallocate assets; "
               + "\nEnter STOP (or anything starting with S "
               + "or s) to stop); or"
               + "\nPress ENTER or anything else to continue:");
      return  (input.length() == 0)  ?  ' '  :  input.charAt(0);
   }  //======================
}
```

The `askUsersMenuChoice` method gets the user's decision at the end of the chosen wait period. Begin by displaying the three choices: stop the program, reallocate assets and continue for another period, or leave the assets allocated as they are and continue for another period. If the user simply presses the ENTER key, that should be interpreted as leaving things ride for another period. Figure 9.3 shows the UML class diagram for the Investor class.

A portfolio consists primarily of five Asset objects. All five of the Asset classes are treated the same, except that the money market is sometimes handled differently. So one Portfolio instance variable should be an array of Asset objects, with the money market stored in component zero for convenience. You also need to keep track of how much money is stored in each asset class, so another Portfolio instance variable should be an array of double values for this money information.

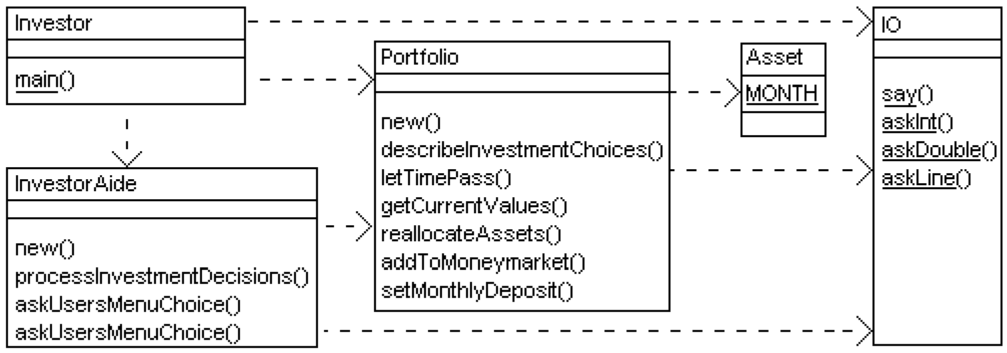

Figure 9.3 UML class diagram for the Investor program

You could call these two variables `itsAsset` and `itsValue`. Then `itsValue[0]` is the amount of money currently invested in the money market `itsAsset[0]`, `itsValue[1]` is the amount of money currently invested in `itsAsset[1]`, etc. These two arrays are called **parallel arrays**, because the kth component of one tells you something about the kth component of the other. Parallel arrays do not often occur in Java because, more often than not, a single array of objects that combine both parts is preferable. In the present situation, however, calculations involving a single asset will probably be complex enough without involving the current value of the holding. That is, the Asset class has enough to do if it just deals with the nature of the asset; the Portfolio class can track the current values.

The Portfolio constructor initializes the array values. The `itsValue` array defaults to all zeros, but it is clearer to explicitly state this as an initializer list `{0,0,0,0,0}`. For this first version of the software, you could simply keep all the investments in the money market fund. This really simplifies the development. So you may as well use the same average return and volatility for all five investments; for now, the only difference between any two Asset objects is in the name. A typical statement in the constructor is:

```
   itsAsset[2] = new Asset("large caps", 1.0002, 1.0001)
```

The second parameter is the daily average of returns (0.02% daily) and the third parameter is the daily volatility of returns (plus or minus 0.01% daily). These values are unrealistic, because different asset classes have different averages and volatilities. So these values will be changed when Bill the expert reports back with his conclusions.

Six Portfolio methods

The `describeInvestmentChoices` and `reallocateAssets` methods can be left undone for now; there is nothing to describe or reallocate in this Version 1, since all assets are kept in the money market. The `addToMoneyMarket` method simply adds the parameter value to `itsValue[0]`; the other values in that array are initially zero. And the `getCurrentValues` method simply prints out the value of the money market.

When you consider the `setMonthlyDeposit` method, you realize that a Portfolio object needs an instance variable to keep track of this amount so it can use it every 21 days. And a Portfolio object also needs to keep track of how many days have passed since it was first created, so it knows when the next 21-day period has passed.

These Portfolio instance variables could be called `itsMonthlyDeposit` and `itsAgeInDays`. You could initialize `itsMonthlyDeposit` to zero for every newly constructed Portfolio object, indicating that no deposits or withdrawals are made monthly. To simplify Version 1, ignore the monthly deposit in the calculations. You can also initialize `itsAgeInDays` to zero in the variable declaration, since it will always be zero at the time a Portfolio object is created.

The `letTimePass` method is to find the effect of waiting a given number of days. Since all assets are in the money market, you only need to multiply the assets by the daily interest rate multiplier of 1.0002 once for each day that passes. The standard library method call `Math.pow(amt,power)` gives the exponential amt^{power}.

The Portfolio class so far is in Listing 9.7; the last four methods listed will require further work in the next version of the software. You can use the stubbed version of the Asset class in the earlier Listing 9.4 as is for this Version 1 of the software.

Listing 9.7 The Portfolio class, Version 1

