Threads

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control that begins execution in the program's `main()` method.

4.1 Creating Java Threads

There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the `Thread` class and to override its `run()` method. However, the most common technique is to define a class that implements the `Runnable` interface. The `Runnable` interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

When a class implements `Runnable`, it must define a `run()` method. The code implementing the `run()` method is what runs as a separate thread.

Figure 4.1 shows the Java version of a multithreaded program that determines the summation of a non-negative integer. The `Summation` class implements the `Runnable` interface. Thread creation is performed by creating an object instance of the `Thread` class and passing the constructor a `Runnable` object.

Creating a `Thread` object does not specifically create the new thread; rather, it is the `start()` method that actually creates the new thread. Calling the `start()` method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the `run()` method, making the thread eligible to be run by the JVM. (Note that we never call the `run()` method directly. Rather, we call the `start()` method, and it calls the `run()` method on our behalf.)

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}

public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         // create the object to be shared
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.1 Java program for the summation of a non-negative integer.

When the summation program runs, two threads are created by the JVM. The first is the parent thread, which starts execution in the `main()` method. The second thread is created when the `start()` method on the `Thread` object is invoked. This child thread begins execution in the `run()` method of the `Summation` class. After outputting the value of the summation, this thread terminates when it exits from its `run()` method.

Sharing of data between threads occurs easily in Win32 and Pthreads, as shared data are simply declared globally. As a pure object-oriented language, Java has no such notion of global data; if two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads. In the Java program shown in Figure 4.1, the main thread and the summation thread share the object instance of the `Sum` class. This shared object is referenced through the appropriate `getSum()` and `setSum()` methods. (You might wonder why we don't use a `java.lang.Integer` object rather than designing a new `Sum` class. The reason is that the `java.lang.Integer` class is **immutable**—that is, once its integer value is set, it cannot change.)

Recall that the parent threads in the Pthreads and Win32 libraries use `pthread_join()` and `WaitForSingleObject()` (respectively) to wait for the summation threads to finish before proceeding. The `join()` method in Java provides similar functionality. Notice that `join()` can throw an `InterruptedException`, which we choose to ignore for now. We discuss handling this exception in Chapter 5.

Java actually identifies two different types of threads: (1) daemon (pronounced "demon") and (2) non-daemon threads. The fundamental difference between the two types is the simple rule that the JVM shuts down when all non-daemon threads have exited. Otherwise, the two thread types are identical. When the JVM starts up, it creates several internal daemon threads for performing tasks such as garbage collection. A daemon thread is created by invoking the `setDaemon()` method of the `Thread` class and passing the method the value `true`. For example, we could have set the thread in the program shown in Figure 4.1 as a daemon by adding the following line after creating—but before starting—the thread:

```
thrd.setDaemon(true);
```

For the remainder of this text, we will refer only to non-daemon threads unless otherwise specified.

4.2 The JVM and the Host Operating System

As we have discussed, the JVM is typically implemented on top of a host operating system (see Figure 16.10). This setup allows the JVM to hide the implementation details of the underlying operating system and to provide a consistent, abstract environment that allows Java programs to operate on any platform that supports a JVM. The specification for the JVM does not indicate how Java threads are to be mapped to the underlying operating system, instead leaving that decision to the particular implementation of the JVM. For example, the Windows XP operating system uses the one-to-one model; therefore, each

Java thread for a JVM running on such a system maps to a kernel thread. On operating systems that use the many-to-many model (such as Tru64 UNIX), a Java thread is mapped according to the many-to-many model. Solaris initially implemented the JVM using the many-to-one model (the green threads library mentioned earlier). Later releases of the JVM used the many-to-many model. Beginning with Solaris 9, Java threads were mapped using the one-to-one model. In addition, there may be a relationship between the Java thread library and the thread library on the host operating system. For example, implementations of a JVM for the Windows family of operating systems might use the Win32 API when creating Java threads; Linux and Solaris systems might use the Pthreads API.

