Chapter 10: Concurrency in Java

Java provides concurrency through threads. A thread is a flow of control that is implemented in Java by the Thread class. This chapter puts the concurrency features of Java threads into the framework of concurrency introduced in Chapter 9.

10.1 Concurrent Units

Java follows the class model for the definition of concurrent units. In particular, a concurrent unit is defined as an object of the Thread class or of some class derived from the Thread class.

Every Thread object may have a run() method defined for its class. That run() method is the concurrent action associated with that thread. There are two ways of doing this. First, a thread may have its run() method attached to it by a constructor that has as its parameter an object of a class that implements the Runnable interface. The Runnable interface is an interface that requires a run() method, and the run() method for the constructed thread will be the run() method of the Runnable object that is attached through the constructor.

An example of such a Java implementation is found in Figure 10.1. The class Hello is created with a Runnable interface. It includes instance variables name and count, which are given values by its constructor. The run() method prints the hello message repeatedly for a total of count times. The main program in HelloTest constructs two Thread objects that use the Hello run() method. When the start() message is sent to a Thread object, the Thread object is initialized and then its run() method is called implicitly. The method start() is a member of the Thread class. The HelloTest main method in Figure 10.1 results in the threads one and two running concurrently. After starting threads one and two, main then completes its execution, terminating when its two running threads terminate their execution.

A second approach to defining threads in Java is to define the run() method by deriving a class from class Thread and including a definition for the run() method in that class definition. This approach is illustrated in Figure 10.2. This example duplicates the functionality of Figure 10.1, but creates a new class derived from Thread, HelloThread, with the run() method defined within that class. Note that this approach moves the initialization of the instance variables from the constructor of the Runnable to the constructor of the Thread.

10.2 Invocation of Concurrent Units

The preceding section shows that a Java Thread is invoked by calling its run() method indirectly by sending it a start() message. In this section, we look at this invocation process in more detail.

When the start() message is sent to a Thread object, the object initiates a concurrent execution of its run() method. The thread of execution that sends the start() message then continues its execution. The newly started thread will continue execution until its run() method completes or until another message to the Thread object intervenes. There are two ways this intervention can occur.

A Thread object can be halted by a stop() message. Upon receipt of this message, the thread is irrevocably halted. It is also possible to halt a thread temporarily by a suspend() message. A suspended thread can be continued by a resume() message. Figure 10.3 shows examples of stop(), suspend(), and resume() message calls. Threads one and two begin concur-
rent execution, after which two is temporarily suspended. Thread one is then halted permanently by the stop() message. Following this, two resumes it execution to completion.

Another message that controls execution of a thread is sleep(). The sleep() message has a single long parameter and causes the receiving thread to suspend for the number of milliseconds specified by the long parameter. Any Java thread of execution is suspended for m milliseconds when it executes the statement

Thread.sleep(m);

This is a call to the static method sleep() of class Thread.

Finally, a thread can suspend itself until another thread is completed. This is done by the message

aThread.join();

This suspends the thread that executes this call until aThread is completed, either through normal termination or through a stop() call.

Figure 10.4 gives examples of sleep() and join() calls. The method join() with a single long parameter m will cause the executing thread to suspend until aThread completes or until it has waited m milliseconds, whichever occurs first.

10.3 Data Sharing

Java threads support data sharing in two ways. First, data can be shared by passing the same data object into a number of threads as a parameter to its constructor. This permits each thread to operate on that object, therefore sharing the data. An example of this type of data sharing is given in Figure 10.5. Here, each thread of class PutTake can put or take data from a common object of the class Vector. Each thread decides on its action by generating a random integer. If the generated integer is even, the thread puts an item, and if it is odd and the Vector is non-empty, the thread takes an item from the Vector. Otherwise, the thread sleeps for 100 milliseconds.

The second approach is to pass a second Java thread to a given thread via its constructor parameter, thus giving the constructed thread access to the data space of the passed thread. Figure 10.6 shows a very simple example of this kind of data sharing. Here two Thread classes, First and Second, are defined so that they share a single int data value. The shared data is a member of an instance of Second that is passed through the First constructor as a parameter, giving each First instance a partner instance of Second with which it shares data.

