Chapter 5: Exceptions and Threads

In the first four chapters we saw the basics of object-oriented programming, getting a glimpse of some of the graphics capabilities of Java. We have also seen a fair amount of examples and definitions about how to put together applications, complete with menus, dialog boxes, and other graphics user interface elements. In chapter 6 will further explore Java's graphical abilities, as well as take a look at its multimedia capabilities.

As it turns out, to properly create more advanced programs we will often need the ability to have one program do several activities at once, such as loading a picture while displaying a frame, or playing some music while doing a long computation. Java does support concurrently executing pieces of code combined in one program using a concept called "threads". Therefore, we need to investigate threads before going into details about images and animations. However, threads are not as simple as our previous programs: threads of concurrently executing code can be "disturbed" for many reasons. Therefore, they will need a mechanism to effectively and flexibly communicate any problems to another part of a program that might rely on the thread to finish its part of the task.

This mechanism of handling potential errors while a particular thread executes turns out to be so useful that Java has adopted it as a general error-handling scheme called a "try-catch" block. To be sure, Java's error-handling mechanism does not originate with threads, but was adopted from a similar mechanism in the programming language C++. However, we were able to avoid, for the most part, dealing with this mechanism in our previous examples, but to continue we need to understand the basic workings of this scheme. Therefore, we will start this chapter by explaining the "try-catch" error-handling mechanism and then introduce threads. With these foundations we can discuss Swing, loading images, playing sounds, and other more advanced techniques in the next chapters.

Quick View

Here is a quick overview of the topics covered in this chapter.

(*) These sections are optional

5.1. Error and Exception Handling

Programming errors are a common problem while you are creating a program, applet, or a class. In fact, virtually nobody can create a reasonably complex application that is free of errors right from the start. Therefore, the goal is not to create a problem-free class on the first try, but to extensively test and rewrite the class until you can be reasonably sure that it is error-free. Even if you have tested a program or class extensively and all tests indicate that it performs as you intended, there will always be circumstance beyond your control. For example:

- If your program gets data from a network connection, what if the server delivering the data goes down?
- If you are printing a long document, what if the printer runs out of paper?
- What should your program do with a user who does not follow directions and enters a string when explicitly asked to enter an integer?
- What if your program asks the user to initialize an array of a specific size, but there is not enough memory available to accommodate the size the user specified?
- What if your program tries to divide zero by zero?

Therefore, since programs run in an imperfect world and are used by imperfect people, there is a need for an elaborate and flexible mechanism to deal with more or less unexpected or unusual errors. Indeed, when a class encounters an error, it should attempt to:

- return to a safe and known state
- enable the users to execute other commands
- save all work if possible
- exit without further harm if necessary

Before finding out how to achieve these goals, let's take a look at an example where we intentionally produce a few errors:

Example 5.01: Creating intentional errors

Create a program that attempts to reference a variable pointing to `null`, tries to access an array element with an incorrect index, and wants to retrieve a character from a string at a negative index. Execute the program and describe the error messages that you see.

The first thing we need to decide is whether to use a GUI, or Frame-based program or a simple program without GUI elements. We have already seen error messages for non-GUI based programs in chapter 2, so we decide on using a `Frame` with buttons to produce the errors this time:

- use one field of type `Button` for each error to produce, plus one more button to quit the program[1]
- add a field containing an array, another field with a well-defined `String`, and one more field with a `String` set to null
- create the class by extending `Frame` and implementing `ActionListener`
- in the constructor, layout the buttons in a grid and show the frame

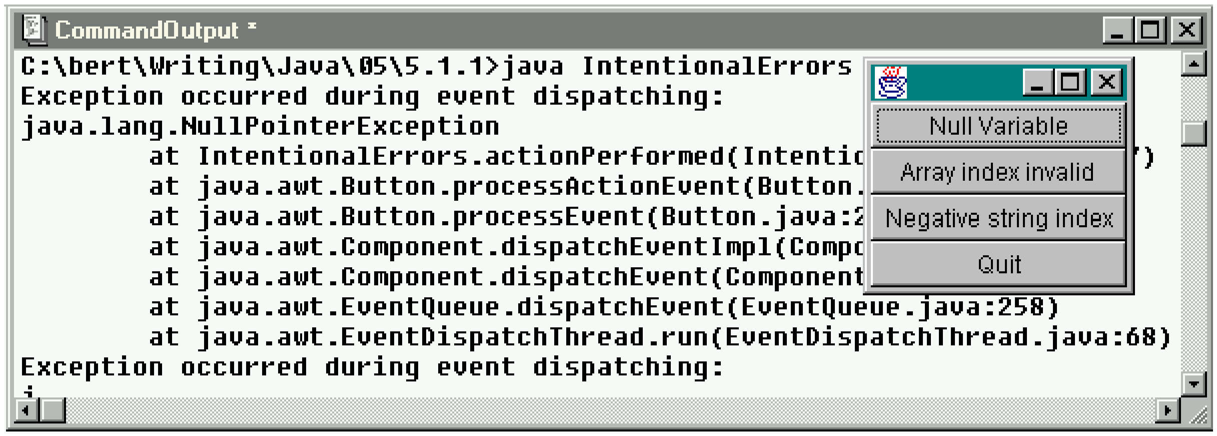
[1] At this stage it is unclear what happens when a program encounters an error. It might completely quit, in which case an explicit quit button would not be necessary. But as we will see in definition 5.02, frame-based programs will try to continue running even if they encounter an error.

- in the `actionPerformed` method, create the various errors when the corresponding buttons are pressed
- in the `main` method, instantiate an instance of the class

Here is the code:

```
import java.awt.*;
import java.awt.event.*;

public class IntentionalErrors extends Frame implements ActionListener
{  public Button nullVariable = new Button("Null Variable");
   public Button arrayIndex   = new Button("Array index invalid");
   public Button stringIndex  = new Button("Negative string index");
   public Button quit         = new Button("Quit");
   private double A[] = new double[10];
   private String string = "Bert";
   private String nada = null;

   public IntentionalErrors()
   {  setLayout(new GridLayout(4,1));
      add(nullVariable);     nullVariable.addActionListener(this);
      add(arrayIndex);       arrayIndex.addActionListener(this);
      add(stringIndex);      stringIndex.addActionListener(this);
      add(quit);             quit.addActionListener(this);
      validate();
      pack();
      setVisible(true);
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == quit)
         System.exit(0);
      else if (e.getSource() == nullVariable)
         System.out.println(nada.substring(0,3));
      else if (e.getSource() == arrayIndex)
         A[10] = 10;
      else if (e.getSource() == stringIndex)
         System.out.println(string.charAt(-1));
   }
   public static void main(String args[])
   { IntentionalErrors ie = new IntentionalErrors(); }
}
```

When we compile and run this class, we see the following error messages:



```
C:\bert\Writing\Java\05\5.1.1>java IntentionalErrors
Exception occurred during event dispatching:
java.lang.NullPointerException
        at IntentionalErrors.actionPerformed(Intentic
        at java.awt.Button.processActionEvent(Button.
        at java.awt.Button.processEvent(Button.java:2
        at java.awt.Component.dispatchEventImpl(Compo
        at java.awt.Component.dispatchEvent(Component
        at java.awt.EventQueue.dispatchEvent(EventQueue.java:258)
        at java.awt.EventDispatchThread.run(EventDispatchThread.java:68)
Exception occurred during event dispatching:
```

Figure 5.01: IntentinalErrors program after clicking the Null Variable button

- When clicking on the `nullVariable` button:

```
Exception occurred during event dispatching:
java.lang.NullPointerException
   at IntentionalErrors.actionPerformed(IntentionalErrors.java:27)
   at java.awt.Button.processActionEvent(Button.java:254)
   at java.awt.Button.processEvent(Button.java:227)
   at java.awt.Component.dispatchEventImpl(Component.java:1764)
   at java.awt.Component.dispatchEvent(Component.java:1704)
   at java.awt.EventDispatchThread.run(EventDispatchThread.java:63)
```

- When clicking on the `arrayIndex` button:

```
Exception occurred during event dispatching:
java.lang.ArrayIndexOutOfBoundsException: 10
   at IntentionalErrors.actionPerformed(IntentionalErrors.java:28)
   at java.awt.Button.processActionEvent(Button.java:254)
   at java.awt.Button.processEvent(Button.java:227)
   at java.awt.Component.dispatchEventImpl(Component.java:1764)
   at java.awt.Component.dispatchEvent(Component.java:1704)
   at java.awt.EventDispatchThread.run(EventDispatchThread.java:63)
```

- When clicking on the `stringIndex` button:

```
Exception occurred during event dispatching:
java.lang.StringIndexOutOfBoundsException: String index out of range:-1
   at java.lang.String.charAt(String.java:398)
   at IntentionalErrors.actionPerformed(IntentionalErrors.java:31)
   at java.awt.Button.processActionEvent(Button.java:254)
   at java.awt.Button.processEvent(Button.java:227)
   at java.awt.Component.dispatchEventImpl(Component.java:1764)
   at java.awt.Component.dispatchEvent(Component.java:1704)
   at java.awt.EventDispatchThread.run(EventDispatchThread.java:63)
```

The first thing to notice is that these errors are not called errors but exceptions. For each exception, a pretty accurate and easy to understand message is printed out:

- a `NullPointerException` occurred in the `actionPerformed` method in line 27 `of the IntentionalErrors.java` source code
- an `ArrayIndexOutOfBoundsException` supposedly occurred in line 28 of the `actionPerformed` method `of the IntentionalErrors.java` source code (the offending line is actually in line 29 of the source code)
- a `StringIndexOutOfBoundsException occurred in line 31 of the actionPerformed method of the IntentionalErrors.java` source code

As the error messages indicate, each method causing the error was in turn called by other methods and we see the entire hierarchy of method calls leading to our exception.

Another important thing to notice is that even though we get these error messages, our program does not crash. We are still able to press other buttons, including the `quit` button to exit the program nicely. We will see later that this is a common feature of how the JVM handles errors.

■

Now it is time to investigate errors and exceptions in more detail before explaining what we can do to intercept these errors and provide our own error-handling mechanism.

The Throwable Hierarchy

Java classifies possible errors into a hierarchy of classes, starting with the base class `Throwable`. We need to first understand the basic classes involved in this hierarchy before giving any examples:

The `Throwable` Classes

The `Throwable` *class is the superclass of all errors and exceptions in the Java language. Only objects of this class or extending this class can be handled by Java's error-handling mechanism. A* `Throwable` *class contains a snapshot of the current state of the program and it usually contains a message string that gives more information about the particular error. The Java API defines the* `Throwable` *class as follows:*

```
public class Throwable extends Object
{  // Constructors
   public Throwable()
   public Throwable(String message)
   // Selected methods
   public String getMessage()
   public String toString()
   public void printStackTrace()
}
```

All classes that extend `Throwable` *should override both constructors, one without input argument and one with a* `String` *input argument.*[2]

When a Java program encounters an unanticipated error, it – somehow – instantiates an object that extends this `Throwable` class. That object will contain information about the error and the current state of the object that caused the error. It can be queried to print out this information on the system console. In addition, by appropriately interpreting the object your code can make an attempt to catch the error, try to set it right, or at the very least perform proper cleanup and damage control.

Throwing an Exception

Almost any method in a class can change the normal flow of execution by throwing an exception. When that happens, the method exits immediately and does not return a value. Also, control is not returned to the code calling the method but instead to an error handler that can deal with this particular error condition, if possible.

If no error handler can be found, then non-GUI programs will stop immediately, while GUI-based programs will try to continue running, displaying a default error message on the system console.

The `Throwable` class is the immediate parent class for two other classes, the `Error` class and the `Exception` class[3]:

[2] In "real life" you will hardly ever create a class that directly extends `Throwable`. It is much more common to extend `Exception` instead, as described in definition 5.06
[3] See figure 5.02

The `Error` and `Exception` Classes

The `Error` class indicates that a serious problem has occurred due to an abnormal condition. If such an error occurs, little can be done to correct the situation. Any method can cause an `Error` without any further specifications, but it rarely happens.

The `Exception` class indicates error conditions that a reasonable application should have attempted to correct. Methods that anticipate some types of an `Exception` need to declare this in their header using the special keyword `throws`. Some types of `Exception` can occur without further specifications.

Both classes extend `Throwable` and have constructors similar to `Throwable`.

In other words, all errors that can occur in a Java program are classified into two categories:

- *Errors*, which are hopefully not going to occur in your program. If they do, there's not much than can be done and your program will most likely crash.
- *Exceptions*, which may occur in any program. If they do occur, the program should attempt to rectify the situation or at the very least return the program to a known and controlled state, allowing the user to continue working in your application.

One type of exception that could occur frequently is the so-called "`Runtime`" exception (as happened in our previous example). As is usual in Java, these types of errors are defined via an appropriate class:

Software Engineering Tip: The `RuntimeException` class and its subclasses can be instantiated at any time during the normal operation of the Java Virtual Machine. They include, among others, arithmetic exceptions, attempting to access a variable set to null, and trying to refer to an array element with an invalid index. The `RuntimeException` class extends the `Exception` class.

Exceptions of this type should be prevented by standard programming techniques, not using the try-catch mechanism to be defined later.[4]

Example 5.02: Preventing runtime exceptions

In our previous example, we have intentionally caused various runtime exceptions. Prevent these exceptions using standard programming techniques.

Of course, since we caused these exceptions intentionally and the particular code really does not serve any meaningful purpose, we could simply remove the offending lines. However, runtime errors are usually easy to prevent using several well-placed `if` statements:

The offending lines are all in the `actionPerformed` method, so we only list that method again with the new `if` statements in bold and italics.

```
public void actionPerformed(ActionEvent e)
{  if (e.getSource() == quit)
      System.exit(0);
   else if (e.getSource() == nullVariable)
```

[4] It is possible to use the try-catch block introduced in definition 5.07 to catch a `Runtime` exception. However, these types of exceptions should not occur in the first place and can usually be prevented with appropriate `if` statements.

```
            if (nada != null)
               System.out.println(nada.substring(0,3));
         else if (e.getSource() == arrayIndex)
            if (10 < A.length)
               A[10] = 10;
         else if (e.getSource() == stringIndex)
            if (-1 >= 0)
               System.out.println(string.charAt(-1));
      }
```

■

In this particular example, the `if` statements may seem somewhat artificial. However, they illustrate a general principle for preventing a `RuntimeException`:

Software Engineering Tip: The two most common runtime exceptions are accessing a variable that points to `null` and accessing an invalid element in an array. To prevent these exceptions:

- before accessing a reference variable that may not be initialized properly, first test if it is pointing to `null` before accessing it
- when there is a possibility that an array element could be accessed with an index that is out of range for the particular array, first test if the index variable is greater than or equal to zero and less then the length of the array

In other words, a `RuntimeException` could occur virtually everywhere in a program and it is easily preventable without too much trouble. That separates this type of exceptions from othes:

Implicit and Explicit Errors

An implicit error or exception is any error that derives from the `Error` *or the* `RuntimeException` *class. Every other type of exception is called an explicit exception.*

- *Implicit exceptions can occur in any class or method. In the case of a* `RuntimeException` *it is easily prevented using strategically placed if statements. In the case of an* `Error`*, there is not much than can be done about it.*
- *Explicit exceptions can only occur in methods that specifically declare this in their method header (see below for details).*

Here is an illustration that shows how the major classes involved in the `Throwable` hierarchy are related:

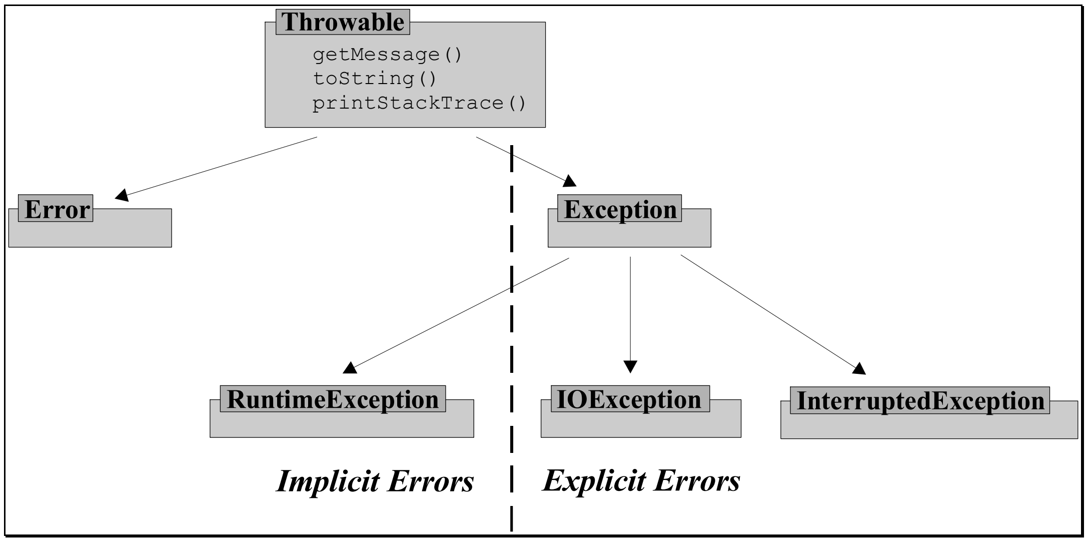


Figure 5.02: Exception and Error Hierarchy

In other words, we should not worry about an `Error` (since there is little to be done about it). We should also make an effort to prevent a `RuntimeException` from occurring, and if one does occur, the blame rests with the programmer. Explicit exceptions, however, are a different story, and we have different options for dealing with them.

