Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Lecture 5 – Concurrency

Learning Objectives

After successfully completing the module, students will be able to do the following:

1. Describe how concurrency, parallelism and multi-threading work in Java programming.
2. Create and use threads.
3. Evaluate thread safety and race conditions.
4. Use synchronized statements and synchronized methods.
5. Synchronize threads using locks.
6. Detect deadlock in programming.
7. Use Thread pool, Executors, ExecutorService.

Module 5 Study Guide and Deliverables

Topics: Lecture 5: Concurrency

Readings: • Module 5 online content

- Deitel & Deitel, Chapter 23
- Instructor's live class slides

Assignments: Assignment 4 due Tuesday, June 14 at 6:00 AM ET

Live Classrooms: • Tuesday, June 7, from 6:00 PM to 8:00 PM ET
                          • Live Office: Schedule with facilitators as long as there are questions

Module Welcome and Introduction

met_cs622_18_su1_ebraude_mod5 video cannot be displayed here

Concurrency Overview

Concurrency, or parallel execution of computing tasks, is almost indispensable in modern programming. Typical examples of concurrency include:

- Web sites (e.g., Google) handling multiple (often numerous) simultaneous users.
- Mobile apps doing (some of) their processing on servers.
- Graphical user interfaces requiring background work to continue.

Concurrent components make a program run faster. True concurrency is possible when multiple processing units are available, with each processing unit running a piece of the total task. Even when only one processor is available, an illusion of concurrency can be created by time-sharing (interleaving) the different pieces on the single processor. Note that:

- Concurrency (parallelism) can be simulated by time-sharing when there are more tasks than physical processors (the tasks take turns to run on processors).
- The schedule for time-sharing is often unpredictable and nondeterministic.

Thread

A thread in a computer program is a flow of execution from a start state to an end state. More specifically, a thread is a "locus of control" inside a running program, indicated by the position in the code and the run-time stack, representing the current point in the computation. We informally say that a thread performs a task; thus multi-threading (or multi-tasking) is generally taken to mean the same thing as concurrency.

Process

A process is an instance of a running program. A process has at least one thread and can create (spawn) multiple, other threads. A Java program starts with one thread, the main thread, corresponding to the main() method. The main thread can create additional threads, as we will now see with code examples.

Creating and Handling Threads

Each thread is created as an instance of the class "Thread". We will discuss two ways for using Thread objects in concurrent programming:

- Create an instance of a Thread class each time the application requires the initiation of a (new) asynchronous task.
- Let thread management be handled by an "Executor" (this allows us to abstract away explicit creation of Thread objects).

In the following examples, we focus on the first approach. Executors are discussed later in the module.

The following program shows the creation and execution of two threads (corresponding to the two Thread objects t1 and t2) in the same program. Two runnable objects, f and s, are first created, and each Thread constructor is called with a runnable object as the argument. (We often refer to runnable objects like f and s as tasks.) Note that:

- Thread is a class
- Runnable is an interface
- The Thread class already implements Runnable
- The argument passed to the Thread constructor must be a Runnable
- The threads spawned by a process run asynchronously.

```
public class TestThread
{
    public static void main(String[] args) throws InterruptedException
    {
        First f = new First();
        Thread t1 = new Thread(f);
        t1.start();

        Second s = new Second();
        Thread t2 = new Thread(s);
        t2.start();
    }
}
```

```
class First implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 100; i++)
                System.out.println("i = " + i);
    }
}


class Second implements Runnable
{
    public void run()
    {
        for (int j = 0; j < 100; j++)
            System.out.println("And j = " + j);
    }
}
```

The start() method of the Thread class automatically invokes the run() method of the corresponding Runnable object (e.g., t1.start() automatically calls First's run()). The thread terminates when the run() method completes.

The output of the above program shows (apparently arbitrarily) interspersed prints of the i and j values. If you do not see the interspersing when you run your program, increase the loop counters to, say, 500. Another way to make the interspersing appear (even with small values of the loop counter) is to use the sleep() method (discussed later in this module) of the Thread class inside run().

In the above program, the main() method could also have been written as follows (without explicit creation of named objects):

```
 public static void main(String[] args) throws InterruptedException
{
        (new Thread(new First())).start();
        (new Thread(new Second())).start();
}
```
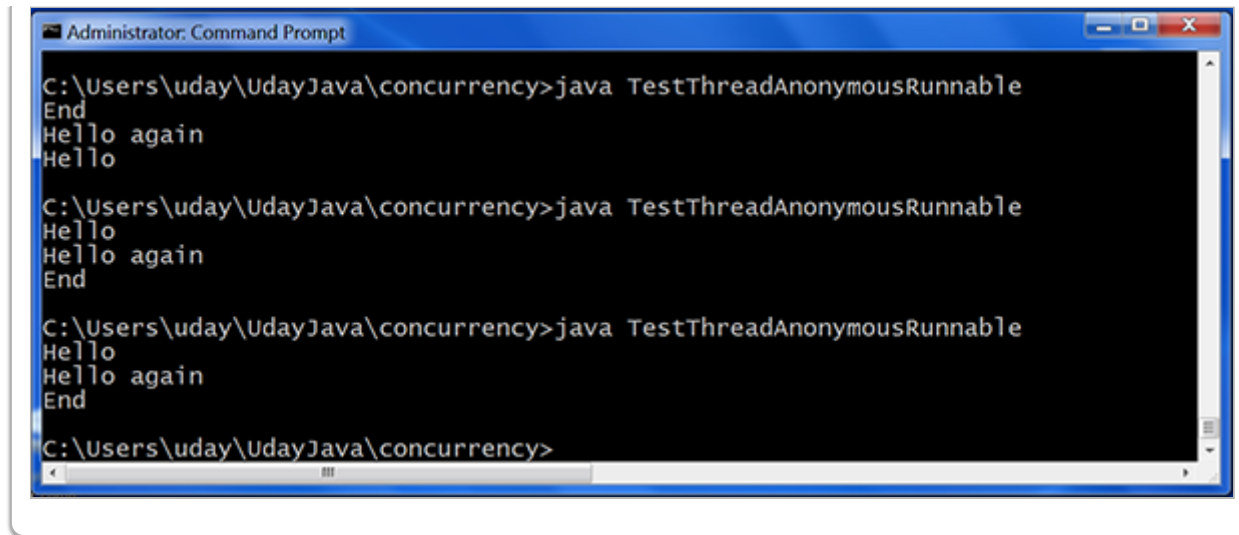
Anonymous Runnable Object

In creating threads, one can use anonymous runnable objects, as in the following example:

```
//Illustrating anonymous runnable object
public class TestThreadAnonymousRunnable
{
    public static void main(String[] args)
                    throws InterruptedException
    {
        Thread t1 = new Thread(new Runnable()    // Anonymous Runnable object
                            {
                                public void run()
                                {
                                    System.out.println("Hello");
                                }
                            }
                        );
        Thread t2 = new Thread() {
                                public void run()
                                {
                                    System.out.println("Hello again");
                                }
                        };
        t1.start();
        t2.start();
        System.out.println("End");
    }
}
```

The output of the above program is the three strings ("Hello", "Hello again", and "End") appearing in *any* order:

Output of the program - anonymous runnable object

Main Thread

The above program (class TestThreadAnonymousRunnable) makes it easy to observe the presence of the three concurrent threads in the program – in addition to t1 and t2, we have the *main thread* that is created by the Java Virtual Machine (JVM). The code in the main() method executes in the main thread. When main() ends (and "End" is printed), the program itself may continue running because there might still be unfinished tasks. That is, the program terminates only when all the threads (including the main thread) are finished.

Difference between run() and start()

The method start() of the Thread class starts a new thread, but run() (of the Runnable interface, which is overridden in the user's Runnable class) continues in the existing thread.