10.4 Synchronization

Synchronization in Java occurs through the presence of a synchronization lock that is associated with each Java object. The lock for an object can be requested by any thread that has access to that object. If the lock is already in the possession of another thread, the requesting thread waits until the lock is made available, at which time it competes for possession with all other threads waiting on the lock. If the lock is available when requested, the requesting thread takes possession of the lock and holds it until the lock is released.

The synchronization lock for an object may be requested in either of two ways. First, it may be requested for the scope of an executable block. The syntax for such a request is

synchronized (anObject) {
    /* block where lock for anObject is held*/
} // Lock for anObject is released upon exit from the block.
The synchronization lock for anObject is first requested when the synchronized clause is encountered. If the lock is available, the thread proceeds to execute the block. Upon completion of the block, the thread relinquishes the lock. If the lock is not available when the synchronized clause is reached, the thread waits until it can obtain the lock before proceeding with the block.

Figure 10.7 is a synchronized version of the PutTake class in Figure 10.5. This implementation requires that a thread acquire the synchronization lock for Vector v before it performs an operation on v. This prevents the occurrence of a “race condition” where two threads perform operations on v simultaneously giving unreliable results.

The second way a synchronization lock can be requested in Java is by a thread calling a synchronized method. A method is tagged as synchronized by the keyword synchronized appearing in its definition. A synchronized method, when called, requests the lock for the receiving object. If it is available, that lock then belongs to the calling thread for the duration of the execution of the method. When the synchronized method terminates, the lock for the receiving object is made available again. The PutTake version using method synchronization is given in Figure 10.8. Here a new class, DataShare, is provided and each method call on an object of this class is synchronized, resulting in an implicit request for that object’s synchronization lock.

10.5 Interprocess Communication

Java implements interprocess communication through the wait() and notify() methods. The implementation of these methods is integrally related to the synchronization lock and these methods can be used to implement both the mail and phone models of interprocess communication.

In order to illustrate the rather complex semantics of wait() and notify() in Java, we consider a simple example involving two threads, t1 and t2, and an object x. A summary of this simple interaction is shown in Figure 10.9. Here we assume that thread t1 has somehow obtained the lock for x. At some point, while holding that lock, t1 executes

\[ x\text{.wait();} \]

The execution of this method causes 1) t1 to relinquish the lock on x, and 2) t1 to enter a sleep state where it will remain until another thread executes x.notify().

We assume here that the notifying thread is t2. Therefore, at some later time t2 obtains the lock on x, since it cannot send x a notify() message otherwise. When t2 executes the call x.notify(), one of the threads that is in a sleep state due to a wait() on object x is awakened. For our example, we assume that the awakened thread is t1, though this may not be the case if more than one thread is waiting on x. When t1 is awakened, it immediately and implicitly submits a request for the lock of x, but it does not immediately receive that lock since that lock is still held by t2. When t2 later relinquishes the lock on x, t1 then competes for that lock with all the other threads that have active requests. Eventually, t1 will receive that lock. At that point, t1 resumes execution at the point immediately after the x.wait() call.

One other feature of the wait() method is that a version is provided that has a single long parameter. This parameter specifies a number of milliseconds after which the waiting thread will awaken in the event that it is not awakened before that time by a notify() call.

Also, there is a notifyAll() method that, when called, awakens all threads currently sleeping as a result of a wait() call on the receiver. Once again, none of these awakened threads will
obtain the lock immediately, but they will all compete for it when the thread calling notify-All() relinquishes the lock.

Figure 10.10 illustrates a Java simulation of the mail model of interprocess communication. Two Thread classes are shown in this figure, Sender, which sends messages to a MessageBuffer, and Receiver, which retrieves the messages. The mail model requires no synchronization between the Sender and Receiver so the Sender proceeds to the completion of its send() message as soon as it has placed the message in the MessageBuffer.

Figure 10.11 illustrates the phone model of interprocess communication. In this case, the Sender thread obtains the lock for test, the String that it is sending, before test is sent. After sending test, Sender sleeps until a notify() for test is executed. The Receiver thread, after obtaining a String from the MessageBuffer, sends notify() to the received message String, reawakening its sending thread. Figure 10.11 illustrates the phone model that does not allow the sending thread to proceed until the receiving thread has finished processing the message. To implement the other model, where the receiver acknowledges as soon as receiving the message and permits the sending thread to continue executing as soon as that acknowledgement occurs, the run() method of Receiver would be as in Figure 10.12.