Here is a list of the more commonly occurring exception classes[5]. For more details, please refer to the Java API. First, here are those exceptions that are usually preventable (most extend `RuntimeException`):

Exceptions in Package `java.lang`	
`ArithmeticException`	An exceptional arithmetic condition has occurred.
`ArrayIndexOutOfBoundsException`	An array has been accessed with an illegal index.
`ArrayStoreException`	An attempt has been made to store the wrong type of object into an array
`ClassCastException`	Attempted to cast an object to a subclass of which it is not an instance.
`IllegalArgumentException`	A method has been passed an incorrect argument.
`IllegalThreadStateException`	A thread is not in an appropriate state for the requested operation.
`InterruptedException`	A thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class Thread.
`NoSuchFieldException`	Signals that the class doesn't have a field of a specified name.
`NoSuchMethodException`	A particular method cannot be found.
`NullPointerException`	An application attempts to use null in a case where an object is required.
`NumberFormatException`	An application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.
`SecurityException`	Indicates a security violation.

[5] To be sure, many other exceptions can occur in other packages, but the ones listed above occur most frequently and are quite sufficient to understand the idea.

StringIndexOutOfBoundsException	A String index is either negative or greater than or equal to the size of the string.

Table 5.03: Exceptions in package java.lang

The `java.awt` package also contains a few exceptions, but they rarely occur:

Exceptions in Package `java.awt`	
AWTException	An Abstract Window Toolkit (AWT) exception has occurred.
IllegalComponentStateException	An AWT component is not in an appropriate state for the requested operation.

Table 5.04: Exceptions in package java.awt

The next group of exception relates to input/output (I/O) operations. These errors occur commonly when reading data from disk or from a network connection. Usually, little can be done from stopping them from occurring, but a lot can be done to stop them from doing any damage.

Exceptions in Package `java.io`	
EOFException	An end of file or end of stream has been reached unexpectedly during input. Mainly used by data input streams since most other input streams return a special value on end of stream. (see chapter 9 for more information about streams).
FileNotFoundException	A file could not be found.
IOException	An I/O exception of some sort has occurred.
InterruptedIOException	An I/O operation has been interrupted.

Table 5.05: Exceptions in package java.io

Finally, there are exceptions related to network programming from the java.net package. We will see them again in detail in chapter 9 and 10, but they are listed here for completeness.

Exceptions in Package `java.net`	
ConnectException	Error while attempting to connect to a remote address and port, typically because the connection was refused
MalformedURLException	An incorrect URL was created.
NoRouteToHostException	Error while attempting to connect to a remote address and port, typically because the remote host cannot be reached.
ProtocolException	Error in the underlying protocol, such as a TCP error.
UnknownHostException	The IP address of a host could not be determined.
UnknownServiceException	An unknown service exception has occurred.

Table 5.06: Exceptions in package java.net

These exceptions are related in various hierarchies. Here are these exceptions again, this time sorted by their relationships with each other:

```
java.lang.Exception
        AWTException
        InterruptedException
        IOException
                EOFException
                FileNotFoundException
                InterruptedIOException
                MalformedURLException
                ProtocolException
                SocketException
                        ConnectException
                        NoRouteToHostException
                UnknownHostException
                UnknownServiceException
        NoSuchFieldException
```

```
NoSuchMethodException
RuntimeException
      ArithmeticException
      ArrayStoreException
      ClassCastException
      IllegalArgumentException
            NumberFormatException
      IllegalStateException
            IllegalComponentStateException
      IndexOutOfBoundsException
            ArrayIndexOutOfBoundsException
            StringIndexOutOfBoundsException
      NullPointerException
SecurityException
```

Figure 5.07: Relationships of common exception classes

Throwing an Exception

We have seen how *implicit* exceptions are thrown automatically whenever they occur. *Explicit* exceptions, however, are thrown by methods only when these methods explicitly declare that possibility:

Throwing an Explicit Exception

To throw an explicit exception, the method header must specify the type of exception(s) this method can encounter, using the syntax:

```
[modifiers] returnType methodName throws listOfExceptionTypes
{  // body of message, including
   throw new ExceptionType(inputList);
}
```

where `listOfExceptionTypes` *is a comma-separated list of special classes, each of which extends the* `Exception` *class, and* `ExceptionType` *is of one of these classes.*

A method that overrides an inherited method can only throw exceptions of the same type as its parent method, or less. In particular, if the parent method does not throw any exceptions neither can the inherited one.

Example 5.03: Throwing an explicit exception

If you were to create a method `double stringToDouble(String number)` that attempts to convert an input string to a `double` number, what are the possible issues that you might be facing? Would "throwing an exception" be a proper way to deal with any problems? If so, create the appropriate method header.

Aside from the fact that we would have to write the actual conversion code, a method to convert a `String` to a `double` may not always succeed: if the input string does not represent a `double` number, it should fail. But since the method `stringToDouble` *must* return a double number – as promised in the method header – the question is which number should be returned? If the input string does not represent a proper `double`, our method, of course, could return 0.0 as a default value. But then a method that is *using* `stringToDouble` does not know whether 0.0 was indeed the correctly converted string or the default value because string conversion failed.

One can cook up some roundabout ways to notify the calling method of any errors that may have occurred, but it would probably not be an elegant solution. Instead, we simply indicate that our method could throw a `NumberFormatException`:

```
double stringToDouble(String number) throws NumberFormatException
{  //  actual conversion code.
   // If no error occurs:
      return convertedNumber;
   else
      throw new NumberFormatException("tried to convert: " + number);
}
```

If no error occurs during the conversion, our method will return the converted number. If an error occurs, it instead throws a `NumberFormatException`, which seems appropriate for this type of exception. That implies that in this case our method does not actually have to return a double, even though it specifies in the method header that it would do so under normal circumstances. If the error is thrown, the method that has called `stringToDouble` will be searched for a proper error handling code[6], and will not continue on its normal path of execution.

■

Of course you can also create your own custom-made exceptions that can be thrown by your methods. After all, the above definition only specifies to throw an object of a class that extends `Exception`. If none of the available exception classes are appropriate for whatever situation you have in mind, simply create your own:

Creating your own `Exception`

If a method needs to throw an exception but none of the build-in exception classes seem appropriate, you can create your own `Exception` *class as follows:*

- *Create a class that extends* `Exception`*, or another existing exception class*
- *Give your class a default constructor with no arguments that calls the constructor of its* `super` *class with an appropriate error message as input*
- *Add a second constructor with a single* `String` *argument that calls the constructor of its* `super` *class with an appropriate error message as input, including the input string*

Example 5.04: Throwing a custom-made exception

In order to save data into a database, you need to make sure that some text that a user enters into a form does not exceed a certain length. Design an appropriate object to handle this situation.

The appropriate object might be a dialog window or a frame containing text fields. Before the data is – somehow – written to the database, a method could simply cut off any extra characters if an input string is too long. However, that approach is not acceptable for two reasons:

- we must inform the user if our program modifies the data that the user enters and expects to save to the database (perhaps the user might be able to rephrase the data if informed of the problem)
- we have not really identified the object that may cause and possibly correct the problem

[6] See definition 5.07

The problem lies in the fact that a text field can contain text of any length. Why not design a special text field that can contain only strings of up to a fixed length? Therefore, our object should extend `TextField` in such a way that it should not be possible to extract from it a string that exceeds a certain length. If one tries the object will not return the text but instead throw an exception.

Therefore, we want to create two objects, or rather classes:

- a `LongStringException` to indicate what this particular exception will mean
- a `ShortTextField` extending `TextField` that will throw an exception when someone tries to extract text that is too long

Note that we should first check the available exceptions in our above table to see if one would be appropriate for this situation. In our case, none of them seems appropriate so we will design our own:

```
public class LongStringException extends Exception
{  public LongStringException ()
   {  super("String too long"); }
   public LongStringException (String msg)
   {  super(msg); }
}
```

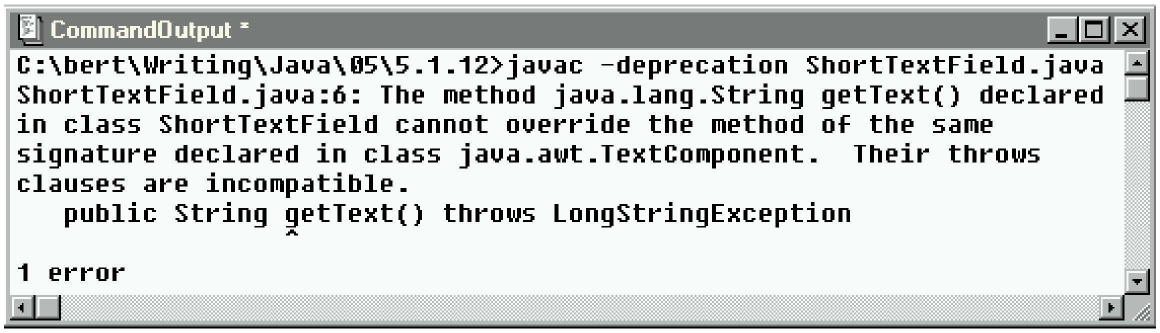
That's all we have to do to create a new exception. Note that almost all custom-made exceptions are that easy to create:

- find an appropriate name for that class and extend the `Exception` class, or another appropriate subclass of `Exception`
- create two constructors, one with void input and the other to input a particular message string

As for our class extending `TextField`, we know from chapter four that the method to retrieve text from a `TextField` is String `getText()`, so we would try to override that method:

```
import java.awt.*;

public class ShortTextField extends TextField
{  private final static int MAX = 10;

   public String getText() throws LongStringException
   {  if (super.getText().length() <= MAX)
         return super.getText();
      else
         throw new LongStringException("String too long. Max: " + MAX);
   }
}
```

This class, however, will not compile but instead give us the following error:



```
C:\bert\Writing\Java\05\5.1.12>javac -deprecation ShortTextField.java
ShortTextField.java:6: The method java.lang.String getText() declared
in class ShortTextField cannot override the method of the same
signature declared in class java.awt.TextComponent.  Their throws
clauses are incompatible.
   public String getText() throws LongStringException
                 ^

1 error
```

Figure 5.08: Error message compiling the LongStringException class

Our method `getText` overrides the method in `TextField`. But the original method does not throw an exception, so according to our definition neither can this method. Therefore, we need to create a new method for our `ShortTextField`, where the changed text is in bold and italics:

```
import java.awt.*;

public class ShortTextField extends TextField
{  private final static int MAX = 10;

   public String getShortText() throws LongStringException
   {  if (getText().length() <= MAX)
         return getText();
      else
         throw new LongStringException("String too long. Max: " + MAX);
   }
}
```

When the method `getShortText` is used, it will only return the text if it is of "legal" length or less. Otherwise, it will instead throw a `LongStringException` and the JVM will start searching for an appropriate error handler to handle this "error".

■

Note that we have not actually achieved our goal: it is still possible to use the inherited `getText` method of the `ShortTextField` class to retrieve the text even if it is longer than the legal length. But as long as the method `getShortText` is used, it will only release the text if it is of "legal" length.[7]

Since it is so easy to throw an exception, you might be tempted to throw them in many of the classes you design. After all, whenever you encounter a problem, you simply throw an exception instead of figuring out how to handle the particular situation. However, throwing an exception does not really solve the problem, it just passes it along to some other method. Eventually, *someone* needs to handle the exception. Therefore, before learning how to create the code to handle exceptions, here is a rule of thumb:

Software Engineering Tip: When creating a method that needs to deal with difficult situations, you may decide to instead throw an exception: But

- If a simple test can prevent the exception from occurring, prevent it (this applies in particular to `RuntimeExceptions`)
- If you know how to properly handle the exception, do so and do not throw an exception

[7] Another, perhaps more appropriate solution could have been to attach a `KeyListener` to the `TextField` that would count the number of characters entered and refuse to allow more characters if the maximum size is reached.

If you are unsure how to handle the exception, throw it and therefore leave it to another method to handle it appropriately. This also applies when you call another method, which in turn throws an exception: you can decide to "re-throw" that exception to propagate it instead of handling it.

Catching an Exception

Now that we know how easy it is to throw exceptions we need to find out how to create the proper code to handle them. In fact, when exceptions are properly caught they can actually simplify your code a lot and avoid a multitude of `if` statements. The idea of handling an exception is simple:

- you know that a method may create an exception because the method must advertise that fact in the method header
- you "try" to use the method that may create an exception, hoping that no exception will occur
- you provide code to "catch" the exception if and when it happens in addition to the code that hopes the exception does not happen

Try-Catch Error Handling

To provide error-handling for code that might throw one or more exceptions, you embed the code that might create an exception inside a `try` *block and the code that is supposed to handle the exception(s) inside one or more* `catch` *blocks, specifying which exception each* `catch` *block can handle.*[8]

- *If any of the code inside the* `try` *block throws an exception of the type mentioned in a* `catch` *block, the rest of the code in the* `try` *block is skipped and the code of the appropriate* `catch` *block will execute instead.*
- *If no code inside the* `try` *block creates an exception, all* `catch` *blocks are skipped entirely. In either case, execution resumes immediately after the try-catch blocks.*
- *If an exception is thrown that is not mentioned in any* `catch` *block, the method exits immediately.*

Syntax:

```
try
{  /* code block that might throw exceptions */ }
catch(ExceptionType e)
{  /* code to handle an exception of ExceptionType */ }
[ catch (OtherExceptionType oe)
{ /* code to handle an exception of OtherExceptionType */ } ]
```

[8] Each catch block can only handle one type of exception, but it could handle a superclass of the particular exception that could occur. For example, if a method throws a `MalformedURLException` as well as a `ConnectException` both can be caught in a catch block for an `IOException` because both classes are subclasses of `IOException`.

Example 5.05: Catching exceptions in a try-catch block

In the first example of this section, we saw an example of a program that intentionally created exceptions, which were subclasses of `RuntimeException`. Redo this program, embedding the code into an appropriate try-catch error handling block(s).

Recall the `actionPerformed` method that contained the code that created the various exceptions:

```
public void actionPerformed(ActionEvent e)
{  if (e.getSource() == quit)
      System.exit(0);
   else if (e.getSource() == nullVariable)
      System.out.println(nada.substring(0,3));
   else if (e.getSource() == arrayIndex)
      A[10] = 10;
   else if (e.getSource() == stringIndex)
      System.out.println(string.charAt(-1));
}
```

As we have seen, the exceptions that are generated by this method are:

- `NullPointerException`: when printing the string nada
- `ArrayIndexOutOfBoundsException`: when accessing `A[10]`
- `StringIndexOutOfBoundsException`: when accessing `string.charAt(-1)`

Therefore, we could embed the code into the following try-catch block:

```
public void actionPerformed(ActionEvent e)
{  try
   {  if (e.getSource() == quit)
         System.exit(0);
      else if (e.getSource() == nullVariable)
         System.out.println(nada.substring(0,3));
      else if (e.getSource() == arrayIndex)
         A[10] = 10;
      else if (e.getSource() == stringIndex)
         System.out.println(string.charAt(-1));
   }
   catch(StringIndexOutOfBoundsException se)
   {  System.out.println("incorrect string index");
   }
   catch(NullPointerException ne)
   {  System.out.println("oops, a null pointer");
   }
   catch(ArrayIndexOutOfBoundsException ae)
   {  System.out.println("incorrect array index");
   }
}
```

Then, instead of the previous error messages we would see the above error strings (try it out). Note that the order in which the catch blocks are listed does not matter. If an exception is generated, the JVM will search for the appropriate error handler regardless of where it is located.