The following program shows that only one thread runs in this program, by printing the lone thread's thread ID (1) and thread name (main).

```
public class ThreadStartAndRun
{
    public static void main(String[] args) throws InterruptedException
    {
```

```
                    First  f1 = new First(100);
                    First  f2 = new First(200);

                    Thread t1 = new Thread(f1);
                    Thread t2 = new Thread(f2);

                    t1.run();
                    t2.run();
                    f1.run();
                    f2.run();
              }
        }


      class First implements Runnable
      {
           private int i;

           public First(int i)
           {
                    this.i = i;
           }

           public void run()
           {
                    System.out.println("i = " + i +
                                        "Thread ID = " + Thread.currentThread().getId() +
                                        "Thread Name = " + Thread.currentThread().getName());
           }
      }
```
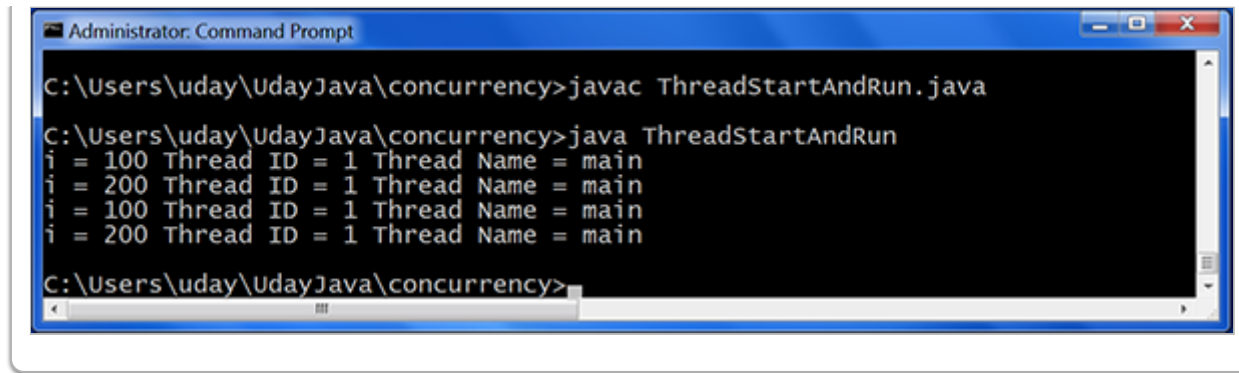
Output of the program - only one thread runs in the program

Replacing the run() statements in the above program with two start() calls, we have the following code, which prints the ID and name of two threads created in main(). (How many threads does this program execute?)

```
public class Thread4
{
    public static void main(String[] args) throws InterruptedException
    {
        First  f1 = new First(100);
        First  f2 = new First(200);

        Thread t1 = new Thread(f1);
        Thread t2 = new Thread(f2);

        t1.start(); t2.start();       // Two separate threads
    }
}
class First implements Runnable
{
    private int i;

    public First(int i) {this.i = i;}

    public void run()
    {
```

```java
            System.out.println("i = " + i +
                                " Thread ID = " + Thread.currentThread().getId() +
                                " Thread Name = " + Thread.currentThread().getName());
        }
    }
```

Output of the program - concurrent threads

```
Administrator: Command Prompt

C:\Users\uday\UdayJava\concurrency>java Thread4
i = 200 Thread ID = 11 Thread Name = Thread-1
i = 100 Thread ID = 10 Thread Name = Thread-0

C:\Users\uday\UdayJava\concurrency>java Thread4
i = 100 Thread ID = 10 Thread Name = Thread-0
i = 200 Thread ID = 11 Thread Name = Thread-1

C:\Users\uday\UdayJava\concurrency>
```

Let us reiterate the fact that the above program runs three, not two, concurrent threads. We can easily show that by modifying the program to print the ID and name of the main thread, in addition to the two threads corresponding to t1 and t2. The result of the modification is given below, along with two sample runs of the program showing that the three threads can finish in different orders.

```java
public class Thread4a
{
    public static void main(String[] args) throws InterruptedException
    {
        First  f1 = new First(100);
        First  f2 = new First(200);

        Thread t1 = new Thread(f1);
        Thread t2 = new Thread(f2);

        t1.start(); t2.start();        // Two separate threads
```
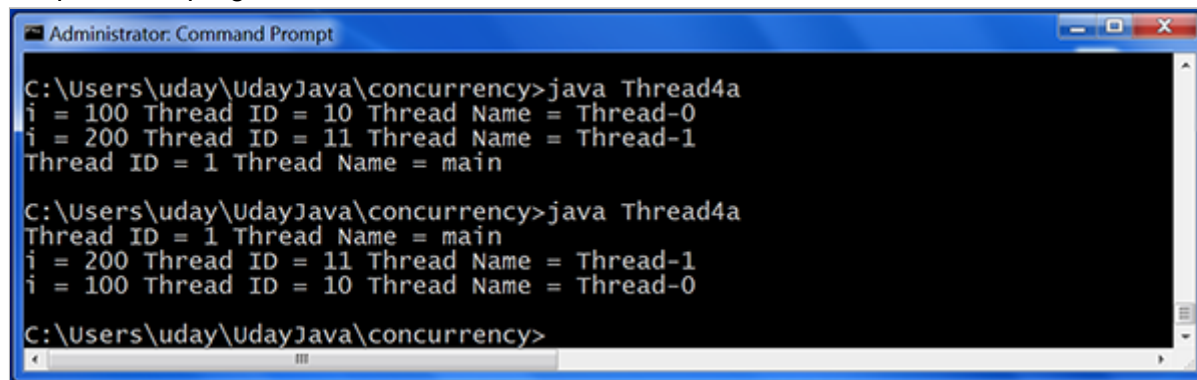
```java
            System.out.println("Thread ID = " + Thread.currentThread().getId() +
                               " Thread Name = " + Thread.currentThread().getName());
        }
    }


    class First implements Runnable
    {
        private int i;

        public First(int i) {this.i = i;}

        public void run()
        {
          System.out.println("i = " + i +
                            " Thread ID = " + Thread.currentThread().getId() +
                            " Thread Name = " + Thread.currentThread().getName());
        }
    }
```

Output from two sample runs follows:

Output of the program - concurrent threads

Alternative Method of Creating a Thread

It is possible to define a subclass of the Thread class and write the implementation of the new subclass's run() method, as follows:

```
public class MyThread extends Thread
{
   public void run()
   {
           // code
   }
}
```

And then create and start a thread as follows:

```
public static void main(String args[])
{
   (New MyThread()).start();
}
```

This approach is not recommended and will not be discussed further in this module.

Thread's sleep() Method

Of the methods in the Thread class, the static method sleep(), often comes in handy. It is used for pausing execution temporarily. Thread.sleep(arg) causes the current thread to suspend execution for a specified amount of time (the amount depends on the argument). See the example program below.

```
// Illustrating sleep()
public class TestThread6
{
   public static void main(String[] args)
   {
      System.out.printf("Main thread with thread id %3d begins%n",
                        Thread.currentThread().getId());

      (new Thread(new First())).start();
      (new Thread(new First())).start();
```

```java
            System.out.printf("Main thread with thread id %3d finishes%n",
                             Thread.currentThread().getId());
        }
    }


class First implements Runnable
{
    public void run()
    {
        System.out.printf("Thread id %3d starting%n", Thread.currentThread().getId());

        int r = (int) (3000 * Math.random());
        try
        {
            Thread.sleep(r);
        }
        catch(InterruptedException ex)
        {
            ex.printStackTrace();
            Thread.currentThread().interrupt();
        }
        System.out.printf("Thread id %3d slept for %4d ms%n", Thread.currentThread().getId(), r);
    }
}
```
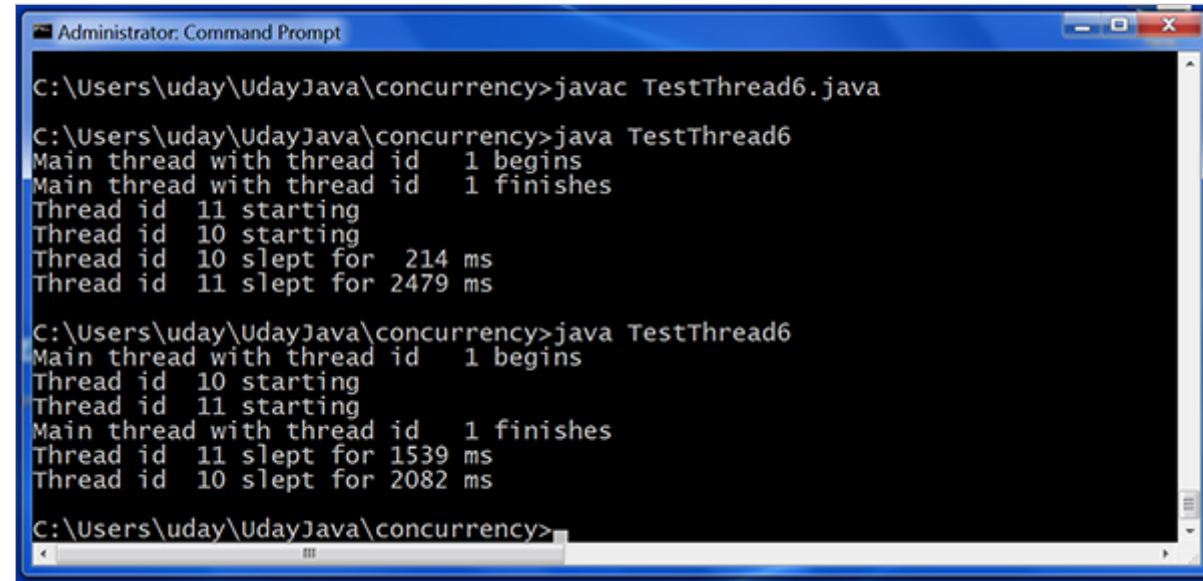
The above program causes each of the two concurrent threads to "sleep" for a random number of milliseconds between zero and 3000. Here are two important features of the sleep() method worth remembering:

- sleep() is a **static** method in Thread.
- The sleep() method throws a checked exception, namely InterruptedException. Because it is a checked exception, the method that invokes sleep()—that is, the run() method—must either catch the exception using try-catch blocks (and process it) or declare that it (the run() method) throws InterruptedException. Now, the method run() in the interface Runnable does NOT have a "throws" declaration (see the original Java documentation). Therefore the user program cannot make run() declare an interrupt, which means the interrupt must be caught in run() (and not re-thrown).