In the case of the example of Figures 10.10 through 10.12, the MessageBuffer class, though not shown here, would have all of its methods synchronized. This would guarantee that whenever a thread is performing an operation on a MessageBuffer, that no other thread will try to operate on the same MessageBuffer while the first operation is in progress. Note that this synchronization will result in the threads obtaining the lock on their MessageBuffers while the other synchronization we have discussed dealt with locks on the String making up the message itself.

Figure 10.1

```
public class Hello implements Runnable {
    protected String name;
    protected long count;

    public Hello(String name, long count) {
        this.name = name;
        this.count = count;
    }

    public void run() {
        for (int i=0; i<count; i++)
            System.out.println("Hello from "+name+i);
    }
}

public class HelloTest {
    public static void main(String args[]) {
        Thread one = new Thread(new Hello("one",1000));
        Thread two = new Thread(new Hello("two",5));
        one.start();
        two.start();
        System.out.println("Hello from Main");
    }
}
```
Figure 10.2

public class HelloThread extends Thread {
    protected String name;
    protected long count;

    public HelloThread(String name, long count) {
        this.name = name;
        this.count = count;
    }

    public void run() {
        for (int i=0; i<count; i++)
            System.out.println("Hello from " + name);
    }
}

public class HelloThreadTest {
    public static void main(String args[]) {
        HelloThread one = new HelloThread("one",1000);
        HelloThread two = new HelloThread("two",5);
        one.start();
        two.start();
        System.out.println("Hello from Main");
    }
}

Figure 10.3

public class HelloTest3 {
    public static void main(String args[]) {
        Thread one = new Thread(new Hello("one",1000));
        Thread two = new Thread(new Hello("two",1000));
        one.start();
        two.start();
        two.suspend();
        System.out.println("two has been suspended");
        one.stop();
        System.out.println("one has been stopped");
        two.resume();
        System.out.println("two has been resumed");
    }
}

Figure 10.4

public class DrowsyThread extends Thread {
    protected String name;
    protected long count;
    protected long nap;

    public DrowsyThread(String name, long count, long nap) {

this.name = name;
this.count = count;
this.nap = nap;
}

public void run() {
for (int i=0; i<count; i++) {
try {
    Thread.sleep(nap);
} catch (Exception ex) {} 
    System.out.println(name + ":" + i);
}
}

public class DrowsyTest {
    public static void main(String args[]) {
        DrowsyThread one = new DrowsyThread("one",5,200);
        DrowsyThread two = new DrowsyThread("two",3,1000);
        one.start();
        two.start();
        try {
            one.join();
        } catch (Exception ex) {}
        System.out.println("end of main");
    }
}

Figure 10.5
import java.util.*;
import java.util.Vector;

public class PutTake extends Thread {
    protected Vector v;
    protected String name;
    protected static Random rng = new Random();

    public PutTake(String name, Vector v) {
        this.name = name;
        this.v = v;
    }

    public void run() {
    for (int i=0; i<100; i++) {
        if (rng.nextInt()%2 == 0) {
            v.addElement(name + i);
            System.out.println("put " + name + i);
        }
        else if (v.size()>0) {
            System.out.println(name + ":take " + (String)v.firstElement());
            v.removeElementAt(0);
        } else try {sleep(100);} catch (Exception ex) {}
    }
}
import java.util.Vector;

public class PutTakeTest {
    public static void main(String args[]) {
        Vector v = new Vector();
        PutTake one = new PutTake("one", v);
        PutTake two = new PutTake("two", v);
        PutTake three = new PutTake("three", v);
        one.start();
        two.start();
        three.start();
    }
}

Figure 10.6

public class First extends Thread {
    Second partner;

    public First(Second partner) {
        this.partner = partner;
    }

    public void setValue(int value) {
        partner.setValue(value);
    }

    public int getValue() {
        return partner.getValue();
    }
}

public class Second extends Thread {
    int value;

    public void setValue(int i) {
        value = i;
    }

    public int getValue() {
        return value;
    }
}

Figure 10.7

import java.util.*;
import java.util.Vector;

public class PutTake extends Thread {
    protected Vector v = new Vector();
    protected String name;
    protected static Random rng = new Random();

    public PutTake(String name, Vector v) {
        this.name = name;
        this.v = v;
    }

    public void run() {
        for (int i=0; i<100; i++) {
            if (rng.nextInt()%2 == 0) {
                synchronized(v) {
                    v.addElement(name + i);
                }
                System.out.println("put " + name + i);
            } else if (v.size()>0) {
                synchronized(v) {
                    System.out.println(name + ":take " + (String)v.firstElement());
                    v.removeElementAt(0);
                }
            } else try {sleep(100);} catch (Exception ex) {}
        }
    }
}

import java.util.Vector;

public class PutTakeTest {
    public static void main(String args[]) {
        Vector v = new Vector();
        PutTake one = new PutTake("one",v);
        PutTake two = new PutTake("two",v);
        PutTake three = new PutTake("three",v);
        one.start();
        two.start();
        three.start();
    }
}