Actually, all of these exceptions extend `RuntimeException`, so we could have chosen to instead catch all of them in one error-handling block:

```
public void actionPerformed(ActionEvent e)
{  try
   {  if (e.getSource() == quit)
```

```
            System.exit(0);
         else if (e.getSource() == nullVariable)
            System.out.println(nada.substring(0,3));
         else if (e.getSource() == arrayIndex)
            A[10] = 10;
         else if (e.getSource() == stringIndex)
            System.out.println(string.charAt(-1));
      }
      catch(RuntimeException se)
      {  System.out.println("caught a RuntimeException: " + se);
      }
   }
```

If you run this code, you will notice that we will only see the string specified in the `System.out.println` method, but not the entire sequence of methods leading up to the exception. If you do want to see that sequence, you can call on the method `printStackTrace` that every object extending `Exception` inherits:

```
catch(RuntimeException se)
{  System.out.println("caught a RuntimeException: " + se);
   se.printStackTrace();
}
```

■

Of course, it would be complete overkill to catch these exceptions in a try-catch block. After all, they could have – and should have – been prevented from occurring with a few simple `if` statements, as illustrated above. However, catching exceptions *can* be put to good use[9]:

Example 5.06: A reusable DoubleField class

Create a class `DoubleField` that is similar to a `TextField`, but it returns the value of the string contained in the field as a double, if possible, or a default return value if the string does not represent a double value. Then use that new field in a fully functioning applet.

The new class should look like a `TextField` so it will extend `TextField`. In addition to the method `getText` and `setText` that it will inherit we also add methods `getNumber` and `setNumber` that will set and return the appropriate values, and catch any errors if necessary. In addition, the class will provide for the possibility of defining the default return value in case an error occurs, and allow for querying it if an error occurred the last time the class was asked to convert a `String` into a `double` value. Here is the code:

```
import java.awt.TextField;

public class DoubleField extends TextField
{  private boolean error = false;
   private double defValue = 0.0;

   public DoubleField(double number)
   {  super(Double.toString(number)); }
   public void setNumber(double number)
   {  setText(Double.toString(number)); }
   public double getNumber()
   {  try
      {  double number = Double.valueOf(getText()).doubleValue();
         setNumber(number);
```

[9] In chapter 8 we will see a lot of examples of how catching exceptions can be quite useful and shorten code significantly.

```
            error = false;
            return number;
         }
         catch(NumberFormatException nfe)
         {  setText("INVALID");
            error = true;
            return defValue;
         }
      }
      public void setDefault(double _defValue)
      {  defValue = _defValue; }
      public boolean getError()
      {  return error; }
   }
```

To test our new class, we will use the `AddingMachine`[10] applet that we have created in example 4.09. The code to convert strings into numbers used to be part of the applet. Now our `DoubleField` will handle that task automatically. Here is the new code, with the important lines in bold and italics:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class AddingMachine extends Applet implements ActionListener
{  // fields
   private Button addButton = new Button("Add");
   private Button subButton = new Button("Subtract");
   private Button resetButton = new Button("Reset");
   private DoubleField input = new DoubleField (10);
   private DoubleField output = new DoubleField (10);
   private int total = 0;
   // methods
   public void init()
   {  setLayout(new FlowLayout());
      add(addButton);     add(subButton);
      add(input);         add(resetButton);
      add(output);
      addButton.addActionListener(this);
      subButton.addActionListener(this);
      resetButton.addActionListener(this);
   }
   public void start()
   {  reset();
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == addButton)
         total += input.getNumber();
      else if (e.getSource() == subButton)
         total -= input.getNumber();
      else if (e.getSource() == resetButton)
         reset();
      output.setNumber(total);
   }
   private void reset()
   {  total = 0;
      input.setNumber(0);
      output.setNumber(0);
```

[10] That applet, incidentally, could handle only integer values at that time and produced unwanted error messages if an input number did not represent a proper integer. That is because the `Integer` method `parseInt` throws an error that extends `RuntimeException` and therefore does not need to be caught explicitly.

```
        }
    }
```

In other words, the `DoubleField` will automatically handle the necessary conversions, if possible, and catch any errors if necessary. As it so happens, the default return value of 0.0 that the `DoubleField` generates in case of an error is appropriate for our example because it is the neutral element of addition and subtraction.

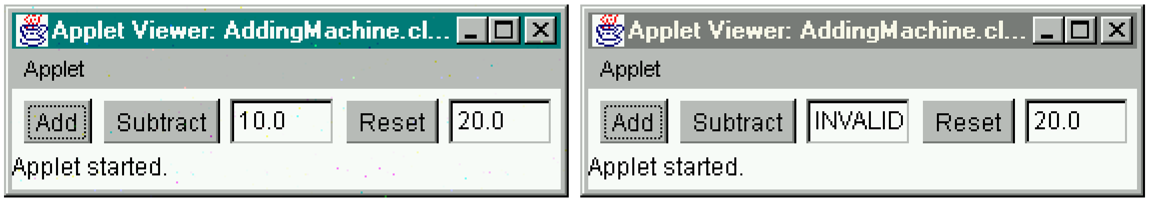


Figure 5.09: `AddingMachine` *with* `DoubleField`*, once used correctly and the second time incorrectly*

If another default value would work better, we could have used the `setDefault` method of a `DoubleField` to specify that default return value. In addition, we could have used the `getError` method immediately after a call to `getNumber` to find out whether an error has occurred or not.

■

And of course we could have designed our class to pass on the possible `NumberFormatException` to whoever uses the `getNumber` method (in which case the default return value would not be necessary). In fact, one of the exercises will ask you to do this.

`finally` – the Last Word on Exceptions

Occasionally there may be code that needs to be performed whether an exception is thrown or not. A method could, for example, acquire a `Graphics` object, then enter into a `try-catch` block. After the method is done, it needs to dispose the `Graphics` object, regardless of whether an exception occurred or not. While you could of course put the same code into the `try` as well as the `catch` block, Java offers a more appropriate solution for this situation:

The `finally` block for Exception Handling

A `try-catch` *block can be followed by an additional code block prefaced by the keyword* `finally`*. That* `finally` *code block will execute regardless of whether the* `try` *block executed successfully – no exceptions thrown – or whether the* `catch` *block executed because of a thrown exception.*

```
try
{  /* code block that might throw exceptions */ }
catch(ExceptionType e)
{  /* code to handle an exception of ExceptionType */ }
[ catch (OtherExceptionType oe)
{ /* code to handle an exception of OtherExceptionType */ } ]
[ finally
{ /* code to execute regardless of try or catch block executing */ }
```

Most classes that handle exceptions get by without a `finally` block. However, it may be useful in situations where system resources have been acquired that need to be released "manually" no matter whether an exception occurred or not. Here is a final example, illustrating exactly which line is executed in which situation in a `try-catch-finally` block.

Example 5.07: Using finally with a try-catch block

Create a class extending `Frame` that contains the `ShortTextField` from example 5.04 as well as a `Quit` and a `Go` button. When the `Go` button is pressed, retrieve text from the `ShortTextField`. Since that class might throw a `LongStringException`, we need to embed the code into a `try-catch` block. In addition, use a `finally` block and carefully examine which lines of code execute in what situation.

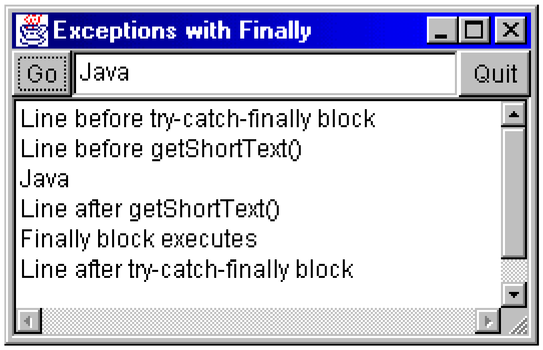
We use the standard code to extend `Frame`, implement `ActionListener`, add the buttons and the `ShortTextField` to the `Frame` and activate the buttons by adding them to the `ActionListener`. We also use a `TextArea` to display the output.

The code containing the `try-catch-finally` block appears inside the `actionPerformed` method and displays lines of text in the display area so that we can follow exactly which line is executing:

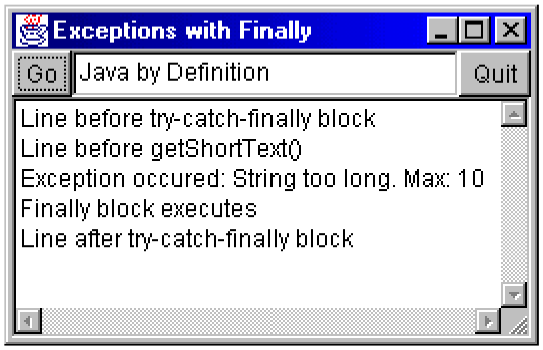
```
import java.awt.*;
import java.awt.event.*;

public class FinalException extends Frame implements ActionListener
{  public Button quit = new Button("Quit"), enter = new Button("Go");
   public ShortTextField  text = new ShortTextField();
   public TextArea        display = new TextArea(3,25);

   public FinalException()
   {  super("Exceptions with Finally");
      quit.addActionListener(this);  enter.addActionListener(this);
      Panel input = new Panel();     input.setLayout(new BorderLayout());
      input.add("West", enter);      input.add("Center", text);
      input.add("East", quit);
      setLayout(new BorderLayout());
      add("Center",display);
      add("North", input);
      validate(); pack(); setVisible(true);
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == quit)
         System.exit(0);
      if (e.getSource() == enter)
      {  display.setText("");
         display.append("Line before try-catch-finally block\n");
         try
         {  display.append("Line before getShortText()\n");
            display.append(text.getShortText() + "\n");
            display.append("Line after getShortText()\n");
         }
         catch(LongStringException se)
         {  display.append("Exception occured: " + se.getMessage()+"\n");
         }
         finally
         {  display.append("Finally block executes\n");
         }
         display.append("Line after try-catch-finally block\n");
```

```
               }
            }
            public static void main(String args[])
            {  FinalException fe = new FinalException(); }
         }
```

Here is what will happen when you run this program and enter a short "valid" string as well as a long "invalid" string, clicking on the "Go" button in either case:



```
Line before try-catch-finally block
Line before getShortText()
Java
Line after getShortText()
Finally block executes
Line after try-catch-finally block
```



```
Line before try-catch-finally block
Line before getShortText()
Exception occured: String too long. Max: 10
Finally block executes
Line after try-catch-finally block
```

Figure 5.10: Following the flow of execution in a try-catch-finally block

In the picture on the left no exception occurs because the text in our `ShortTextField` contains only the four-letter word "Java". We can see that first the line before the `try-catch` block executes, then the entire `try` block, then the `finally` block, and then the line following the entire error-handling code.

In the picture on the right an exception occurs since the text "Java by Definition" is too long for our `ShortTextField`. We can see that the first line of the `try` block executed, then – after the exception occurred – execution switched to the `catch` block. The `finally` block executed as before, followed by the line after the error-handling code.

■

Software Engineering Tip: If you are using a method that can throw an exception, you need to embed it in a `try-catch` block with an optional `finally` block. As a rule of thumb:

- If you can find a meaningful way to catch the exception, you should do so.
- Otherwise you should consider propagating or re-throwing it after some processing. You can also convert one exception into another type and throw the new exception.
- Alternatively, you could decide to completely ignore the exception and add it to the *`throws`* statement in your method header to have someone else handle it.

You can catch multiple exceptions in one block by catching a common superclass but you should never use an empty catch block to suppress an exception[11] completely. Instead of

```
catch(Exception e)
{  }   // do not use this empty catch block
```

use as a minimum an appropriate output statement such as

[11] Since every exception extends Exception, this 'empty' catch block would always work. However, the exception, of course, may still occur but no attempt is made to deal with it in any meaningful way. The user is not even informed that anything out of the ordinary has happened. That will almost certainly jeopardize the integrity of your program.

```
catch(Exception e)
{  System.err.println("Exception occurred: " + e); }
```

5.2. Basic Threads

Now that we understand how to handle exceptions we are ready to look at some multitasking support that Java provides. In this section we will introduce threads and show several examples of using them. Programs with multiple threads can be difficult to handle, so we will restrict our attention to the basics of using threads only.

Multitasking

Multitasking is the ability of a computer operating system to run several programs or tasks simultaneously. To be precise, systems with a single CPU can only run one task at any given time. However, the operating system can switch multiple programs so quickly in and out of the CPU that they appear to execute simultaneously.

All modern operating systems such as Windows 95 and better, Macintosh System 7 and above, and Unix provide automatic support for multitasking in a transparent way. In other words, the user does not really have to be aware of the necessary – and tricky – details needed to keep track of which part of what program is currently using the CPU. A user simply starts one or more programs and all of them will appear to work concurrently. For example, while a web browser is downloading a page from the web containing lots of images, the user is busy typing a letter in a word processor. It is expected that the browser will continue retrieving the data from the web regardless of how long the letter is that is being typed.

This idea of concurrently running several tasks can be pushed one level higher: why not give *one* program the ability to perform different tasks simultaneously. This ability is called multithreading and is also built into most modern operating systems (but not Windows 3.1).

Multithreading

Multithreading is the ability of one class or program to execute and manage multiple strands, or threads, of execution. Each execution thread can execute independently of other threads, or in coordination with other threads, as desired. The class has control over its threads, can determine which ones receive priority, which have access to other class resources, which should execute and which should remain dormant, etc. The underlying operating system must provide support for multithreading but the execution of threads is controlled by the class or classes who own the threads.

In short, multitasking is the simultaneous execution of more than one program and is controlled by the operating system. Multithreading is the simultaneous execution of more than one thread within one class and is controlled by a class (with support from the operating system).

Most people use multithreaded programs without thinking about the details of the underlying process.

- A web browser downloads data and displays parts of the available data while continuing to download more. One thread of execution downloads data and informs another thread that more data has arrived. The second thread then formats and displays the information as it becomes available.
- Many word processors perform spell checking "on the fly", i.e. as the user is typing, the program analyzes the text and flags incorrectly spelled words "immediately". One thread of execution listens to the keyboard and displays the properly formatted characters on the screen while another thread "reads" the typed text and flags any spelling mistakes.

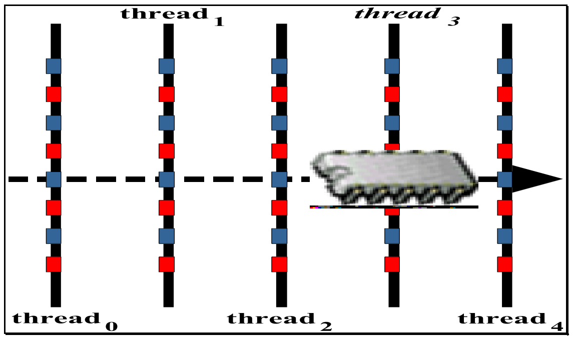


Figure 5.11: Threads simulating Multitasking

Java provides extensive support for creating multithreaded programs and tries its best to negotiate the underlying details with the operating system without user intervention. As we will see in this section, Java's mechanism for creating simple programs running two or more independent threads is surprisingly easy, and even creating multiple synchronized threads is not hard. While Java can not eliminate all technical details and can occasionally create "deadlock situations" (see section 5.3) its support for multithreading is so nice that it is worth investigating in some detail. The most extensive use of thread-based programs involves programs displaying sound, images, or advanced graphics. It is a good idea to become familiar with multithreading before delving into Swing and Multimedia programming.

The `Thread` Class

Thread

A thread is a strand of execution in a program that is executing independently of other threads. Each thread is in a specific state and has a given priority. The JVM schedules threads according to their priority and state. A thread can exist in one of four states:

- ***new*** *a thread has been created but has not yet been started*
- ***runnable*** *a thread has been instantiated and started. The JVM can give the thread time slices to run (but may not necessarily do so)*
- ***blocked*** *a thread has been instantiated and started and used to be runnable. It is not currently executing but has been put on hold (also called "wait" state)*
- ***dead*** *a thread has expired and can not be started again*

Threads can have priorities from `Thread.MIN_PRIORITY` *(=1) to* `Thread.MAX_PRIORITY` *(=10), with normal priority being equal to* `Thread.NORM_PRIORITY` *(= 5). Runnable threads with higher priority will be scheduled to execute before runnable threads of lower priority.*[12]

As usual in Java, everything is based on classes and threads are no exception. Therefore, we should first investigate the `Thread` class (see the Java API for `java.lang.Thread`) before we can produce any examples:

The `Thread` Class

The `Thread` *class is used to instantiate new threads. Using its methods, one can manipulate the priority of a thread and move it in and out of its various states:*

- *a thread is moved into the new state by instantiating it via the* `new` *operator*
- *a thread is moved into the runnable state by calling on its* `start` *method. A runnable thread is scheduled to execute the code inside its* `run` *method*
- *a thread is moved into the blocked state by calling its* `sleep` *or* `wait` *method (inherited from* `Object)`*, or because the thread performs a "blocking" operation*[13]
- *a thread is moved into the dead state if its* run *method is done, an exception has occurred, or the* `System.exit` *method has been executed*[14]

For details on the methods of the `Thread` *class, please see table 5.13.*

In other words, to create and execute a thread, one needs to perform four steps:

- create a class that extends `Thread` (so that we can override the `run` method)
- implement the `run` method with the code that the thread is supposed to execute
- instantiate an object of that new class (which will be a thread) via the `new` keyword
- start the thread by calling its `start` method

The thread will automatically finish when its `run` method is done.

Example 5.08: Extending the Thread class

Create a class that extends `Thread` and counts from 1 to 8. Then instantiate three objects of that class in another test class, start the threads, and describe what happens.

To create our first class we need to extend `Thread` and put the counting code into the `run` method that we override:

```
public class CountingThread extends Thread
{  public void run()
```

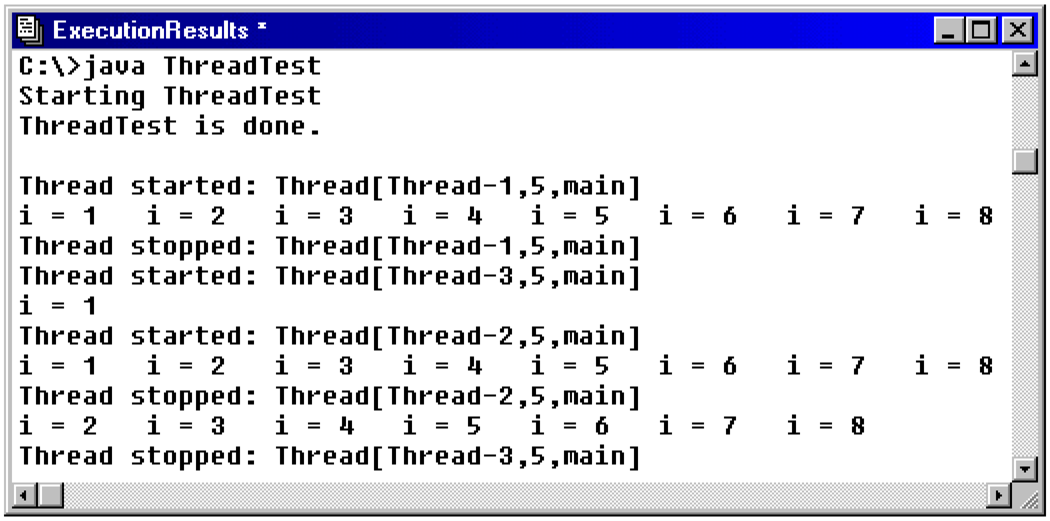
[12] In Java 1.0 threads with the same priority behaved differently on different platforms. Under Windows, they executed simultaneously (with the operating system giving each thread time slices for execution), while under Sun Solaris (Unix) the thread started first would need to finish its execution before another thread of the same priority could start. In newer versions of Java this problem is resolved.
[13] In older versions of Java it was legal to call `suspend` to put a thread into blocked state and `resume` to wake the thread up again. The use of these methods is no longer recommended since Java 1.2 and should be avoided.
[14] In older versions of Java it was legal to call `stop` to put a thread into dead state. Calling that method is no longer recommended and should be avoided.

```
      {  System.out.println("Thread started: " + this);
         for (int i = 0; i < 9; i++)
           System.out.print("i = " + (i+1) + "\t");
         System.out.println("Thread stopped: " + this);
      }
   }
```

Then we create another class to instantiate and start three objects of type `CountingThread` in its `main` method, as described in definition 5.12.

```
   public class ThreadTest
   {  public static void main(String args[])
      {  System.out.println("Starting ThreadTest");
         CountingThread thread1 = new CountingThread();
         thread1.start();
         CountingThread thread2 = new CountingThread();
         thread2.start();
         CountingThread thread3 = new CountingThread();
         thread3.start();
         System.out.println("ThreadTest is done.");
       }
   }
```

When we compile the classes and execute `ThreadTest`, it produces some happy chaos that looks similar to the following:[15]



```
C:\>java ThreadTest
Starting ThreadTest
ThreadTest is done.

Thread started: Thread[Thread-1,5,main]
i = 1   i = 2   i = 3   i = 4   i = 5   i = 6   i = 7   i = 8
Thread stopped: Thread[Thread-1,5,main]
Thread started: Thread[Thread-3,5,main]
i = 1
Thread started: Thread[Thread-2,5,main]
i = 1   i = 2   i = 3   i = 4   i = 5   i = 6   i = 7   i = 8
Thread stopped: Thread[Thread-2,5,main]
i = 2   i = 3   i = 4   i = 5   i = 6   i = 7   i = 8
Thread stopped: Thread[Thread-3,5,main]
```

Figure 5.12: Output of ThreadTest

This example illustrates that the three threads are indeed running completely independently from each other. In fact, there really are not three threads, but (at least) four[16]: the three that we created by our class and one thread running our `main` method to start everything. In particular, here is what happened during our particular execution of `ThreadTest`:

- the `main` method starts, instantiates the threads, then exits right away. before any of our three threads had a chance to start executing

[15] Each time this program is executed the output can be slightly different because we have no control over when exactly each thread starts executing. Verify that by executing the `ThreadTest` program multiple times.
[16] Using a utility program such as "System Monitor" under Windows you can verify that the operating system uses 5 threads to execute `ThreadTest`. The fifth thread is most likely the automatic garbage collector (see section 3.1).

- the first thread starts, counts to eight, then stops
- the third thread starts before the second, but only manages to count to 1
- the second thread starts, gets to count to 8 and finishes
- finally the third thread resumes execution, finishes its counting, and exits

■

This example illustrates several facts:

- it is very easy to create independently running threads of execution, with the JVM taking care of most of the technical details
- it is impossible to know exactly which thread is running at what time since that will be determined by the operating system
- threads are running truly independently of each other
- threads are running independently of the thread (or program) that started them

To facilitate control over thread execution, Java provides a variety of methods:

Method Header	Description
`public` **`Thread()`**	Allocates, but does not start, a new `Thread` object with name "Thread-"+n (n integer).
`public` **`Thread(Runnable target)`**	Allocates a new `Thread` object and uses the `run` method of target as opposed to its own run method
`public static native void` **`yield()`**	Causes the currently executing thread object to pause and allow other threads to execute.
`public static void` **`sleep(long millis)`** `throws InterruptedException`	The currently executing thread ceases execution for the specified number of milliseconds.
`public synchronized void` **`start()`**	This thread begins execution and the JVM calls the `run` method of this thread.
`public void` **`run()`**	Unless the thread was constructed using a separate `Runnable run` object, this method does nothing. Otherwise, the Runnable object's `run` method is called.
`public final native boolean` **`isAlive()`**	Tests if this thread is alive, i.e. it has been started and has not yet died.
`Public final void` **`setPriority(int new)`**	Changes the priority of this thread.
`public static Thread` **`currentThread()`**	Returns a reference to the currently executing thread object
`public final void` **`notify()`**	Wakes up a single thread that is waiting on this object's monitor (inherited from `Object`).
`public final void` **`notifyAll()`**	Wakes up all threads that are waiting on this object's monitor (inherited from `Object`).
`public final void` **`wait()`** `throws InterruptedException`	Causes current thread to wait until another thread invokes `notify` or `notifyAll` for this object (inherited from `Object`).

Table 5.13: Description of selected constructors and methods in the Thread.class

Since threads require a certain amount of processor time, they should *always* be cooperative and signal the processor when they do not actually need processing. The `sleep` and `yield` methods of a thread serve precisely that purpose:

- If you want to have a short break in your computation, you should call – inside the run method – the thread's sleep method. That will not only pause this thread, but also allow the operating system to schedule other threads waiting to be processed.
- If your computation in the run loop really does not lend itself to taking a break, at least call yield in the run loop of the thread. That again will give the operating system a chance to schedule other waiting threads if necessary.

While a thread automatically stops once its run method finishes, there are occasions when you might want to stop a thread *before* that method is done. That is easily accomplished using a boolean variable and the following rule of thumb[17]:

Software Engineering Tip: Once a thread is in the runnable state, it will continue to execute until its *run* method is done. To provide an easy mechanism for stopping a thread at any time, use the following steps:

- extend the Thread class
- add a boolean variable running to the class and initialize it to false
- override the start method to first set running to true, then call super.start()
- provide a public method halt that sets running to false
- use a while loop in the run method similar to the following:

```
public void run()
{  while (running)
   { /* your code here */ }
}
```

If the halt method of such a thread class is called it will set running to false and therefore cause the run method to stop executing, finishing the thread.

Example 5.09: A program using multiple threads

In this example the goal is to create a program that simulates a "table" with twenty or so moving "marbles" on it. Each "marble" is a class that extends Thread and is responsible for its own movement. The "table" should extend Frame and contain buttons to start and stop the marbles. The marbles should be bouncing on the sides of the table so that they do not leave it and be initially placed at random locations on the table.

Let's start with the "Marble" class: the basic idea is not so difficult:

- The class will extend Thread so that it can independently execute in its own thread.
- We will follow the above rule of thumb to allow the marbles to be stopped at any time.
- The fields for the class are x and y for the current coordinates of the marble.
- To move a marble to a new (x,y) location, we recompute its coordinates in the run method, which will have the effect of independently moving it.
- Since there should be several moving marbles, we will call on the sleep method inside the run method to give the operating system time to move other marbles.

[17] The Thread class does provide a stop method to immediately interrupt a thread, but that method is not safe and may leave other objects in an inconsistent state. Since Java 1.2 the stop method should no longer be used and has been deprecated.

- To make sure the marble "bounces" off the edges of the table, we need to know the size of the table. Therefore, we pass a reference to the table into this class via the constructor so that a marble can query the table for its size.

Here is the code so far:

```
import java.awt.*;

public class Marble extends Thread
{  private int xdir = 2*(1-2*(int)Math.round(Math.random()));
   private int ydir = 2*(1-2*(int)Math.round(Math.random()));
   private boolean running = false;
   private Table table = null;
   protected int x, y;
   public Marble(Table _table, int _x, int _y)
   {  table = _table;
      x = _x;
      y = _y;
      start();
   }
   public void start()
   {  running = true;
      super.start();
   }
   public void halt()
   {  running = false; }
   public void run()
   {  while (running)
      {  move();
         try
         {  sleep(50); }
         catch(InterruptedException ie)
         { System.err.println("Thread interrupted"); }
      }
   }
   private void move()
   {  x += xdir;
      y += ydir;
      if ((x > table.getSize().width) || (x < 0))
         xdir *= (-1);
      if ((y > table.getSize().height) || (y < 0))
         ydir *= (-1);
   }
}
```

As described in the above Software Engineering Tip the `run` method contains a loop that can be controlled by changing the value of running to false via the halt method. In the constructor of the class we pass the initial values of the (x,y) location for the marble into the object.

Also, the "bouncing" is handled in a straightforward manner: add a certain value to the current `x` and `y` coordinate of the marble. Then check if it is now outside the drawing area of the table. If so, make the marble "reverse direction" by switching the sign of the value that is added to the `x` or `y` coordinates, respectively. Note that the `move` method is marked `private` so that only the thread itself can call it, but the `x` and `y` fields are marked as `protected` so that they can be referred to by `Table` (see below).[18]

[18] To be picky, marking `x` and `y` as `protected` leaves them open for changes from outside the class even though the `move` method is marked `private`. While this approach saves space, a better idea would be to mark `x` and `y` as `private` and add methods `getX` and `getY` to obtain their values.

As a little trick, the initial values of `xdir` and `ydir` are choosen at random so that they are either 2 or –2, because:

- `Math.random()` returns a random double between 0.0 and 1.0
- `Math.round(Math.random())` returns either 0.0 or 1.0
- `2*(int)Math.round(Math.random())` returns either 0 or 2
- `1 - 2*(int)Math.round(Math.random())` returns either –1 or 1
- `2*(1 - 2*(int)Math.round(Math.random()))` returns either –2 or 2

This, of course, does not settle the question of how to actually draw the marble. Before we go into that, let's take a look at our Table class:

- We clearly need three fields for the buttons `Quit`, `Start`, and `Stop`. In addition, we will use an array to store references to 20 marbles (or some number like that)
- In the constructor, we layout everything, resize the frame for 300 by 300 pixels, and make it visible. We also initialize the marbles at random locations.
- In the `actionPerformed` method we use `for` loops to call the methods `start` and `halt` of the marble array when the corresponding buttons are pressed.
- There is, of course, the standard `main` method to get everything started.

```
import java.awt.*;
import java.awt.event.*;

public class Table extends Frame implements ActionListener
{  private Button quit  = new Button("Quit");
   private Button start = new Button("Start");
   private Button stop  = new Button("Stop");
   private Marble marbles[]   = new Marble[20];

   public Table()
   {  super("Table with Marbles");
      setLayout(new FlowLayout());
      add(quit);  quit.addActionListener(this);
      add(start); start.addActionListener(this);
      add(stop);  stop.addActionListener(this);
      validate(); setSize(300,300);
      setVisible(true);
      for (int i = 0; i < marbles.length; i++)
      {  int x = (int)(getSize().width*Math.random());
         int y = (int)(getSize().height*Math.random());
         marbles[i] = new Marble(this, x, y);
      }
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == stop)
         for (int i = 0; i < marbles.length; i++)
            marbles[i].halt();
      if (ae.getSource() == start)
         for (int i = 0; i < marbles.length; i++)
         {  marbles[i].halt();
            marbles[i] = new Marble(this, marbles[i].x, marbles[i].y);
         }
      if (ae.getSource() == quit)
         System.exit(0);
   }
   public void paint(Graphics g)
   { /* still to do */ }
   public static void main(String args[])
   {  Table table = new Table(); }
}
```

If the `start` button is called, we first make sure all current threads are done by calling their `halt` methods before initializing a new set of marbles. In particular, when we initialize a new set of marbles we use coordinates of the current location for each marble. That will cause the movement of each marble to resume where it stopped, even though they are completely new objects. That's the reason for marking the x and y fields of a Marble as protected.[19]

We still need to solve the problem of how the marbles are going to actually appear on the screen. Of course, each marble should draw itself but it does not have a `Graphics` object with which to do the drawing. Our solution will be to pass a `Graphics` object from the `Table`'s `paint` method to an appropriate method for each of the marbles in our array. Hence the `Table`'s `paint` method will be:

```
public void paint(Graphics g)
{  for (int i = 0; i < marbles.length; i++)
      if (marbles[i] != null)
         marbles[i].draw(g);
}
```

The corresponding `draw` method in the `Marble` class is also simple; it just draws a filled oval of some specific size at the current (x,y) coordinates. We therefore add the following method to the `Marble` class:

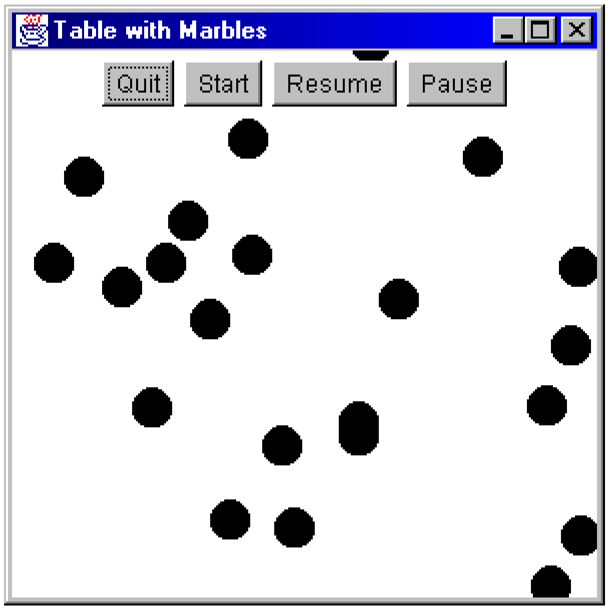
```
public void draw(Graphics g)
{  g.setColor(Color.black);
   g.fillOval(x,y,20,20);
}
```

But this will still not work ! The marbles are running independently, and each of their `run` methods (when started) will continuously change to coordinates at which the marble should be drawn. However, the actual method to *do* the drawing is the `paint` method in the `Table` class, and that is – currently – only called when the frame needs updating. So to make sure that the table is redrawn any time a marble changes its coordinates we place a call to the `Table`'s `repaint` method in the `run` method of the `Marble`. Here's the new, modified method, with one additional line in bold and italics:

```
public void run()
{  while (true)
   {  move();
      try
      {  sleep(40); }
      catch(InterruptedException ie)
      { System.err.println("Thread interrupted"); }
      table.repaint();
   }
}
```

While we can not see this movement on paper, here is nevertheless a screen shot of the frame:

[19] There is actually a slight "mistake" in our class. In order for the marbles to continue their movement after clicking on start, we need to copy the complete state of the old marble into a new one. We do that with the (x,y) coordinates but fail to do it with the current values of `xdir` and `ydir`. Hence, every time `start` is clicked the marbles will get a new random direction for their movement.



Frame 5.14: Tables with Marbles, each executing in its own thread

It is worth reflecting for just a second what is going on:

- When the `Start` button is pressed, each Marble starts executing its `run` method.
- The `run` method in each marble computes new coordinates, sleeps for 40 milliseconds, then calls the `repaint` method of the table.
- That, in turn, results in updating the image and creates the illusion of continuous movement.

■

Now we have some idea about using threads. We have so far extended the `Thread` class to produce a thread, but that's not always necessary. In fact, most programs need a thread only to override its `run` method and very little else. One solution would be, of course, to have the program extend, say, `Frame` *and* `Thread` but as we know Java does not support multiple inheritance. Instead, Java provides an `Interface` called `Runnable` to enable any class to use additional threads without having to extend the `Thread` class.

Threads via the `Runnable` Interface

The `Runnable` interface in Java is an alternate method to using threads in a program. In many situations, a class already extends a class like `Frame` or `Applet`, so it can not extend `Thread` also. One could of course create another class that subclasses `Thread` and use that class to do some of the work. However, in many situations a much simpler approach is possible using the `Runnable` interface.

The `Runnable` Interface

The `Runnable` *interface provides a means for classes to execute, or be controlled by, a thread without extending the* `Thread` *class. The standard use of this interface involves a class implementing the* `public void run` *method and using a field of type* `Thread`*. When instantiating* *that thread, a reference to itself is used as input parameter which will link the* `run` *method of the class to the thread so that it can be controlled via the thread's* `start`*,* `sleep`*, and* `yield` *methods.*

```
[modifier] class ClassName [extends SuperClassName]
                            implements Runnable [, OtherInterfaceList]
{   /* fields, constructors and methods
    public void run()
    {   /* code for the run method */
    }
}
```

Using this interface, a class can extend other classes and in addition use one (or more) threads. Even if a class could extend `Thread`, it may want to use the `Runnable` interface if all that would be necessary is to implement the `run` method of the thread.

Example 5.10: Displaying the time using Runnable

Create a program that prints out the current time every second (like clockwork).

Our first problem, of course, is to get the current time (and date) from the system. Searching through the Java classes, we find a class in the `java.util` package called `Date` which will serve the bill just fine: instantiating a new `Date` object will give us the current time, and the `toString` method can be used to get its string representation (see below for a definition of the `Date` class). The next decision we have to make is whether we need a thread, and if so, whether we should extend the `Thread` class or use the `Runnable` interface.

First, since something is to occur every second a thread is perfect: we can wake up a thread once per second, and have it print out the time at that instance. Moreover, that is pretty much all we need to do, and a call to the `System.out.println` method will fit nicely into the `run` method. Therefore, since our use of a thread is simple, we choose to use the `Runnable` interface:

```
import java.util.Date;

public class ShowSeconds implements Runnable
{  private Thread clocker = null;
   private Date now = new Date();

   public ShowSeconds()
   {  clocker = new Thread(this);
      clocker.start();
   }
   public void run()
   {  while (true)
      {  now = new Date();
         System.out.println(now);
         try
         {  clocker.sleep(1000); }
         catch(InterruptedException ie)
         {  System.err.println("Thread error: " + ie); }
      }
   }
   public static void main(String args[])
   {   ShowSeconds time = new ShowSeconds(); }
}
```

The thread `clocker` is initialized in the constructor with the current instance of the `ShowSeconds` class as input parameter (which works since the class implements `Runnable`). That links the `run` method of the `clocker` thread to the `run` method of our class `ShowSeconds`. Therefore, when the thread is started, the `run` method of our class begins executing. Just as before this method contains an infinite loop. However, this time we forgot to provide any mechanism to allow us to `stop` the thread, so this time the run look truly is infinite – this is not at all good programing.

We have to actually instantiate a new `ShowSeconds` instance as opposed to using only the static main method. If we want to associate a thread with the `run` method of our class, we need to pass a reference `this` to the class itself into the thread's constructor. That only works if there *is* an instance of `this` class, as opposed to a class using only static methods.

■

As mentioned, the above class will run forever (or to be more precise, it will not stop by itself and needs outside intervention before quitting). That, of course, is not what we should do. In addition, we might want to create a fully functioning `Clock` class that can be reused in other applications at any time instead of create a simple, standalone program for demonstration purposes only. Before creating that class, let us take a look at the `Date` class that Java provides:

The `Date` Class

The `Date` *class is used to represent an instance in time, including the date. The actual representation of the* `Date` *is Universal Time (UT)*[20]*, but..*

```
public class Date extends Object
   public Date()
   /* Creates a Date and initializes it to the current time */
   public long getTime()
   /* number of milliseconds since January 1, 1970, 00:00:00 */
   public void setTime(long time)
   /* represent the number of milliseconds since January 1, 1970 */
   public boolean before(Date when)
   public boolean after(Date when)
   public boolean equals(Object obj)
   public String toString()
```

Java also provides the `Calendar` *and* `DateFormat` *class to properly format time based on a country's time convention.*

Now let's create a reusable, self-running clock that can be properly started and stopped any time. We have already seen how to start and stop a class extending `Thread` but when we use the `Runnable` interface a different technique is more appropriate.

Software Engineering Tip: When using Threads you have to be sure that all threads that have been instantiated and started are truly stopped when they are not needed. Otherwise, they may consume system resources without any tangible benefit. Classes that implement the `Runnable` interface may want to use the following framework:

```
public class ClassName extends SuperClass implements Runnable
{  private Thread threadName = null;
   // class-specific fields and methods, could include
```

[20] The details are actually not easy because of the intricacies of time measurement For further details on "time", see http://tycho.usno.navy.mil, as well as the Java API for the `Calendar` and `DateFormat` classes.

```
      // a call to "start" to actually start the thread
      public void start()
      {  if (threadName == null)
         {  threadName = new Thread(this);
            threadName.start();
         }
      }
      public void stop()21
      {  threadName = null; }
      public void run()
      {   Thread currentThread = Thread.currentThread();
          while (threadName == currentThread)
          { /* class-specific code calling threadName.sleep(sleepTime)
               or threadName.yield() */}
      }
```

Naming two of the above methods `start` and `stop` does not mean that they are called automatically; they only provide convenient means to properly start and stop the thread.

The `start` method checks if the thread is currently `null`. If so, it indicates that the thread is not running so we instantiate a new thread and tie it to the `run` method of our class. Then we call the thread's `start` method, which in turn begins the `run` loop. Before the `while` loop of the `run` method starts, we use the `static currentThread` method of the `Thread` class to obtain a reference to the currently executing thread. That reference should be identical to the thread field initialized in the `start` method. The `while` loop will then execute as long as both references agree. In the `stop` method we change the thread field to point to `null`. That will stop the `run` method the next time the condition in the `while` loop is check, and also marks the thread for automatic garbage collection.[22]

Example 5.11: A reusable Clock class

Create a `Clock` class that will emulate a digital clock. It should have the possibility of being started and stopped at any time, but otherwise run completely independent of anything else.

The first thing we need to determine is which class we should extend. One option would be to extend `Canvas` and display the date string in the `paint` method. However, there is a simpler option: our time (converted to a string) is non-editable text, so why not display it in a `Label`? In fact, why not have our `Clock` class extend `Label`? Now that the difficult question is solved, the rest is easy: we follow the above rule of thumb and add a field `clocker` of type `Thread` to our class as well as the methods `start`, `stop`, and `run`, as outlined in the above Software Engineering Tip. In the `while` loop of the run method we set the text of the label to the new system time every second by putting the thread to sleep for 1000 milliseconds. The constructor initializes the label with the current time. Here is the code:

```
import java.awt.Label;
import java.util.Date;

public class Clock extends Label implements Runnable
{  private Thread clocker = null;

   public Clock()
```

[21] In our earlier Software Engineering Tip for stopping a thread we called the method to stop a thread `halt` instead of `stop`. That was necessary because a class extending `Thread` inherits a `final` method `stop`, which cannot be overridden. In our current example there is no `final stop` method so we can choose this name instead of `halt`.
[22] The idea is similar to the one used in our earlier Software Engineering Tip for stopping a thread but there we needed an additional boolean variable to control the run method. Here we can use the state of the thread field itself.

```
         {  setText(new Date().toString()); }
         public void start()
         {  if (clocker == null)
            {  clocker = new Thread(this);
               clocker.start();
            }
         }
         public void stop()
         {  clocker = null; }
         public void run()
         {  Thread currentThread = Thread.currentThread();
            while (clocker == currentThread)
            {  setText(new Date().toString());
               try
               {  clocker.sleep(1000);
               }
               catch (InterruptedException ie)
               {  setText("Clock broken"); }
            }
         }
      }
```

Again, we seem to have created an infinite loop in our `run` method, but we can use the stop method to interrupt the loop. Therefore, the loop is not really infinite and our class should work perfectly fine.

■

This class can now be used in any program or applet, and it will tell the time independently of anything else that is going on in that class.

Example 5.12: Using the Clock class

Create a simple applet that uses the prior `Clock` class.

The `Clock` extends `Label`, so it can be positioned with a layout manager. We also use two buttons to start and stop our clock. The code is completely straightforward.

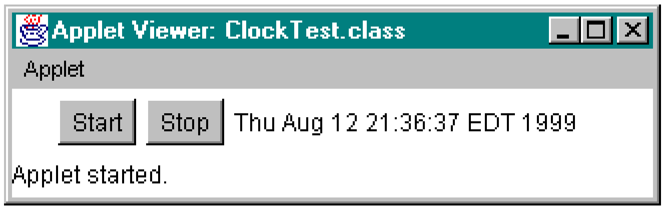
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ClockTest extends Applet implements ActionListener
{  private Button start = new Button("Start");
   private Button stop  = new Button("Stop");
   private Clock  clock = new Clock();

   public void init()
   {  setLayout(new FlowLayout());
      add(start); start.addActionListener(this);
      add(stop);  stop.addActionListener(this);
      add(clock);
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == stop)
         clock.stop();
      else if (ae.getSource() == start)
         clock.start();
   }
}
```

This applet will place a clock via the `FlowLayout` manager, and the clock will start ticking as soon as the `start` button is clicked. But now we have a problem due precisely to the fact that threads run independently of everything else: if a user looks at a web page containing this applet, the clock will start running as it is supposed to. But when a user leaves the page, the clock will continue to run. That is, of course, unnecessary and in fact wastes system resources: why should the clock continue to run even if nobody is looking at it?

Therefore, we need to implement the `start` and `stop` methods of an `Applet` that are called automatically (see *The `Applet` Class* in section 4.3) when a user leaves and revisits the web page containing the applet, so we implement these methods as follows:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class ClockTest extends Applet implements ActionListener
{  private Button start = new Button("Start");
   private Button stop  = new Button("Stop");
   private Clock  clock = new Clock();

   public void init()
   {  setLayout(new FlowLayout());
      add(start); start.addActionListener(this);
      add(stop);  stop.addActionListener(this);
      add(clock);
   }
   public void start()
   {  clock.start(); }
   public void stop()
   {  clock.stop(); }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == stop)
         clock.stop();
      else if (ae.getSource() == start)
         clock.start();
   }
}
```

Now the clock will start as soon as the applet has finished loading (because the `start` method is called automatically), and it will stop when the user leaves the page containing the applet (because the `stop` method is automatically called).



```
Thu Aug 12 21:36:37 EDT 1999
Applet started.
```

Figure 5.15: The Clock class embedded in an applet

■

Here is another popular example of using a thread via the `Runnable` interface.

Example 5.13: A reusable Ticker class

Create a reusable "Ticker Tape" class, i.e. a class that displays some text that moves slowly across its display area from right to left. Then create an applet that tests this class together with the previous `Clock` class.

As usual, we first need to decide which class we want to extend. This time, we will extend a `Canvas` and implement the `Runnable` interface, since the `paint` method of a `Canvas` allows us complete control over the placement of the string.

The general idea is simple: set an integer field `xPos` to the width of the screen and another one `yPos` to the height of the string. Then draw the string at position (`xPos`, `yPos`). In the `run` method, decrease `xPos` and redraw the string: it will appear to move to the left. Continue to decrease `xPos` until the last character of the string has "left" the display area, at which time you want to reset `xPos` to the width of the display area. In other words, our basic class will look as follows (using the above framework):

```
import java.awt.*;

public class Ticker extends Canvas implements Runnable
{  private int xPos, yPos;
   private Thread runner = null;
   private String text   = null;

   public Ticker(String _text)
   {  text = _text;
      start();
   }
   public void start()
   {  if (runner == null)
      {  runner = new Thread(this);
         runner.start();
      }
   }
   public void stop()
   {  runner = null; }
   public void run()
   {  Thread currentThread = Thread.currentThread();
      while (runner == currentThread)
      {  computeCoordinates();
         repaint();
         try
         {  runner.sleep(50); }
         catch (InterruptedException ie)
         {  System.err.println("Error: " + ie);
         }
      }
   }
   public void paint(Graphics g)
   {   g.drawString(text, xPos, yPos); }
   private void computeCoordinates()
   {  /* to be implemented */ }
}
```

That, of course, leaves out the method `computeCoordinates` to reset the coordinates: we want to reset the decrement `xPos` until the *end* of the string is about to leave the drawing area. Since at that time the `xPos` would be equal to the width of the text, it means that we need to be able to determine the width of a `String`. But that width (and the height as well) depends on the particular font selected, its point size, and its appearance (bold, italic, normal). Therefore, before we can continue, we must search the Java API for classes appropriate to give us this information. In section 4.7 we have already introduced the `Font` class, but we also need a class to measure the sizes of characters in a particular font.

The `FontMetrics` Class

The `FontMetrics` *class is used to retrieve information about a particular font, and to measure the dimensions of a* `String` *in a particular* `Font`.