Output of the program - sleep() method


```
Administrator: Command Prompt

C:\Users\uday\UdayJava\concurrency>javac TestThread6.java

C:\Users\uday\UdayJava\concurrency>java TestThread6
Main thread with thread id    1 begins
Main thread with thread id    1 finishes
Thread id   11 starting
Thread id   10 starting
Thread id   10 slept for   214 ms
Thread id   11 slept for  2479 ms

C:\Users\uday\UdayJava\concurrency>java TestThread6
Main thread with thread id    1 begins
Thread id   10 starting
Thread id   11 starting
Main thread with thread id    1 finishes
Thread id   11 slept for  1539 ms
Thread id   10 slept for  2082 ms

C:\Users\uday\UdayJava\concurrency>
```

Thread's join() Method

The join() method allows one thread to wait for the completion of another. (Note that unlike sleep(), join() is not a static method; it is an instance method.)

The following code illustrates join().
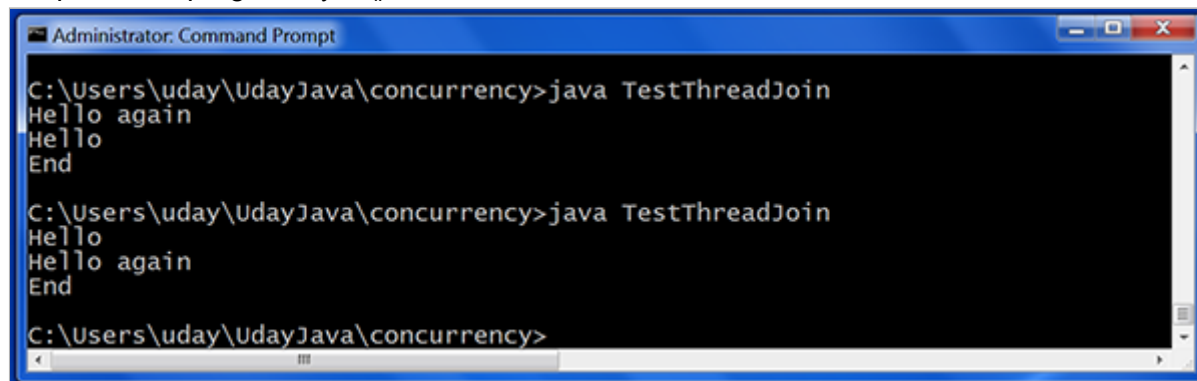
```
//Illustrating join()

public class TestThreadJoin
{
    public static void main(String[] args)
                throws InterruptedException
    {
        Thread t1 = new Thread() {
                public void run()
                {
                    System.out.println("Hello");
```

```
                                }
                        };

                Thread t2 = new Thread() {
                                public void run()
                                {
                                    System.out.println("Hello again");
                                }
                        };
                t1.start();
                t2.start();
                t1.join();
                t2.join();

                System.out.println("End");
            }
        }
```

The t1.join() statement causes the current thread, that is, the main thread (which obviously is not t1) to wait for t1 to complete. Similarly, t2.join() forces the current thread (main thread) to wait for t2's completion. Without the joins, the output could have the "End" appearing in any of the three possible positions: at the beginning, in between the two "Hello"s, or at the end. With the two joins, however, the "End" is guaranteed to be at the end. Two sample runs of the program are shown in the screenshot below.

Output of the program - join() method

```
Administrator: Command Prompt

C:\Users\uday\UdayJava\concurrency>java TestThreadJoin
Hello again
Hello
End

C:\Users\uday\UdayJava\concurrency>java TestThreadJoin
Hello
Hello again
End

C:\Users\uday\UdayJava\concurrency>
```

Thread Safety and Race Condition

A major concern in concurrent programming is thread safety, or how to maintain consistency of the data under simultaneous operations from multiple threads. Even apparently simple computation can result in bad data under multi-threading. Consider the example of updating a variable x:

```
x = x + 1;
```

When the above incrementing is done by multiple, concurrent threads, the final value of x after all the threads complete may not always be correct. This might at first sight seem impossible, because isn't x = x + 1 an *atomic* (indivisible) operation? The answer is no in general. (It may or may not be atomic; there is no way for us to know in advance.) It is quite likely that the above incrementing operation really consists of three lower-level components:

- Get (fetch) the current value of x into a temporary storage, temp;
- Increment this value by 1;
- Store the new value back to x;

If two threads T1 and T2 try to execute x = x + 1 at the same time, the exact sequence of interleaved low-level operations may look like this (assume the initial value of x is zero):

T1	T2	x
Get value		0
Increment temporary (temp is 1)	Get value(0)	0
Write back (x is now 1)	Increment temporary (temp is 1)	1
	Write back (x is now 1)	1

Thus the final value of x will be 1, not 2. The above scenario is just one possibility; other scenarios of possible interleaving exist. This is an example of a **race condition** (the correctness of the program depends on the relative timing of events in concurrent computation). We say that T1 and T2 in the above example are in a race. Changing the increment statement code x = x + 1; to

- x++;

- ++x;

- x += 1;

will not help us avoid the race condition. The Java compiler or the processor makes NO commitment about low-level operations.

Methods for ensuring data integrity (thread safety) can be divided into four broad categories:

- **Thread synchronization**: When the data must be shared between threads, keep two or more threads from accessing it at the same time.
- **Threadsafe datatype**: Use a datatype that does the synchronization automatically, like a BlockingQueue or a synchronized collection wrapper.
- **Immutability**: Make the shared data immutable, so the question of data corruption does not arise.
- **No sharing**: Don't share the data between threads. Ensure that only one thread has access to it.

This module focuses on the first method, thread synchronization.

Shared Static Data

The following is an example of unsafe multithreading where each of 5000 concurrent threads updates the same integer variable. The shared variable in this example is a static integer field, called count, in the class MyRunnable.

```
import java.util.*;

public class ThreadSafetyUnsafe1
{
    public static void main(String[] args) throws InterruptedException
    {
        ArrayList<Thread> threads = new ArrayList<Thread>();

        for (int i = 1; i <= 5000; i++)
                threads.add(new Thread(new MyRunnable()));
```

```java
            for (Thread t : threads)    t.start();
            for (Thread t : threads)    t.join();

            System.out.println("End: count = " + MyRunnable.getCount());
        }
    }


    class MyRunnable implements Runnable
    {
        private static int count = 0;

        public void run()
        {
            count++;
        }

        public static int getCount() { return count; }
    }
```
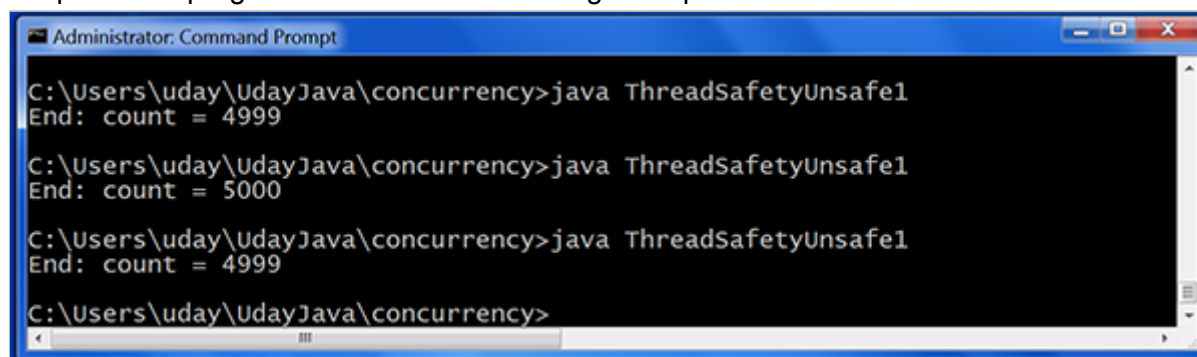
The final value of count could be less than 5000 because of interleaving of the threads (as shown in the screenshot below).

