11

Threads

In this chapter we'll investigate the facilities Java has to enable you to overlap the execution of segments of a single program. As well as ensuring your programs run more efficiently, this capability is particularly useful when your program has, of necessity, to do a number of things at the same time. For example: a server program on a network that needs to communicate with multiple clients.

In this chapter you will learn:

- What a thread is and how you can create threads in your programs.
- How to control interactions between threads.
- What synchronization means and how to apply it in your code.
- What deadlocks are, and how to avoid them.
- How to set thread priorities.
- How to get information about the threads in your programs.

Understanding Threads

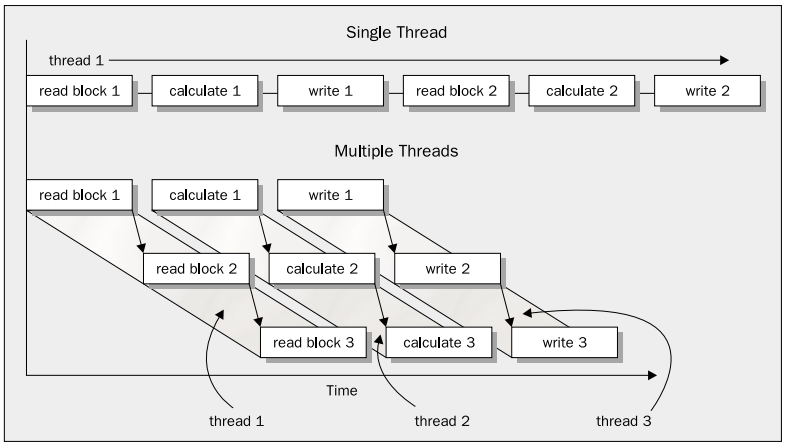
Many programs, of any size, contain some code segments that are more or less independent of one another, and that may execute more efficiently if the code segments could be overlapped in time. Threads provide a way to do this. Of course, if like most people your computer only has one processor, you can't execute more than one computation at any instant, but you can overlap input/output operations with processing. Another reason for using threads is to allow processes in a program that need to run continuously, such as a continuously running animation, to be overlapped with other activities in the same program. Java applets in a web page are executed under the control of your browser, and threads make it possible for multiple applets to be executing concurrently. In this case the threads serve to segment the activities running under the control of the browser so that they appear to run concurrently. If you only have one processor, this is an illusion created by your operating system since only one thread can actually be executing instructions at any given instant, but it's a very effective illusion. To produce animation, you typically put some code that draws a succession of still pictures in a loop that runs indefinitely.

The code to draw the picture generally runs under the control of a timer so that it executes at a fixed rate, say 20 times per second. Of course, nothing else can happen in the same thread while the loop is running. If you want to have another animation running, it must be in a separate thread. Then the multitasking capability of your operating system can allow the two threads to share the available processor time, thus allowing both animations to run. We will explore how we can program animations in Chapter 16.

Let's get an idea of the principles behind how threads operate. Consider a very simple program that consists of three activities:

- Reads a number of blocks of data from a file
- Performs some calculation on each block of data
- Writes the results of the calculation to another file

You could organize the program as a single sequence of activities. In this case the activities, read file, process, write file, run in sequence, and the sequence is repeated for each block to be read and processed. You could also organize the program so that reading a block from the file is one activity, performing the calculation is a second activity, and writing the results is a third activity. Both of these situations are illustrated below.

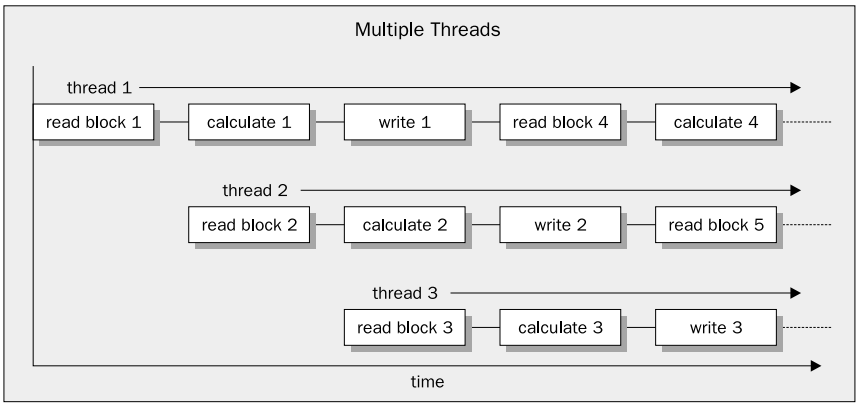


Once a block of data has been read, the computation process can start, and as soon as the computation has been completed, the results can be written out. With the program executing each step in sequence (that is, as a single thread), as shown in the top half of the diagram, the total time for execution will be the sum of the times for each of the individual activities. However, suppose we were able to execute each of the activities independently, as illustrated in the lower half of the diagram. In this case, reading the second block of data can start as soon as the first block has been read, and in theory we can have all three activities executing concurrently. This is possible even though you only have one processor because the input and output operations are likely to require relatively little processor time while they are executing, so the processor can be doing other things while they are in progress. This can have the effect of reducing the total execution time for the program. These three processes that we have identified that run more or less independently of one another – one to read the file, another to process the data and a third to write the results – are called **threads**. Of course, the first example at the top of the diagram has just one thread that does everything in sequence. Every Java program has at least one thread.

However, the three threads in the lower example aren't completely independent of one another. After all, if they were, you might as well make them independent programs. There are practical limitations too – the potential for overlapping these threads will be dependent on the capabilities of your computer, and of your operating system. However, if you can get some overlap in the execution of the threads, the program is going to run faster. There's no magic in using threads though. Your computer has only a finite capacity for executing instructions, and if you have many threads running you may in fact increase the overall execution time because of the overhead implicit in managing the switching of control between threads.

An important consideration when you have a single program running as multiple threads is that the threads are unlikely to have identical execution times, and, if one thread is dependent on another, you can't afford to have one overtaking the other – otherwise you'll have chaos. Before you can start calculating in the example in the diagram, you need to be sure that the data block the calculation uses has been read, and before you can write the output, you need to know that the calculation is complete. This necessitates having some means for the threads to communicate with one another.

The way we have shown the threads executing in the previous diagram isn't the only way of organizing the program. You could have three threads, each of which reads the file, calculates the results and writes the output, as shown here.

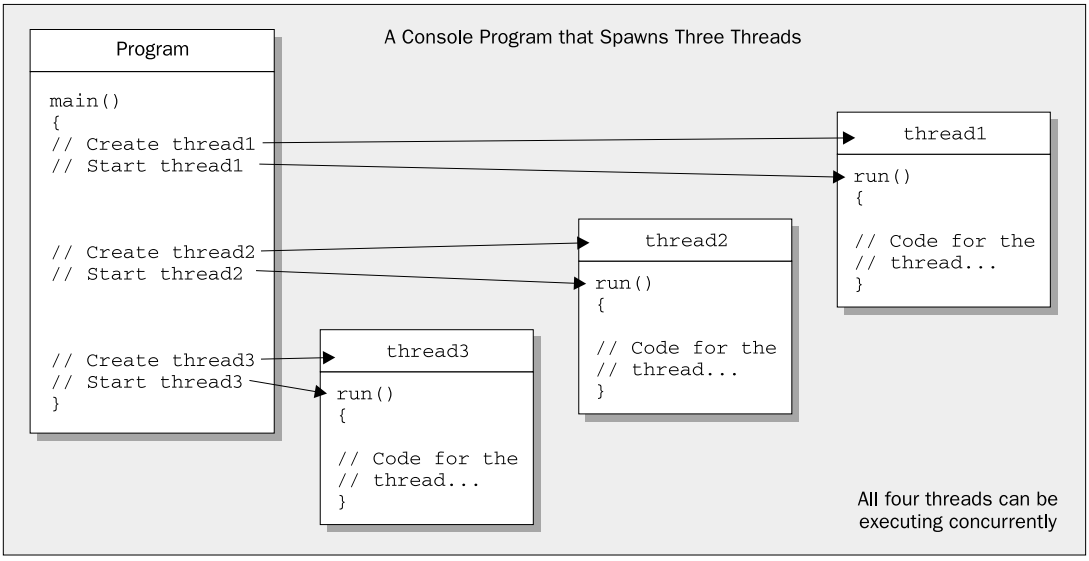


Now there's a different sort of contention between the threads. They are all competing to read the file and write the results, so there needs to be some way of preventing one thread from getting at the input file while another thread is already reading from it. The same goes for the output file. There's another aspect of this arrangement that is different from the previous version. If one thread, *thread1* say, reads a block, *block4* perhaps, that needs a lot of time to compute the results, another thread, *thread2* say, could conceivably read a following block, *block5* maybe, calculate and write the results for *block5*, before *thread1* has written the results for *block4*. If you don't want the results appearing in a different sequence from the input, you should do something about this. Before we delve into the intricacies of making sure our threads don't get knotted, let's first look at how we create a thread.

Creating Threads

Your program always has at least one thread: the one created when the program begins execution. With a program, this thread starts at the beginning of `main()`. With an applet, the browser is the main thread. When your program creates a thread, it is in addition to the thread of execution that created it. As you might have guessed, creating an additional thread involves using an object of a class, and the class you use is `java.lang.Thread`. Each additional thread that your program creates is represented by an object of the class `Thread`, or of a subclass of `Thread`. If your program is to have three additional threads, you will need to create three such objects.

To start the execution of a thread, you call the `start()` method for the `Thread` object. The code that executes in a new thread is always a method called `run()`,which is `public`, accepts no arguments and doesn't return a value. Threads other than the main thread in a program always start in the `run()`method for the object that represents the thread. A program that creates three threads is illustrated diagrammatically here.



For a class representing a thread in your program to do anything, you must implement the `run()` method as the version defined in the `Thread` class does nothing. Your implementation of `run()` can call any other methods you want. Our illustration shows `main()` creating all three threads, but that doesn't have to be the case. Any thread can create more threads.

Note that you don't call the `run()` method to start a thread, you call the `start()` method for the object representing the thread and that causes the `run()` method to be called. When you want to stop the execution of a thread that is running, you call the `stop()` method for the `Thread` object.