```
public abstract class FontMetrics extends Object
{  // Constructor
   protected FontMetrics(Font font)
   // Selected Methods
   public int getLeading()
   public int getAscent()
   public int getDescent()
   public int getHeight()    // = leading + ascent + descent
   public int charWidth(char ch)
   public int stringWidth(String str)
}
```

The `FontMetrics` *class is an abstract class, and can therefore not be instantiated. Instead, every Component offers a method* `getFontMetrics` *that will return an appropriate reference to a* `FontMetrics` *object.*

In order to measure a String in a particular font, we need to make sure that the string is really drawn in that font, then use an instance of a `FontMetrics` class to measure its height and width. In our class, we will accomplish this by adding fields for the current height, width, and font, and setting their values via the `FontMetrics` class. Here is the modified class, with the changed code in bold and italics:

```
public class Ticker extends Canvas implements Runnable
{  private int xPos, yPos;
   private int height, width;
   private Thread runner = null;
   private String text   = null;
   private FontMetrics metrics = null;
   private Font font = new Font("Monospaced", Font.BOLD, 12);

   public Ticker(String _text)
   {  text = _text;
      metrics = getFontMetrics(font);
      width = metrics.stringWidth(text);
      height = metrics.getHeight();
      xPos = getSize().width;
      yPos = height;
      start();
   }
   public void start() { /* no change */ }
   public void stop()  { /* no change */ }
   public void run()   { /* no change */ }
   public void paint(Graphics g)
   {   g.setFont(font);
       g.drawString(text, xPos, yPos);
   }
   private void computeCoordinates()
   {  /* to be implemented */ }
}
```

Now that we know the dimensions, in pixels, of the string, we can also implement the `computeCoordinates` method that is being referred to in the run method:

```
private void computeCoordinates()
{  if (xPos < -width)
      xPos = getSize().width;
   else
      xPos -= 2;
}
```

At this point it seems our class is complete. However, we would like to position a Ticker object using a layout manager. When a layout manager wants to add an object, it queries that object as to which size it would like to have. Then it tries to accommodate the object, based on that information, as best as possible under the constraints of that manager. The method that is queried is the `getPreferredSize` method, which is part of every component. Since our canvas, however, does not override this method, a layout manager will not properly size our `Ticker` class unless we implement `getPreferredSize` (see Software Engineering Tip in section 4.7). Therefore, we add one more method to our class before testing it:

```
public Dimension getPreferredSize()
{  return new Dimension(width, height+3); }
```

which will inform a layout manager that our `Ticker` wants to be at least as big as the dimensions of the text it is displaying.[23]

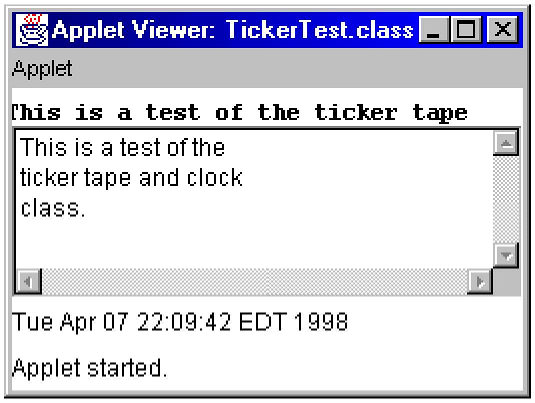
Now we have a reusable "ticker tape" class that we can add to any class we create. Here is a simple example:

```
import java.applet.*;
import java.awt.*;

public class TickerTest extends Applet
{  private Ticker ticker = new Ticker("Testing the ticker tape class");
   private Clock  clock  = new Clock();
   private TextArea display = new TextArea();

   public void init()
   {  setLayout(new BorderLayout());
      add("North", ticker);
      add("Center", display);
      add("South", clock);
      display.append("This is a test of the\n");
      display.append("ticker tape and clock\n");
      display.append("class.");
   }
   public void start()
   {  ticker.start();
      clock.start();
   }
   public void stop()
   {  ticker.stop();
      clock.stop();
   }
}
```

[23] It is not necessary to add the `getPreferredSize` to the `Clock` class in example 5.11. That class extends `Label`, not `Canvas`, and a labels preferred size is automatically as big as the text it contains initially.

This will place the ticker tape at the top of the applet, the clock at the bottom, and some display area in the middle. The clock and the ticker tape will run independently of everything else, using their own threads.

Note that our applet implements the `start` and `stop` methods to properly stop (and restart) the threads when a user leaves and reenters the web pages containing this applet. As we mentioned, threads should usually be stopped when a user is not viewing an applet to conserve system resources.


Applet Viewer: TickerTest.class
Applet
This is a test of the ticker tape
This is a test of the ticker tape and clock class.
Tue Apr 07 22:09:42 EDT 1998
Applet started.
Figure 5.16: Testing the Ticker and Clock class

■

As a final example of threads, here is a class and a test program to figure out:

Example 5.14: A reusable TimedCounter class to generate an ActionEvent

Take a look at the class and test program below and explain what is going to happen when the test program is started.

```
import java.awt.*;
import java.awt.event.*;

public class TimedCounter implements Runnable
{  private ActionListener listener = null;
   private Object object = null;
   private Thread thread = null;
   private int timer     = 0;
   private int time       = 5;

   public TimedCounter(ActionListener _listener, Object _obj, int _time)
   {  time   = _time ;
      object = _obj;
      listener = AWTEventMulticaster.add(listener, _listener);
   }
   public void start()
   {  if (thread == null)
      {  thread = new Thread(this);
         thread.start();
      }
   }
   public void stop()
   {  thread = null; }
   public void run()
   {  Thread currentThread = Thread.currentThread();
      while ((currentThread == thread) && (timer < time * 1000))
      {  timer += 100;
         try
         {  thread.sleep(100); }
         catch(Exception e)
         {  System.err.println("Unknown error " + e); }
         if (timer >= time * 1000)
         {  ActionEvent e = new ActionEvent(object,
                                     ActionEvent.ACTION_PERFORMED,
                                            "time up");
            listener.actionPerformed(e);
         }
      }
   }
}

import java.awt.*;
import java.awt.event.*;

public class TestProgram extends Frame implements ActionListener
{  private Button quit = new Button("Quit");
   private TimedCounter counter = null;

   public TestProgram()
   {  quit.addActionListener(this);
      counter = new TimedCounter(this, quit, 5);
      setLayout(new FlowLayout());
      add(quit);
      validate(); pack(); setVisible(true);
      counter.start();
   }
   public void actionPerformed(ActionEvent e)
   {  if (e.getSource() == quit)
      {  counter.stop();
         System.exit(0);
      }
   }
   public static void main(String args[])
   {  TestProgram tp = new TestProgram(); }
}
```

Clearly the `TestProgram`, when started, will instantiate and display a frame with a single button labeled "Quit". If you click on that button the frame closes and the program quits. The purpose of the first class is to attach a timer to an active component so that an action event with the proper signature is generated automatically after a specified time has expired. The test program starts the thread in its constructor, which will cause the `run` method of `TimedCounter` to execute. That method starts counting every 100 milliseconds, until either the value of `timer` is at least as large as `time` or the `stop` method has interrupted the thread. After a given number of seconds (in this case 5), the class generates an action event which has as its id the identity of that object used in the constructor of the class. In this case it means that after 5 seconds the `TimedCounter` class will generate an event that simulates a click on the `quit` button. In other words, if the frame appears you have 5 seconds before the frame will shut down automatically because the `TimedCounter` will simulate the appropriate action event. If you click the button before the 5 seconds are up, the thread will be stopped because the `actionPerformed` method in the `TestProgram` class calls `counter.stop()`.

This `TimedCounter` class can be attached to any object that generates action events to put a counter on that object that will automatically fire the corresponding action event as if a user has generated the event. With this class you can, for example, create programs that offer users certain choices and if the user does not react after a fixed amount of time the corresponding choices are activated automatically. Since each counter uses its own thread, many such counters can be attached to action objects and they will all count down independently of each other.

■

We will conclude our introduction to threads by briefly discussion synchronization problems (and solutions) when using multiple threads.[24]

5.3. Synchronized Threads and Methods

When you create a class using threads, all threads created have complete access to the fields and methods of the class or to public fields and methods of any class for which the thread has a reference. For classes using a single thread, that does not usually pose a problem, but for classes that use multiple threads this can result in serious problems.

Data Corruption using Multiple Threads

For example, two threads may have access to the same integer field. One thread reads the value of the field and performs some action based on that value. While the first thread is performing its action, the second thread changes the value of the integer. Now the action of the first thread is no longer in sync with the value of the integer, but the first thread does not know about that change.

Example 5.15 Data corruption using slow and fast threads

Create a class that contains an integer field `number` with a positive random value and a method `performWork` to perform two types of work: a "slow" task and a "quick" task The type of work to perform should depend on the input parameter to the method. Instead of performing the work itself by calling the method directly, however, the class should instantiate two threads, a "fast" and a "slow" one, which in turn call `performWork` as follows:

- The "Slow Worker" should call `performWork` and cause it to display the random number, put the thread to sleep for 2 seconds to simulate a long-lasting task, then display the number again.
- The "Fast Worker" should cause `performWork` to modify the number to negative 1, without any delays, simulating the quick task.

The first class is straight forward, according to the instructions. When instantiating the two threads, we need to make sure to pass a reference to the current instance of the class to the threads so that both threads have access to the `performWork` method. We also pass a parameter along to indicate which type of work each thread is supposed to perform. Finally, we add a `main` method so that this class is executable:

```
public class CorruptedData
{  protected static int DISPLAY = 1, CHANGE = 2;
   private WorkThread slowWorker = null;
   private WorkThread fastWorker = null;
   private int number = 0;

   public CorruptedData()
   {  number = (int)(10*Math.random());
      slowWorker = new WorkThread(this, DISPLAY);
```

[24] This is particularly relevant for chapter 6 where we introduce Swing components. Swing components are not "thread-safe" and should be synchronized when used in programs with threads.

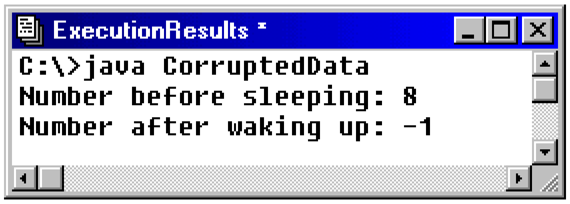
```
            fastWorker = new WorkThread(this, CHANGE);
         }
         public void performWork(int type)
         {  if (type == DISPLAY)
            {  System.out.println("Number before sleeping: " + number);
               try
               { slowWorker.sleep(2000);
               }
               catch(InterruptedException ie)
               {  System.err.println("Error: " + ie); }
               System.out.println("Number after waking up: " + number);
            }
            if (type == CHANGE)
               number = -1;
         }
         public static void main(String args[])
         {  CorruptedData cd = new CorruptedData(); }
      }
```

Note that we can instantiate the threads only in the constructor, not in the `main` method or field definitions because they contain a reference to `this` which is not available in the `static main` method.

The thread class is again easy to implement: it simply defines fields for a reference to the main class and to the type of work to perform, starts the thread, and in the `run` method calls the `performWork` method of the main class, using the appropriate parameter.

```
      public class WorkThread extends Thread
      {  private CorruptedData data = null;
         private int work           = 0;

         public WorkThread(CorruptedData _data, int _work)
         {  data = _data;
            work = _work;
            start();
         }
         public void run()
         {  data.performWork(work);
         }
      }
```

When we compile and run the `CorruptedData` class, we will see the output similar to the following:



```
C:\>java CorruptedData
Number before sleeping: 8
Number after waking up: -1
```

Figure 5.17: Data Corruption by multiple Threads

The program starts and instantiates two threads, both of which start running. The "slow thread" prints out the value of `number`, then sleeps for two seconds. When it wakes up it prints out the value again: it has – seemingly inexplicably – changed to negative one, even though there is no change of that sort occuring in the currently executing branch of the `if` statement. Of course, what has happened was that while the slow thread was sleeping (or perfoming another long and perhaps more useful task depending on the value of `number`), the second "stealth" method has *changed* the value of number. The first thread, however, still operates under the assumption that number has its original value, which could lead to all kinds of undesired results.

■

This example might seem somewhat contrived but it is not hard to create a more realistic example that will illustrate what can happen if multiple threads have concurrent access to data:

Example 5.16 Data corruption in a bank account simluation

Create a class Bank that contains an array of 8 integers, simulating savings accounts, each initially containing $100,000. The Bank class should contain a method transfer that can transfer money from one savings account to another, provided the first one has enough funds. That method should "waste some time" between taking money out of one account and putting it into another account. In addition, there should be a method showAccounts that displays a summary of all accounts. The Bank class should also instantiate 8 threads simulating customers that own the accounts, one customer per account. These customer threads then continue to transfer random amounts of money (up to $1,000 per transaction) from their own account into another account choosen at random, using the Bank's transfer method to conduct the transfer (if there are sufficient funds). Run this program in a frame, using buttons to display the account summary and to restart the computations. Explain.

The basic idea of the Bank class is fairly clear: it contains a field for the integer array representing the accounts and another field for an array of Customer threads. We will also add a field to count the number of transactions performed. In the constructor we will initialize the accounts and customer threads. The transfer method needs to have three input parameters: the account from which to take the money, the account into which to deposit the money, and the amount of money to transfer. Before conducting the transfer we should check if there are sufficient funds in the withdrawal account. To waste time between computations, we will call on the static sleep method of the Thread class. Here is the basic layout of the important methods of our class, not counting any GUI coding:

```
import java.awt.*;
import java.awt.event.*;

public class Bank
{  protected final static int NUM_ACCOUNTS = 8;
   private   final static int WASTE_TIME   = 1;
   private int      accounts[] = new int[NUM_ACCOUNTS];
   private Customer customer[] = new Customer[NUM_ACCOUNTS];
   private int      counter    = 0;
   public Bank()
   {  for (int i = 0; i < accounts.length; i++)
         accounts[i] = 100000;
      start();
   }
   public void transfer(int from, int into, int amount)
   {  if ((accounts[from] >= amount) && (from != into))
      {  int newAmountFrom = accounts[from] - amount;
         int newAmountTo   = accounts[into] + amount;
         wasteSomeTime();
         accounts[from] = newAmountFrom;
         accounts[into] = newAmountTo;
      }
   }
   private void start()
   {  stop();
      for (int i = 0; i < accounts.length; i++)
         customer[i] = new Customer(i, this);
   }
   private void stop()
   {  for (int i = 0; i < accounts.length; i++)
         if (customer[i] != null)
            customer[i].halt();
   }
   private void wasteSomeTime()
   {  try
      {  Thread.sleep(WASTE_TIME); }
      catch(InterruptedException ie)
      {  System.err.println("Error: " + ie); }
   }
   private void showAccounts()
   {  /* shows account information for all accounts */ }
   public static void main(String args[])
   {  Bank bank = new Bank(); }
}
```

The important method is `transfer`: it first checks whether the `from` account contains enough funds and the `from` account is different from the `into` account. Then it computes the new amounts in each account after transferring the money, but proceeds to waste some time before actually adjusting the accounts.

Before we add the GUI code to the `Bank` class, let's introduce the `Customer` class. That class extends `Thread` and follows our Software Engineering Tip for starting and stopping a thread. In addition to the `boolean` field `running` described in that definition, we need two additional fields for references to the `Bank` object and for the id of the account corresponding to this customer. We then start the thread, which in its `run` method determines random values for the amount to transfer and the account into which to deposit the money.