4.3 Java Thread States

A Java thread may be in one of six possible states in the JVM:

1. **New**. A thread is in this state when an object for the thread is created with the `new` command but the thread has not yet started.
2. **Runnable**. Calling the `start()` method allocates memory for the new thread in the JVM and calls the `run()` method for the thread object. When a thread's `run()` method is invoked, the thread moves from the new to the runnable state. A thread in the runnable state is eligible to be run by the JVM. Note that Java does not distinguish between a thread that is eligible to run and a thread that is currently running. A running thread is still in the runnable state.
3. **Blocked**. A thread is in this state as it waits to acquire a lock—a tool used for thread synchronization. We cover such tools in Chapter 5.
4. **Waiting**. A thread in this state is waiting for an action by another thread. For example, a thread invoking the `join()` method enters this state as it waits for the thread it is joining on to terminate.
5. **Timed waiting**. This state is similar to waiting, except a thread specifies the maximum amount of time it will wait. For example, the `join()` method has an optional parameter that the waiting thread can use to specify how long it will wait until the other thread terminates. Timed waiting prevents a thread from remaining in the waiting state indefinitely.
6. **Terminated**. A thread moves to the this state when its `run()` method terminates.

Figure 4.2 illustrates these different thread states and labels several possible transitions between states. It is important to note that these states relate to the Java virtual machine and are not necessarily associated with the state of the thread running on the host operating system.

The Java API for the `Thread` class provides several methods to determine the state of a thread. The `isAlive()` method returns `true` if a thread has been started but has not yet reached the Terminated state; otherwise, it returns `false`. The `getState()` method returns the state of a thread as an enumerated

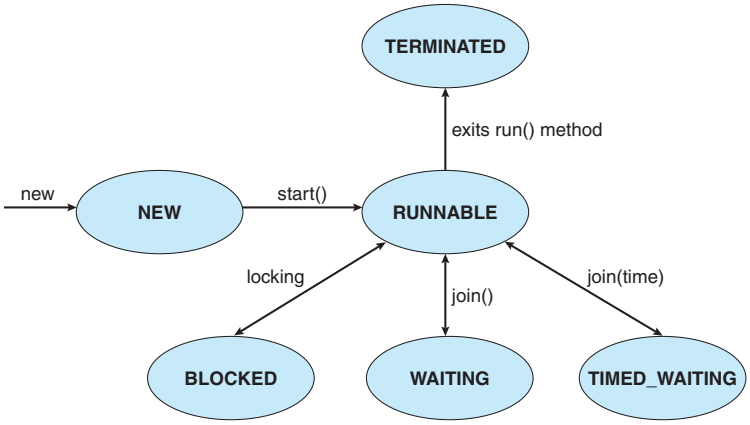

Figure 4.2 Java thread states.

data type as one of the values from above. The source code available with this text provides an example program using the `getState()` method.

4.4 Solution to the Producer–Consumer Problem

We conclude our discussion of Java threads with a complete multithreaded solution to the producer–consumer problem that uses message passing. The class `Factory` in Figure 4.3 first creates a message queue for buffering messages, using the `MessageQueue` class developed in Chapter 3. It then creates separate producer and consumer threads (Figure 4.4 and Figure 4.5, respectively) and

```
import java.util.Date;

public class Factory
{
   public static void main(String args[]) {
      // create the message queue
      Channel<Date> queue = new MessageQueue<Date>();

      // Create the producer and consumer threads and pass
      // each thread a reference to the MessageQueue object.
      Thread producer = new Thread(new Producer(queue));
      Thread consumer = new Thread(new Consumer(queue));

      // start the threads
      producer.start();
      consumer.start();
   }

}
```

Figure 4.3 The `Factory` class.

```
import java.util.Date;

class Producer implements Runnable
{
   private Channel<Date> queue;

   public Producer(Channel<Date> queue) {
      this.queue = queue;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item and enter it into the buffer
         message = new Date();
         System.out.println("Producer produced " + message);
         queue.send(message);
      }
   }
}
```

Figure 4.4 Producer thread.

passes each thread a reference to the shared queue. The producer thread alternates among sleeping for a while, producing an item, and entering that item into the queue. The consumer alternates between sleeping and then retrieving an item from the queue and consuming it. Because the `receive()` method of the `MessageQueue` class is nonblocking, the consumer must check to see whether the message that it retrieved is null.