```
public class Portfolio
{
   public static final int NUM_ASSETS = 5;
   /////////////////////////////////
   private Asset[]  itsAsset = new Asset[NUM_ASSETS];
   private double[] itsValue = {0,0,0,0,0};
   private double   itsMonthlyDeposit = 0;
   private int      itsAgeInDays = 0;

   public Portfolio()
   {  itsAsset[0] = new Asset ("money mkt   ", 1.0002, 1.0001);
      itsAsset[1] = new Asset ("2-yr bonds  ", 1.0002, 1.0001);
      itsAsset[2] = new Asset ("large caps  ", 1.0002, 1.0001);
      itsAsset[3] = new Asset ("small caps  ", 1.0002, 1.0001);
      itsAsset[4] = new Asset ("emerg mkts  ", 1.0002, 1.0001);
   }  //======================

   public void addToMoneyMarket (double amountAdded)
   {  itsValue[0] += amountAdded;
   }  //======================

   public void setMonthlyDeposit (double deposit)
   {  itsMonthlyDeposit = deposit;
   }  //======================

   public String describeInvestmentChoices()
   {  return "No description of choices for Version 1";
   }  //======================

   public void reallocateAssets()
   {  IO.say ("No reallocation of assets for Version 1");
   }  //======================

   public void letTimePass (int numDays)
   {  itsAgeInDays += numDays;
      itsValue[0] *= Math.pow (1.0002, numDays);
   }  //======================

   public String getCurrentValues()
   {  return itsAgeInDays + ", " + itsValue[0];
   }  //======================
}
```

Exercise 9.24 Revise the `askInitialSetup` method in Listing 9.6 to force the user to enter a positive number for the initial investment.
Exercise 9.25** Write an IO method `public static int askChar()` that returns the first character of the line of input, capitalizing it if it is a lowercase letter. It should return a blank if the user simply presses the ENTER key. Then rewrite the logic of the investor classes to use this method appropriately.

9.7 Version 2 Of Iterative Development

You should not try to do too much at one time when developing the implementation of a program:

- Set a reasonable target that will give meaningful results when the program is run.
- Develop the logic to reach that target, which is Version 1 of the iterative development.
- Test the program thoroughly to make sure it does what it is supposed to do at that point.
- Repeat this partial development and testing through several additional versions until the program is complete.

This program has so much complexity already that you should try for the minimum possible additional logic in Version 2 to give a useful result. So you decide to have Portfolio's `letTimePass` method simply add `itsMonthlyDeposit` to the money market fund. You also ignore negative asset values. Now the logic can be worked out as shown in the accompanying design block. The coding is in the upper part of Listing 9.8 (see next page).

DESIGN for the letTimePass method
```
1. For each of the numDays days do...
        1a. Get that day's multiplier from each Asset object and multiply it by the
            current value in that asset class.
        1b. Add 1 to itsAgeInDays.
        1c. If itsAgeInDays is a multiple of 21, then...
                1cc. Add the amount of the constant monthly deposit to the value of the
                     money market fund.
```

The getCurrentValues method

The user will want to be able to look at the status of the holdings for the last few times a reallocation was made. To have so much data on the screen at once, you print to the terminal window. The `getCurrentValues` method provides a title for the column of five lines and then the name and current value of each of the five asset classes. This makes you realize that you need to add a `getName()` method to the Asset class. The values printed should be in columns, so the tab character `'\t'` is needed.

Also, investors will not be interested in the amount to the exact penny (or beyond), since they have several thousand dollars in each investment. So the values should be rounded to the nearest dollar. The standard way to do that is to add 0.5 and then truncate with the `(int)` cast. The coding for the `getCurrentValues` method is in the middle part of Listing 9.8.

Listing 9.8 Revisions of four Portfolio methods for Version 2

```
   /** Update the status for price changes for several days. */

   public void letTimePass (int numDays)
   {  for (;  numDays > 0;  numDays--)
      {  for (int k = 0;  k < NUM_ASSETS;  k++)
            itsValue[k] *= itsAsset[k].oneDaysMultiplier();
         itsAgeInDays++;
         if (itsAgeInDays % Asset.MONTH == 0)
            itsValue[0] += itsMonthlyDeposit;
      }
   }  //======================

   /** Return the values for each holding.  */

   public String getCurrentValues()
   {  String s = "\nname \t\tcurrent value \n";
      for (int k = 0;  k < NUM_ASSETS;  k++)
         s += itsAsset[k].getName() + "\t$"
                          + (int) (itsValue[k] + 0.5) + "\n";
      return s;
   }  //======================

   /** Return description of all available investment choices. */

   public String describeInvestmentChoices()
   {  String s = Asset.getHeading() + "\n";
      for (int k = 0;  k < NUM_ASSETS;  k++)
         s += itsAsset[k].description();
      return s;
   }  //======================