There are two ways in which you can define a class that is to represent a thread. One way is to define your class as a subclass of `Thread` and provide a definition of the method `run()` that overrides the inherited method. The other possibility is to define your class as implementing the interface, `Runnable`, which declares the method `run()`, and then create a `Thread` object in your class when you need it. We will look at and explore the advantages of each approach in a little more detail.

Try It Out — Deriving a Subclass of Thread

We can see how this works by using an example. We'll define a single class, `TryThread`, that we'll derive from `Thread`. Execution starts in the method `main()`:

```
import java.io.IOException;

public class TryThread extends Thread
{
  private String firstName;          // Store for first name
  private String secondName;         // Store for second name
  private long aWhile;               // Delay in milliseconds

  public TryThread(String firstName, String secondName, long delay)
  {
    this.firstName = firstName;      // Store the first name
    this.secondName = secondName;    // Store the second name
    aWhile = delay;                  // Store the delay
    setDaemon(true);                 // Thread is daemon
  }

  public static void main(String[] args)
  {
    // Create three threads
    Thread first = new TryThread("Hopalong ", "Cassidy ", 200L);
    Thread second = new TryThread("Marilyn ", "Monroe ", 300L);
    Thread third = new TryThread("Slim ", "Pickens ", 500L);

    System.out.println("Press Enter when you have had enough...\n");
    first.start();                       // Start the first thread
    second.start();                      // Start the second thread
    third.start();                       // Start the third thread
    try
    {
      System.in.read();                  // Wait until Enter key pressed
      System.out.println("Enter pressed...\n");
    }
    catch (IOException e)                // Handle IO exception
    {
      System.out.println(e);             // Output the exception
    }
    System.out.println("Ending main()");
    return;
  }

  // Method where thread execution will start
  public void run()
  {
    try
    {
      while(true )                           // Loop indefinitely...
      {
```

```
            System.out.print(firstName);         // Output first name
            sleep(aWhile);                       // Wait aWhile msec.
            System.out.print(secondName + "\n"); // Output second name
          }
        }

        catch(InterruptedException e)       // Handle thread interruption
        {
          System.out.println(firstName + secondName + e);     // Output the exception
        }
      }
    }
```

If you compile and run the code, you'll see something like this:

```
Press Enter when you have had enough...

Hopalong Marilyn Slim Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Pickens
Slim Monroe
Marilyn Cassidy
Hopalong Cassidy
Hopalong Monroe
Marilyn Pickens
Slim Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Cassidy
Hopalong Monroe
Marilyn Pickens
Slim Cassidy
Hopalong Cassidy
Hopalong Monroe
Marilyn
Enter pressed...

Ending main()
```

How It Works

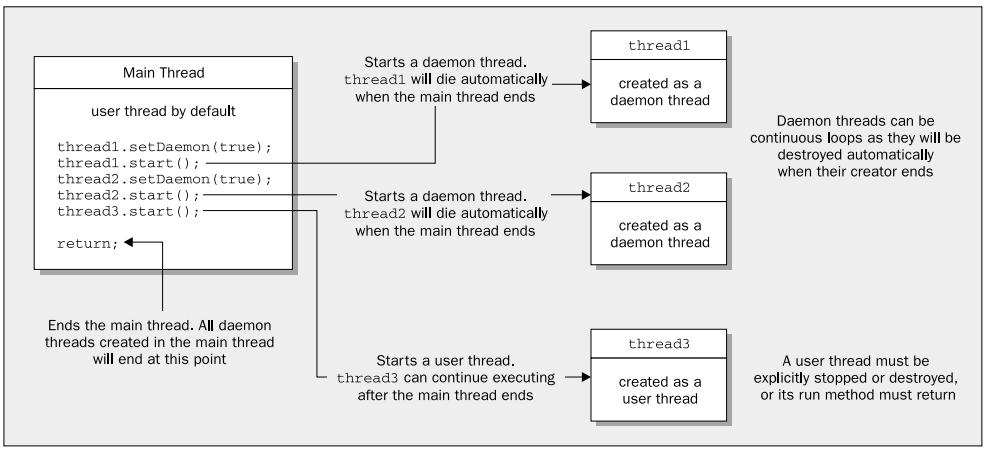
There are three instance variables in our class `TryThread` and these are initialized in the constructor. The two `String` variables hold first and second names, and the variable `aWhile` stores a time period in milliseconds. The constructor for our class, `TryThread()`, will automatically call the default constructor, `Thread()`, for the base class.

Our class containing the method `main()` is derived from `Thread`, and implements `run()`, so objects of this class represent threads. The fact that each object of our class will have access to the method `main()` is irrelevant – the objects are perfectly good threads. Our method `main()` creates three such objects: `first`, `second` and `third`.

Daemon and User Threads

The call to `setDaemon()`, with the argument `true` in the `TryThread` constructor, makes the thread that is created a **daemon thread**. A daemon thread is simply a background thread that is subordinate to the thread that creates it, so when the thread that created the daemon thread ends, the daemon thread dies with it. In our case, the method `main()` creates the daemon threads so that when `main()` returns, all the threads it has created will also end. If you run the example a few times pressing *Enter* at random, you should see that the daemon threads die after the `main()` method returns, because, from time to time, you will get some output from one or other thread after the last output from `main()`.

A thread that isn't a daemon thread is called a **user thread**. The diagram below shows two daemon threads and a user thread that are created by the main thread of a program.



A user thread has a life of its own that is not dependent on the thread that creates it. It can continue execution after the thread that created it has ended. The default thread that contains `main()` is a user thread, as shown in the diagram, but `thread3` shown in the diagram could continue to execute after `main()` has returned. Threads that run for a finite time are typically user threads, but there's no reason why a daemon thread can't be finite. Threads that run indefinitely should usually be defined as daemon threads simply because you need a means of stopping them. A hypothetical example might help you to understand this so let's consider how a network server handling transactions of some kind might work in principle.

A network server might be managed overall by a user thread that starts one or more daemon threads to listen for requests. When the server starts up the operator starts the management thread and this thread creates daemon threads to listen for requests. Each request recognized by one of these daemon threads might be handled by another thread that is created by the listening thread, so that each request will be handled independently. Where processing a transaction takes a finite time, and where it is important that the requests are completed before the system shuts down, the thread handling the request might be created as a user thread to ensure that it runs to completion, even if the listening thread that created it stops. When the time comes to shut the system down, the operator doesn't have to worry about how many listening threads are running. When the main thread is shut down, all the listening threads will also shut down because they are daemon threads. Any outstanding threads dealing with specific transactions will then run to completion.

Note that you can only call `setDaemon()` for a thread before it starts; if you try to do so afterwards, the method will throw an `IllegalThreadStateException` exception. Also, a thread that is itself created by a daemon thread will be daemon by default.

Creating Thread Objects

In the method `main()`, we create three `Thread` variables that store three different objects of our class `TryThread`. As you can see, each object has an individual name pair as the first two arguments to its constructor, and a different delay value passed as the third argument. All objects of the class `TryThread` are daemon threads because we call `setDaemon()`with the argument `true` in the constructor. Since the output can continue indefinitely, we display a message to explain how to stop it.

Once you've created a thread, it doesn't start executing by itself. You need to set it going. As we said earlier, you don't call the `run()` method for the `Thread` object to do this, you call its `start()` method. Thus we start the execution of each of the threads represented by the objects, `first`, `second` and `third`, by calling the `start()` method that is inherited from `Thread` for each object. The `start()` method starts the object's `run()` method executing, then returns to the calling thread. Eventually all three threads are executing in parallel with the original application thread, `main()`.

Implementing the run() Method

The `run()`method contains the code for the thread execution. The code in this case is a single, infinite `while` loop which we put in a `try` block because the `sleep()`method that is called in the loop can throw the exception caught by the `catch` block. The code in the loop outputs the first name, calls the method `sleep()` inherited from `Thread` and then outputs the second name. The `sleep()` method suspends execution of the thread for the number of milliseconds that you specify in the argument. This gives any other threads that have previously been started a chance to execute. This allows the possibility for the output from the three threads to become a little jumbled.

Each time a thread calls the method `sleep()`, one of the other waiting threads jumps in. You can see the sequence in which the threads execute from the output. From the names in the output you can deduce that they execute in the sequence `first`, `second`, `third`, `first`, `first`, `second`, `second`, `first`, `first`, `third` and so on. The actual sequence depends on your operating system scheduler so this is likely to be different on different computers. The execution of the `read()` method that is called in `main()` is blocked until you press *Enter*, but all the while the other threads continue executing. The output stops when you press *Enter* because this allows the main thread to continue, and execute the `return`. Executing `return` ends the thread for `main()` and since the other threads are daemon threads they also die when the thread that created them dies, although as you may have seen, they can run on a little after the last output from `main()`.

Stopping a Thread

If we did not create the threads in the last example as daemon threads, they would continue executing independently of `main()`. If you are prepared to terminate the program yourself (use *Ctrl+C* in a DOS session running Java) you can demonstrate this by commenting out the call to `setDaemon()` in the constructor. Pressing *Enter* will end `main()`, but the other threads will continue indefinitely.

A thread can signal another thread that it should stop executing by calling the `interrupt()` method for that `Thread` object. This in itself doesn't necessarily stop the thread. It just sets a flag in the thread that needs to be checked in the `run()` method to have any effect. As it happens the `sleep()` method checks whether the thread has been interrupted, and throws an `InterruptedException` if it has been. You can see that in action by altering the previous example a little.

Try It Out — Interrupting a Thread

Make sure the call to the `setDaemon()` method is still commented out in the constructor, and modify the `main()` method as follows:

```
public static void main(String[] args)
{
  // Create three threads
  Thread first = new TryThread("Hopalong ", "Cassidy ", 200L);
  Thread second = new TryThread("Marilyn ", "Monroe ", 300L);
  Thread third = new TryThread("Slim ", "Pickens ", 500L);

  System.out.println("Press Enter when you have had enough...\n");
  first.start();                      // Start the first thread
  second.start();                     // Start the second thread
  third.start();                      // Start the third thread
  try
  {
    System.in.read();                 // Wait until Enter key pressed
    System.out.println("Enter pressed...\n");

    // Interrupt the threads
    first.interrupt();
    second.interrupt();
    third.interrupt();
  }
  catch (IOException e)               // Handle IO exception
  {
    System.out.println(e);            // Output the exception
  }
  System.out.println("Ending main()");
  return;
}
```

Now the program will produce output that is something like:

```
Press Enter when you have had enough...

Slim Hopalong Marilyn Cassidy
Hopalong Monroe
Marilyn Cassidy
Hopalong Pickens
Slim Cassidy
Hopalong Monroe
Marilyn
```

```
Enter pressed...

Ending main()
Marilyn Monroe java.lang.InterruptedException: sleep interrupted
Slim Pickens java.lang.InterruptedException: sleep interrupted
Hopalong Cassidy java.lang.InterruptedException: sleep interrupted
```

How It Works

Since the method `main()` calls the `interrupt()` method for each of the threads after you press the *Enter* key, the `sleep()` method called in each thread registers the fact that the thread has been interrupted and throws an `InterruptedException`. This is caught by the `catch` block in the `run()` method and produces the new output you see. Because the `catch` block is outside the `while` loop, the `run()` method for each thread returns and each thread terminates.

