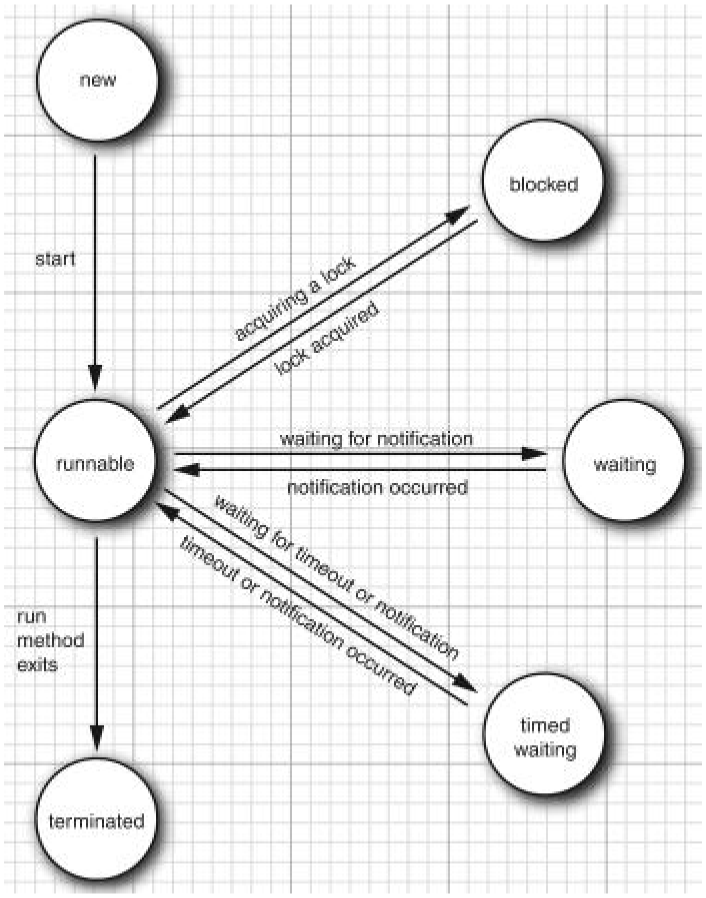
Lab 2: Process and Thread Management

Total points: 100

I. Multithreading in Java

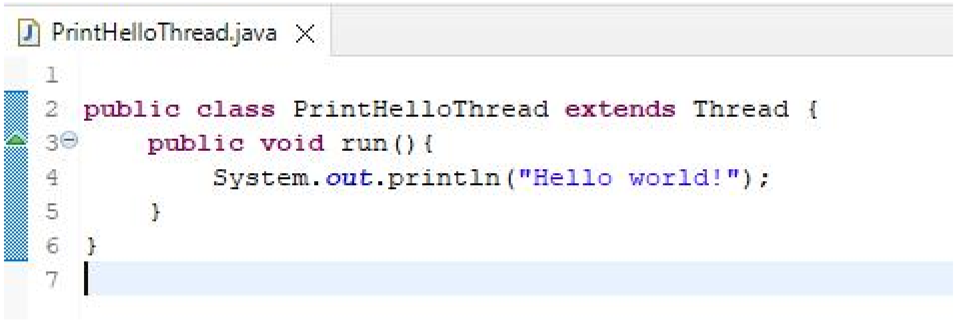
1. Thread States



2. Creating and running a task in a separate thread

a. Creating a subclass of `Thread`

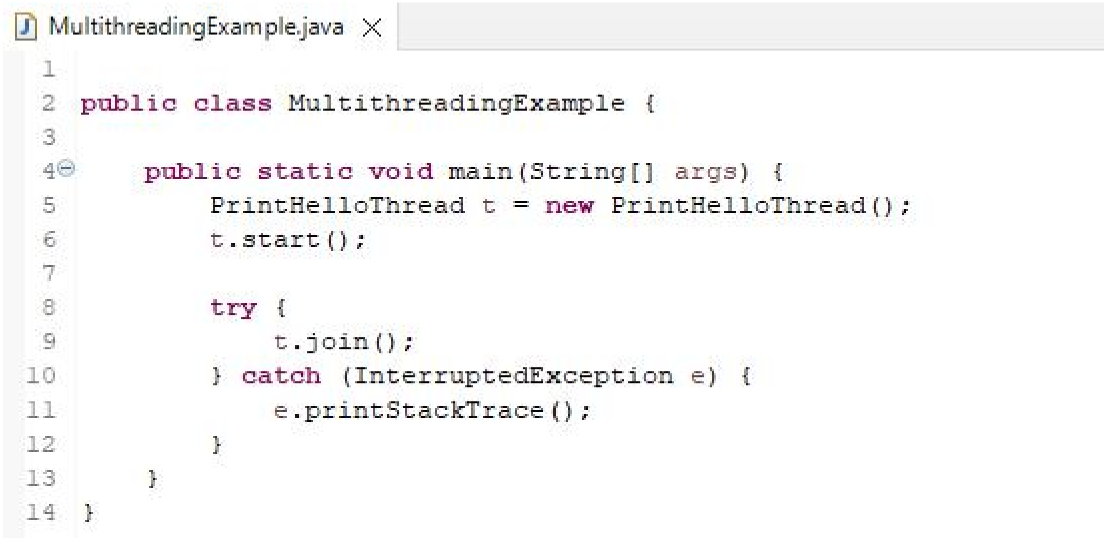
The first way is to create a subclass of `Thread` and override the `run()` method. The `run()` method is what is executed by the thread after you call `start()`.



```java

public class PrintHelloThread extends Thread {