4.5 Thread Cancellation

Java threads can be asynchronously terminated using the `stop()` method of the `Thread` class. However, this method has been **deprecated**. Deprecated methods are still implemented in the current API, but their use is discouraged. We discuss why `stop()` was deprecated in Section 7.2.

It is also possible to cancel a Java thread using deferred cancellation. As just described, deferred cancellation works by having the target thread periodically check whether it should terminate. In Java, checking involves use of the `interrupt()` method. The Java API defines the `interrupt()` method for the `Thread` class. When the `interrupt()` method is invoked, the **interruption status** of the target thread is set. A thread can periodically check its interruption status by invoking either the `interrupted()` method or the `isInterrupted()` method, both of which return `true` if the interruption status of the target thread is set. (It is important to note that the `interrupted()` method clears

```
import java.util.Date;

class Consumer implements Runnable
{
   private Channel<Date> queue;

   public Consumer(Channel<Date> queue) {
      this.queue = queue;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = queue.receive();

         if (message != null)
            System.out.println("Consumer consumed " + message);
      }
   }
}
```

Figure 4.5 Consumer thread.

the interruption status of the target thread, whereas the `isInterrupted()` method preserves the interruption status.) Figure 4.6 illustrates how deferred cancellation works when the `isInterrupted()` method is used.

An instance of an `InterruptibleThread` can be interrupted using the following code:

```
Thread thrd = new Thread(new InterruptibleThread());
thrd.start();
. . .
thrd.interrupt();
```

In this example, the target thread can periodically check its interruption status via the `isInterrupted()` method and—if it is set—clean up before terminating. Because the `InterruptibleThread` class does not extend `Thread`, it cannot directly invoke instance methods in the `Thread` class. To invoke instance methods in `Thread`, a program must first invoke the `static` method `currentThread()`, which returns a `Thread` object representing the thread that is currently running. This return value can then be used to access instance methods in the `Thread` class.

It is important to recognize that interrupting a thread via the `interrupt()` method only sets the interruption status of a thread; it is up to the target thread to check this interruption status periodically. Traditionally, Java does not wake a thread that is blocked in an I/O operation using the `java.io`

```
class InterruptibleThread implements Runnable
{
   /**
    * This thread will continue to run as long
    * as it is not interrupted.
    */
   public void run() {
      while (true) {
         /**
          * do some work for awhile
          * . . .
          */

         if (Thread.currentThread().isInterrupted()) {
            System.out.println("I'm interrupted!");
            break;
         }
      }
      // clean up and terminate
   }
}
```

Figure 4.6 Deferred cancellation using the `isInterrupted()` method.

package. Any thread blocked doing I/O in this package will not be able to check its interruption status until the call to I/O is completed. However, the `java.nio` package introduced in Java 1.4 provides facilities for interrupting a thread that is blocked performing I/O.

4.6 Thread Pools

The `java.util.concurrent` package includes an API for thread pools, along with other tools for concurrent programming. The Java API provides several varieties of thread-pool architectures; we focus on the following three models, which are available as `static` methods in the `java.util.concurrent.Executors` class:

1. Single thread executor—`newSingleThreadExecutor()`—creates a pool of size 1.
2. Fixed thread executor—`newFixedThreadPool(int size)`—creates a thread pool with a specified number of threads.
3. Cached thread executor—`newCachedThreadPool()`—creates an unbounded thread pool, reusing threads in many instances.