You can check whether a thread has been interrupted by calling the `isInterrupted()` method for the thread. This returns `true` if `interrupt()` has been called for the thread in question. Since this is an instance method, you can use this to determine in one thread whether another thread has been interrupted. For example, in `main()` you could write:

```
if(first.isInterrupted())
  System.out.println("First thread has been interrupted.");
```

Note that this only determines whether the interrupted flag has been set by a call to `interrupt()` for the thread – it does not determine whether the thread is still running. A thread could have its interrupt flag set and continue executing – it is not obliged to terminate because `interrupt()` is called. To test whether a thread is still operating you can call its `isAlive()` method. This returns `true` if the thread has not terminated.

The instance method `isInterrupted()` has no effect on the interrupt flag in the thread – if it was set, it remains set. However, the static method `interrupted()` in the `Thread` class is different. It tests whether the currently executing thread has been interrupted and, if it has, it clears the interrupted flag in the current `Thread` object and returns `true`.

When an `InterruptedException` is thrown, the flag that registers the interrupt in the thread is cleared, so a subsequent call to `isInterrupted()` or `interrupted()` will return `false`.

Connecting Threads

If you need to wait in one thread until another thread dies, you can call the `join()` method for the thread which you expect isn't long for this world. Calling the `join()` method with no arguments, will halt the current thread for as long as it takes the specified thread to die:

```
thread1.join();     // Suspend the current thread until thread1 dies
```

You can also pass a `long` value to the `join()` method to specify the number of milliseconds you're prepared to wait for the death of a thread:

```
thread1.join(1000); // Wait up to 1 second for thread1 to die
```

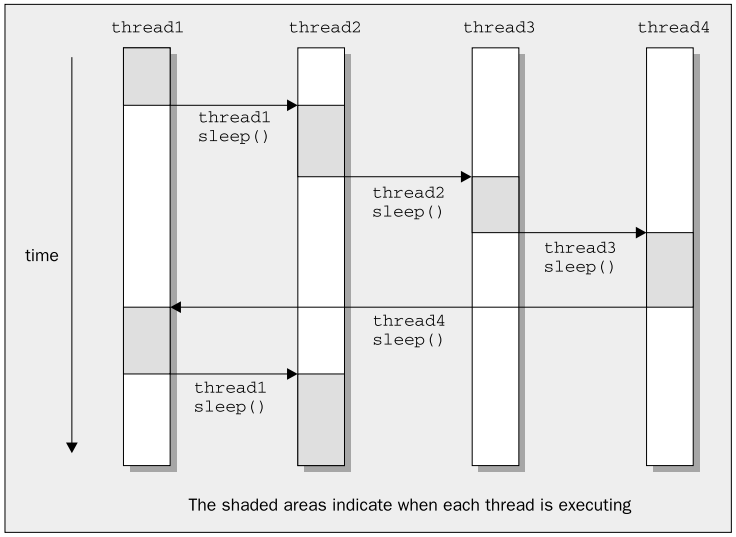
If this is not precise enough, there is a version of `join()` with two parameters. The first is a time in milliseconds and the second is a time in nanoseconds. The current thread will wait for the duration specified by the sum of the arguments. Of course, whether or not you get nanosecond resolution will depend on the capability of your hardware.

The `join()` method can throw an `InterruptedException` if the current thread is interrupted by another thread, so you should put a call to `join()` in a `try` block and catch the exception.

Thread Scheduling

The scheduling of threads depends to some extent on your operating system, but each thread will certainly get a chance to execute while the others are 'asleep', that is, when they've called their `sleep()` methods. If your operating system uses preemptive multitasking, as Windows 98 does, the program will work without the call to `sleep()` in the `run()` method (you should also remove the `try` and `catch` blocks, if you remove the `sleep()` call). However, if your operating system doesn't schedule in this way, without the `sleep()` call in `run()`, the `first` thread will hog the processor, and will continue indefinitely.

The diagram below illustrates how four threads might share the processor over time by calling the `sleep()` method to relinquish control.



Note that there's another method, `yield()`, defined in the `Thread` class, that gives other threads a chance to execute. You would use this when you just want to allow other threads a look-in if they are waiting, but you don't want to suspend execution of the current thread for a specific period of time. When you call the `sleep()` method for a thread, the thread will not continue for at least the time you have specified as an argument, even if no other threads are waiting. Calling `yield()` on the other hand will cause the current thread to resume immediately if no threads are waiting.

Implementing the Runnable Interface

As an alternative to defining a new subclass of `Thread`, we can implement the interface `Runnable` in a class. You'll find that this is generally much more convenient than deriving a class from `Thread` because you can derive your class from a class other than `Thread`, and it can still represent a thread. Because Java only allows a single base class, if you derive your class from `Thread`, it can't inherit functionality from any other class. The interface `Runnable` only declares one method, `run()`, and this is the method that will be executed when the thread is started.

Try It Out — Using the `Runnable` Interface

To see how this works in practice, we can write another version of the previous example. We'll call this version of the program `JumbleNames`:

```
import java.io.IOException;

public class JumbleNames implements Runnable
{
  private String firstName;                          // Store for first name
  private String secondName;                         // Store for second name
  private long aWhile;                               // Delay in milliseconds

  // Constructor
  public JumbleNames(String firstName, String secondName, long delay)
  {
    this.firstName = firstName;                      // Store the first name
    this.secondName = secondName;                    // Store the second name
    aWhile = delay;                                  // Store the delay
  }

  // Method where thread execution will start
  public void run()
  {
    try
    {
      while(true)                                    // Loop indefinitely...
      {
        System.out.print(firstName);                 // Output first name
        Thread.sleep(aWhile);                        // Wait aWhile msec.
        System.out.print(secondName+"\n");           // Output second name
      }
    }
    catch(InterruptedException e)                    // Handle thread interruption
    {
      System.out.println(firstName + secondName + e);     // Output the exception
    }
  }

  public static void main(String[] args)
  {
    // Create three threads
```

```
      Thread first = new Thread(new JumbleNames("Hopalong ", "Cassidy ", 200L));
      Thread second = new Thread(new JumbleNames("Marilyn ", "Monroe ", 300L));
      Thread third = new Thread(new JumbleNames("Slim ", "Pickens ", 500L));

      // Set threads as daemon
      first.setDaemon(true);
      second.setDaemon(true);
      third.setDaemon(true);

      System.out.println("Press Enter when you have had enough...\n");
      first.start();                                  // Start the first thread
      second.start();                                 // Start the second thread
      third.start();                                  // Start the third thread
      try
      {
        System.in.read();                             // Wait until Enter key pressed
        System.out.println("Enter pressed...\n");
      }
      catch (IOException e)                           // Handle IO exception
      {
        System.out.println(e);                        // Output the exception
      }
      System.out.println("Ending main()");
      return;
    }
  }
```

How It Works

We have the same data members in this class as we had in the previous example. The constructor is almost the same as previously. We can't call `setDaemon()` in this class constructor because our class isn't derived from `Thread`. Instead, we need to do that in `main()` after we've created the objects representing the threads. The `run()` method implementation is also very similar. Our class doesn't have `sleep()` as a member, but because it's a `public static` member of the class `Thread`, we can call it in our `run()` method by using the class name.

In the method `main()`, we still create a `Thread` object for each thread of execution, but this time we use a constructor that accepts an object of type `Runnable` as an argument. We pass an object of our class `JumbleNames` to it. This is possible because our class implements `Runnable`.

Thread Names

Threads have a name, which, in the case of the `Thread` constructor we're using in the example, will be a default name composed of the string `"Thread*"` with a sequence number appended. If you want to choose your own name for a thread, you can use a `Thread` constructor that accepts a `String` object specifying the name you want to assign to the thread. For example, we could have created the `Thread` object `first`, with the statement:

```
  Thread first = new Thread(new JumbleNames ("Hopalong ", "Cassidy ", 200L),
                            "firstThread");
```

This assigns the name `"firstThread"` to the thread. Note that this name is only used when displaying information about the thread. It has no relation to the identifier for the `Thread` object, and there's nothing, apart from common sense, to prevent several threads being given the same name.

You can obtain the name assigned to a thread by calling the `getName()` method for the `Thread` object. The name of the thread is returned as a `String` object. You can also change the name of a thread by calling the `setName()` method defined in the class `Thread`, and passing a `String` object to it.

Once we've created the three `Thread` objects in the example, we call the `setDaemon()`method for each. The rest of `main()` is the same as in the original version of the previous example, and you should get similar output when you run this version of the program.

Managing Threads

In both the examples we've seen in this chapter, the threads are launched and then left to compete for computer resources. Because all three threads compete in an uncontrolled way for the processor, the output from the threads gets muddled. This isn't normally a desirable feature in a program. In most instances where you use threads, the way in which they execute will need to be managed so that they don't interfere with each other.

Of course, in our examples, the programs are deliberately constructed to release control of the processor part way through outputting a name. While this is very artificial, similar situations can arise in practice, particularly where threads are involved in a repetitive operation. It is important to appreciate that a thread can be interrupted while a source statement is executing. For instance, suppose a thread executes the statement:

```
i = i+1;
```

It is quite possible for the thread execution to be interrupted while the execution of this statement is still in progress, perhaps after the value of `i` has been fetched to increment it, but before the result has been stored back in `i`. Without the proper controls, another thread that has access to `i` could alter it at this point. The effect would be that the increment of `i` by `1` in this thread would be lost.