    public void run(){
        System.out.println("Hello world!");
    }
}

```

The `start()` call will return as soon as the thread is started. It will not wait until the `run()` method is done. The `run()` method will execute as if executed by a different CPU. When the `run()` method executes it will print out the text “`Hello world!`”.

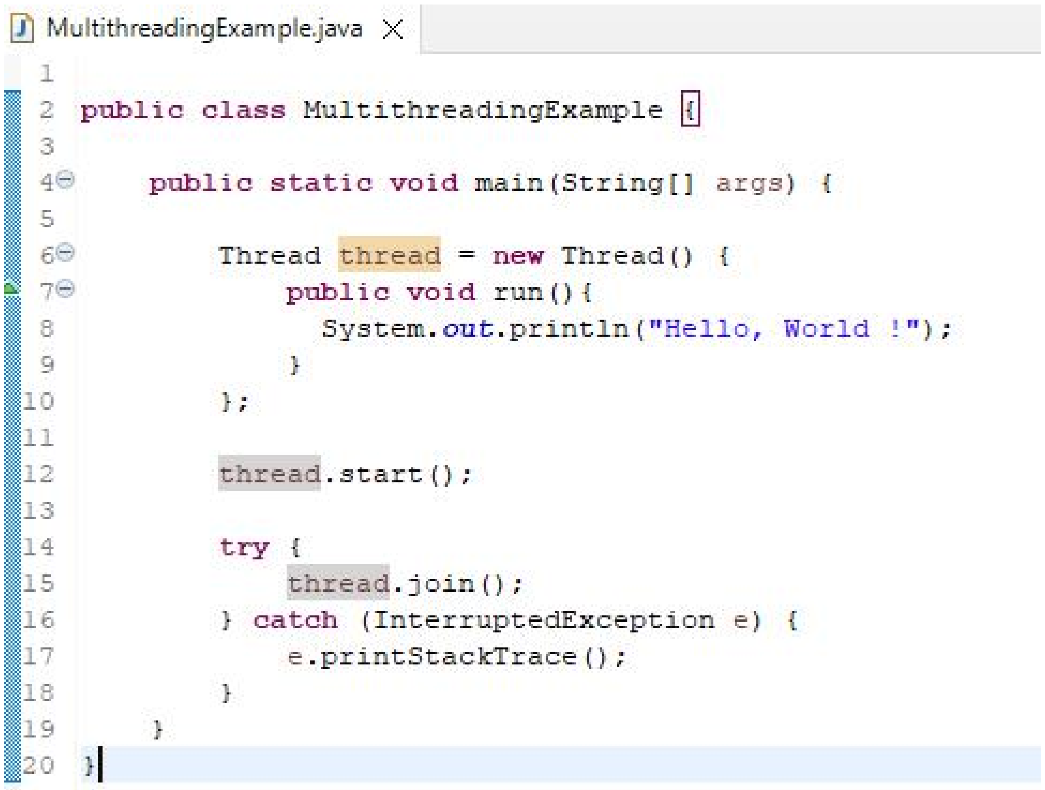
Do not call the `run()` method directly. The `start()` method will call the `run()` method in a new created thread.