Output of the program - unsafe multithreading example 1

To make the problem appear really spectacular, one can deliberately break down the incrementing step by introducing a temporary variable and introducing delays by inserting sleep() inside run():

```java
import java.util.*;

public class ThreadSafetyUnsafe1DelayAdded
{
    public static void main(String[] args) throws InterruptedException
    {
      ArrayList<Thread> threads = new ArrayList<Thread>();

      for (int i = 1; i <= 5000; i++)
            threads.add(new Thread(new MyRunnable()));

      for (Thread t : threads)    t.start();
      for (Thread t : threads)    t.join();

      System.out.println("End: count = " + MyRunnable.getCount());
    }
}

class MyRunnable implements Runnable
{
    private static int count = 0;

    public void run()
    {
            int temporary = count+1;
            try
            {
                Thread.sleep(10);
            }
            catch(InterruptedException ex)
            {
                ex.printStackTrace();
                Thread.currentThread().interrupt();
            }
```
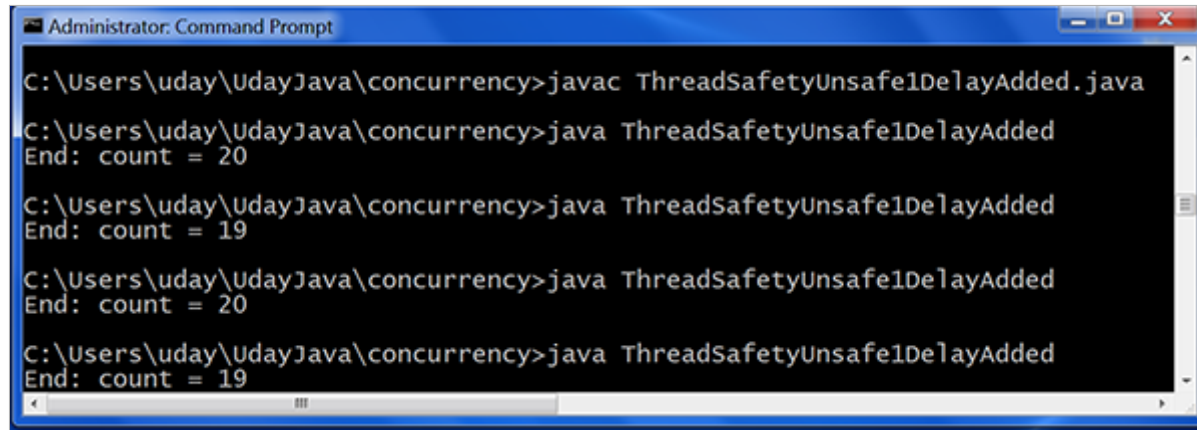
```
            count = temporary;
        }


    public static int getCount() { return count; }
}
```

The problem of unsafe concurrency now looks much worse than in the previous case:

Output of the program - unsafe multithreading example 2

The problem can be solved by using what is known as a *synchronized statement* (or *synchronized block*).

Synchronized Statement

The form of the synchronizing statement or synchronizing block (a block is a sequential list of statements) is as follows:

```
synchronized (obj)
{
    // One or more statements;
}
```

where obj is an object reference. A thread that wants to execute the above synchronized block first tries to acquire a lock on the named object, obj. If the attempt is successful, the thread enters the block and executes the statement(s). The thread releases the lock upon exit from the block. As long as one thread "locks" the object obj, no other thread can acquire a lock on the same object. This ensures mutual exclusion of the code in the block.

A lock is a mechanism (an abstraction) that allows at most one thread to "own" the lock at a time. As mentioned earlier, *acquire* and *release* are the two operations on locks. If a thread tries to acquire a lock that is currently owned by another thread, it (the first thread) must wait ("block") until the other thread releases the lock before it can acquire it.

The following program illustrates how we can use a synchronizing block to solve the thread safety problem we have been discussing:

```java
import java.util.*;

public class ThreadSafetyUnsafe1MadeSafe
{
    public static void main(String[] args) throws InterruptedException
    {
        ArrayList<Thread> threads = new ArrayList<Thread>();

        for (int i = 1; i <= 5000; i++)
            threads.add(new Thread(new MyRunnable()));

        for (Thread t : threads)    t.start();
            for (Thread t : threads)    t.join();

            System.out.println("End: count = " + MyRunnable.getCount());
        }
}

class MyRunnable implements Runnable
{
    private static int count = 0;
    private static Object lockobj = new Object();

    public void run()
```

```
        {
            synchronized (lockobj)
            {
                int temporary = count+1;
                try
                {
                    Thread.sleep(10);
                }
                catch(InterruptedException ex)
                {
                    ex.printStackTrace();
                    Thread.currentThread().interrupt();
                }
                count = temporary;
            }
        }

        public static int getCount() { return count; }
    }
```

Note that the lock in this example is obtained on an Object called lockobj, which must first be created as a private static field of class MyRunnable. Note, in particular, that a non-static Object won't do in this example, because we want all the threads to use a lock on the *same* object. Omitting the "static" before "Object lockobj" in class MyRunnable would create 5000 different locks, corresponding to as many different object instances of class MyRunnable (see the i-loop in main()). The above program is thread-safe and correctly produces a final count of 5000. (Would it be okay to declare lockobj as a *public* static field in MyRunnable?)

In the above example, synchronization was achieved with the help of an explicitly chosen object (lockobj) as the lock. A related concept, *synchronized method*, to be discussed shortly, uses an implicit lock object.

Monitor

Just as the synchronized statement protects (that is, ensures mutual exclusion of) one or more statements, all of the statements in a method body can be protected by using the object itself (the "this" reference) as a lock in a synchronized statement. Consider the class Counter:

```
public class Counter
{
    private int c = 0;
    public void increment() {c++;}
    public void decrement() {c--;}
    public int getvalue() {return c;}
}
```

Given the above class, we can protect (guard) the inside of the increment() method by using a synchronized statement where the lock is acquired on "this":

```
public void increment()
{
        synchronized(this)
        {
            c++;
        }
}
```

Applying the mechanism to each method in Counter, we have:

```
public class Counter
{
    private int c = 0;

  public void increment()
   {
        synchronized(this)
        {
            c++;
        }
   }
```

```
        public void decrement()
        {
               synchronized(this)
               {
                     c--;
               }
        }
        public int getvalue()
        {
               synchronized(this)
               {
                     return c;
               }
        }
     }
```

When all of the statements in a method are guarded, the "synchronized" keyword can be placed just before the method name, making the syntax less cluttered:

```
     public class Counter
     {
        private int c = 0;

        public synchronized void increment() {c++;}
        public synchronized void decrement() {c--;}
        public synchronized int getvalue() {return c;}
     }
```

The above pattern is called a "monitor". A class, whose methods are mutually exclusive, is a monitor.

Note that a method's constructor cannot be made synchronized (in fact it should be unnecessary by the very definition of construction – only one thread can construct an object).

Thread Safety - Shared Reference

Let us consider another example of unsafe concurrency and how to fix this problem. In the following program, the Counter object sharedcounter is being shared among the threads, but the threads are not properly synchronized, resulting in possible wrong values of sharedcounter's attribute c.

```java
import java.util.*;

public class ThreadSafetyUnsafe2A
{
     public static void main(String[] args) throws InterruptedException
     {
        Counter sharedcounter = new Counter();       // shared reference

        ArrayList<Thread> allthreads = new ArrayList<Thread>();

        for (int i = 1; i <= 10000; i++)
            allthreads.add(new Thread(new First(i, sharedcounter)));

        for (Thread t: allthreads)
            t.start();

        for (Thread t: allthreads)
            t.join();

        System.out.println("End: final counter value = " + sharedcounter.getvalue());
     }
}


class First implements Runnable
{
     private int i;
     private Counter ctr;

     public First(int i, Counter c)
```
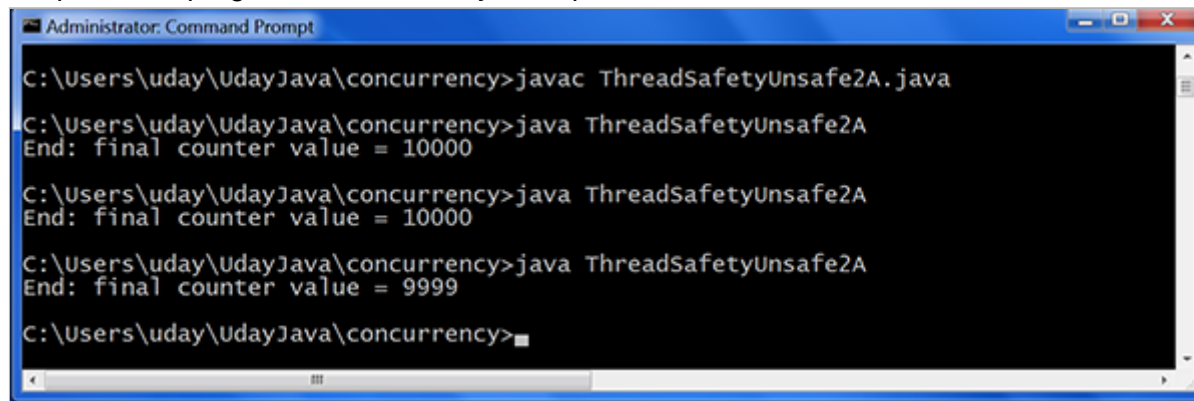
```
        {
            this.i = i;
            this.ctr = c;
        }

        public void run()
        {
            if (i > 0)
                ctr.increment();
        }
    }


    class Counter
    {
        private int c = 0;
        public void increment() {c++;}
        public int getvalue() {return c;}
    }
```