Where two or more threads share a common resource, such as a file or a block of memory, you'll need to take steps to ensure that one thread doesn't modify a resource while that resource is still being used by another thread. Having one thread update a record in a file while another thread is part way through retrieving the same record, is a recipe for disaster. One way of managing this sort of situation is to use **synchronization** for the threads involved.

Synchronization

The objective of synchronization is to ensure that, when several threads want access to a single resource, only one thread can access it at any given time. There are two ways in which you can use synchronization to manage your threads of execution:

- You can manage code at the method level – this involves synchronizing methods.
- You can manage code at the block level – using synchronizing blocks.

We'll look at how we can use synchronized methods first.

Synchronized Methods

You can make a subset (or indeed all) of the methods for any class object mutually exclusive, so that only one of the methods can execute at any given time. You make methods mutually exclusive by declaring them in the class using the keyword `synchronized`. For example:

```
class MyClass
{
  synchronized public void method1()
  {
    // Code for the method...
  }

  synchronized public void method2()
  {
    // Code for the method...
  }

  public void method3()
  {
    // Code for the method...
  }
}
```

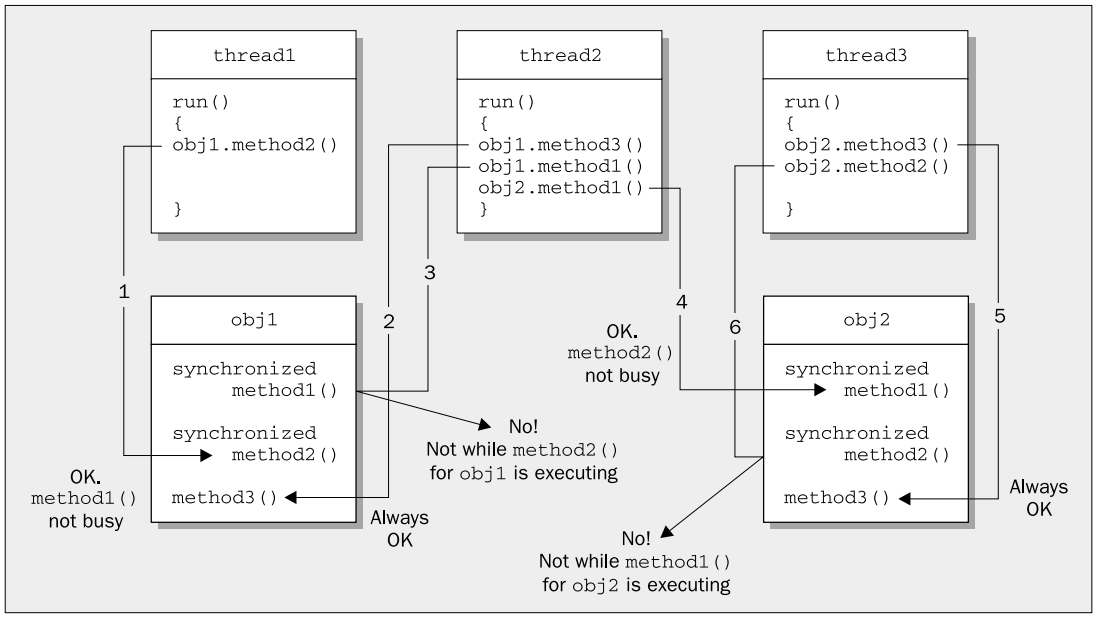
Now, only one of the synchronized methods in a class object can execute at any one time. Only when the currently executing synchronized method for an object has ended can another synchronized method start for the same object. The idea here is that each synchronized method has guaranteed exclusive access to the object while it is executing, at least so far as the other synchronized methods for the class object are concerned.

The synchronization process makes use of an internal **lock** that every object has associated with it. The lock is a kind of flag that is set by a process referred to as **locking**, or a **lock action**, when a synchronized method starts execution. Each synchronized method for an object checks to see whether the lock has been set by another method. If it has, it will not start execution until the lock has been reset by an **unlock action**. Thus, only one synchronized method can be executing at one time, because that method will have set the lock that prevents any other synchronized method from starting.

> **Note that there's no constraint here on simultaneously executing synchronized methods for two *different* objects of the same class. It's only concurrent access to any one object that is controlled.**

Of the three methods in `myClass`, two are declared as `synchronized`, so for any object of the class, only one of these methods can execute at one time. The method that isn't declared as synchronized, `method3()`, can always be executed by a thread, regardless of whether a synchronized method is executing.

It's important to keep clear in your mind the distinction between an object which has instance methods that you declared as `synchronized` in the class definition, and the threads of execution that might use them. A hypothetical relationship between three threads and two objects of the class `myClass` is illustrated in the following diagram:



The numbers on the arrows in the diagram indicate the sequence of events. **No!** indicates that the thread waits until the method is unlocked so it can execute it. While `method1()` in `obj2` is executing, `method2()` for the same object can't be executed. The synchronization of these two instance methods in an object provides a degree of protection for the object, in that only one synchronized method can mess with the data in the object at any given time.

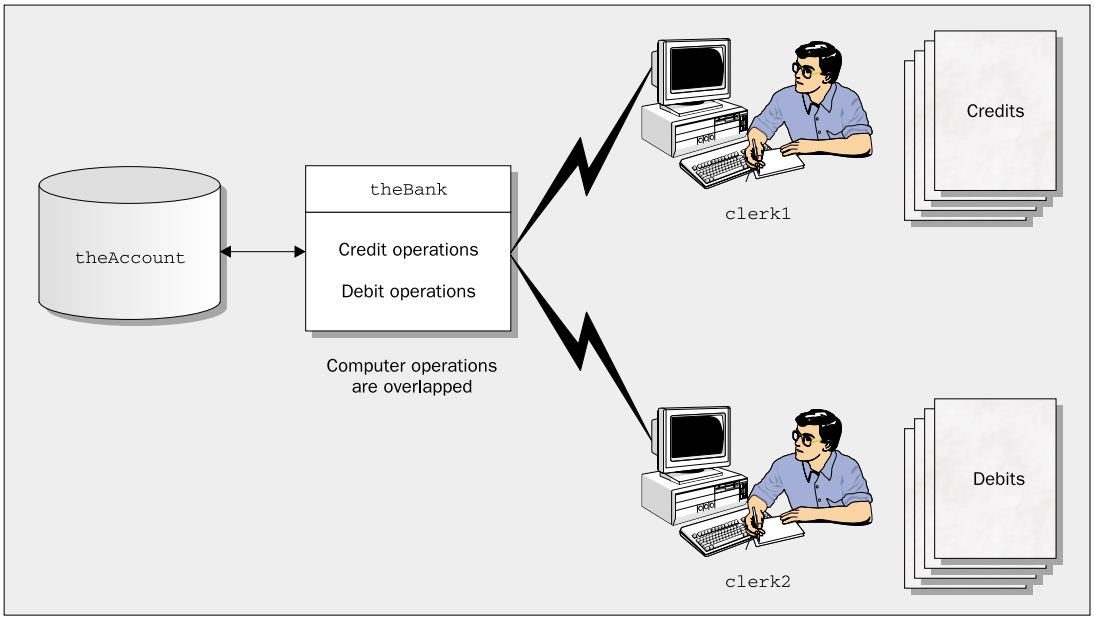
However, each object is independent of any other object when it comes to synchronized instance methods. When a thread executes a synchronized method for an object, it is assured exclusive access to the object insofar as the synchronized methods in that object are concerned. Another thread, though, can still call the same method for a different object. While `method1()` is being executed for `obj1`, this doesn't prevent `method1()` for `obj2` being executed by some other thread. Also, if there's a method in an object that has not been declared as `synchronized` – `method3()` in `obj1` for example – any thread can call that at any time, regardless of the state of any synchronized methods in the object.

If you apply synchronization to `static` methods in a class, only one of those `static` methods in the class can be executing at any point in time, and this is per class synchronization and the class lock is independent of any locks for objects of the class.

An important point of principle that you need to understand is that the only method that is necessarily part of a thread in a class object that represents a thread is the `run()` method. Other methods for the same class object are only part of the thread if they are called directly or indirectly by the `run()` method. All the methods that are called directly or indirectly from the `run()` method for an object are all part of the same thread, but they clearly don't have to be methods for the same `Thread` object. Indeed they can be methods that belong to any other objects, including other `Thread` objects that have their own `run()` methods.

Using Synchronized Methods

To see how synchronization can be applied in practice, we'll construct a program that provides a simple model of a bank. Our bank is a very young business with only one customer account initially, but we'll have two clerks each working flat out to process transactions for the account, one handling debits and the other handling credits. The objects in our program are illustrated here:



The bank in our model is actually a computer that performs operations on the account, and the account is stored separately. Each clerk can communicate directly with the bank. We'll be defining four classes that we will use in our program to model banking operations:

- A `Bank` class to represent the bank computer.
- An `Account` class to represent the account at the bank.
- A `Transaction` class to represent a transaction on the account – a debit or a credit for instance.
- A `Clerk` class to represent a bank clerk.

We will also define a class containing the method `main()` that will start the process off and determine how it all works.

> **As we develop the code, we won't necessarily get it right first time, but we will improve as we find out more about how to program using threads. This will expose some of the sorts of errors and complications that can arise when you're programming using threads.**

Try It Out — Defining a Bank Class

The bank computer is the agent that will perform the operations on an account so we will start with that. We can define the `Bank` class that will represent this as:

```
// Define the bank
class Bank
{
  // Perform a transaction
  public void doTransaction(Transaction transaction)
  {
    int balance = transaction.getAccount().getBalance();   // Get current balance

    switch(transaction.getTransactionType())
    {
      case Transaction.CREDIT:
        // Credits require a lot of checks...
        try
        {
          Thread.sleep(100);
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
        balance += transaction.getAmount();          // Increment the balance
        break;

      case Transaction.DEBIT:
        // Debits require even more checks...
        try
        {
          Thread.sleep(150);
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
        balance -= transaction.getAmount();          // Decrement the balance
        break;
```

```
        default:                                          // We should never get here
          System.out.println("Invalid transaction");
          System.exit(1);
     }
      transaction.getAccount().setBalance(balance);    // Restore the account
balance
  }
}
```

How It Works

The `Bank` class is very simple. It keeps no records of anything locally as the accounts will be identified separately, and it only has one method that carries out a transaction. The `Transaction` object will provide all the information about what the transaction is, and to which account it applies. We have only provided for debit and credit operations on an account, but the switch could easily be extended to accommodate other types of transactions. Both of the transactions supported involve some delay while the standard nameless checks and verifications, that all banks have, are carried out. The delay is simulated by calling the `sleep()` method belonging to the `Thread` class.