MultithreadingExample.java

```java

public class MultithreadingExample {

    public static void main(String[] args) {
        PrintHelloThread t = new PrintHelloThread();
        t.start();

        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- You can also create an anonymous subclass of `Thread` like this:



MultithreadingExample.java

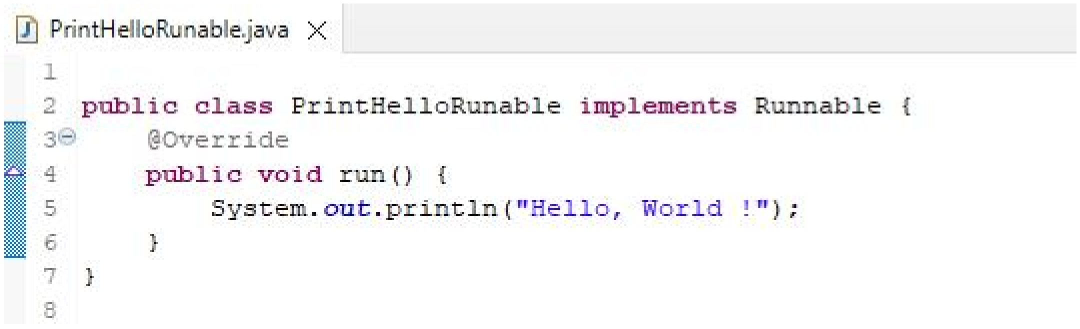
```java

public class MultithreadingExample {

    public static void main(String[] args) {

        Thread thread = new Thread() {
            public void run(){
              System.out.println("Hello, World !");
            }
        };

        thread.start();

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

b. Implementing the `Runnable` interface

The second way is to create a class that implements the `java.lang.Runnable` interface. A Java object that implements the `Runnable` interface can be executed by a Java Thread.

The `Runnable` interface is a standard Java Interface that comes with the Java platform. The `Runnable` interface only has a single method `run()`. Here is basically how the `Runnable` interface looks:

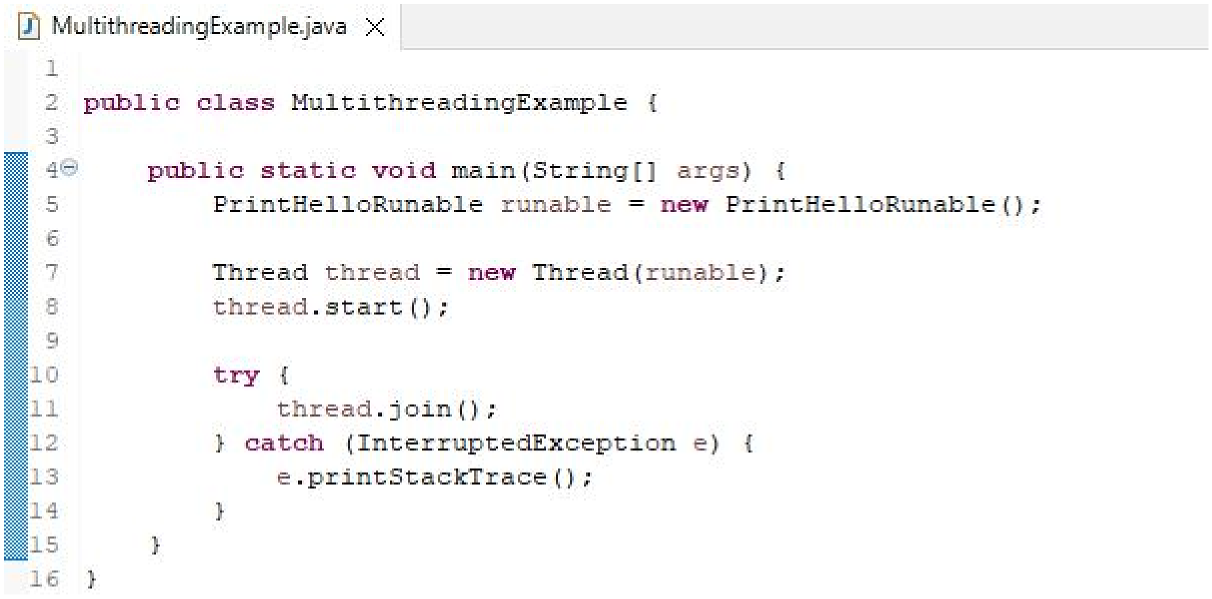
```
public interface Runnable() {

    public void run();

}
```



PrintHelloRunable.java

```

public class PrintHelloRunable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello, World !");
    }
}