Output of the program - thread safety example 1

The solution is to make the method increment() synchronized (nothing else in the program need be changed; of course, please read the section on synchronized/unsynchronized mix later in this module):

```
class Counter
{
   private int c = 0;
   public synchronized void increment() {c++;}
   public int getvalue() {return c;}
}
```

Alternatively, the Counter object (instance) can be used as the lock in the synchronized statement to protect the inside of the increment() method:

```
class Counter
{
   private int c = 0;
   public void increment()
   {
        synchronized(this)
        {
            c++;
        }
   }
   public int getvalue() {return c;}
}
```

Let us take up another example program for studying the thread safety issue. The following program, which is a slight modification of the previous example, is thread-unsafe. Note how a shared object reference is passed.

```
import java.util.*;

public class ThreadSafetyUnsafe2
{
     public static void main(String[] args) throws InterruptedException
     {
        Counter sharedcounter = new Counter();        // shared reference

        ArrayList<Thread> allthreads = new ArrayList<Thread>();
```

```java
            for (int i = 1; i <= 10000; i++)
                allthreads.add(new Thread(new First(i, sharedcounter)));

            for (Thread t: allthreads)
                t.start();

            for (Thread t: allthreads)
                t.join();

            System.out.println("End: final counter value = " + sharedcounter.getvalue());

        }
    }



class First implements Runnable
{
    private int i;
    private Counter ctr;

    public First(int i, Counter c)
    {
        this.i = i;
        this.ctr = c;
    }
    public void changeCounter(Counter cntr)
    {
        if (i > 0)
            cntr.increment();
    }
    public void run()
    {
        changeCounter(ctr);

    }
}
```
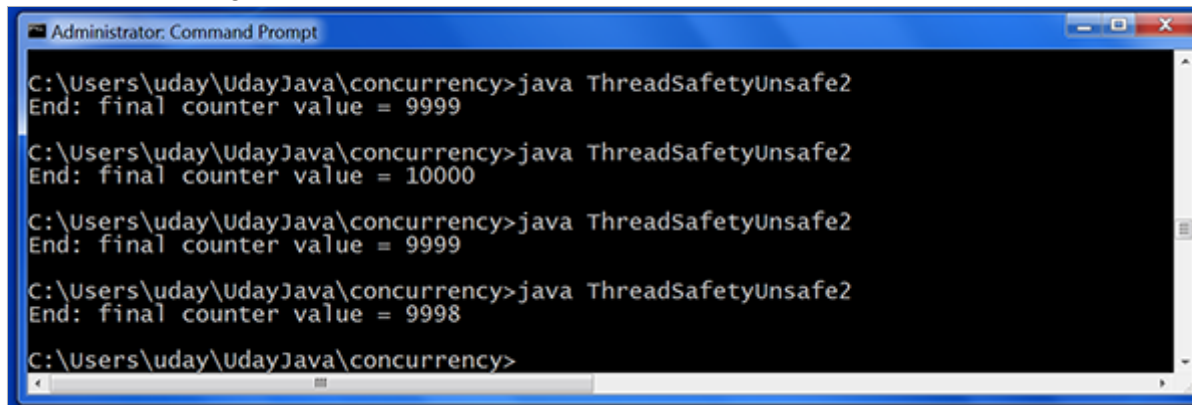
```
class Counter
{
    private int c = 0;
    public void increment() {c++;}
    public int getvalue() {return c;}
}
```

Some sample runs are shown below:

Output of the program - thread safety example 2

```
Administrator: Command Prompt

C:\Users\uday\UdayJava\concurrency>java ThreadSafetyUnsafe2
End: final counter value = 9999

C:\Users\uday\UdayJava\concurrency>java ThreadSafetyUnsafe2
End: final counter value = 10000

C:\Users\uday\UdayJava\concurrency>java ThreadSafetyUnsafe2
End: final counter value = 9999

C:\Users\uday\UdayJava\concurrency>java ThreadSafetyUnsafe2
End: final counter value = 9998

C:\Users\uday\UdayJava\concurrency>
```

In an attempt to make the program thread-safe, is it a good idea to make the changeCounter() method synchronized, as shown below?

```
public synchronized void changeCounter(Counter cntr)
    {
        if (i > 0)
            ntr.increment();
    }
```

The answer is no, because this synchronized changeCounter creates a lock on a **First** object, not on a **Counter** object. Multi-threading in this example was intended not on a single First object, but on a single Counter object. In other words, the 10,000 threads on as many different First objects can indeed run in interleaved fashion (because this "synchronized" creates 10,000 locks, not a single lock), making the program unsafe. A correct solution is to make the increment() method (of class Counter) synchronized.

Next, let us try an alternative approach for solving the same problem. Does it help to use a synchronized statement (with a lock on the cntr object) inside the changeCounter() method, as shown below?

```
public void changeCounter(Counter cntr)
{
    if(i > 0)
    {
        synchronized(cntr)
        {
            cntr.increment();
        }
    }
}
```

The answer is yes.

Synchronized-unsynchronized Mix

The following program mixes synchronized and unsynchronized methods (of the same class) in concurrent threads and is therefore unsafe. The output is potentially wrong (unpredictable).

```
import java.util.*;

public class SynchUnsynchMixUnsafe
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter sharedcounter = new Counter();      // shared reference

        ArrayList<Thread> allthreads = new ArrayList<Thread>();
```

```java
        for (int i = 1; i <= 10000; i++)
            allthreads.add(new Thread(new First(i, sharedcounter)));

        for (Thread t: allthreads)
            t.start();

        for (Thread t: allthreads)
            t.join();

        System.out.println("End: final counter value = " + sharedcounter.getvalue());

    }
}


class First implements Runnable
{
    private int i;
    private Counter ctr;

    public First(int i, Counter c)
    {
        this.i = i;
        this.ctr = c;
    }

    public void run()
    {
        if (i % 2 == 0)
            ctr.safe_increment();
        else
            ctr.unsafe_increment();

    }
}
```

```
class Counter
{
    private int c = 0;
    public synchronized void safe_increment() {c++;}
    public void unsafe_increment() {c++;}  // not synchronized
    public int getvalue() {return c;}
}
```

Note that:

- In a synchronized method that is non-static (i.e., instance), the lock is on the object, never on the method. For a synchronized static method, the lock is on the class.

For synchronized (instance) methods, the following facts are important:

- It is *not* possible for two (or more) threads running synchronized methods on the *same* object to interleave.
- When one thread is running a synchronized method on an object, all other threads trying to run synchronized methods (the same synchronized method or different synchronized methods) on the *same* object must block (that is, wait) until the first thread finishes with the object.
- However, it *is* possible for one thread running a synchronized method on an object to interleave with another thread running a non-synchronized (ordinary) method on the *same* object.