Of course, during this time, other things in other threads may be going on. There are no instance variables to initialize in a `Bank` object so we don't need a constructor. Since our `Bank` object works using a `Transaction` object, let's define the class for that next.

Try It Out — Defining a Transaction on an Account

The `Transaction` class could represent any transaction on an account, but we are limiting ourselves to debits and credits. We can define the class as:

```
class Transaction
{
  // Transaction types
  public static final int DEBIT = 0;
  public static final int CREDIT = 1;
  public static String[] types = {"Debit","Credit"};

  // Constructor
  public Transaction(Account account, int transactionType, int amount)
  {
    this.account = account;
    this.transactionType = transactionType;
    this.amount = amount;
  }

  public Account getAccount()
  {  return account;  }

  public int getTransactionType()
  {  return transactionType;  }

  public int getAmount()
  {  return amount;  }
```

```
    public String toString()
    {
      return types[transactionType] + " A//C: " + ": $" + amount;
    }

    private Account account;
    private int amount;
    private int transactionType;
  }
```

How It Works

The identification of a transaction is specified by the `transactionType` field that must be one of the values defined for transaction types. We should build in checks in the constructor to ensure only valid transactions are created, but we'll forego this to keep the code volume down, and you certainly know how to do this sort of thing by now. A transaction records the amount for the transaction and a reference to the account to which it applies, so a `Transaction` object specifies a complete transaction. The methods are very straightforward, just accessor methods for the data members that are used by the `Bank` object, plus the `toString()` method in case we need it.

Try It Out — Defining a Bank Account

We can define an account as:

```
// Defines a customer account
public class Account
{
  // Constructor
  public Account(int accountNumber, int balance)
  {
    this.accountNumber = accountNumber;            // Set the account number
    this.balance = balance;                        // Set the initial balance
  }

  // Return the current balance
  public int getBalance()
  {  return balance;  }

  // Set the current balance
  public void setBalance(int balance)
  {  this.balance = balance;  }

  public int getAccountNumber()
  {  return accountNumber;  }

  public String toString()
  {
    return "A//C No. "+accountNumber+" : $"+balance;
  }

  private int balance;                             // The current account balance
  private int accountNumber;                       // Identifies this account
}
```

How It Works

The `Account` class is also very simple. It just maintains a record of the amount in the account as a balance, and provides methods for retrieving and setting the current balance. Operations on the account are performed externally by the `Bank` object. We have a bit more than we need in the `Account` class at the moment, but the methods we don't use in the current example may be useful later.

Try It Out — Defining a Bank Clerk

A clerk is a slightly more complicated animal. He or she retains information about the bank, details of the current transaction and is responsible for initiating debits and credits on an account by communication with the central bank. Each clerk will work independently of the others so they will each be a separate thread:

```
public class Clerk implements Runnable
{
  private Bank theBank;               // The employer - an electronic marvel
  private Transaction inTray;         // The in-tray holding a transaction

  // Constructor
  public Clerk(Bank theBank)
  {
    this.theBank = theBank;           // Who the clerk works for
    inTray = null;                    // No transaction initially
  }

  // Receive a transaction
  public void doTransaction(Transaction transaction)
  {  inTray = transaction;  }

  // The working clerk...
  public void run()
  {
    while(true)
    {
      while(inTray == null)           // No transaction waiting?
      {
        try
        {
          Thread.sleep(150);          // Then take a break...
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
      }

      theBank.doTransaction(inTray);
      inTray = null;                  // In-tray is empty
    }
  }
```

```
    // Busy check
    public boolean isBusy()
    {
      return inTray != null;            // A full in-tray means busy!
    }
  }
```

How It Works

A `Clerk` object is a thread since it implements the `Runnable` interface. Each clerk has an in-tray, capable of holding one transaction, and while the in-tray is not `null`, the clerk is clearly busy. A clerk needs to be aware of the `Bank` object that is employing him or her, so a reference is stored in `theBank` when a `Clerk` object is created. A transaction is placed in the in-tray for a clerk by calling his or her `doTransaction()` method. You can check whether a clerk is busy by calling the `isBusy()` member which will return `true` if a transaction is still in progress.

The real work is actually done in the `run()` method. If the in-tray is empty, indicated by a `null` value in `inTray`, then there's nothing to do, so after sleeping a while the loop goes around again for another look at the in-tray. When a transaction has been recorded, the method in `theBank` object is called to carry it out and the `inTray` is reset to `null`.

All we need now is the class to drive our model world which we'll call `BankOperation`. This class only requires the method `main()`, but there are quite a lot of things to do in this method so we'll put it together piece by piece.

Try It Out — Defining the Operation of the Bank

Apart from setting everything up, the `main()` method has to originate transactions on the accounts and pass them on to the clerks to be expedited. We will start with just one account and a couple of clerks. Here's the basic structure:

```
import java.util.Random;

public class BankOperation
{
  public static void main(String[] args)
  {
    int initialBalance = 500;     // The initial account balance
    int totalCredits = 0;         // Total credits on the account
    int totalDebits =0;           // Total debits on the account
    int transactionCount = 20;    // Number of debits and credits

    // Create the account, the bank and the clerks...

    // Create the threads for the clerks as daemon, and start them off

    // Generate the transactions of each type and pass to the clerks

    // Wait until both clerks are done

    // Now output the results
  }
}
```

The import for the `Random` class is there because we will need it in a moment. To create the bank object, the clerks, and the account we need to add the following code:

```
// Create the bank, the clerks, and the account...
Bank theBank = new Bank();                        // Create a bank
Clerk clerk1 = new Clerk(theBank);                // Create the first clerk
Clerk clerk2 = new Clerk(theBank);                // Create the second clerk
Account account = new Account(1, initialBalance);// Create an account
```

The next step is to add the code to create the threads for the clerks and start them going:

```
// Create the threads for the clerks as daemon, and start them off
Thread clerk1Thread = new Thread(clerk1);
Thread clerk2Thread = new Thread(clerk2);
clerk1Thread.setDaemon(true);                     // Set first as daemon
clerk2Thread.setDaemon(true);                     // Set second as daemon
clerk1Thread.start();                             // Start the first
clerk2Thread.start();                             // Start the second
```

The code to generate the transactions looks a lot, but is quite repetitive:

```
// Generate transactions of each type and pass to the clerks
Random rand = new Random();                       // Random number generator
Transaction transaction;                          // Stores a transaction
int amount;                                       // stores an amount of money
for(int i = 1; i <= transactionCount; i++)
{
  amount = 50 + rand.nextInt(26);                 // Generate amount of $50 to $75
  transaction = new Transaction(account,             // Account
                                Transaction.CREDIT, // Credit transaction
                                amount);            //  of amount
      totalCredits += amount;                     // Keep total credit tally

  // Wait until the first clerk is free
  while(clerk1.isBusy())
  try
  {
    Thread.sleep(25);                             // Busy so try later
  }
  catch(InterruptedException e)
  {
    System.out.println(e);
  }

  clerk1.doTransaction(transaction);              // Now do the credit

  amount = 30 + rand.nextInt(31);                 // Generate amount of $30 to $60
  transaction = new Transaction(account,             // Account
                                Transaction.DEBIT,  // Debit transaction
                                amount);            //  of amount
  totalDebits += amount;                          // Keep total debit tally
```

```
    // Wait until the second clerk is free
    while(clerk2.isBusy())
    try
    {
      Thread.sleep(25);                          // Busy so try later
    }
    catch(InterruptedException e)
    {
      System.out.println(e);
    }

    clerk2.doTransaction(transaction);           // Now do the debit
  }
```

Once all the transactions have been processed, we can output the results. However, the clerks could still be busy after we exit from the loop, so we need to wait for both of them to be free before outputting the results. We can do this with a `while` loop:

```
  // Wait until both clerks are done
  while(clerk1.isBusy() || clerk2.isBusy())
  try
  {
    Thread.sleep(25);
  }
  catch(InterruptedException e)
  {
    System.out.println(e);
  }
```

Lastly, we output the results:

```
  // Now output the results
  System.out.println(
            "Original balance    : $" + initialBalance+"\n" +
            "Total credits       : $" + totalCredits+"\n" +
            "Total debits        : $" + totalDebits+"\n" +
            "Final balance       : $" + account.getBalance() + "\n" +
            "Should be           : $" + (initialBalance + totalCredits -
                                                        totalDebits));
```

How It Works

The variables in the `main()`method that track the total debits and credits, and record the initial account balance, are to help us figure out what has happened after the transactions have been processed. The number of debits and credits to be generated is stored in `transactionCount`, so the total number of transactions will be twice this value. We have added five further blocks of code to perform the functions indicated by the comments, so let's now go through each of them in turn.

The `Account` object is created with the account number as 1 and with the initial balance stored in `initialBalance`. We pass the bank object, `theBank`, to the constructor for each of the `Clerk` objects, so that they can record it.

The `Thread` constructor requires an object of type `Runnable`, so we can just pass the `Clerk` objects in the argument. There's no problem in doing this because the `Clerk` class implements the interface `Runnable`. You can always implicitly cast an object to a type which is any superclass of the object, or any interface type that the object class implements.

All the transactions are generated in the `for` loop. The handling of debits is essentially the same as the handling of credits, so we'll only go through the code for the latter in detail. A random amount between $50 and $75 is generated for a credit transaction by using the `nextInt()` method for the `rand` object of type `Random` that we create. You'll recall that `nextInt()` returns an `int` value in the range 0 to one less than the value of the argument, so by passing 26 to the method, we get a value between 0 and 25 returned. We add 50 to this and, hey presto, we have a value between 50 and 75. We then use this amount to create a `Transaction` object that represents a credit for the account. To keep a check on the work done by the clerks, we maintain the total of all the credits generated in the variable `totalCredits`. This will allow us to verify whether or not the account has been updated properly.

Before we pass the transaction to `clerk1`, we must make sure that he or she isn't busy. Otherwise we would overwrite the clerk's in-tray. The `while` loop does this. As long as the `isBusy()` method returns `true`, we continue to call the `sleep()` method for a twenty five millisecond delay, before we go round and check again. When `isBusy()` returns `false`, we call the `doTransaction()`method for the clerk with the reference to the `transaction` object as the argument. The `for` loop will run for twenty iterations, so we'll generate twenty random transactions of one or other type.