   /** Accept a reallocation of investments by the user.  */

   public void reallocateAssets()
   {  IO.say ("How will you invest your current assets?"
               + "\nUnallocated amounts go in a money market.");
      for (int k = 1;  k < NUM_ASSETS;  k++)
      {  double amount = IO.askDouble ("How much in "
                         + itsAsset[k].getName() + "? ");
         itsValue[0] += itsValue[k] - amount;
         itsValue[k] = amount;
      }
      IO.say ("That leaves $" + (int) (itsValue[0] + 0.5)
                    + " in the money market.");
   }  //======================
```

The describeInvestmentChoices method

The `describeInvestmentChoices` method calls `itsAsset[k].description` for each of the Asset objects in order, after giving an appropriate heading. The description of an Asset object should include possible outcomes of investing in that asset: what one could expect in a good year and in a bad year, as well as in an average year.

The Asset class "knows" how many values it will display and what they mean, and this may change in future revisions. So you should ask the Asset class to provide appropriate headings for the columns of values. You need to add a `getHeading` method to the Asset class for this purpose. This method should be a class method, since the headings do not depend on any particular Asset object's status. That is, you are asking a question of the Asset class as a whole, not of any particular Asset object.

The reallocateAssets method

For the `reallocateAssets` method, go through each of the asset classes one at a time, skipping the money market fund (cash is special). For each non-cash asset class, ask the user for the amount that is to be in it. Then replace `itsValue[k]` by that amount (`k` ranging 1 through 4). You also need to add to or subtract from the money market fund the amount of change in `itsValue[k]`. Listing 9.8 gives Version 2 of these four methods for the Portfolio class.

Note that you are changing the object design, which you may have thought was complete several stages ago. This just means that the Asset class is required to provide certain additional services that were not apparent from the main logic alone.

Exercise 9.26 (harder) Modify Portfolio's `letTimePass` method to charge 5% higher interest when the money market balance is negative. This roughly corresponds to interest rates on margin loans. Hint: You need to add `0.05 / Asset.YEAR` to get 5% annual interest compounded daily.
Exercise 9.27* Modify Portfolio's `getCurrentValues` method to print out with the heading, "After Y years, M months, and D days:" using the correct values for Y, M, D.
Exercise 9.28** Modify Portfolio's `letTimePass` method to distribute the monthly deposit proportionally over all five asset classes (e.g., if large-caps are 40% of the whole and the deposit is $200, put $80 in large-caps).
Exercise 9.29** Modify Portfolio's `reallocateAssets` method to charge 1% of all assets sold other than money market assets. This approximates the real-life cost of buying and selling investments, for commissions and the like. Debit the money market fund for this cost. Example: If the only change is for small caps to go up from $30,000 to $40,000 and for large caps to go down from $50,000 to $44,000, charge 1% of the $6,000 sale. So the money market fund will go down by a total of $4,060.

9.8 Version 3 Of Iterative Development

When you look at what an Asset object has to be able to do, you can deduce what it needs to know: `itsName`, `itsReturn`, and `itsRisk`. Those are the obvious instance variables used to calculate daily changes. But when asked to print the descriptions of an asset, the user does not want to see daily returns and volatility. The user wants an indication of what the return would be in a good year, what it would be in a bad year, and what it would be in an average year. So the `description` instance method should return `itsName` and those three numbers calculated from `itsReturn` and `itsRisk`. The `getHeading` class method should return four corresponding column titles.

Question: How do you calculate a good year and a bad year? Since the sequence of daily returns is a set of random numbers, theory of statistics can provide the answers. However, its answers for the distribution of values from a random distribution are for cases where you have the sum of random values. They do not directly apply to the product of random values.

Solution: Do a statistical analysis on the logs of the random values. The result for one year of 252 days is the sum of the 252 random daily values if they are expressed in logarithms. The **Central Limit Theorem** says that the potential values for a sum of many random values are extremely close to a normal distribution (the notorious bell-shaped curve). Specifically, the distribution of the sum of 252 random daily values will be very close to a normal distribution that has a mean of 252 times the daily mean and a standard deviation of 15.87 times the daily standard deviation (because the square root of 252 is 15.87).

Suppose Bill the expert tells you that the two multipliers that describe the long-term behavior of a certain kind of stock are the **daily average** of 100.04% and the **daily volatility** of 101%. This daily average means that you multiply the previous day's value by 100.04% to get the new day's value if it is an average day. Over a year of 252 days, that comes to 10.6% gain, since 1.0004^{252} is 1.106 (you can check this in a spreadsheet with the expression `power(1.0004,252)` or just `1.0004^252`).

But mutual funds go up and down in value; they do not always go up by a fixed amount. The daily volatility takes this into account. For a good day in the market, multiply the average day's result by 101%; for a bad day, divide it by 101%. Bill says you should (for now) approximate market uncertainty by calculating either a good day or a bad day at random with equal likelihood. This means that you either add or subtract `log(1.01)` to `log(1.0004)` to modify the previous day's result, so `log(1.01)` is the standard deviation of the daily returns (when everything is expressed in logs).

To simplify the calculations, you should store the logarithms of daily averages and volatilities rather than the numbers themselves. The average return for N days is the sum of the N returns. And the variability in returns for N days is `Math.sqrt(N)` times the variability for one day. That is, we can apply standard statistical analysis to logarithms of returns. Since the square root of 252 is 15.87, the annual volatility according to the Central Limit Theorem is $1.01^{15.87}$, which is 1.1711, or 17.11% in everyday terms.

Math methods

This program uses four class methods from the standard library Math class that are described in Chapter Six:

- `Math.sqrt(x)` gives the square root of a non-negative number `x`;
- `Math.pow(x, n)` gives `x` to the nth power;
- `Math.log(x)` gives the natural logarithm of a positive number `x`; and
- `Math.exp(x)` gives the antilog of a number `x`.

In particular, `Math.exp(Math.log(x))` is the same as `x` for any positive number `x`, and the logarithm of the product of two numbers is found by adding the logarithms of the two individual numbers. The Asset class also uses a random-number-producing object from the `java.util.Random` class: `random.nextInt(n)` is one of the equally-likely numbers in the range 0 to n-1 inclusive.

The Asset constructor takes the three values passed in as parameters and stores them in its three instance variables, first converting the return and risk to logarithms. The `getName` method has the standard logic, and the `oneDaysMultiplier` method returns one of two results with equal (50%) probability: The market goes 1% higher or 1% lower than the daily average, which in the very long run produces something relatively close to the daily average (according to the Law of Large Numbers).

The description method

In a normal distribution, only 16% of the time do you get a value more than one standard deviation above the mean. A good year can be considered to be one standard deviation above the mean, because the market is that good (or better) only one year in six. Similarly, a bad year can be considered to be one standard deviation below the mean, because something that bad (or worse) happens only one year in six. One standard deviation is `itsReturn * Math.sqrt(YEAR)`, which is the (log) daily return times the square root of the number of observations. These Asset methods are in Listing 9.9.

Listing 9.9 The Asset class, Version 2