In order to realize that the unsafe nature of the concurrent computation in the immediately preceding program is not due solely to unsynchronized-unsynchronized interleaving (that program had 5000 threads each running an unsynchronized method, in addition to another 5000 threads that each ran a synchronized method; both the methods belonged to the same class), let us look at the following program where all threads but one run a synchronized method:

```
// all but one synchronized
import java.util.*;

public class SynchUnsynchMixUnsafe2
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter sharedcounter = new Counter();      // shared reference
```

```java
        ArrayList<Thread> allthreads = new ArrayList<Thread>();

        for (int i = 1; i <= 10; i++)
            allthreads.add(new Thread(new First(i, sharedcounter)));

        for (Thread t: allthreads)
            t.start();

        for (Thread t: allthreads)
            t.join();

        System.out.println("End: final counter value = " + sharedcounter.getvalue());
        System.out.flush();
    }
}


class First implements Runnable
{
    private int i;
    private Counter ctr;

    public First(int i, Counter c)
    {
        this.i = i;
        this.ctr = c;
    }

    public void run()
    {
        if (i != 5)
            ctr.safe_increment(i);
        else
            ctr.unsafe_increment(i); //only 1 thread is unsafe; others safe
    }
}
```

```java
class Counter
{
    private static int serial = 0;
    private int c = 0;

    public Counter()
    {
        serial++; // keeps track of how many objects created
    }

    public synchronized void safe_increment(int i)
    {
        System.out.println("Th-ID "+ Thread.currentThread().getId() +
                            " serial " + serial + " i= " + i + " Entering safe " + this);
        System.out.flush();
        for (int k = 0; k < 5000; k++)
                c++;
        try {
          Thread.sleep(5);
        }
        catch(InterruptedException ex)
        {
            ex.printStackTrace();
        }
        for (int k = 5000; k < 10000; k++)
                    c++;
        System.out.println("Th-ID "+ Thread.currentThread().getId() +
                            " serial " + serial + " i= " + i + " Exiting safe " + this);
        System.out.flush();
    }

    public void unsafe_increment(int i)  // not synchronized
    {
        System.out.println("Th-ID "+ Thread.currentThread().getId() +
```
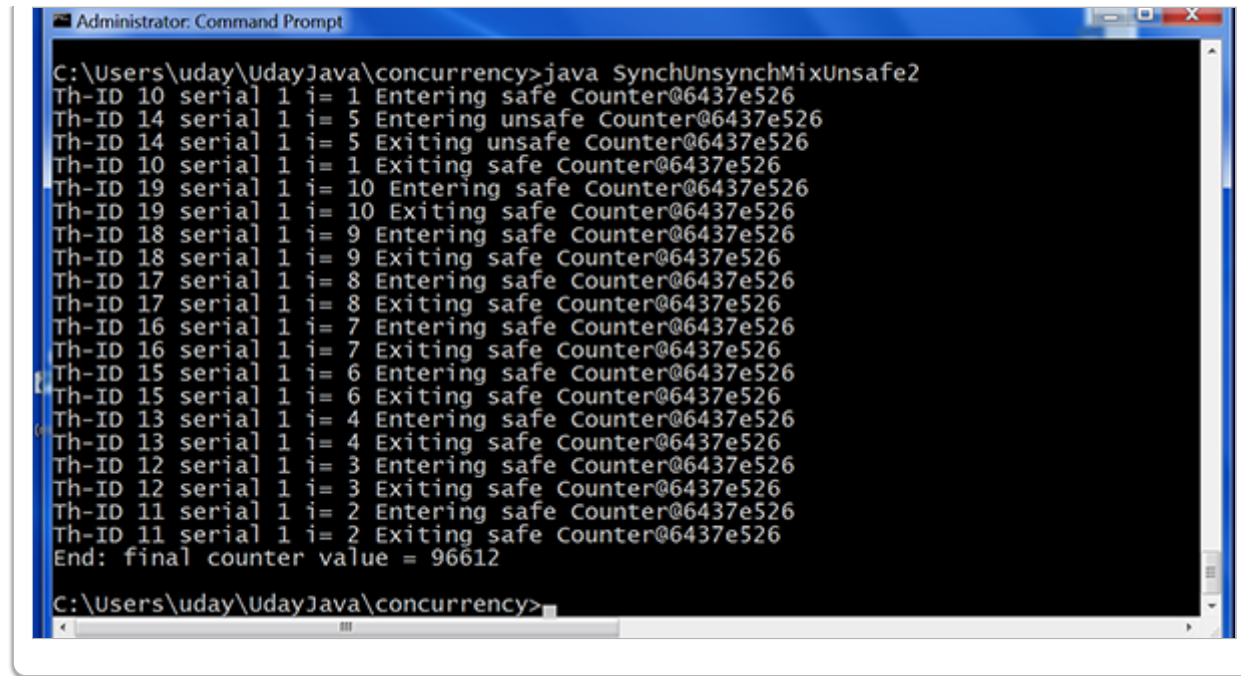
```
                              " serial " + serial + " i= " + i + " Entering unsafe " + this);
           System.out.flush();
           for (int k = 0; k < 5000; k++)
                  c++;
           try {
              Thread.sleep(5);
           }
           catch(InterruptedException ex)
           {
               ex.printStackTrace();
           }
           for (int k = 5000; k < 10000; k++)
                  c++;
           System.out.println("Th-ID "+ Thread.currentThread().getId() +
                              " serial " + serial + " i= " + i + " Exiting unsafe " + this);
           System.out.flush();
        }


      public int getvalue() {return c;}
     }
```

Here's some sample output that demonstrates synchronized-unsynchronized interleaving:

Output of the program - synchronized-unsynchronized interleaving

```
Administrator: Command Prompt

C:\Users\uday\UdayJava\concurrency>java SynchUnsynchMixUnsafe2
Th-ID 10 serial 1 i= 1 Entering safe Counter@6437e526
Th-ID 14 serial 1 i= 5 Entering unsafe Counter@6437e526
Th-ID 14 serial 1 i= 5 Exiting unsafe Counter@6437e526
Th-ID 10 serial 1 i= 1 Exiting safe Counter@6437e526
Th-ID 19 serial 1 i= 10 Entering safe Counter@6437e526
Th-ID 19 serial 1 i= 10 Exiting safe Counter@6437e526
Th-ID 18 serial 1 i= 9 Entering safe Counter@6437e526
Th-ID 18 serial 1 i= 9 Exiting safe Counter@6437e526
Th-ID 17 serial 1 i= 8 Entering safe Counter@6437e526
Th-ID 17 serial 1 i= 8 Exiting safe Counter@6437e526
Th-ID 16 serial 1 i= 7 Entering safe Counter@6437e526
Th-ID 16 serial 1 i= 7 Exiting safe Counter@6437e526
Th-ID 15 serial 1 i= 6 Entering safe Counter@6437e526
Th-ID 15 serial 1 i= 6 Exiting safe Counter@6437e526
Th-ID 13 serial 1 i= 4 Entering safe Counter@6437e526
Th-ID 13 serial 1 i= 4 Exiting safe Counter@6437e526
Th-ID 12 serial 1 i= 3 Entering safe Counter@6437e526
Th-ID 12 serial 1 i= 3 Exiting safe Counter@6437e526
Th-ID 11 serial 1 i= 2 Entering safe Counter@6437e526
Th-ID 11 serial 1 i= 2 Exiting safe Counter@6437e526
End: final counter value = 96612

C:\Users\uday\UdayJava\concurrency>
```