The third `while` loop works in the same way as the previous check for a busy clerk – the loop continues if either of the clerks is busy.

Lastly, we output the original account balance, the totals of credits and debits, and the final balance plus what it should be for comparison. That's all we need in the method `main()`, so we're ready to give it a whirl. Remember that all four classes need to be in the same directory.

Running the Example

Now, if you run the example the final balance will be wrong. You should get results something like the following:

```
Original balance    : $500
Total credits       : $1252
Total debits        : $921
Final balance       : $89
Should be           : $831
```

Of course, your results won't be the same as this, but they should be just as wrong. The customer will not be happy. His account balance is seriously out – in the bank's favor of course, as always. So how has this come about?

The problem is that both clerks are operating on the same account at the same time. Both clerks call the `doTransaction()` method for the `Bank` object, so this method is executed by both clerk threads. Separate calls on the same method are overlapping.

Try It Out — Synchronizing Methods

One way we can fix this is by simply declaring the method that operates on an account as `synchronized`. This will prevent one clerk getting at it while it is still in progress with the other clerk. To implement this you should amend the `Bank` class definition as follows:

```
// Define the bank
class Bank
{
  // Perform a transaction
  synchronized public void doTransaction(Transaction transaction)
  {
    // Code exactly as before...
  }
}
```

How It Works

Declaring this method as `synchronized` will prevent a call to it from being executed while another is still in operation. If you run the example again with this change, the result will be something like:

```
Original balance    : $500
Total credits       : $1201
Total debits        : $931
Final balance       : $770
Should be           : $770
```

The amounts may be different because the transaction amounts are random, but your final balance should be the same as adding the credits to the original balance and subtracting the debits.

As we saw earlier, when you declare methods in a class as `synchronized`, it prevents concurrent execution of those methods within a single object, *including concurrent execution of the same method*. It is important not to let the fact that there is only one copy of a particular method confuse you. A given method can be potentially executing in any number of threads – as many threads as there are in the program in fact. If it was not synchronized, the `doTransaction()` method could be executed concurrently by any number of clerks.

Although this fixes the problem we had in that the account balance is now correct, the bank is still amazingly inefficient. Each clerk is kicking their heels while another clerk is carrying out a transaction. At any given time a maximum of one clerk is working. On this basis the bank could sack them all bar one and get the same throughput. We can do better, as we shall see.

Synchronizing Statement Blocks

In addition to being able to synchronize methods on a class object, you can also specify a statement or a block of code in your program as `synchronized`. This is more powerful, since you specify which particular object is to benefit from the synchronization of the statement or code block, not just the object that contains the code as in the case of a synchronized method. Here we can set a lock on any object for a given statement block. When the block that is synchronized on the given object is executing, no other code block or method that is synchronized on the same object can execute. To synchronize a statement, you just write:

```
synchronized(theObject)
  statement;            // Synchronized with respect to theObject
```

No other statements or statement blocks in the program that are synchronized on the object, `theObject`, can execute while this statement is executing. Naturally, this applies even when the statement is a call to a method, which may in turn call other methods. The statement here could equally well be a block of code between braces. This is powerful stuff. Now we can lock a particular object while the code block that is working is running.

To see precisely how you can use this in practice, let's create a modification of the last example. Let's up the sophistication of our banking operation to support multiple accounts. To extend our example to handle more than one account, we just need to make some changes to `main()`. We'll add one extra account to keep the output modest, but we'll modify the code to handle any number.

Try It Out — Handling Multiple Accounts

We can modify the code in `main()` that creates the account and sets the initial balance to create multiple accounts as follows:

```
public class BankOperation
{
  public static void main(String[] args)
  {
    int[] initialBalance = {500, 800};  // The initial account balances
    int[] totalCredits = new int[initialBalance.length];  // Total cr's
    int[] totalDebits = new int[initialBalance.length];   // Total db's
    int transactionCount = 20;          // Number of debits and of credits

    // Create the bank and the clerks...
    Bank theBank = new Bank();                  // Create a bank
    Clerk clerk1 = new Clerk(theBank );         // Create the first clerk
    Clerk clerk2 = new Clerk(theBank );         // Create the second clerk
    // Create the accounts, and initialize total credits and debits
    Account[] accounts = new Account[initialBalance.length];
    for(int i = 0; i < initialBalance.length; i++)
    {
      accounts[i] = new Account(i+1, initialBalance[i]); // Create accounts
      totalCredits[i] = totalDebits[i] = 0;
    }

    // Create the threads for the clerks as daemon, and start them off

    // Create transactions randomly distributed between the accounts

    // Wait until both clerks are done

    // Now output the results
  }
}
```

We now create an array of accounts in a loop, the number of accounts being determined by the number of initial balances in the `initialBalance` array. Account numbers are assigned successively starting from 1. The code for creating the bank and the clerks and for creating the threads and starting them, is exactly the same as before. The shaded comments that follow the code indicate the other segments of code in `main()` that we need to modify.

The next piece we need to change is the creation and processing of the transactions:

```
// Generate transactions of each type and pass to the clerks
Random rand = new Random();
Transaction transaction;                          // Stores a transaction
int amount;                                       // Stores an amount of money
int select;                                       // Selects an account
for(int i = 1; i <= transactionCount; i++)
{
  // Generate a random account index for credit operation
  select = rand.nextInt(accounts.length);
  amount = 50 + rand.nextInt(26);                 // Generate amount of $50 to $75
  transaction = new Transaction(accounts[select],       // Account
                                Transaction.CREDIT, // Credit transaction
                                amount);            //  of amount
  totalCredits[select] += amount;                 // Keep total credit tally

  // Wait until the first clerk is free
  while(clerk1.isBusy())
  try
  {
    Thread.sleep(25);                             // Busy so try later
  }
  catch(InterruptedException e)
  {
    System.out.println(e);
  }

  clerk1.doTransaction(transaction);              // Now do the credit

  // Generate a random account index for debit operation
  select = rand.nextInt(accounts.length);
  amount = 30 + rand.nextInt(31);                 // Generate amount of $30 to $60
  transaction = new Transaction(accounts[select],   // Account
                                Transaction.DEBIT,  // Debit transaction
                                amount);            //  of amount
  totalDebits[select] += amount;                  // Keep total debit tally

  // Wait until the second clerk is free
  while(clerk2.isBusy())
  try
  {
    Thread.sleep(25);                         // Busy so try later
  }
  catch(InterruptedException e)
  {
```

```
      System.out.println(e);
    }

    clerk2.doTransaction(transaction);            // Now do the debit
  }
```

The last modification we must make to the method `main()` is for outputting the results. We now do this in a loop. seeing as we have to process more than one account:

```
for(int i = 0; i < accounts.length; i++)
  System.out.println("Account Number:"+accounts[i].getAccountNumber()+"\n"+
     "Original balance    : $" + initialBalance[i] + "\n" +
     "Total credits       : $" + totalCredits[i] + "\n" +
     "Total debits        : $" + totalDebits[i] + "\n" +
     "Final balance       : $" + accounts[i].getBalance() + "\n" +
     "Should be           : $" + (initialBalance[i] + totalCredits[i] -
                                  totalDebits[i]) + "\n");
```

This is much the same as before except that we now extract values from the arrays we have created. If you run this version it will, of course, work perfectly. A typical set of results are:

```
Account Number:1
Original balance    : $500
Total credits       : $659
Total debits        : $614
Final balance       : $545
Should be           : $545

Account Number:2
Original balance    : $800
Total credits       : $607
Total debits        : $306
Final balance       : $1101
Should be           : $1101
```

How It Works

We now allocate arrays for the initial account balances, the total of credits and debits for each account and the accounts themselves. The number of initializing values in the `initialBalance[]` array will determine the number of elements in each of the arrays. In the `for` loop, we create each of the accounts with the appropriate initial balance, and initialize the `totalCredits[]` and `totalDebits[]` arrays to zero.

In the modified transactions loop, we select the account from the array for both the debit and the credit transactions by generating a random index value which we store in the variable, `select`. The index, `select`, is also used to keep a tally of the total of the transactions of each type.

This is all well and good, but by declaring the methods in the class `Bank` as `synchronized`, we're limiting the program quite significantly. No operation of any kind can be carried out while any other operation is in progress. This is unnecessarily restrictive since there's no reason to prevent a transaction on one account while a transaction for a different account is in progress. What we really want to do is constrain the program to prevent overlapping of operations on the same account, and this is where declaring blocks of code to be synchronized on a particular object can help.

Let's consider the methods in the class `Bank` once more. What we really want is the code in the `doTransaction()` method to be synchronized so that simultaneous processing of the same account is prevented, not that processing of different accounts is inhibited. What we need to do is synchronize the processing code for a transaction on the account object that is involved.

Try It Out — Applying `synchronized` Statement Blocks

We can do this with the following changes:

```
class Bank
{
  // Perform a transaction
  public void doTransaction(Transaction transaction)
  {
    switch(transaction.getTransactionType())
    {
      case Transaction.CREDIT:
      synchronized(transaction.getAccount())
      {
         // Get current balance
         int balance = transaction.getAccount().getBalance();

        // Credits require require a lot of checks...
        try
        {
          Thread.sleep(100);
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
        balance += transaction.getAmount();            // Increment the balance...
        transaction.getAccount().setBalance(balance);   // Restore account balance
        break;
      }
      case Transaction.DEBIT:
      synchronized(transaction.getAccount())
      {
        // Get current balance
        int balance = transaction.getAccount().getBalance();
```

```
            // Debits require even more checks...
            try
            {
              Thread.sleep(150);
            }
            catch(InterruptedException e)
            {
              System.out.println(e);
            }
            balance -= transaction.getAmount();            // Increment the balance...
            transaction.getAccount().setBalance(balance);  // Restore account balance
            break;
          }
          default:                                         // We should never get here
            System.out.println("Invalid transaction");
            System.exit(1);
        }
      }
    }
```

How It Works

The expression in parentheses following the keyword `synchronized` specifies the object for which the synchronization applies. Once one synchronized code block is entered with a given account object, no other code block or method can be entered that has been synchronized on the same object. For example, if the block performing credits is executing with a reference to the object `accounts[1]` returned by the `getAccount()` method for the transaction, the execution of the block carrying out debits cannot be executed for the same object, but it could be executed for a different account.