In the Java API, thread pools are structured around the `Executor` interface, which appears as follows:

```
public interface Executor
{
   void execute(Runnable command);
}
```

Classes implementing this interface must define the `execute()` method, which is passed a `Runnable` object such as the following:

```
public class Task implements Runnable
{
   public void run() {
    System.out.println("I am working on a task.");
    . . .
   }
}
```

For Java developers, this means that code that runs as a separate thread using the following approach:

```
Thread worker = new Thread(new Task());
worker.start();
```

can also run as an `Executor`:

```
Executor service = new Executor;
service.execute(new Task());
```

A thread pool is created using one of the factory methods in the `Executors` class:

- `static ExecutorService newSingleThreadExecutor()`
- `static ExecutorService newFixedThreadPool(int nThreads)`
- `static ExecutorService newCachedThreadPool()`

Each of these factory methods creates and returns an object instance that implements the `ExecutorService` interface. `ExecutorService` extends the `Executor` interface, allowing us to invoke the `execute()` method on this object. However, `ExecutorService` also provides additional methods for managing termination of the thread pool.

The example shown in Figure 4.7 creates a cached thread pool and submits tasks to be executed by a thread in the pool. When the `shutdown()` method is invoked, the thread pool rejects additional tasks and shuts down once all existing tasks have completed execution. In Chapter 5, we provide a programming exercise to design and implement a thread pool.

4.7 Thread-Specific Data

At first glance, it may appear that Java has no need for thread-specific data, since all that is required to give each thread its own private data is to create

```
import java.util.concurrent.*;

public class TPExample
{
   public static void main(String[] args) {
      int numTasks = Integer.parseInt(args[0].trim());

      // Create the thread pool
      ExecutorService pool = Executors.newCachedThreadPool();

      // Run each task using a thread in the pool
      for (int i = 0; i < numTasks; i++)
         pool.execute(new Task());

      // Shut down the pool. This shuts down the pool only
      // after all threads have completed.
      pool.shutdown();
   }
}
```

Figure 4.7 Creating a thread pool in Java.

threads by subclassing the `Thread` class and to declare instance data in this class. Indeed, as long as threads are constructed in that way, this approach works fine. However, when the developer has no control over the thread-creation process—for example, when a thread pool is being used—then an alternative approach is necessary.

The Java API provides the `ThreadLocal` class for declaring thread-specific data. `ThreadLocal` data can be initialized with either the `initialValue()` method or the `set()` method, and a thread can inquire as to the value of `ThreadLocal` data using the `get()` method. Typically, `ThreadLocal` data are declared as `static`. Consider the `Service` class shown in Figure 4.8, which declares `errorCode` as `ThreadLocal` data. The `transaction()` method in this class can be invoked by any number of threads. If an exception occurs, we assign the exception to `errorCode` using the `set()` method of the `ThreadLocal` class. Now consider a scenario in which two threads—say, thread 1 and thread 2—invoke `transaction()`. Assume that thread 1 generates exception A and thread 2 generates exception B. The value of `errorCode` for thread 1 and thread 2 will be A and B, respectively. Figure 4.9 illustrates how a thread can inquire as to the value of `errorCode()` after invoking `transaction()`.

Programming Problems

4.1 Exercise 3.3 in Chapter 3 involves designing a client–server program where the client sends a message to the server and the server responds with a count containing the number of characters and digits in the message. However, this server is single-threaded, meaning that the server cannot respond to other clients until the current client closes

```
class Service
{
   private static ThreadLocal errorCode =
     new ThreadLocal();

   public static void transaction() {
      try {
         /**
          * some operation where an error may occur
          . . .
          */
      }
      catch (Exception e) {
          errorCode.set(e);
      }
   }

   /**
    * Get the error code for this transaction
    */
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

Figure 4.8 Using the `ThreadLocal` class.

its connection. Modify your solution to Exercise 3.3 so that the server services each client in a separate request.

4.2 Write a multithreaded Java program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

4.3 Modify the socket-based date server (Figure 3.11) in Chapter 3 so that the server services each client request in a separate thread.