Unsafe Concurrency – Public Access

Thread safety analyses must be made extremely carefully. A naïve application of synchronized statements or synchronized methods may not guarantee safe concurrency. Public fields, for example, can cause problems, as shown in the following program where the variable c is declared public in class Counter:

```
import java.util.*;

public class UnsafePublicAccess
{
     public static void main(String[] args) throws InterruptedException
     {
         Counter sharedcounter = new Counter();      // shared reference
```

```java
            ArrayList<Thread> allthreads = new ArrayList<Thread>();


            for (int i = 1; i <= 400000; i++)
                 allthreads.add(new Thread(new First(i, sharedcounter)));


            for (Thread t: allthreads)
                t.start();


            for (Thread t: allthreads)
                t.join();


            System.out.println("End: final counter value = " + sharedcounter.getvalue());
         }
     }



    class First implements Runnable
    {
        private int i;
        private Counter ctr;

        public First(int i, Counter c)
        {
            this.i = i;
            this.ctr = c;
        }


        public void run()
        {
            if (i > 0)
                ctr.increment(); // lock released upon exit from increment()

            ctr.c++;  //because field c is public in Counter
         }
     }
```
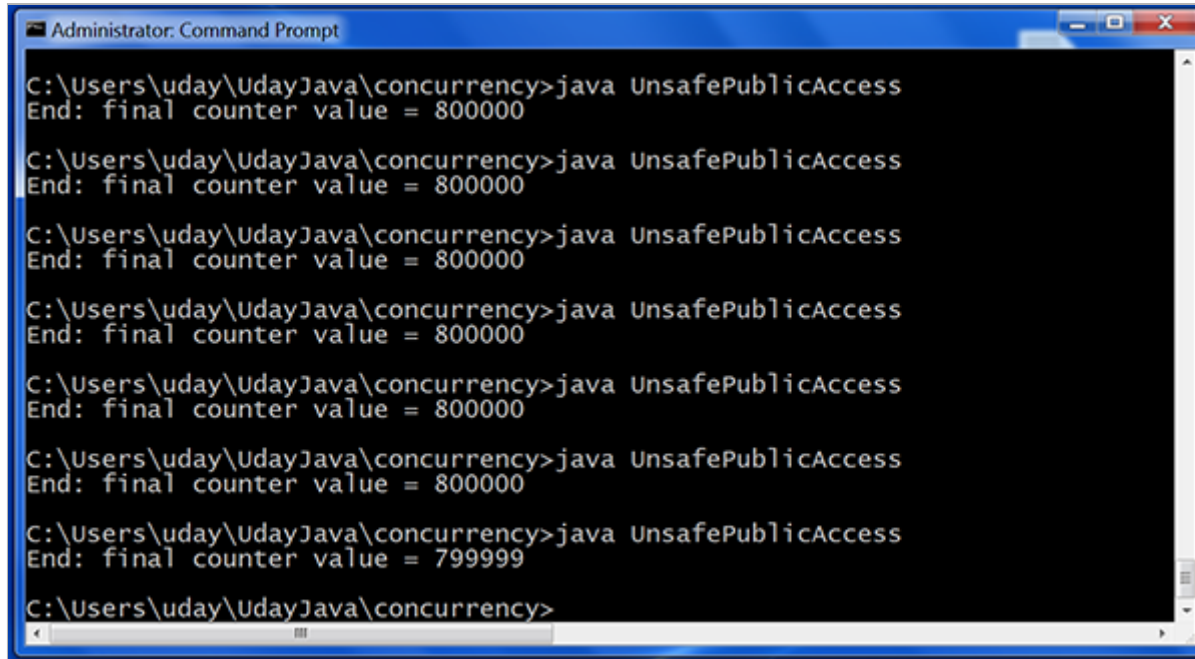
```
class Counter
{
    public int c = 0;   // public; unsafe

    public synchronized void increment() {c++;}
    public int getvalue() {return c;}
}
```

The above program is not thread-safe because making c public allows a client to alter c directly, without having to go through the (synchronized) increment() method. Some sample output is shown below.

Output of the program - unsafe public access

```
C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 800000

C:\Users\uday\UdayJava\concurrency>java UnsafePublicAccess
End: final counter value = 799999

C:\Users\uday\UdayJava\concurrency>
```

Lock Interface and ReentrantLock Class

In addition to synchronized statements and synchronized methods, explicit, named locks can be used to achieve synchronization. Java provides a Lock interface and several Lock classes for this purpose.

The following idiom is common in Java for using a Lock (read more about Interface Lock):

```
Lock lock = new …;
lock.lock();
try
{
     // use the resource protected by this lock
}
finally
{
  lock.unlock();
}
```

We revisit the class ThreadSafetyUnsafe2A, illustrating the use of ReentrantLock, a concrete implementation of the Lock interface, in the following program.

```
// Lock interface and ReentrantLock class
//Class ThreadSafetyUnsafe2A revisited

import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class ThreadSafetyUnsafe2AMadeSafeReentrant
{
     public static void main(String[] args) throws InterruptedException
     {
        Counter sharedcounter = new Counter();      // shared reference

        ArrayList<Thread> allthreads = new ArrayList<Thread>();

        for (int i = 1; i <= 1000; i++)
```

```
            allthreads.add(new Thread(new First(i, sharedcounter)));

        for (Thread t: allthreads)
            t.start();

        for (Thread t: allthreads)
            t.join();

        System.out.println("End: final counter value = " + sharedcounter.getvalue());

    }
}

class First implements Runnable
{
    private int i;
    private Counter ctr;

    public First(int i, Counter c)
    {
        this.i = i;
        this.ctr = c;
    }

    public void run()
    {
        if (i > 0)
            ctr.increment();
    }
}

class Counter
{
    private int c = 0;
```

```java
    private ReentrantLock rlock = new ReentrantLock();

    public void increment()
    {
        rlock.lock();   // acquire rlock
        try
        {
            c++;
            Thread.sleep(5);
        }
        catch(InterruptedException ex)
        {
            ex.printStackTrace();
        }
        finally
        {
            rlock.unlock();   //release rlock
        }
    }

    public int getvalue() {return c;}
}
```

In the above program, a single Counter object is shared among the many threads. Thread-safety is ensured by defining a private ReentrantLock object rlock as a field in class Counter and then using that lock (rlock) to protect the incrementing statement (c++) in method increment(). (The sleep() statement is inserted just for illustrative purposes.)

Thread Pool – Executors and ExecutorService

A *thread pool* enables us to execute concurrent tasks efficiently. Java provides the Executor and ExecutorService interfaces for facilitating multithreading.

The following examples creates and manages three concurrent First tasks. The newCachedThreadPool() method (of Executors) obtains an ExecutorService that is capable of creating new threads as they are needed by the program. ExecutorService's execute() method takes a Runnable as its argument and executes that (we do not have to invoke start() on a thread explicitly). Method execute() returns immediately; the program does not wait for each Runnable to finish. The shutdown() method of ExecutorService is used to stop accepting new Runnables, but does not affect the continuation of the execution of already-submitted Runnables. The ExecutorService terminates when all of the previously submitted Runnables have finished.

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

public class ThreadPoolExecutorService
{
    public static void main(String[] args) throws InterruptedException
    {
     Counter sharedcounter = new Counter();

     ExecutorService ex = Executors.newCachedThreadPool();

     ex.execute(new First(1, sharedcounter));
     ex.execute(new First(2, sharedcounter));
     ex.execute(new First(3, sharedcounter));

     ex.shutdown();

     System.out.println("Final count = " + sharedcounter.getvalue());
    }
}


class First implements Runnable
{
    private int i;
    private Counter ctr;

    public First(int i, Counter c)
    {
```

```java
        this.i = i;
        this.ctr = c;
    }
    public void changeCounter(Counter cntr)
    {
        if (i > 0)
        {
            synchronized(cntr) {
                    cntr.increment();
            }
        }
    }

    public void run()
    {
        changeCounter(ctr);
    }
}


class Counter
{
    private int c = 0;

    public void increment()
    {
            System.out.println("Thread Id: " + Thread.currentThread().getId());
            c = c + 1;
    }

    public int getvalue() {return c;}
}
```
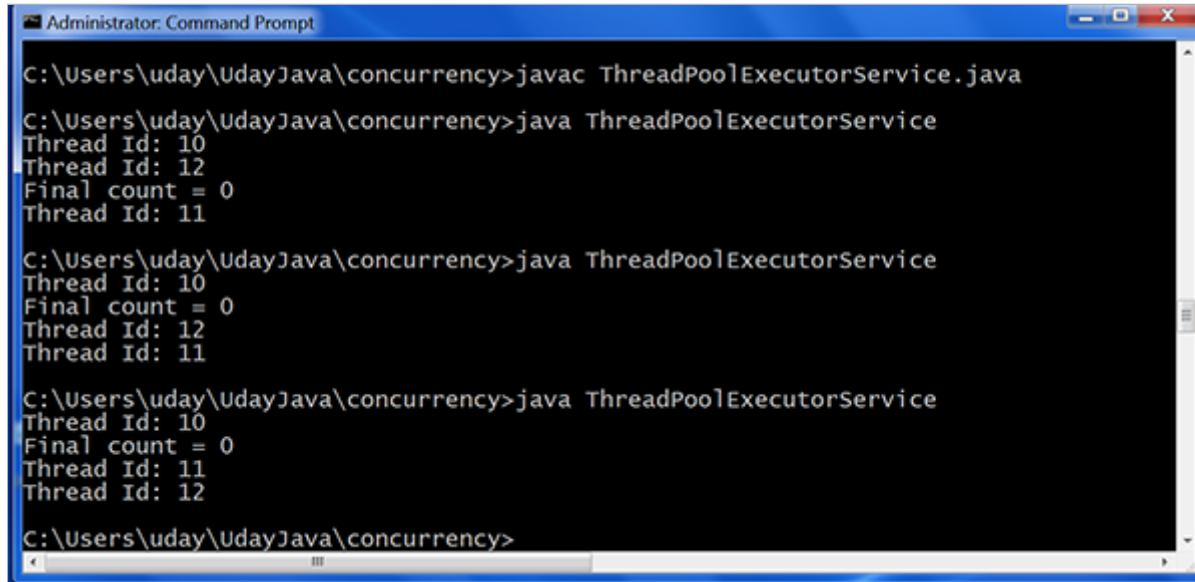
Sample output is shown below. That the final (incorrect) count is printed before all the three tasks have completed should come as no surprise.

Output of the program - Executors and ExecutorService

The isTerminated() method of ExecutorService returns true when all the threads have finished. This is illustrated in the following code and the sample output. Note that the final count is correct this time.