```

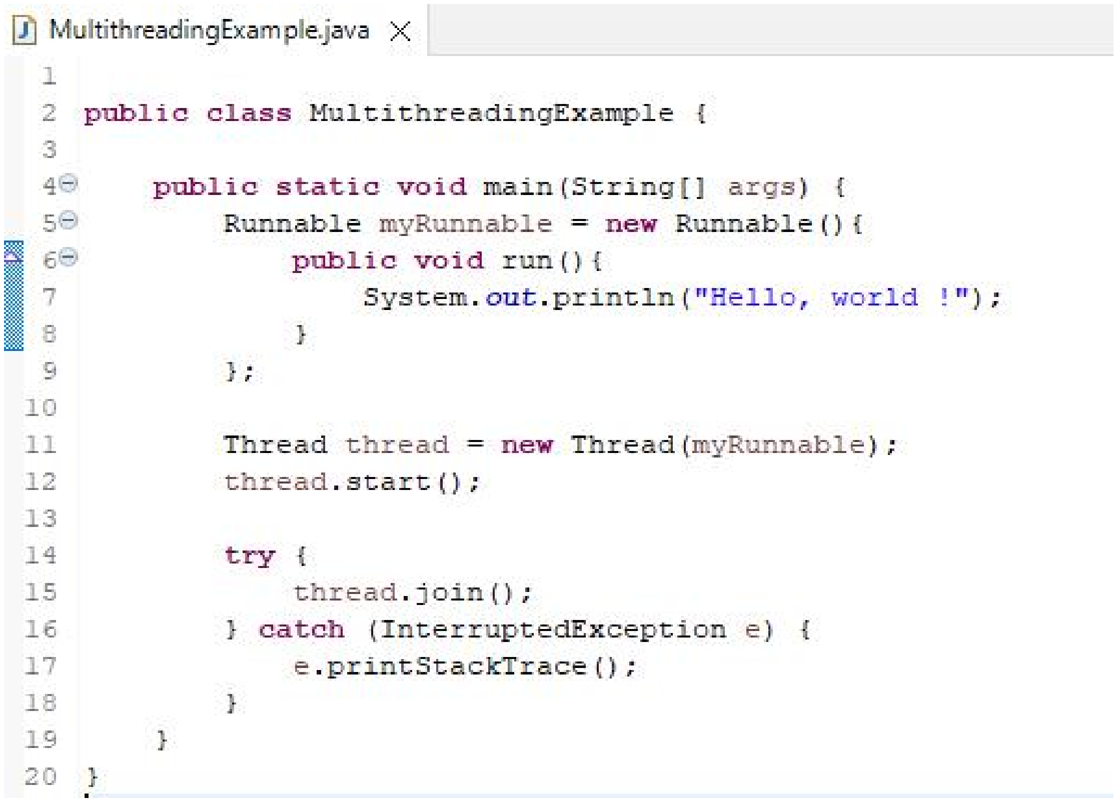
Then we create and start a new `Thread` with a `Runnable`