```
public class Customer extends Thread
{  private Bank    bank    = null;
   private int     id      = -1;
   private boolean running = false;
   public Customer(int _id, Bank _bank)
   {  bank = _bank;
      id   = _id;
      start();
   }
   public void start()
   {  running = true;
      super.start();
   }
   public void halt()
   {  running = false; }
   public void run()
   {  while (running)
      {  int into   = (int)(Bank.NUM_ACCOUNTS * Math.random());
         int amount = (int)(1000 * Math.random());
         bank.transfer(id, into, amount);
         yield();
      }
   }
}
```

Note in particular that our thread is a good citizen and calls on `yield` in its `run` method to allow other threads to do their work.[25] Finally, we add some simple buttons and other GUI elements to the `Bank` class as follows:

- First, we change the definition of the class so that it extends `Frame` and implements `ActionListener`:

```
public class Bank extends Frame implements ActionListener
```

- Next, we add buttons, a status label, and a display area as fields to the class:

```
private Label    status     = new Label("Transfers Completed: 0");
private TextArea display    = new TextArea();
private Button   show       = new Button("Show Accounts");
private Button   start      = new Button("Restart");
private Button   stop       = new Button("Stop");
```

- Then we modify the constructor to layout these components appropriately, using an anonymous inner class to close the frame if the user clicks on the standard close box:

```
public Bank()
{  super("Mystery Money");
   Panel buttons = new Panel(); buttons.setLayout(new FlowLayout());
   buttons.add(show);  show.addActionListener(this);
   buttons.add(start); start.addActionListener(this);
   buttons.add(stop);  stop.addActionListener(this);
   setLayout(new BorderLayout());
   add("North", status); add("South",  buttons); add("Center", display);
   for (int i = 0; i < accounts.length; i++)
      accounts[i] = 100000;
   start();
   validate(); setSize(300, 300); setVisible(true);
   addWindowListener(new WindowAdapter()
         { public void windowClosing(WindowEvent we)
           {  System.exit(0); }
         });
}
```

- We now add a line displaying the number of transfers as the last line of the `transfer` method:

```
status.setText("Transfers completed: " + counter++);
```

- We implement the `showAccounts` method to display its information in the `display` text area:

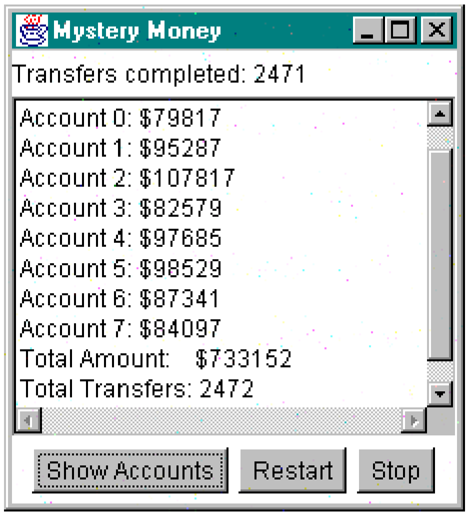
```
private void showAccounts()
{  int sum = 0;
   for (int i = 0; i < accounts.length; i++)
   {  sum += accounts[i];
      display.append("\nAccount " + i + ": $" + accounts[i]);
   }
   display.append("\nTotal Amount:    $" + sum);
   display.append("\nTotal Transfers: " + counter + "\n");
}
```

[25] At this point we could already compile the two classes and execute `Bank`. However, so far we do not call on `showAccounts` (we didn't even implement that method) so we will not be able to get any information about our accounts. Therefore, we need to wait until we have added the appropriate GUI elements.

- And finally we add an `actionPerformed` method to react to clicks on our buttons:

```
public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == show)
         showAccounts();
      else if (ae.getSource() == start)
         start();
      else if (ae.getSource() == stop)
         stop();
   }
```

Before executing the `Bank` class let us reflect on the financial situation of our bank: initially, each of the eight account contains $100,000 and the customers only shuffle money from their own account into other accounts. Therefore, the total sum of money available in all accounts should always remain at a constant value of $800,000, regardless of how many transfers have taken place. However, when we run this program for a while, then click on the `Stop` and the `Show Accounts` button, we will see output similar to the following:



```
Transfers completed: 2471
Account 0: $79817
Account 1: $95287
Account 2: $107817
Account 3: $82579
Account 4: $97685
Account 5: $98529
Account 6: $87341
Account 7: $84097
Total Amount:    $733152
Total Transfers: 2472
```

Figure 5.18: Running Customer threads on shared Bank data[26]

After only 2471 transfers the total has inexplicably dropped to $733,152. This, of course, should not happen – money has mysteriously disappeared. A bank using this program for conducting its transactions would certainly not stay in business for long. If you click on `Restart` to let the program continue for a while, then again click on `Stop` and `Show Accounts`, more money would disappear.

This is disturbing and we need to explain why the money in the `Bank` class seems to evaporate into thin air.[27] Here is what the program is doing, put into a "real world" context:

Customer 1 whose account is currently at, say, $50,000, wants to take out, say, $500 and transfer it to another account. The customer therefore goes to the bank and inquires about the current balance. The bank clerk says there are $50,000 in the account, and customer 1 is therefore able to conduct the transaction. The clerk computes the new account balances for the customer account to $45,500 and adds the money to the new account. Before the clerk updates the actual accounts, however, he is

[26] If you run this program the numbers displayed will be different from the ones displayed here. But the total amount will almost certainly be different from the expected $800,000.
[27] . We will learn in the next section how to ensure that multiple threads are properly synchronized so that they can not result in this kind of data corruption.

called to the phone (simulated by the `wasteSomeTime` method). When he returns, he resets the accounts according to the figures he computed just a second ago. But in the short time the clerk was on the phone, another customer may have transferred, say, $200 into the account of customer 1 which now has a value of $50,200. Nonetheless, the clerk resets the customer's account to $45,500, and the $200 that briefly were in the account have disappeared.[28]

■

A situation like this is actually much more common when performing input/output operations on files, since data transfer to and from file takes much longer than data transfer in main memory. But even for non-IO programs the problem is real as the above example shows.

Synchronizing Threads

To prevent this error from occurring, Java provides a simple yet powerful synchronization mechanism.

`Synchronized` Methods

Any method in a class can be designated as `synchronized` *to prevent data corruption by multiple threads. When a thread enters a* `synchronized` *method, the JVM guarantees that this thread will be able to finish before any other thread will be given access to the method. If a another thread tries to access a* `synchronized` *method already activated by a thread, the second thread is put on hold and must wait until the first thread finishes and releases the method. To declare a method as* `synchronized`*, use:*

```
[modifier] synchronized returnType methodName([parameterList])
{ /* method body */ }
```

Example 5.17: Synchronizing slow and fast threads to prevent data corruption

In example 5.15 using a "slow" and a "fast" thread, use proper synchronization to prevent data corruption:

Recall the `performWork` method in the `CorruptedData` class:

```
public void performWork(int type)
{  if (type == DISPLAY)
   {  System.out.println("Number before sleeping: " + number);
      try
      { slowWorker.sleep(2000); }
      catch(InterruptedException ie)
      {  System.err.println("Error: " + ie); }
      System.out.println("Number after waking up: " + number);
   }
   if (type == CHANGE)
```

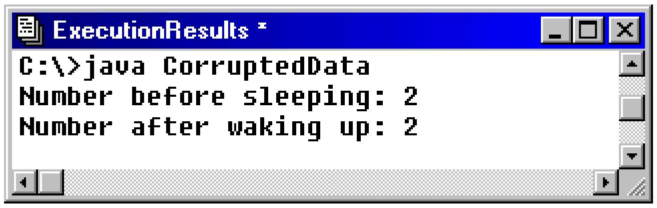
[28] The transfer method is somewhat contrived. A more straightforward implementation would simply say `account[from] -= amount` and `account[into] += amount`. If you use those lines it is *much* less likely that data corruption occurs but the principle is still the same: in the (very) brief time the right sides are computed, the account balance on the left side *could* be changed by another thread, resulting in data loss.

```
            number = -1;
        }
```

This method is activated by two threads: the first thread displays the number, sleeps, then displays it again. While this thread was sleeping, the second thread was allowed access to the method and changed the value of number (see example 5.15 for details). To prevent this data corruption, simply flag this method as `synchronized`, leaving everything else unchanged:

```
        public synchronized void performWork(int type)
        { /* rest of the method is unchanged */ }
```

When we now execute this class with the above modification, the output will be similar to:



```
C:\>java CorruptedData
Number before sleeping: 2
Number after waking up: 2
```

Figure 5.19: Data Corruption prevented using Synchronized Methods

Nothing else has changed in the code, but since the `performWork` method is now designated as `synchronized`, it will not allow the second thread to execute it until the first thread has finished. Of course, after both threads execute the number will be -1 but the "slow" thread was guaranteed that the value of number will not change while it is working (or in this case, sleeping).

■

Using this simple synchronization technique will also allow us to prevent our bank in example 5.16 from going out of business:

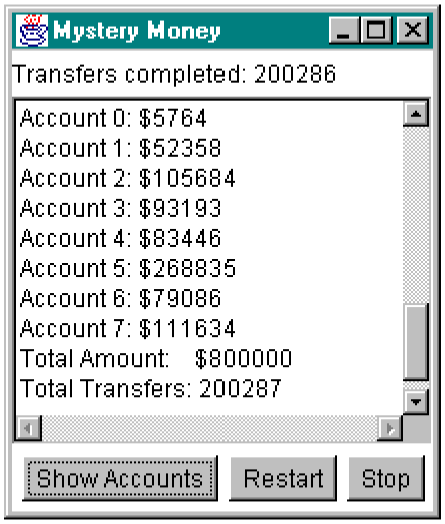
Example 5.18 Synchronizing the bank account simulation

In the class Bank create in example 5.16 use proper synchronization to prevent data corruption.

Again, a simple change will insure the integrity of our bank accounts: while one customer thread is performing a transaction, no other thread should be allowed to change any account. We therefore change the header of the `transfer` method from that example to a `synchronized` method:

```
        public synchronized void transfer(int from, int into, int amount)
```

Now we can perform any amount of transactions, and any time a thread wants to use this method to transfer money, it must wait until the currently executing thread has finished its business, therefore insuring the integrity of the bank's accounting books. To verify, we should execute the new `Bank` class including the `synchronized transfer` method for a large number of transactions. The total sum should always be constant at $800,000. Here is an excerpt of the output, showing that after more than 200,000 transactions the total sum is indeed still $800,000:



```
Transfers completed: 200286
Account 0: $5764
Account 1: $52358
Account 2: $105684
Account 3: $93193
Account 4: $83446
Account 5: $268835
Account 6: $79086
Account 7: $111634
Total Amount:   $800000
Total Transfers: 200287
```

Figure 5.20: Running Customer threads on shared Bank data via synchronized Method[29]

■

The mechanism that Java employs to ensure the integrity of synchronized methods involves monitors:

Monitors

An object that can deny threads access and notify threads when access is again possible is called a monitor. Any object that contains one or more synchronized methods is a monitor.

We do not have time to explain more details about monitors but we will show an example where Java's convenient but simple synchronization principle can result in deadlocked or stalled programs.

Wait, Notify, and Deadlocks

Java's mechanism to prevent simultaneous access to objects by threads is convenient, easy to use, and works without problems in most situations. However, there can be cases when threads begin to block each other in such a way that neither of them can continue because each one waits for the other to make a first move. At the same time, each thread also *prevents* the other from making the first move, thus resulting in a deadlocked situation.

Before we give one example of a deadlock situation, we need to investigate some explicit methods that objects can use to control threads

The Methods `wait` and `notify`

Every Java class extends the basic `Object` *class. Therefore, every class inherits from* `Object` *the methods* `wait` *and* `notify`*. Both methods can only be called in a* `synchronized` *method.*

[29] You can also click on `Show Accounts` without first stopping all threads. In that case the total amount in the accounts will likely not be equal to $800,000 because as we sum up the value of the accounts, threads continue to change individual accounts. Only after all threads are stopped will the total amount be $800,000.

- `public final void wait() throws InterruptedException`*: Causes a thread to release ownership of its monitor and wait until it is notified by another thread of a change in this object. The waiting thread will then resume ownership of the monitor and continue executing. If a waiting thread is not explicitly notified, it will wait forever.*
- `public final native void notify()`*: Informs a waiting thread that the state of an object has changed and that the waiting thread can resume ownership of the thread's monitor.*

Both methods should only be called by a thread that is the owner of this object's monitor, i.e. by threads that have executed a synchronized method of that object. Only one thread at a time can own an object's monitor.

Note that you do not *have* to call either the `wait` or `notify` method in order to perform proper synchronization. However, if you *do* call upon the `wait` method, you must make sure that a matching `notify` method is also called. If there is a `wait` without a corresponding `notify`, a thread will be in a wait state that it can no longer exit.

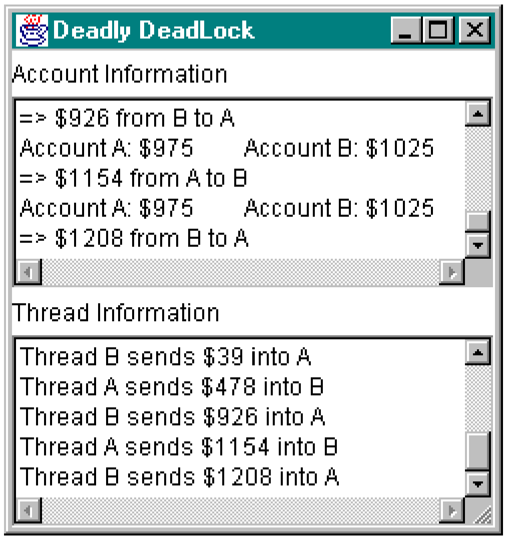
Using a fairly simple (and unfortunate) combination of `wait` and `notify` calls can result in a deadlock situation where two threads wait on each other's `notify` method to be called but neither will call its `notify` method because it is waiting on the other thread to do so.

Example 5.19: A deadlock situation

Below are two classes, one extending thread and the other containing calls to the `wait` and `notify` method (they are modeled after the `Bank` and `Customer` classes in example 5.16). Execute `DeadLock` and observe what happens. Explain.

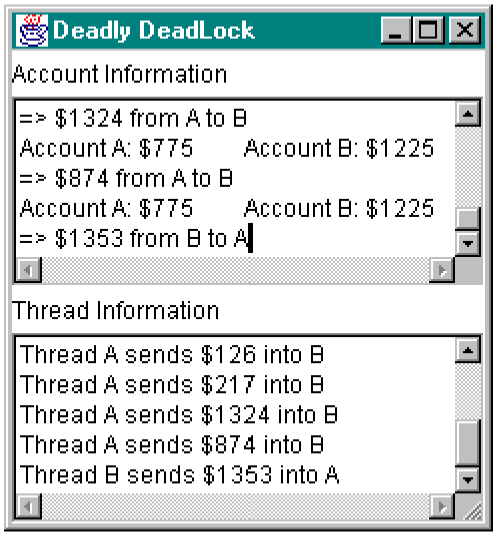
```
import java.awt.*;
import java.awt.event.*;

public class DeadLock extends Frame
{  protected static final String[] NAMES = {"A", "B"};
   private int accounts[] = {1000, 1000};
   private TextArea info = new TextArea(5,40);
   private TextArea status  = new TextArea(5,40);
   public DeadLock()
   {  super("Deadly DeadLock");
      setLayout(new GridLayout(2,1));
      add(makePanel(info, "Accounts")); add(makePanel(status, "Threads"));
      validate(); pack(); show();
      DeadLockThread A = new DeadLockThread(0, this, status);
      DeadLockThread B = new DeadLockThread(1, this, status);
      addWindowListener(new WindowAdapter()
         {  public void windowClosing(WindowEvent we)
            {  System.exit(0); }
         });
   }
   public synchronized void transfer(int from, int into, int amount)
   {  info.append("\nAccount A: $" + accounts[0]);
      info.append("\tAccount B: $" + accounts[1]);
      info.append("\n=> $" + amount + " from " + NAMES[from] +
                  " to " + NAMES[into]);
      while (accounts[from] < amount)
      {  try
         {  wait(); }
         catch(InterruptedException ie)
         {  System.err.println("Error: " + ie); }
      }
      accounts[from] -= amount;
      accounts[into] += amount;
      notify();
   }
   private Panel makePanel(TextArea text, String title)
   {  Panel p = new Panel();
      p.setLayout(new BorderLayout());
      p.add("North", new Label(title)); p.add("Center", text);
      return p;
   }
   public static void main(String args[])
   {  DeadLock bank = new DeadLock(); }
}

import java.awt.TextArea;

public class DeadLockThread extends Thread
{  private DeadLock bank;
   private int  id;
   private TextArea display;
   public DeadLockThread(int _id, DeadLock _bank, TextArea _display)
   {  bank    = _bank;
      id      = _id;
      display = _display;
      start();
   }
   public void run()
   {  while (true)
      {  int amount = (int)(1500 * Math.random());
         display.append("\nThread " + DeadLock.NAMES[id] + " sends $" +
                        amount + " into " + DeadLock.NAMES[(1-id)]);
         try
         {  sleep(20);
         }
         catch (InterruptedException ie)
         {  System.err.println("Interrupted"); }
         bank.transfer(id, 1-id, amount);
      }
   }
}
```

Both classes compile fine and when we execute `DeadLock` money will start to get transferred between the two accounts (compare example 5.16). However, after a short while the program will hang – no further output is produced, yet no error message is shown. The program simply stops working. If you execute it again, the same thing will happen but not necessarily after the same number of steps. This suggests some kind of exceptional condition that only occurs in special situations.