```
import java.util.Random;   // goes at the top of the file

class Asset
{
   public static final int MONTH = 21;
   public static final int YEAR = 12 * MONTH;
   private static Random random = new Random();
   /////////////////////////////////
   private String itsName;
   private double itsReturn;
   private double itsRisk;


   public Asset (String name, double average, double volatility)
   {  itsName   = name;
      itsReturn = Math.log (average);
      itsRisk   = Math.log (volatility);
   }  //======================

   public String getName()
   {  return itsName;
   }  //======================

   public static String getHeading()
   {  return "name\tgood year\tavg year\tbad year";
   }  //======================

   public String description()
   {  double rate = itsReturn * YEAR;
      double risk = itsRisk * Math.sqrt (YEAR);
      return itsName + toPercent (rate + risk)
             + "%\t" + toPercent (rate)
             + "%\t\t" + toPercent (rate - risk) + "% \n";
   }  //======================

   private double toPercent (double par)
   {  return ((int)(1000 * Math.exp(par) + 0.5) - 1000) / 10.0;
   }  //======================

   public double oneDaysMultiplier()
   {  return random.nextInt (2) == 0   // 50-50 chance
               ?  Math.exp (itsReturn + itsRisk)
               :  Math.exp (itsReturn - itsRisk);
   }  //======================
}
```

The clients say they would like to see expected returns from investments as annual percentage rates rounded to the nearest tenth of a percent. For instance, a multiplier of 1.0317 should be expressed as 3.2%, and a multiplier of 1.03148 should be expressed as 3.1%. The standard way to do this is as follows:

1. Multiply the number to be rounded by 1000, e.g., 1.0317 -> 1031.7, 1.03148 -> 1031.48. Use 1000 because you want it rounded to the nearest thousandth.
2. Add 0.5, which pushes XX.5 or higher to the next higher place but leaves the other cases the same in the part before the decimal point, e.g., 1031.7 -> 1032.2, but 1031.48 -> 1031.98.
3. Truncate with the `(int)` cast, e.g., 1032.2 -> 1032, 1031.98 -> 1031.
4. Subtract 1000 and then divide by 10.0 to get the percentage accurate to tenths, e.g., 1032 -> 3.2, 1031 -> 3.1.

A private `toPercent` method in the Asset class accepts a number as input, "unlogs" it, and makes the conversion just described, so it is in a form ready to print. This completes Version 2 of the software.

More statistics considerations

The bell-shaped curve is not a close approximation for distributions of sums of less than about thirty random values. But it is worth seeing how close it comes for just five values (since you can calculate those values in your head). `Math.sqrt(5)` is roughly 2.24, so somewhat more than 16% of the sums of five values randomly chosen from {+1, -1} should theoretically be higher than 2.24 (one standard deviation). A 5-day result has 1 chance in 32 of going up every day (since 2^5 is 32), and a 5-in-32 chance of going up on 4 of the 5 days (and thus down on the other day). So there is a 6-in-32 chance of going up by 3 or more of `itsRisk`. And 6-in-32 is roughly 19%. Conclusion: There is a roughly 19% probability of a week going up by at least 1 standard deviation. Similarly, there is a roughly 19% probability of a week going down by at least 1 standard deviation. 19% is reasonably close to 16%.

Bill the expert has finished his research into market performance and has three results for you. First, assuming a long-term fairly steady inflation rate of 4%, the average annual returns from the various investments should be around 5% for money markets, 7% for 2-year bonds, 10% for large-cap American stocks, 11% for small-cap American stocks, and 12% for foreign stocks with significant exposure to emerging markets. A quick spreadsheet calculation (e.g., `1.05^(1/252) = 1.00020` for money markets) gives the daily averages to use in the Asset constructors. These values are in the middle column of Listing 9.10.

Listing 9.10 Replacement for the Portfolio constructor, Version 3

```
public Portfolio()
{  itsAsset[0] = new Asset ("money mkt   ", 1.00020, 1.0001);
   itsAsset[1] = new Asset ("2-yr bonds  ", 1.00027, 1.004);
   itsAsset[2] = new Asset ("large caps  ", 1.00038, 1.011);
   itsAsset[3] = new Asset ("small caps  ", 1.00041, 1.015);
   itsAsset[4] = new Asset ("foreign     ", 1.00045, 1.020);
}  //======================
```

Bill also has found that daily volatility for money markets is quite low, but it rises very quickly as the daily average rises. He gives you the values in the last column of Listing 9.10 to use.

Second, Bill suggests using an actual normal distribution to approximate daily market behavior. You already have a random-number-producing object named `random` from the Random class. The **`nextGaussian`** method of the Random class produces a double value that is distributed normally with a mean of 0.0 and a standard deviation of 1.0. So the body of Asset's `oneDaysMultiplier` becomes as follows:

```
   return Math.exp (itsReturn + random.nextGaussian() * itsRisk);
```

The messiest part of what Bill has for you is an estimate of the influence of the trend of past market changes on the next day's market change. Many stock market experts say that such trends cannot be used profitably by individual investors, but other stock market experts disagree. In any case, the clients want to have a day's market change depend about one-third on previous trends and about two-thirds on chance. That is left as a major programming project.

Exercise 9.30 Explain why the algorithm for rounding to the nearest thousandth does not work if you subtract 1000 before applying `(int)` instead of after.
Exercise 9.31 What changes would you make in Listing 9.9 to have it round to hundredths of a percent instead of tenths of a percent?
Exercise 9.32 What changes would you make in Listing 9.9 if a good year is considered 1.64 standard deviations above the mean (1-chance-in-20 of that) and a bad year is 1.64 standard deviations below the mean (also 1-chance-in-20)?
Exercise 9.33 (harder) Write an independent method that finds the common logarithm of a given number, i.e., using base ten instead of base e. Throw an IllegalArgumentException when the parameter is not positive.
Exercise 9.34* The output from the program at the end of each time period puts the value of each asset class on a separate line. The client feels that it would be easier to see the patterns over time if they were all on one line separated by tabs. Change the coding to print just two lines at the end of each time period: The names of the five classes on one line and the values on the second line.
Exercise 9.35* Calculate the probability that an asset goes up by at least 1 standard deviation in a 6-day period, similar to the calculation given for a 5-day period.
Exercise 9.36* Revise Listing 9.10 to get the ten numbers from a BufferedReader.
Exercise 9.37** The prospectus for a mutual fund often gives a chart showing how an initial investment of $10,000 would have grown over the years if you let it ride. Add an instance variable to each Asset object named `itsIndex` to track this amount. This `itsIndex` value is initially $10,000. Each time you call Asset's `oneDaysMultiplier`, it changes `itsIndex` by the appropriate amount. Add an instance method `getIndex()` to obtain the current value of `itsIndex` for that mutual fund. Have Portfolio's `getCurrentValues` method return the current value of the index.

9.9 Additional Java Statements: switch And break

Java has several statements that are not used elsewhere in this book. For three of them, the `switch` statement, the simple `return` statement, and the alternate constructor invocation, that is because the proper conditions for them to be used do not come up very often. The other statements mentioned here, in the opinion of this author, should never be used in well designed coding except for the use of the `break` statement in a `switch` statement.

The switch statement

The `switch` statement is used only when you need a multiway-selection structure that chooses one of many possibilities (usually five or more) depending on the value of a particular integer expression. For example, let us say the "weekday number" of Sunday is 1, of Monday is 2, of Tuesday is 3, etc. The method in Listing 9.11 then returns the weekday number of a day of the week, given the String of characters that form its name. It does this by computing the sum of the integer equivalents of the first and fourth characters and going directly to the corresponding `case` clause.

Listing 9.11 Example of a switch statement

```
/** Return the number of the weekday with the given name. */

public static int weekdayNumber (String day)  // independent
{  switch (day.charAt (0) + day.charAt (3)) // 1st & 4th chars
   {  case 'F' + 'd':   // Friday
         return 6;
      case 'M' + 'd':   // Monday
         return 2;
      case 'S' + 'd':   // Sunday
         return 1;
      case 'W' + 'n':   // Wednesday
         return 4;
      case 'T' + 'r':   // Thursday
         return 5;
      case 'T' + 's':   // Tuesday
         return 3;
      case 'S' + 'u':   // Saturday
         return 7;
   }
   return 0;
}  //======================
```

The seven cases are listed here in increasing order of the **switch control value** in the parentheses after `switch`, though they need not be. The last `return` statement is executed if none of the seven cases apply (perhaps the value in `day` is misspelled).

The requirements for a **switch statement** are strict, which make it less useful than it otherwise might be:

- The control value must be an int value or its equivalent (char, short, or byte). So the method in Listing 9.11 could not have `switch (day) { case "Sunday":...`
- The expression for each case must be computable at compile time. So it cannot involve a non-final variable.
- All case expressions must differ in their integer values.

The advantage of the `switch` statement is speed: It takes the flow-of-control directly to the right statement. A multiway if-statement would take less room in a program, but some `day` values could require testing five or six conditions.

You can have the last alternative of a `switch` statement say `default:` rather than specifying a case. All values of the control value for which no case is listed end up in that default case. Without the default case, the switch statement does nothing for such values.

The fall-through effect for switch statements

The statements in each case of the `switch` statement normally finish with a `return` statement or some other way to leave the case. Otherwise the flow-of-control will continue into the next case ("fall-through"). You usually do not want this. For instance, you might write the following coding to assign the month number for a month beginning with 'J', where any month name with the wrong fourth letter switches into the default case:

```
switch (month.charAt (3)) // 4th char if month begins with 'J'
{  case 'e':     // June
      num = 6;
   case 'u':     // January
      num = 1;
   case 'y':     // July
      num = 7;
   default:
      num = 0;
}
```

But that would be wrong. For instance, "January" switches to the second case which sets `num` to 1; then it falls through to the next case, setting `num` to 7; then it falls through to the default case, setting `num` to 0. You can prevent this fall-through if you put a `break` statement at the end of each case except the default. The `break` statement terminates the `switch` statement immediately, going on to execute the next statement in the method. The corrected version is shown in Listing 9.12.

Listing 9.12 A switch statement with breaks

```
switch (month.charAt (3))
{  case 'e':     // June
      num = 6;
      break;
   case 'u':     // January
      num = 1;
      break;
   case 'y':     // July
      num = 7;
      break;
   default:
      num = 0;
}
```

The advantage of this fall-through property is that you can have several case phrases for a single group of statements. For instance, the following returns the first letter of the day with the given day number:

```
switch (dayNumber)
{  case 1:  case 7:
      return 'S';
   case 3:  case 5:
      return 'T';
   case 4:
      return 'W';
   case 6:
      return 'F';
   case 2:
      return 'M';
}
```

Caution If the statements within one of the `case` clauses include a looping statement, a `break` statement inside that looping statement will not break out of the `switch` statement. That `break` statement will only break out of the looping statement.

The simple return statement

The word `return` with a semicolon immediately after it is a legal Java statement. It can only be used in a constructor or in a method that returns nothing, i.e., has the word `void` in the heading. The effect is to terminate the method immediately, without going to the end of its coding. For instance, you might have some coding with the structure shown below on the left. Many people feel this is clearer if rewritten as shown on the right:

```
// without return                    // same thing with return
if (s != null)                       if (s == null)
{  s = s.trim();                        return;
   if (s.length() != 0)              s = s.trim();
   {  // several statements          if (s.length() == 0)
   }                                    return;
}                                    // several statements
```

You have seen several cases of a loop with a continuation condition formed with &&. Often these can be written without && if a `return` statement is used in the body of the loop. Many people prefer to do this. For instance, a while-statement at the end of a method in Listing 7.8 is as follows, except that the italicized part has been added. That addition allows you to omit the underlined part:

```
while (data.getName() != null && itsSize < itsItem.length)
{  if (itsSize >= itsItem.length)
      return;
   itsItem[itsSize] = data;
   itsSize++;
   data = new Worker (file.readLine());
}
```

The Collision JApplet in Listing 8.12 has the following coding at the end of one action method:

```
int c = 0;
while (c < SIZE && itsItem[c].misses (guy))
   c++;
if (c < SIZE && itsItem[c].isHealthy())
   itsItem[c].getsPoison();
```

This can be written instead as follows. Note that the loop control variable does not have to be declared outside the for-statement, because it is not used after the loop:

```
for (int c = 0;  c < SIZE;  c++)
{  if ( ! itsItem[c].misses (guy))
   {  if (itsItem[c].isHealthy())
         itsItem[c].getsPoison();
      return;
   }
}
```

A new use of this

The first statement in a constructor can be `this(someParameters)` instead of `super(someParameters)`, where the parentheses are filled with whatever parameters are appropriate for the constructor method that the `this` statement calls. This "alternate constructor invocation" calls another constructor in the same class instead of a constructor from the superclass.

For instance, if you have two constructors with the headings

```
public Whatever (String name, int age, boolean inUnion)
public Whatever (String name)
```

then the first statement in the second constructor can be the following:

```
this (name, 21, true);
```

This statement replaces the usual `super` statement. It means that the first constructor is executed before anything else is done in the second constructor. In effect, it says that the second constructor uses a default age of 21 and a default unionization status `true`.

Multiple declarations and assignments

You may declare several variables in the same statement if they have the same type. For instance, the following statement declares three int variables, initializing the second one to 3. This should only be done for variables that are strongly related to each other, such as the <x, y> position of a graphical object:

```
int x, y = 3, z;
```

You may also assign several variables the same value at once. The following statement assigns 5 to each of the three named variables:

```
x = y = z = 5;
```

A bare semicolon is considered a statement. For instance, the following coding finds the first positive array component (empty braces `{ }` would be clearer):

```
for (k = 0;  k < size && a[k] > 0;  k++)
   ;
```

The break and continue statements

If you execute the statement `break;` inside the body of a loop, the loop terminates immediately. If you execute the statement `continue;` inside the body of a loop, the current iteration terminates and execution continues with the next iteration. You should know these facts so you can understand Java written by people who think it is alright to do these things. You can also put a label such as `ralph:` on a statement and then use the command `break ralph;` or `continue ralph;` to terminate looping constructs all the way out to the statement labeled as `ralph`.

Exercise 9.38 Rewrite the `switch` statement in Listing 9.12 as a multiway-selection structure.
Exercise 9.39 (harder) Write a `switch` statement to print the Roman numeral corresponding to num == 10, 50, 100, or 500; print "No" for all other values of num.
Exercise 9.40* Write a `switch` statement to print the month number for any of twelve different strings of characters recording months of the year. Assume each string is at least four characters long (add a blank for May). Use a control value that is the sum of two character values, as in Listing 9.11.
Exercise 9.41* Explain why the compiler will not allow you to use `day.charAt(0) + day.charAt(1)` in Listing 9.11 as the control value for the seven alternatives with the obvious case phrases, e.g., 'F'+'e', 'A'+'p'.

*9.10 About Throwable And Error (*Sun Library)*

This chapter tells you all you are likely to need to know about the Exception class and its subclasses. The Exception class is a subclass of the Throwable class, which has one other subclass named Error.

The Throwable class (from java.lang)

The **Throwable** class has the following two constructors and four methods:

- `new Throwable(messageString)` creates a Throwable object with the specified error message.
- `new Throwable()` creates a Throwable object with a null error message.
- `someThrowable.getMessage()` returns the error message belonging to the executor.
- `someThrowable.toString()` returns `ClassName: ErrorMessage` where ClassName is the full class name for the subclass of the object being thrown (e.g., `java.lang.NullPointerException`) and ErrorMessage is the executor's error message. But if the Exception object was created with no error message, then all that it returns is the full class name alone.
- `someThrowable.printStackTrace()` prints the `toString()` value followed by one line for each method that is currently active, as described next. It prints to `System.err`, which is normally the terminal window. This method is overloaded with a second version for printing to a PrintStream parameter and a third version for printing to a PrintWriter parameter.
- `someThrowable.fillInStackTrace()` puts the current values in the StackTrace record so that `printStackTrace` gives the correct value.

A sample output from `printStackTrace` is as follows:

```
java.lang.NullPointerException: null String!
   at SomeClass.meth (SomeClass.java:20)
   at OtherClass.main (OtherClass.java:33)
```

This StackTrace indicates that the statement

```
throw new NullPointerException ("null String!");
```

was executed at line 20 in the SomeClass class, within the method named `meth`. This method was called from line 33 of the OtherClass, from within the `main` method. And that `main` method was called from the runtime system (since otherwise more method calls would be active). How do you know it was caused by an explicit throw statement rather than a simple attempt to evaluate an expression that put a dot on a null value? Because the runtime system throws a NullPointerException without an error message in the latter case.

The purpose of the `fillInStackTrace()` method is to correct the StackTrace when a throw statement re-throws an Exception it caught; that Exception has the StackTrace from its original point of creation, and `fillInStackTrace()` replaces it by the StackTrace for the statement that re-throws it. You can re-throw with a statement such as

```
throw e.fillInStackTrace();
```

because the method returns the Throwable object that is its executor.

The Error class (from java.lang)

The Error subclass of Throwable is generally for abnormal problems that are too serious to be caught -- you should just let the program crash. It has four primary subclasses, all of which are in the `java.lang` package:

- **AWTError** is thrown when a serious Abstract Windowing Toolkit problem happens (something in the graphics part of the program).
- **LinkageError** is thrown when a class depends on another class which has been altered since the dependent class was compiled. For instance, a **ClassCircularityError** indicates that initialization of class X requires prior initialization of another class Y which requires prior initialization of class X. This cannot be resolved without correcting and recompiling the classes. A **NoClassDefFoundError** occurs when the current class X requires another class that existed at the time X was compiled but does not exist at runtime. An **ExceptionInInitializerError** occurs for a problem initializing a class variable, e.g., `private static x = s.length()` when `s` is null.
- **VirtualMachineError** is thrown when the Java Virtual Machine breaks or is out of RAM or otherwise cannot operate correctly. For instance, a **StackOverflowError** happens when a program has a recursive call that goes thousands of levels deep (this generally indicates either an inappropriate application of recursion or an infinite loop). An **OutOfMemoryError** occurs when there is no more RAM available for calling a constructor and the garbage collector cannot find any to recover.
- **ThreadDeath** is thrown by a statement of the form `someThread.stop()`. This statement is **deprecated**, i.e., it is from an earlier version of Java, it is outdated, and it should never be used.

9.11 Review Of Chapter Nine

About the Java language:

- The **Exception** class has several subclasses including the **RuntimeException** class. If a method contains only statements that can throw a RuntimeException, the Exception does not have to be caught. But the program will crash if that Exception arises and is not caught.
- A non-RuntimeException is called a **checked Exception**. Any method containing a statement that can throw a checked Exception must handle the potential Exception in one of two ways: Put the statement within a try/catch statement that can catch it, or put a **throws clause**, `throws SomeException`, at the end of the method heading. The SomeException must be the same class or a superclass.
- A **try/catch statement** `try {...} catch (SomeException e) {...}` has just the one **try-block** but possibly many **catch-blocks**, each with its own class of Exceptions. If an Exception is thrown by a statement in the try-block, the first catch-block that can handle that kind of Exception is executed.
- The Exception class has two constructors that every subclass of Exception should copy. One has no parameters and the other has a String parameter for supplying an error message.
- Every subclass of Exception inherits a `getMessage` method for retrieving the message part of an Exception object.
- You can throw an Exception using the statement `throw new Exception (someMessage)`. The message is optional. This throws a checked Exception. To throw an unchecked Exception, use `throw new RuntimeException (someMessage)` instead. Or use any other subclass of Exception that is appropriate.
- `someRandom.nextGaussian()` returns a value that normally distributed with a mean of 0.0 and a standard deviation of 1.0. The Random class is in the `java.util` package of the Sun standard library.
- Read through the documentation for Exception and its subclasses (partially described in this chapter). Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

About some Sun standard library RuntimeException subclasses:

- `ArithmeticException` is thrown when e.g. you divide by zero.
- `IndexOutOfBoundsException` has two commonly occurring subclasses: When you refer to `b[k]` and `k < 0` or `k >= b.length`, you get an ArrayIndexOutOfBoundsException object, and when you refer to `someString.charAt(k)` and `k < 0` or `k >= someString.length()`, you get a StringIndexOutOfBoundsException object.
- `NegativeArraySizeException` is thrown when e.g. you have `new int[n]` and `n` is a negative number.
- `NullPointerException` is thrown when you put a dot on an object variable that has the null value or you refer to `b[k]` and `b` has the null value. However, the former does not apply if the dot is followed by a class variable or a class method call.
- `ClassCastException` is thrown when e.g. you use `(Worker) x` for a Person variable `x` but `x` refers to a Client or Student object. `(ClassName) x` is allowed only when `x` refers to an object of that class, or a subclass of that class, or an implementation of that interface. However, it is also allowed that x be null.
- `NumberFormatException` is thrown by e.g. `parseDouble` or `parseInt` for an ill-formed numeral.
- All of the above Exception subclasses are in the `java.lang` package, which is automatically available to every class.

Other vocabulary to remember:

- Two (or more) arrays are called **parallel arrays** when the kth component of one tells you something about the kth component of the other. Parallel arrays do not often occur in Java because an object that combines the two (or more) parts is generally preferable.
- When N amounts are to be added to a base value, the average amount is found by adding them up and dividing by N. But when N amounts are to be multiplied by a base value, the average amount is the **geometric average**, found by multiplying them together and taking the Nth root. Alternatively, you take the usual average of their logarithms and then take the anti-log of the result.

Answers to Selected Exercises

9.1
```
if (k >= 0 && k < a.length)
    System.out.println (a[k]);
if (k >= 0 && k < str.length())
    System.out.println (str.charAt (k));
```
9.2
```
if (num >= 0)
    array = new int [num];
```
9.3 Put "try{" in front of the declaration of the rate variable.
Put the following after the right brace at the end of the body of the while-statement:
```
} catch (NumberFormatException e)
{   JOptionPane.showMessageDialog (null, "That was an ill-formed numeral.");
}
```
9.4
```
throw new NullPointerException ("the parameter is null");
```
9.5 The first prompt is "Entry?" The second prompt is "Ill-formed integer: Entry?"
The third prompt is "Ill-formed integer: Ill-formed integer: Entry?"

9.8
```
public boolean checkError()
{   return isKeyboard;
}
```
9.9
```
public static void printStuff (Object[] toPrint)
{   try
    {   for (int k = 0;  k < toPrint.length;  k++)
            printOneItem (toPrint[k]);
    }catch (java.awt.print.PrinterAbortException e)
    {   System.out.println (e.getMessage());
    }
}
```
9.10
```
public static void printStuff (Object[] toPrint)
{   for (int k = 0;  k < toPrint.length;  k++)
    {   try
        {   printOneItem (toPrint[k]);
        }catch (java.awt.print.PrinterAbortException e)
        {   System.out.println (e.getMessage());
        }
    }
}
```
9.14
```
public class BadStringException extends Exception
{
    public BadStringException()
    {   super();
    }
    public BadStringException (String message)
    {   super (message);
    }
}
```

9.15 Insert the following statement at the beginning of each of the two methods:

```
if (itself.length() == 0)
      throw new BadStringException();
```

Also revise the two method headings as follows, since BadStringException is checked:

```
public void trimFront (int n)  throws BadStringException
public String firstWord()  throws BadStringException
```

9.16 No modification is needed in the heading, since it is not a checked Exception.
Insert the following statement at the beginning of the method body:

```
if (interestRate <= 0)
      throw new ArithmeticException ("The rate is negative or zero");
```

9.18 4%. The end result is 1.50 * 1.50 * 0.50 = 1.125, for a 12.5% overall return. This is an average (geometric) return of 4.00%, since 1.0400 to the third power is 1.125.

9.19 1.10 to the 20th power is 6.73, so $700 grows to about 6.73 times $700, i.e., $4710.

9.20 1.07 to the 20th power is 3.87, so $700 grows to about 3.87 times $700, i.e., $2709.
The ratio is 57.5%. So failure to have it in a Roth IRA or a deductible IRA costs you 42.5% of your money.

9.21 10%, because Jo keeps only 7% after taxes, so Jo's investment grows at a rate of 7% in after-tax value.

9.24 Replace the assignment to value in Listing 9.6 by the following:

```
value = IO.askDouble ("How much do you add each month");
while (value <= 0)
      value = IO.askDouble ("The initial investment must be POSITIVE; try again:");
```

9.26 Replace the heading of the inner for-loop of letTimePass by the following five lines:

```
if (itsValue[0] < 0)
      itsValue[0] *= itsAsset[0].oneDaysMultiplier() + 0.05 / Asset.YEAR;
else
      itsValue[0] *= itsAsset[0].oneDaysMultiplier();
for (int k = 1;  k < NUM_ASSETS;  k++)
```

9.30 If the number was originally 1.0317, You multiply by 1000 and add 0.5, giving 1032.7. You can then subtract 1000 and truncate in either order and it makes no difference. But if the number was originally 0.9924, you multiply by 1000 and add 0.5 to get 992.9. If you subtract 1000 before truncating, you get -7.1 which truncates to -7.0, which is the wrong answer. If you truncate before subtracting, you get -8.0, which is the right answer. That is, the problem arises with truncating negative numbers.

9.31 Replace 1000 by 10000 in two places, and replace 10.0 by 100.0.

9.32 Replace the second statement of the description method by the following:

```
double risk = 1.64 * itsRisk * Math.sqrt (YEAR);
```

9.33

```
public static double commonLog (double x)
{     if (x <= 0)
            throw new IllegalArgumentException();
      return Math.log (x) / Math.log (10);
}
```

9.38

```
if (month.charAt (3) == 'e')
      num = 6;
else if (month.charAt (3) == 'u')
      num = 1;
else if (month.charAt (3) == 'y')
      num = 7;
else
      num = 0; // this is 6 lines shorter, though it may execute somewhat more slowly.
```

9.39

```
switch (num)
{     case 10:
            JOptionPane.showMessageDialog (null,  "X");
            break;
      case 50:
            JOptionPane.showMessageDialog (null,  "L");
            break;
      case 100:
            JOptionPane.showMessageDialog (null,  "C");
            break;
      case 500:
            JOptionPane.showMessageDialog (null,  "D");
            break;
      default:
            JOptionPane.showMessageDialog (null,  "No");
}
```