The object in a synchronized code block acts rather like a baton in a relay race that serves to synchronize the runners in the team. Only the runner with the baton is allowed to run. The next runner in the team can only run once they get hold of the baton. Of course, in any race there will be several different batons so you can have several sets of runners. In the same way, you can specify several different sets of `synchronized` code blocks in a class, each controlled by a different object. It is important to realize that code blocks that are synchronized with respect to a particular object don't have to be in the same class. They can be anywhere in your program where the appropriate object can be specified.

Note how we had to move the code to access and restore the account balance inside both synchronized blocks. If we hadn't done this, accessing or restoring the account balance could occur while a synchronized block was executing. This could obviously cause confusion since a balance could be restored by a debit transaction after the balance had been retrieved for a credit transaction. This would cause the effect of the debit to be wiped out.

If you want to verify that we really are overlapping these operations in this example, you can add output statements to the beginning and end of each method in the class `Bank`. Outputting the type of operation, the amount and whether it is the start or end of the transaction will be sufficient to identify them. For example, you could modify the `doTransaction()` method in the `Bank` class to:

```
    public void doTransaction(Transaction transaction)
    {
      switch(transaction.getTransactionType())
      {
        case Transaction.CREDIT:
        synchronized(transaction.getAccount())
        {
           System.out.println("Start credit of " +
                    transaction.getAccount() + " amount: " +
                    transaction.getAmount());
           // Code to process credit...
           System.out.println("  End credit of " +
                    transaction.getAccount() + " amount: " +
                    transaction.getAmount());
           break;
        }
        case Transaction.DEBIT:
        synchronized(transaction.getAccount())
        {
           System.out.println("Start debit of " +
                    transaction.getAccount() + " amount: " +
                    transaction.getAmount());
          // Code to process debit...
          System.out.println("  End debit of " +
                    transaction.getAccount() + " amount: " +
                    transaction.getAmount());
          break;
        }
        default:                                        // We should never get here
          System.out.println("Invalid transaction");
          System.exit(1);
      }
    }
```

This will produce quite a lot of output, but you can always comment it out when you don't need it. You should be able to see how a transaction for an account that is currently being worked on is always delayed until the previous operation on the account is completed. You will also see from the output that operations on different accounts do overlap. Here's a sample of what I got:

```
Start credit of A//C No. 2 : $800 amount: 74
  End credit of A//C No. 2 : $874 amount: 74
Start debit of A//C No. 2 : $874 amount: 52
Start credit of A//C No. 1 : $500 amount: 51
  End debit of A//C No. 2 : $822 amount: 52
  End credit of A//C No. 1 : $551 amount: 51
Start debit of A//C No. 2 : $822 amount: 38
  End debit of A//C No. 2 : $784 amount: 38
Start credit of A//C No. 2 : $784 amount: 74
  End credit of A//C No. 2 : $858 amount: 74
Start debit of A//C No. 1 : $551 amount: 58
Start credit of A//C No. 2 : $858 amount: 53
  End debit of A//C No. 1 : $493 amount: 58
...
```

You can see from the third and fourth lines here that a credit for account 1 starts before the preceding debit for account 2 is complete, so the operations are overlapped. If you want to force overlapping debits and credits on the same account, you can comment out the calculation of the value for `select` for the debit operation in the `for` loop in `main()`. This modification is shown shaded:

```
// Generate a random account index for debit operation
// select = rand.nextInt(accounts.length);
totalDebits[select] += amount;              // Keep total debit tally
```

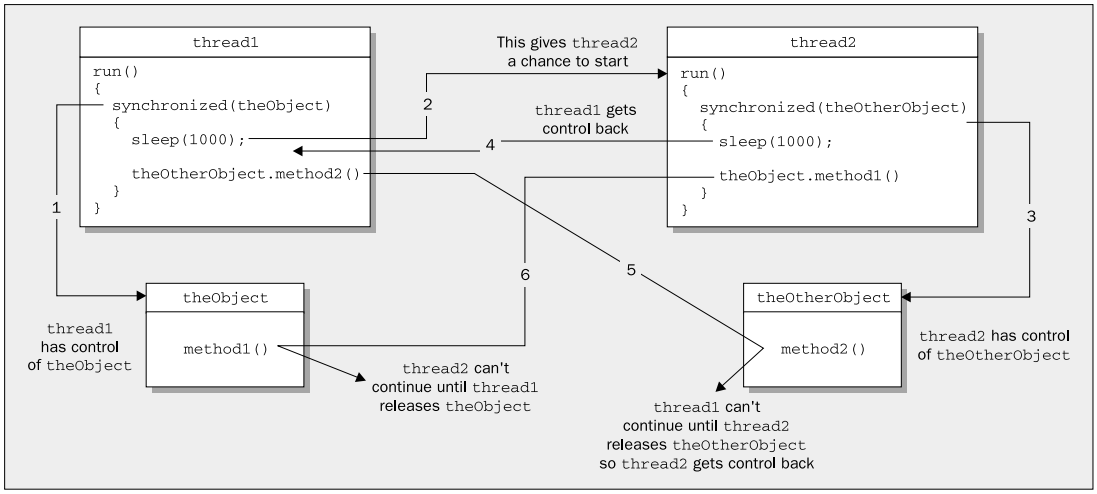
This will make the debit transaction apply to the same account as the previous credit, so the transactions will always be contending for the same account.

Of course, this is not the only way of getting the operations to overlap. Another approach would be to equip accounts with methods to handle their own credit and debit transactions, and declare these as synchronized methods.

While testing that you have synchronization right is relatively easy in our example, in general it is extremely difficult to be sure you have tested a program that uses threads adequately. Getting the design right first is essential, and there is no substitute for careful design in programs that have multiple threads (or indeed any real time program that has interrupt handlers). You can never be sure that a real world program is 100% correct, only show it works correctly most of the time!

Deadlocks

Since you can synchronize code blocks for a particular object virtually anywhere in your program, there's potential for a particularly nasty kind of bug called a **deadlock**. This involves a mutual interdependence between two threads. One way this arises is when one thread executes some code synchronized on a given object, `theObject` say, and then needs to execute another method that contains code synchronized on another object, `theOtherObject` say. Before this occurs though, a second thread executes some code synchronized to `theOtherObject`, and needs to execute a method containing code synchronized to first object, `theObject`. This situation is illustrated here:



The sequence of events is as follows:

- `thread1` starts first, and synchronizes on `theObject`. This prevents any methods for `theObject` being called by any other thread.
- `thread1` then calls `sleep()` so `thread2` can start.
- `thread2` starts and synchronizes on `theOtherObject`. This prevents any methods for `theOtherObject` being called by any other thread.
- `thread2` then calls `sleep()` allowing `thread1` another go.
- `thread1` wakes up and tries to call `method2()` for `theOtherObject`, but it can't until the code block in `thread2` that is synchronized on `theOtherObject` completes execution.
- `thread2` gets another go because `thread1` can't proceed, and tries to call `method1()` for `theObject`. This can't proceed until the code block in `thread1` that is synchronized on `theObject` completes execution.

Neither thread has any possibility of continuing – they are deadlocked. Finding and fixing this sort of problem can be very difficult, particularly if your program is complicated and has other threads which will continue to execute.

You can create a trivial deadlock in the last example by making the `for` loop in `main()` synchronized on one of the accounts. For example:

```
synchronized(accounts[1]){
  for(int i = 1; i <= transactionCount; i++)
  {
    // code for generating transactions etc...
  }
}
```

A deadlock occurs as soon as a transaction for `accounts[1]` arises because the `doTransaction()` method in `theBank` object that is called by a `Clerk` object to handle the transaction will be synchronized to the same object and can't execute until the loop ends. Of course, the loop can't continue until the method in `theBank` object terminates so the program hangs.

In general, ensuring that your program has no potential deadlocks is extremely difficult. If you intend to do a significant amount of programming using threads, you will need to study the subject in much more depth than we can deal with here. A good book on the subject is *Concurrent Programming in Java: Design Principles and Patterns* written by Doug Lea (ISBN 0-201-69581-2).

Communicating between Threads

We've seen how we can lock methods or code blocks using synchronization to avoid the problems that uncontrolled thread execution can cause. While this gives us a degree of control, we're still introducing inefficiencies into the program. In the last example, there were several occasions where we used a loop to wait for a clerk thread to complete an operation before the current thread could sensibly continue. For example, we couldn't pass a transaction to a `Clerk` object while that object was still busy with the previous transaction. Our solution to this was to use a `while` loop to test the busy status of the `Clerk` object from time to time and call the `sleep()` method in between. But there's a much better way.

The `Object` class defines the methods, `wait()`, `notify()` and `notifyAll()`, that you can use to provide a more efficient way of dealing with this kind of situation. Since all classes are derived from `Object`, all classes inherit these methods. You can only call these methods from within a `synchronized` method, or from within a synchronized code block, and an exception of type `IllegalMonitorStateException` will be thrown if you don't. The functions that these methods perform are:

Method	Description
`wait()`	There are three overloaded versions of this method. This version suspends the current thread until the `notify()` or `notifyAll()` method is called for the object to which the `wait()` method belongs. Note that when any version of `wait()` is called, the thread releases the synchronization lock it has on the object, so any other method or code block synchronized on the same object can execute. As well as enabling `notify()` or `notifyAll()` to be called by another thread, this also allows another thread to call `wait()` for the same object. Since all versions of the `wait()` method can throw an `InterruptedException`, you must call it in a `try` block with a `catch` block for this exception, or at least indicate that the method calling it throws this exception.
`wait(long timeout)`	This version suspends the current thread until the number of milliseconds specified by the argument has expired, or until the `notify()` or `notifyAll()` method for the object to which the `wait()` method belongs, is called, if that occurs sooner.
`wait(long timeout, int nanos)`	This version works in the same way as the previous version, except the time interval is specified by two arguments, the first in milliseconds, and the second in nanoseconds.
`notify()`	This will restart a thread that has called the `wait()` method for the object to which the `notify()` method belongs. If several threads have called `wait()` for the object, you have no control over which thread is notified, in which case it is better to use `notifyAll()`. If no threads are waiting, the method does nothing.