```
=> $926 from B to A
Account A: $975       Account B: $1025
=> $1154 from A to B
Account A: $975       Account B: $1025
=> $1208 from B to A
```

```
Thread B sends $39 into A
Thread A sends $478 into B
Thread B sends $926 into A
Thread A sends $1154 into B
Thread B sends $1208 into A
```



```
=> $1324 from A to B
Account A: $775       Account B: $1225
=> $874 from A to B
Account A: $775       Account B: $1225
=> $1353 from B to A
```

```
Thread A sends $126 into B
Thread A sends $217 into B
Thread A sends $1324 into B
Thread A sends $874 into B
Thread B sends $1353 into A
```

Figure 5.21: Two runs of `DeadLock` *execute then "hang" without apparent reason*

Here is why our program is running into a problem in figure 5.21 on the left (the case on the right is similar):

- account A contains $975, account B = $1,025 prior to hanging
- thread A wants to send $1,154 into account B
- account A does not have sufficient funds, so thread A waits to be notified of a change

- now thread B wants to send $1,208 into account A
- account B also does not contain enough money so thread B also waits

Each thread is waiting for the other to `notify` it about a change in the account status, but since both of them are waiting the program comes to a halt. A deadlock situation has arisen and Java, by itself, can do nothing about it. Our threads will wait forever on each other.

■

The deadlock, in principle, would be easy to break. Allow A to transfer the money, leaving temporarily -$179 in its account and $2,179 in account B. Now account B has enough money to transfer $1,208, leaving $1,029 in account A (positive again) and $971 in account B. Unfortunately our program does not allow negative accounts at any time, even temporarily, thus resulting in the deadlock situation (and actually, some banks also don't allow negative balances, even temporarily).

Dead Lock

A deadlock is a situation where all threads executing in a particular application or applet are put into the `wait` *state in such a way that each thread is waiting for another thread to notify it of a change in the state of the object.*
Java can do nothing to avoid deadlock situations, or to automatically resolve them should they occur. Responsibility for avoiding deadlocks rests entirely with the programmer. The programmer must make sure that the code is structured in such a way that a deadlock can not occur.

The hardest part in avoiding deadlocks is to realize under which conditions they could happen. Once that has been figured out, it is usually not too hard to modify the code so that the deadlock can not occur.

Example 5.20: Preventing a deadlock situation

What would have to change in the above classes `DeadLock` and/or `DeadLockThread` to avoid a possible deadlock situation?

A we mentioned before, we could of course allow negative balances even though that would not be desirable from the bank's point of few, so another solution is required.

The deadlock can arise because each thread is allowed to move a random amount between 0 and $1500. Initially, each account contains $1000 for a total sum of $2000. If we restrict the maximum amount of money that each thread could move during one transaction to any number less than $1000, no deadlock can result: the combined total of accounts is $2000, and both accounts are always positive (because of the way the `transfer` method works). That means that at any given time, one of the two accounts must *always* contain $1000 or more. Therefore, *one* of the two threads will always be able to move an amount of less than $1000 into the other account, and therefore one of the two threads will always be able to change the state of the accounts, thus calling `notify` to awaken the waiting thread.

You should modify the `DeadLockThread` program accordingly and run `DeadLock` again. This time the program should continue to move money around until you exit it without any deadlock situation.

■

Incidentally, in our original `Bank`/`Customer` program no deadlock can occur: for one thing, we have, in that class, set a maximum of $1,000 per transfer. With eight total accounts, each containing initially $1,000, at least one of these accounts must again contain $1,000 or more. The thread corresponding to that account will eventually break any possible deadlock between the other threads. More importantly, though, in that earlier example we did not put the threads into waiting mode in the first place. Instead, we simply did not conduct the requested transfer at all if the funds were not sufficient. Therefore, no deadlock could arise in principle in that class. [30]

While Java's support for threads, and in particular for synchronized threads, is indeed very convenient and powerful, troublesome situations can still arise. The problem of deadlocks and how to efficiently resolve them is, in fact, a difficult topic that could easily fill a separate book.

For example, deadlock situations could occur within any GUI program that uses more than one thread, because many of the methods that Java provides for its GUI classes (such as List) are `synchronized` methods and can, in principle cause a deadlock situation. However, most Java programs – while frequently using threads – are content with using only one thread, or using threads in such a way that deadlocks cannot arise. Therefore, we will conclude our discussion of threads at this point and move on to a topic where threads are indeed used almost all the time, implicitly or explicitly: multimedia and advanced graphics techniques.

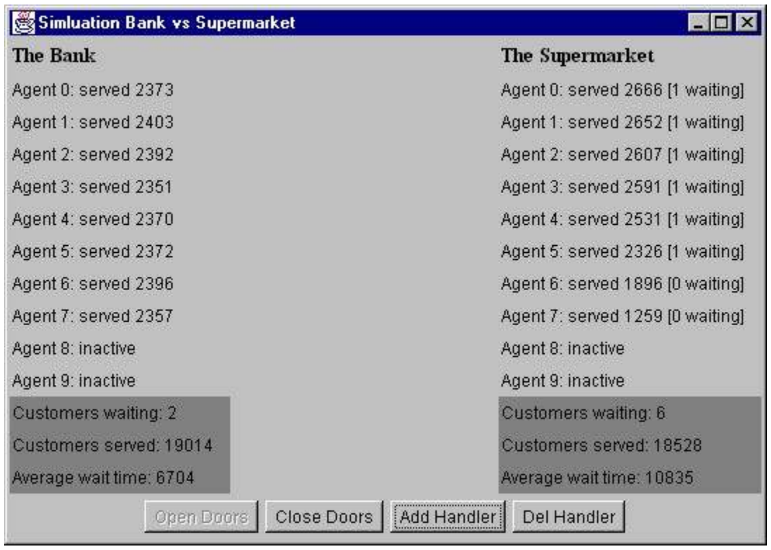
[30] If the limit was bigger than $1000, it would be possible for no transfers to occur at all for some time, which is different from the program hanging.

Case Study: Simulation and Time

```
import java.awt.*;
import java.awt.event.*;

public class Simulator extends Frame implements ActionListener
{  protected static final int NUM_AGENTS = 10;
   protected static final int NUM_INITIAL_AGENTS = 6;
   protected static final int MAX_CUSTOMER_DELAY = 9000;
   protected static final int MAX_TELLER_BREAK = 1000;
   protected static final int MAX_NO_CUSTOMERS = 2000;
   private Button open = new Button("Open Doors");
   private Button close = new Button("Close Doors");
   private Button add = new Button("Add Handler");
   private Button del = new Button("Del Handler");
   private Bank bank = new Bank();
   private Supermarket supermarket = new Supermarket();
   private class WindowCloser extends WindowAdapter
   {   public void windowClosing(WindowEvent we)
      {  bank.stop();
         supermarket.stop();
         System.exit(0);
      }
   }

   public Simulator()
   {  super("Simluation Bank vs Supermarket");
      Panel buttons = new Panel();
      buttons.setLayout(new FlowLayout());
      buttons.add(open);   open.addActionListener(this);
      buttons.add(close);  close.addActionListener(this);
      buttons.add(add);    add.addActionListener(this);
      buttons.add(del);    del.addActionListener(this);
      addWindowListener(new WindowCloser());

      setLayout(new BorderLayout());
      add("West", bank); add("East", supermarket); add("South", buttons);
      validate(); pack(); show();

      bank.start();
      supermarket.start();
   }
   public void actionPerformed(ActionEvent ae)
   {  if (ae.getSource() == open)
      {  bank.openDoor();
         supermarket.openDoor();
      }
      else if (ae.getSource() == close)
      {  bank.closeDoor();
         supermarket.closeDoor();
      }
      else if (ae.getSource() == add)
      {  bank.addAgent();
         supermarket.addAgent();
      }
      else if (ae.getSource() == del)
      {  bank.retireAgent();
         supermarket.retireAgent();
      }
   }
   public static void main(String args[])
   {  Simulator smlt = new Simulator(); }
}


import java.awt.*;
import java.awt.event.*;

public abstract class Business extends Panel implements Runnable
{  protected Agent[] agent = new Agent[Simulator.NUM_AGENTS];
   protected Label[] labelAgent = new Label[Simulator.NUM_AGENTS];
   protected Label labelQueue = new Label("Customers waiting: 0");
   protected Label labelServed = new Label("Customers servers: 0");
   protected Label labelWait = new Label("Average wait: 0");
   protected int numAgents = Simulator.NUM_INITIAL_AGENTS;
   protected int numCustomer = 0;
   protected long totalWait = 0L;
   private Thread thread = null;
   private boolean doorIsOpen = false;

   public Business(String title)
   {  super();
      setup(title);
   }
   public abstract void generateCustomer();
   public abstract Customer requestCustomerFor(int ID);
   public abstract void updateDisplay();
   public void checkoutCustomer(int handled, long waitTime)
   {  numCustomer++;
      totalWait += waitTime;
   }
   public void addAgent()
   {  if (numAgents < Simulator.NUM_AGENTS)
      {  agent[numAgents] = new Agent(this, numAgents);
         agent[numAgents].start();
         numAgents++;
      }
   }
   public void retireAgent()
   {  if (numAgents > 1)
      {  agent[numAgents-1].halt();
         numAgents--;
      }
   }
   public void start()
   {  if (thread == null)
      {  thread = new Thread(this);
         doorIsOpen = true;
         thread.start();
         for (int i = 0; i < numAgents; i++)
            agent[i].start();
      }
   }
   public void stop()
   {  doorIsOpen = false;
      thread = null;
      for (int i = 0; i < numAgents; i++)
         agent[i].halt();
   }
   public void openDoor()
   {  doorIsOpen = true; }
   public void closeDoor()
   {  doorIsOpen = false;   }
   public void run()
   {  while (thread == Thread.currentThread())
      {  try
         {  thread.sleep((int)(Math.random()*Simulator.MAX_NO_CUSTOMERS));
            if (doorIsOpen)
               generateCustomer();
            updateDisplay();
         }
         catch(InterruptedException ie)
         {  System.out.println("Business exception: " + ie); }
      }
   }
   private void setup(String title)
   {  Panel agentPanel = new Panel();
      agentPanel.setLayout(new GridLayout(Simulator.NUM_AGENTS + 3, 1));
      for (int i = 0; i < Simulator.NUM_AGENTS; i++)
      {  labelAgent[i] = new Label("Agent " + i + ": served 0");
         agentPanel.add(labelAgent[i]);
         agent[i] = new Agent(this, i);
      }
      for (int i = numAgents; i < Simulator.NUM_AGENTS; i++)
         labelAgent[i].setText("Agent " + i + ": inactive");
      agentPanel.add(labelQueue);
      agentPanel.add(labelServed);
      agentPanel.add(labelWait);
      setLayout(new BorderLayout());
      add("Center", agentPanel);
      add("North", new Label(title));
   }
}

import java.awt.*;
import java.awt.event.*;

public class Bank extends Business implements Runnable
{  private Queue queue = null;

   public Bank()
   {  super("The Bank");
      queue = new Queue();
   }
   public void updateDisplay()
   {  labelServed.setText("Customers served: " + numCustomer);
      if (numCustomer != 0)
         labelWait.setText("Average wait: " + (totalWait / numCustomer));
      for (int i = 0; i < numAgents; i++)
         labelAgent[i].setText("Agent "+i+": served "+agent[i].getHandled());
      for (int i = numAgents; i < Simulator.NUM_AGENTS; i++)
         labelAgent[i].setText("Agent " + i + ": inactive");
      labelQueue.setText("Customers waiting: " + queue.getSize());
    }
   public void generateCustomer()
   {  queue.insert(new Customer());   }
   public Customer requestCustomerFor(int ID)
   {  return queue.requestCustomer(); }
}


import java.awt.*;
import java.awt.event.*;

public class Supermarket extends Business
```

```
{  private Queue[] queue = new Queue[Simulator.NUM_AGENTS];

   public Supermarket()
   {  super("The Supermarket");
      for (int i = 0; i < queue.length; i++)
         queue[i] = new Queue();
   }
   public void updateDisplay()
   {  labelServed.setText("Customers served: " + numCustomer);
      if (numCustomer != 0)
         labelWait.setText("Average wait: " + (totalWait / numCustomer));
      int totalSize = 0;
      for (int i = 0; i < numAgents; i++)
      {  totalSize += queue[i].getSize();
         String s = "Agent " + i + ": served " + agent[i].getHandled();
         labelAgent[i].setText(s + " [" + queue[i].getSize() + " waiting]");
      }
      for (int i = numAgents; i < Simulator.NUM_AGENTS; i++)
         labelAgent[i].setText("Agent " + i + ": inactive");
      labelQueue.setText("Customers waiting: " + totalSize);
   }
   public void generateCustomer()
   {  int index = 0;
      for (int i = 0; i < numAgents; i++)
         if (queue[i].getSize() < queue[index].getSize())
            index = i;
      queue[index].insert(new Customer());
   }
   public Customer requestCustomerFor(int ID)
   {  // the queue is guarantied to release a new customer
      return queue[ID].requestCustomer();
   }
}


import java.util.Date;

public class Agent extends Thread
{  private boolean running = false;
   private Business business = null;
   private int ID = -1;
   private int numCustomers = 0;
   public Agent(Business _business, int _ID)
   {  business = _business;
      ID = _ID;
   }
   public void start()
   {  running = true;
      super.start();
   }
   public void halt()
   {  running = false; }
   public int getHandled()
   {  return numCustomers; }
   private void releaseCustomer(Customer customer)
   {  numCustomers++;
      business.checkoutCustomer(numCustomers,customer.getWaitTime(new Date()));
   }
   public void run()
   {  while (running)
      {  try
         {  sleep((int)(Math.random()*Simulator.MAX_TELLER_BREAK)+1000);
            Customer customer = business.requestCustomerFor(ID);
```

```
                if (customer != null)
                {  sleep(customer.getDelayTime());
                   releaseCustomer(customer);
                }
             }
             catch(InterruptedException ie)
             {  System.out.println("Teller exception: " + ie); }
          }
       }
    }


    import java.util.Date;

    public class Customer
    {  private Date created;

       public Customer()
       {  created = new Date(); }
       public int getDelayTime()
       {  return (int)(Simulator.MAX_CUSTOMER_DELAY * Math.random()); }
       public long getWaitTime(Date now)
       {  return now.getTime() - created.getTime(); }
    }


    public class Queue
    {  private List customers = new List();

       private synchronized Object performAction(String cmd, Object obj)
       {  if (cmd.equals("insert"))
          {  if (!customers.isFull())
                customers.add(obj);
             notify();
             return null;
          }
          else if (cmd.equals("size"))
          {  return new Integer(customers.getSize());   }
          else if (cmd.equals("retrieve"))
          {  while (customers.getSize() == 0)
             {  try
                {  wait();   }
                catch(InterruptedException ie)
                {  System.out.println("Synch error in queue: " + ie); }
             }
             Customer c = (Customer)customers.get(0);
             customers.delete(0);
             return c;
          }
          else
             return null;
       }
       public void insert(Customer c)
       {  performAction("insert", c);   }
       public int getSize()
       {  return ((Integer)performAction("size", null)).intValue();   }
       public Customer requestCustomer()
       {  return (Customer)performAction("retrieve", null);   }
    }
```



```
The Bank                                  The Supermarket
Agent 0: served 2373                      Agent 0: served 2666 [1 waiting]
Agent 1: served 2403                      Agent 1: served 2652 [1 waiting]
Agent 2: served 2392                      Agent 2: served 2607 [1 waiting]
Agent 3: served 2351                      Agent 3: served 2591 [1 waiting]
Agent 4: served 2370                      Agent 4: served 2531 [1 waiting]
Agent 5: served 2372                      Agent 5: served 2326 [1 waiting]
Agent 6: served 2396                      Agent 6: served 1896 [0 waiting]
Agent 7: served 2357                      Agent 7: served 1259 [0 waiting]
Agent 8: inactive                         Agent 8: inactive
Agent 9: inactive                         Agent 9: inactive
Customers waiting: 2                      Customers waiting: 6
Customers served: 19014                   Customers served: 18528
Average wait time: 6704                   Average wait time: 10835
```

Figure 5.22: Sample run of Simulator

Chapter Summary