```
public class ThreadPoolExecutorService2
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter sharedcounter = new Counter();

        ExecutorService ex = Executors.newCachedThreadPool();

        ex.execute(new First(1, sharedcounter));
        ex.execute(new First(2, sharedcounter));
        ex.execute(new First(3, sharedcounter));

        ex.shutdown();
```

```java
            while (!ex.isTerminated()) System.out.println("Not yet done");

            System.out.println("Final count = " + sharedcounter.getvalue());
        }
    }
```

Sample output from three independent runs:

Run #1

```
    Thread Id: 10
    Thread Id: 11
    Not yet done
    Thread Id: 12
    Not yet done
    Final count = 3
```

Run #2

```
    Thread Id: 10
    Thread Id: 11
    Thread Id: 12
    Not yet done
    Final count = 3
```

Run #3

```
    Thread Id: 10
    Not yet done
    Not yet done
    Thread Id: 12
    Not yet done
    Not yet done
    Not yet done
    Not yet done
    Not yet done
```

```
Not yet done
Not yet done
Not yet done
Not yet done
Not yet done
Not yet done
Not yet done
Thread Id: 11
Not yet done
Final count = 3
```

The awaitTermination() method of ExecutorService waits until all threads are finished or until the specified amount of time has elapsed. Here is an example:

```java
public class ThreadPoolExecutorService2
{
    public static void main(String[] args) throws InterruptedException
    {
        Counter sharedcounter = new Counter();

        ExecutorService ex = Executors.newCachedThreadPool();

        ex.execute(new First(1, sharedcounter));
        ex.execute(new First(2, sharedcounter));
        ex.execute(new First(3, sharedcounter));

        ex.shutdown();

        ex.awaitTermination(50, TimeUnit.MILLISECONDS);

        System.out.println("Final count = " + sharedcounter.getvalue());
    }
}
```

The newFixedThreadPool(int) method creates a fixed number (the number specified in the argument) of threads in a pool. When a thread finishes execution, it can be re-used for another pending task.
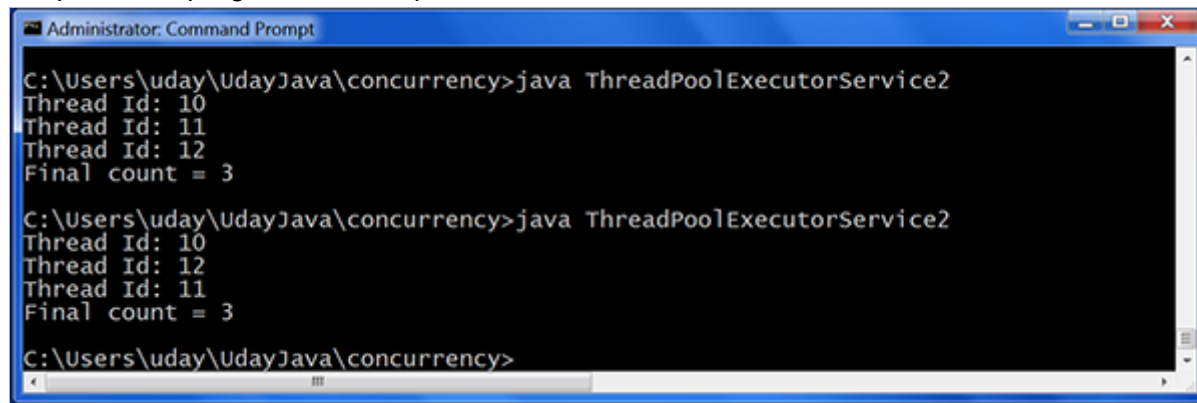
Replacing the line

```
ExecutorService ex = Executors.newCachedThreadPool();
```

in the above code with

```
ExecutorService ex = Executors.newFixedThreadPool(3);
```

will produce the following sample output:

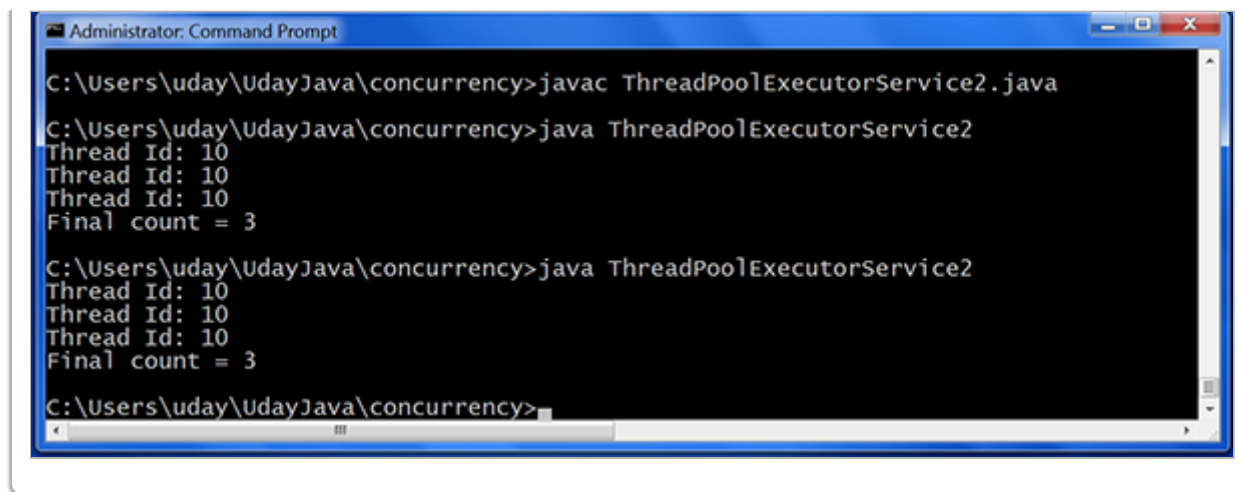Output of the program - thread pool ExecutorService

The effect of thread re-use can be seen by changing the number of threads from 3 to 1 in the line

```
ExecutorService ex = Executors.newFixedThreadPool(1);
```

The corresponding output is shown below:

Output of the program - thread re-use

Deadlock

Deadlock is the situation when two or more threads indefinitely wait for each other. For example, the following situation represents a deadlock: Thread1 waits for the lock on account2 to be released, while Thread2 waits for account1 to be released.

Thread1:

```
synchronized (account1)
{
    account1.withdraw(100);
    synchronized (account2)
    {
        account2.deposit(100);
    }
}
```

Thread2:

```
synchronized (account2)
```

```
        {
            account2.withdraw(100);
            synchronized(account1)
            {
                account1.deposit(100);
            }
        }
```

One way of deadlock avoidance is **resource ordering**, which in the present example would mean that both the threads must acquire locks on the two accounts in the *same* order. One of the following two orders would do:

- account1 followed by account2, or
- account2 followed by account1

Module 5 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 5.1

True or false: A thread is not runnable once it has terminated.

Suggested answer: True.

Test Yourself 5.2

How many threads run in class ThreadSafetyUnsafe2A?

Suggested answer: 10000 + 1(main) = 10001.

Test Yourself 5.3

Consider the class

```
class Alpha
{
    private String title = "An ABC of English Usage";
    public synchronized Alpha(String newtitle) {title = newtitle;}
    public synchronized void change() {title = title.substring(0, 3);}
    public synchronized int getlen() {return title.length();}
}
```

Is there anything wrong?

Suggested answer: The constructor cannot be made synchronized.

Test Yourself 5.4

How many threads run in the following program?

```
public class TestYourselfQuestion
{
        public static void main(String[] args)
        {
                First  f1 = new First(100);
                First  f2 = new First(200);

                Thread t1 = new Thread(f1);
                Thread t2 = new Thread(f2);

                t1.run(); t2.run();
        }
}
class First implements Runnable
```

```java
    {
        private int i;

        public First(int i) {this.i = i;}

        public void run()
        {
            System.out.println("Hi");
        }
    }
```

Suggested answer: One (the main thread).

References

- Oracle. *The Java™ Tutorials: Concurrency*. Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html.
- Oracle. *The Java™ Tutorials: Networking*. Retrieved from https://docs.oracle.com/javase/tutorial/networking/overview/index.html.
- Goetz, B. (2016). *Java Concurrency in Practice.* Addison-Wesley.

Boston University Metropolitan College