```
class Worker implements Runnable
{
   private static Service provider;

   public void run() {
      provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}
```

Figure 4.9 Inquiring the value of `ThreadLocal` data.

4.4 Modify the socket-based date server (Figure 3.11) in Chapter 3 so that the server services each client request using a thread pool.

4.5 The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$
$$fib_1 = 1$$
$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a multithreaded program that generates the Fibonacci sequence using Java. This program should work as follows: The user will enter on the command line the number of Fibonacci numbers that the program is to generate. The program will then create a separate thread that will generate the Fibonacci numbers, placing the sequence in data that can be shared by the threads (an array is probably the most convenient data structure). When the thread finishes execution, the parent thread will output the sequence generated by the child thread. Because the parent thread cannot begin outputting the Fibonacci sequence until the child thread finishes, this will require having the parent thread wait for the child thread to finish, using the techniques described in Section 4.4.

4.6 Write a multithreaded sorting program in Java that works as follows: A collection of items is divided into two lists of equal size. Each list is then passed to a separate thread (a *sorting thread*), which sorts the list using any sorting algorithm (or algorithms) of your choice. The two sorted lists are passed to a third thread (a *merge thread*), which merges the two separate lists into a single sorted list. Once the two lists have been merged, the complete sorted list is output. If we were sorting integer values, this program should be structured as depicted in Figure 4.11.

Perhaps the easiest way of designing a sorting thread is to pass the constructor an array containing `java.lang.Object`, where each `Object` must implement the `java.util.Comparable` interface. Many objects in the Java API implement the `Comparable` interface. For the purposes of this project, we recommend using `Integer` objects. To ensure that the two sorting threads have completed execution, the main thread will need to use the `join()` method on the two sorting threads before passing the

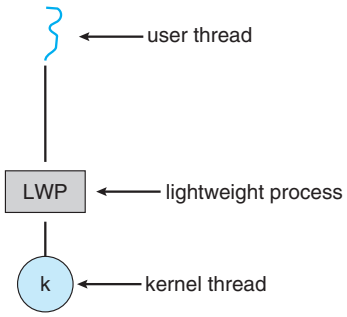

Figure 4.10 Lightweight process (LWP).

two sorted lists to the merge thread. Similarly, the main thread will need to use `join()` on the merge thread before it outputs the complete sorted list. For a discussion of sorting algorithms, consult the bibliography.

4.7 Write a Java program that lists all threads in the Java virtual machine. Before proceeding, you will need some background information. All threads in the JVM belong to a **thread group**, and a thread group is identified in the Java API by the `ThreadGroup` class. Thread groups are organized as a tree structure, where the root of the tree is the **system thread group**. The system thread group contains threads that are automatically created by the JVM, mostly for managing object references. Below the system thread group is the **main thread group**. The main thread group contains the initial thread in a Java program

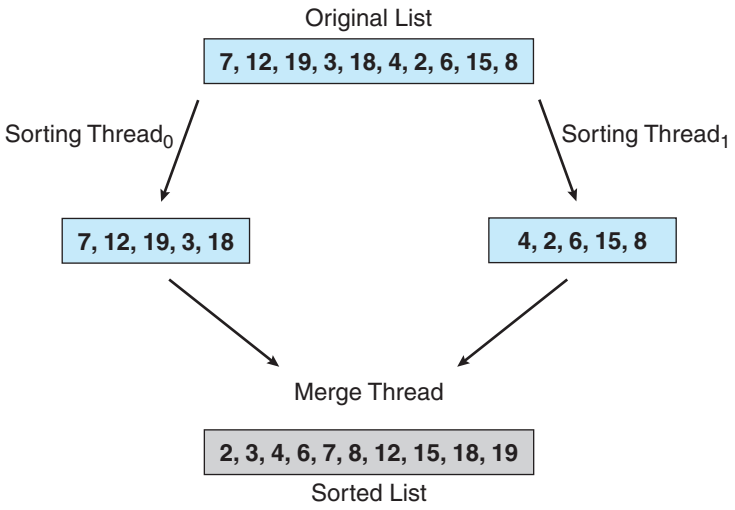

Figure 4.11 Sorting.

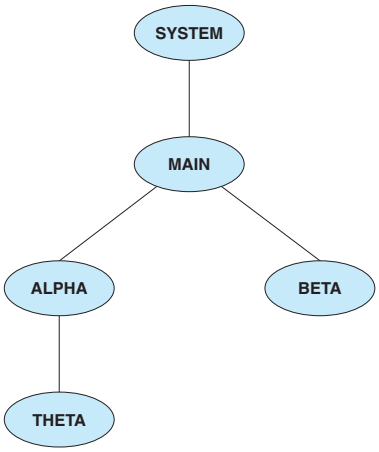

Figure 4.12 Thread-group hierarchy.

that begins execution in the `main()` method. It is also the default thread group, meaning that—unless otherwise specified—all threads you create belong to this group. It is possible to create additional thread groups and assign newly created threads to these groups. Furthermore, when creating a thread group, you may specify its parent. For example, the following statements create three new thread groups: `alpha`, `beta`, and `theta`:

```
ThreadGroup alpha = new ThreadGroup("alpha");
ThreadGroup beta = new ThreadGroup("beta");
ThreadGroup theta = new ThreadGroup(alpha, "theta");
```

The `alpha` and `beta` groups belong to the default—or main—thread group. However, the constructor for the `theta` thread group indicates that its parent thread group is `alpha`. Thus, we have the thread-group hierarchy depicted in Figure 4.12.

Notice that all thread groups have a parent, with the obvious exception of the system thread group, whose parent is `null`.

Writing a program that lists all threads in the Java virtual machine will involve a careful reading of the Java API—in particular, the `java.lang.ThreadGroup` and `java.lang.Thread` classes. A strategy for constructing this program is to first identify all thread groups in the JVM and then identify all threads within each group. To determine all thread groups, first obtain the `ThreadGroup` of the current thread, and then ascend the thread-group tree hierarchy to its root. Next, get all thread groups below the root thread group; you should find the overloaded `enumerate()` method in the `ThreadGroup` class especially helpful. Next, identify the threads belonging to each group. Again, the `enumerate()` method should prove helpful. One thing to be careful of when using the `enumerate()` method is that it expects an array as a parameter. This means that you will need to determine the size of the array before calling `enumerate()`. Additional methods in the `ThreadGroup` API should be useful for determining how to size arrays. Have your output list each thread group and all threads within each group. For example, based on Figure 4.12, your output would appear as follows:

- `system`: all threads in the system group
- `main`: all threads in the main group
- `alpha`: all threads in the alpha group
- `beta`: all threads in the beta group
- `theta`: all threads in the theta group

When outputting each thread, list the following fields:

a. The thread name

b. The thread identifier

c. The state of the thread

d. Whether or not the thread is a daemon

In the source code download for this chapter on WileyPLUS, we provide an example program (`CreateThreadGroups.java`) that creates the `alpha`, `beta`, and `theta` thread groups as well as several threads within each group. To use this program, enter the statement

```
new CreateThreadGroups();
```

at the beginning of your thread-listing program.

4.8 Modify the preceding problem so that the output appears in a tabular format using a graphical interface. Have the left-most column represent the name of the thread group and successive columns represent the four fields of each thread. In addition, whereas the program in the preceding problem lists all threads in the JVM only once, allow this program to periodically refresh the listing of threads and thread groups by specifying a refresh parameter on the command line when invoking the program. Represent this parameter in milliseconds. For example, if your program is named `ThreadLister`, to invoke the program so that it refreshes the list ten times per second, enter the following:

```
java ThreadLister 100
```

Programming Projects

The projects below deal with two distinct topics—naming services and matrix muliplication.

Project 1: Naming Service Project

A naming service such as DNS (domain name system) can be used to resolve IP names to IP addresses. For example, when someone accesses the host `www.westminstercollege.edu`, a naming service is used to determine the IP address that is mapped to the IP name `www.westminstercollege.edu`. This assignment consists of writing a multithreaded naming service in Java using sockets (see Section 3.4).

The `java.net` API provides the following mechanism for resolving IP names:

```
InetAddress hostAddress =
    InetAddress.getByName("www.westminstercollege.edu");
String IPaddress = hostAddress.getHostAddress();
```

where `getByName()` throws an `UnknownHostException` if it is unable to resolve the host name.

The Server

The server will listen to port 6052 waiting for client connections. When a client connection is made, the server will service the connection in a separate thread and will resume listening for additional client connections. Once a client makes a connection to the server, the client will write the IP name it wishes the server to resolve—such as `www.westminstercollege.edu`—to the socket. The server thread will read this IP name from the socket and either resolve its IP address or, if it cannot locate the host address, catch an `UnknownHostException`. The server will write the IP address back to the client or, in the case of an `UnknownHostException`, will write the message "`Unable to resolve host <host name>`." Once the server has written to the client, it will close its socket connection.

The Client

Initially, write just the server application and connect to it via telnet. For example, assuming the server is running on the local host, a telnet session will appear as follows. (Client responses appear in blue.)

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
www.westminstercollege.edu
146.86.1.17
Connection closed by foreign host.
```

By initially having telnet act as a client, you can more easily debug any problems you may have with your server. Once you are convinced your server is working properly, you can write a client application. The client will be passed the IP name that is to be resolved as a parameter. The client will open a socket connection to the server and then write the IP name that is to be resolved. It will then read the response sent back by the server. As an example, if the client is named `NSClient`, it is invoked as follows:

```
java NSClient www.westminstercollege.edu
```

The server will respond with the corresponding IP address or "unknown host" message. Once the client has output the IP address, it will close its socket connection.

Project 2: Matrix Multiplication Project

Given two matrices, A and B, where matrix A contains M rows and K columns and matrix B contains K rows and N columns, the **matrix product** of A and B is matrix C, where C contains M rows and N columns. The entry in matrix C for row i, column j ($C_{i,j}$) is the sum of the products of the elements for row i in matrix A and column j in matrix B. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

   public WorkerThread(int row, int col, int[][] A,
    int[][] B, int[][] C) {
     this.row = row;
     this.col = col;
     this.A = A;
     this.B = B;
     this.C = C;
   }
   public void run() {
     /* calculate the matrix product in C[row] [col] */
   }
}
```

Figure 4.13 Worker thread in Java.

For example, if A is a 3-by-2 matrix and B is a 2-by-3 matrix, element $C_{3,1}$ is the sum of $A_{3,1} \times B_{1,1}$ and $A_{3,2} \times B_{2,1}$.

For this project, you need to calculate each element $C_{i,j}$ in a separate *worker* thread. This will involve creating $M \times N$ worker threads.

Passing Parameters to Each Thread

The parent thread will create $M \times N$ worker threads, passing each worker the values of row i and column j that it is to use in calculating the matrix product. This requires passing two parameters to each thread.

One approach is for the main thread to create and initialize the matrices A, B, and C. This main thread will then create the worker threads, passing the three matrices—along with row i and column j—to the constructor for each worker. Thus, the outline of a worker thread appears in Figure 4.13.

Waiting for Threads to Complete

```
#define NUM_THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.14 Pthread code for joining ten threads.

```
final static int NUM_THREADS = 10;

/* an array of threads to be joined upon */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
   try {
      workers[i].join();
    } catch (InterruptedException ie) { }
}
```

Figure 4.15 Java code for joining ten threads.

Once all worker threads have completed, the main thread will output the product contained in matrix C. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish. Section Section 4.4 describes how to wait for a child thread to complete using Java. Java uses the `join()` method. Completing this exercise will require waiting for multiple threads.

A simple strategy for waiting on several threads using the Java's `join()` is to enclose the join operation within a simple `for` loop. For example, you could join on ten threads in Java in Figure 4.15.