Figure 10.8
protected DataShare ds;
protected String name;
protected static Random rng = new Random();

public PutTake(String name, DataShare ds) {
    this.name = name;
    this.ds = ds;
}

public void run() {
    for (int i=0; i<100; i++) {
        if (rng.nextInt()%2 == 0) {
            ds.addElement(name + i);
            System.out.println("put " + name + i);
        } else if (ds.size()>0) {
            System.out.println(name + ":take " + (String)ds.firstElement());
            ds.removeElementAt(0);
        } else try {sleep(100);} catch (Exception ex) {
        }
    }
}

import java.util.*;
import java.util.Vector;

class DataShare {
    protected Vector v = new Vector();
    public synchronized void addElement(Object o) {
        v.addElement(o);
    }
    public synchronized Object firstElement() {
        return v.firstElement();
    }
    public synchronized void removeElementAt(int i) {
        v.removeElementAt(i);
    }
    public synchronized int size() {
        return v.size();
    }
}

class PutTakeTest {
    public static void main(String args[]) {
        DataShare ds = new DataShare();
        PutTake one = new PutTake("one",ds);
        PutTake two = new PutTake("two",ds);
        PutTake three = new PutTake("three",ds);
one.start();
two.start();
three.start();
}
}

Figure 10.9 Illustration of wait() and notify() calls

<table>
<thead>
<tr>
<th>x lock belongs to</th>
<th>State of t1</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>executing</td>
</tr>
<tr>
<td>t1 obtains lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t1 executing</td>
</tr>
<tr>
<td>t1 executes x.wait()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>none sleeping</td>
</tr>
<tr>
<td>t2 obtains lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t2 sleeping</td>
</tr>
<tr>
<td>t2 executes x.notify()</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t2 waiting for lock</td>
</tr>
<tr>
<td>t2 relinquishes lock for x</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>none waiting for lock</td>
</tr>
<tr>
<td>t1 obtains lock for x and resumes execution immediately after wait() call</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>t1 executing</td>
</tr>
</tbody>
</table>

Figure 10.10 Sender/Receiver Threads using Mail Model Interprocess Communication

public class Sender extends Thread {
    protected MessageBuffer mb;

    public Sender(MessageBuffer mb) {
        this.mb = mb;
        mb.register(this);
    }

    public void send(String test) {
        mb.setValue(test);
    }
}

public void terminate() {
    mb.unregister(this);
}
public class Receiver extends Thread {

    MessageBuffer mb;

    public Receiver(MessageBuffer mb) {
        this.mb = mb;
    }

    public void run() {
        String message;
        while (mb.has_senders() || mb.has_value()) {
            if (mb.has_value()) {
                message = mb.getValue();
                process(message);
            }
        }
    }

    public void process(String m) {
        // code used by Receiver to process the message
    }
}

Figure 10.11 Phone Model with Acknowledgement after processing

public class Sender extends Thread {
    protected MessageBuffer mb;

    public Sender(MessageBuffer mb) {
        this.mb = mb;
        mb.register(this);
    }

    public synchronized void send(String test) {
        mb.setValue(test);
        try {test.wait();} catch (Exception ex) {} }
    }

    public void terminate() {
        mb.unregister(this);
    }
}

public class Receiver extends Thread {

    MessageBuffer mb;
public Receiver(MessageBuffer mb) {
    this.mb = mb;
}

public void run() {
    String message;
    while (mb.has_senders() || mb.has_value()) {
        if (mb.has_value()) {
            message = mb.getValue();
            process(message);
            synchronized(message) {
                message.notify();
            }
        }
    }
}

public void process(String m) {
    try {Thread.sleep(150);} catch (Exception ex){}
    System.out.println(m + " received");
}

Figure 10.12 Phone Model with acknowledgement before processing