Table continued on following page

Method	Description
`notifyAll()`	This will restart all threads that have called `wait()` for the object to which the `notifyAll()` method belongs.

The basic idea of the `wait()` and `notify()` methods is that they provide a way for methods or code blocks that are synchronized on a particular object to communicate. One block can call `wait()` to suspend its operation until some other method or code block synchronized on the same object changes it in some way, and calls `notify()` to signal the change is complete. A thread will typically call `wait()` because some particular property of the object it is synchronized on is not set, or some condition is not fulfilled, and this is dependent on action by another thread. Perhaps the simplest situation is where a resource is busy because it is being modified by another thread, but you are by no means limited to that.

The major difference between calling `sleep()` and calling `wait()` is that `wait()`releases any objects on which the current thread has a lock, whereas `sleep()` does not. It is essential that `wait()` should work this way, otherwise there would be no way for another thread to change things so that the condition required by the current thread is met.

Thus the typical use of `wait()` is:

```
synchronized(anObject)
{
  while(condition-not-met)
    anObject.wait();
  // Condition is met so continue...
}
```

Here the thread will suspend operation when the `wait()` method is called until some other thread synchronized on the same object calls `notify()` (or more typically `notifyAll()`). This latter allows the `while` loop to continue and check the condition again. Of course, it may still not be met, in which case the `wait()` method will be called again so another thread can operate on `anObject`. You can see from this that `wait()` is not just for getting access to an object. It is intended to allow other threads access until some condition has been met. You could even arrange that a thread would not continue until a given number of other threads had called `notify()` on the object to ensure that a minimum number of operations had been carried out.

It is generally better to use `notifyAll()` rather than `notify()` when you have more than two threads synchronized on an object. If you call `notify()` when there are two or more other threads suspended having called `wait()`, only one of the threads will be started but you have no control over which it is. This opens the possibility that the thread that is started calls `wait()` again because the condition it requires is not fulfilled. This will leave all the threads waiting for each other, with no possibility of continuing.

Although the action of each of these methods is quite simple, applying them can become very complex. You have the potential for multiple threads to be interacting through several objects with `synchronized` methods and code blocks. We'll just explore the basics by seeing how we can use `wait()` and `notifyAll()` to get rid of a couple of the `while` loops we had in the last example.

Using wait() and notifyAll() in the Bank Program

In the `for` loop in `main()` that generates the transactions and passes them to the `Clerk` objects, we have two `while` loops that call the `isBusy()` method for a `Clerk` object. These were needed so that we didn't pass a transaction to a clerk while the clerk was still busy. By altering the `Clerk` class, so that it can use `wait()`and `notifyAll()`, we can eliminate the need for these.

Try It Out — Slimming Down the Transactions Loop

We want to make the `doTransaction()` method in the `Clerk` class conscious of the state of the `inTray` for the current object. If it is not `null`, we want the method to wait until it becomes so. To use `wait()` the block or method must be synchronized on an object – in this case the `Clerk` object since `inTray` is what we are interested in. We can do this by making the method synchronized:

```
public class Clerk implements Runnable
{
  synchronized public void doTransaction(Transaction transaction)
  {
    while(inTray != null)
      try
      {
        wait();
      }
      catch(InterruptedException e)
      {
        System.out.println(e);
      }
    inTray = transaction;
    notifyAll();
  }

  // Rest of the class as before...
}
```

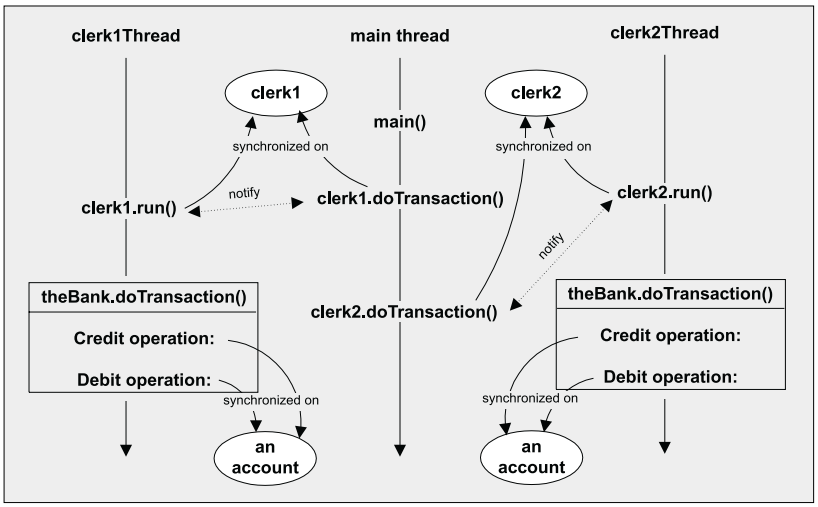
When `inTray` is `null`, the transaction is stored and the `notifyAll()` method is called to notify other threads waiting on a change to this `Clerk` object. If `inTray` is not `null`, this method waits until some other thread calls `notifyAll()` to signal a change to the `Clerk` object. We now need to consider where the `inTray` field is going to be modified elsewhere. The answer is in the `run()` method for the `Clerk` class of course, so we need to change that too:

```
class Clerk
{
  synchronized public void run()
  {
    while(true)
    {
      while(inTray == null)           // No transaction waiting?
        try
        {
          wait();                     // Then take a break until there is
        }
        catch(InterruptedException e)
        {
          System.out.println(e);
        }
```

```
            theBank.doTransaction(inTray);
            inTray = null;                // In-tray is empty
            notifyAll();                  // Notify other threads locked on this clerk
          }
        }

        // Rest of the class as before...
      }
```

Just to make it clear which methods are in what threads, the situation in our program is illustrated below.



Here the `run()` method is synchronized on the `Clerk` object that contains it, and the method waits if `inTray` is `null`. Eventually the `doTransaction()` method for the current object should store a transaction in `inTray`, and then notify the thread that is waiting on the `Clerk` object – which will be the thread running the `theBank.doTransaction()` method for the object – that it should continue.

It may seem odd having two methods in the same object synchronized on one and the same object that owns them, but remember that the `run()` and `doTransaction()` methods for a particular `Clerk` object are in separate threads.

The transaction processing method for the bank can be in both of the clerk threads, whereas the methods that hand over a transaction to a clerk are in the main thread. The diagram also shows which code is synchronized on what objects.

We can now modify the code in the `for` loop in `main()`to pass the transactions directly to the clerks:

```
      // Generate transactions of each type and pass to the clerks
  for(int i = 1; i <= transactionCount; i++)
      {
        // Generate a random account index for credit operation
```

```
        select = rand.nextInt(accounts.length);
        amount = 50 + rand.nextInt(26);            // Generate amount of $50 to $75
        transaction = new Transaction(accounts[select],    // Account
                                      Transaction.CREDIT, // Credit transaction
                                       amount);           //  of amount
        totalCredits[select] += amount;            // Keep total credit tally

        clerk1.doTransaction(transaction);         // Now do the credit

        // Generate a random account index for debit operation
        select = rand.nextInt(accounts.length);
        amount = 30 + rand.nextInt(31);            // Generate amount of $30 to $60
        transaction = new Transaction(accounts[select],   // Account
                                      Transaction.DEBIT,  // Debit transaction
                                      amount);            //  of amount
        totalDebits[select] += amount;             // Keep total debit tally

        clerk2.doTransaction(transaction);         // Now do the debit
      }
```

We have just deleted the loop blocks that were waiting until a clerk became free. This makes our code a lot shorter.

The example will now run without the need for checking whether the `Clerk` objects are busy in the transaction processing loop in `main()`.

With a small change to the `isBusy()` method in the `Clerk` class, we can eliminate the need for the `while` loop before we output the results in `main()`:

```
synchronized public void isBusy()
{
  while(inTray != null)             // Is this object busy?
    try
    {
      wait();                       // Yes, so wait for notify call
    }
    catch(InterruptedException e)
    {
      System.out.println(e);
    }
  return;                           // It is free now
}
```

Now the `isBusy()` method will only return when the clerk object has no transaction waiting or in progress, so no return value is necessary. The `while` loop in `main()` before the final output statements can be replaced by:

```
// Wait if clerks are busy
clerk1.isBusy();
clerk2.isBusy();
```

How It Works

The `doTransaction()` method for a `Clerk` object calls the `wait()` method if the `inTray` field contains a reference to a transaction object, as this means the `Clerk` object is still processing a credit or a debit. This will result in the current thread (which is the main thread) being suspended until the thread corresponding to this `Clerk` object – which is in the `run()` method – calls `notifyAll()` to indicate a change to the clerk.

Because the `run()` method is also synchronized on the `Clerk` object, it too can call `wait()`, in this case, if the `inTray` contains `null`, since this indicates that there is no transaction waiting for the clerk to expedite. A call to the `doTransaction()` method for the `Clerk` object will result in a transaction being stored in `inTray`, and the `notifyAll()` call will wake up the `run()` method to continue execution.

Because we've declared the `isBusy()` method as `synchronized`, we can call the `wait()` method to suspend the current thread if transactions are still being processed. Since we don't return from the method until the outstanding transaction is complete, we have no need of a `boolean` return value.

Thread Priorities

All threads have a priority which determines which thread is executed when several threads are waiting for their turn. This makes it possible to give one thread more access to processor resources than another. Let's consider an elementary example of how this could be used. Suppose you have one thread in a program that requires all the processor resources – some solid long running calculation, and some other threads that require relatively little resource. By making the thread that requires all the resources a low priority thread, you ensure that the other threads get executed promptly, while the processor bound thread can make use of the processor cycles that are left over after the others have had their turn.

The possible values for thread priority are defined in `static` data members of the class `Thread`. These members are of type `int`, and declared as `final`. The maximum thread priority is defined by the member `MAX_PRIORITY` which has the value 10. The minimum priority is `MIN_PRIORITY` defined as 1. The value of the default priority that is assigned to the main thread in a program is `NORM_PRIORITY` which is set to 5. When you create a thread, its priority will be the same as that of the thread that created it.

You can modify the priority of a thread by calling the `setPriority()` method for the `Thread` object. This method accepts an argument of type `int` which defines the new priority for the thread. An `IllegalArgumentException` will be thrown if you specify a priority that is less than `MIN_PRIORITY` or greater than `MAX_PRIORITY`.

If you're going to be messing about with the priorities of the threads in your program, you need to be able to find out the current