MultithreadingExample.java

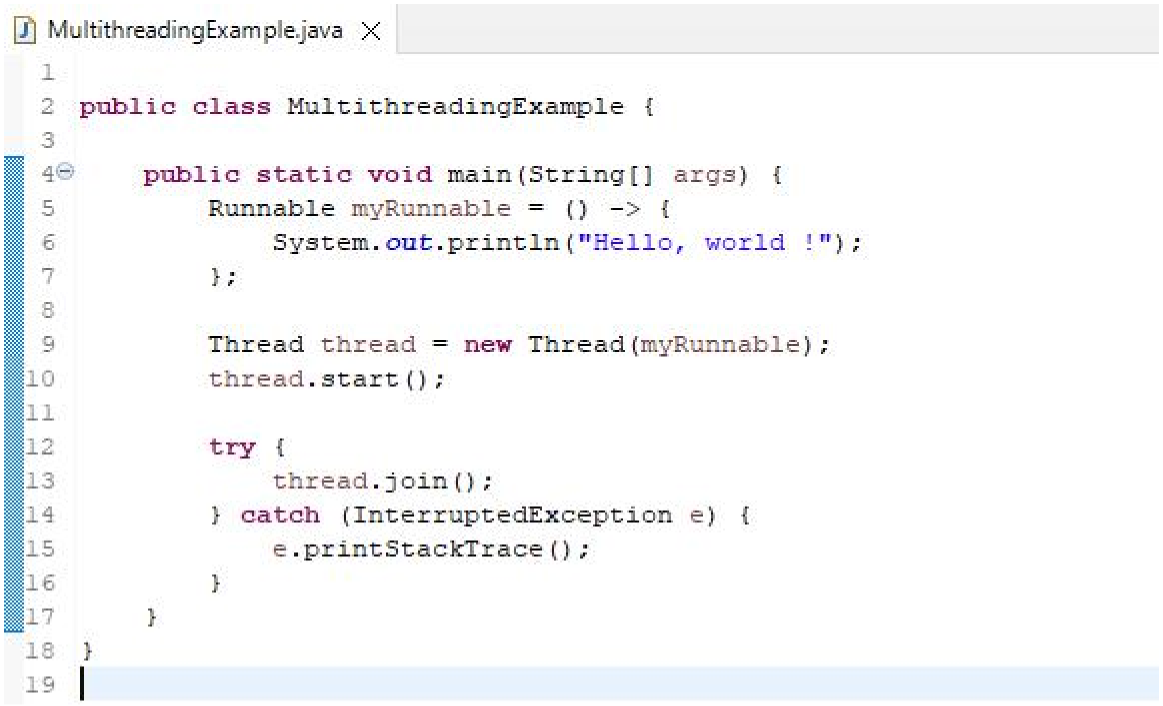
```

public class MultithreadingExample {

    public static void main(String[] args) {
        PrintHelloRunable runable = new PrintHelloRunable();

        Thread thread = new Thread(runable);
        thread.start();

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Anonymous class Implementation of `Runnable`



```java

public class MultithreadingExample {

    public static void main(String[] args) {
        Runnable myRunnable = new Runnable(){
            public void run(){
                System.out.println("Hello, world !");
            }
        };

        Thread thread = new Thread(myRunnable);
        thread.start();

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Java Lambda Implementation of `Runnable`



MultithreadingExample.java

```java

public class MultithreadingExample {

    public static void main(String[] args) {
        Runnable myRunnable = () -> {
            System.out.println("Hello, world !");
        };

        Thread thread = new Thread(myRunnable);
        thread.start();

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

3. Interrupting Threads

A thread terminates when its `run` method returns—by executing a `return` statement, after executing the last statement in the method body, or if an exception occurs that is not caught in the method.

There is no way to force a thread to terminate. However, the `interrupt` method can be used to request termination of a thread.

When the `interrupt` method is called on a thread, the `interrupted` status of the thread is set. This is a `boolean` flag that is present in every thread. Each thread should occasionally check whether it has been interrupted.

```
Runnable r = () -> {
      try {
            . . .
            while(!Thread.currentThread().isInterrupted() && more work to do) {
                  do more work
            }
      }
      catch(InterruptedException e) {
            // thread was interrupted during sleep or wait
      }
      finally {
            cleanup, if required
      }
      // exiting the run method terminates the thread
};
```

The `isInterrupted` check is neither necessary nor useful if you call the `sleep` method (or another interruptible method) after every work iteration. If you call the `sleep` method when the `interrupted` status is set, it doesn't `sleep`. Instead, it clears the status (!) and throws an `InterruptedException`. Therefore, if your loop calls `sleep`, don't check the `interrupted` status. Instead, catch the `InterruptedException`, like this:

```
Runnable r = () -> {
      try {
            . . .
            While(more work to do) {
                  do more work
                  Thread.sleep(delay);
            }
      }
      catch(InterruptedException e) {
            // thread was interrupted during sleep or wait
      }
      finally {
            cleanup, if required
      }
      //exiting the run method terminates the thread
};
```

II. Assignments

Assignment 1 (20 points)

Develop a program which has a shared data and two threads. The shared data contains an integer variable which is set to 0 initially. The first thread will increase the variable a number of times (1000000 times) and the second thread will decrease the variable the same number of times (1000000 times).

a. The program creates two threads and starts the first thread. The program waits for the first thread to terminate and then start the second thread. The program waits for the second thread to terminate and display the variable's value. Explain the result.

b. The program creates, starts two threads and waits for the two threads to terminate and display the variable's value. Explain the result.

Assignment 2 (Problem 4.22) (20 points)

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

```
    90 81 78 95 79 72 85
```

The program will report

```
    The average value is 82

    The minimum value is 72

    The maximum value is 95
```

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and **the parent thread will output the values once the workers have exited**. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

Assignment 3 (Problem 4.23) (20 points)

Write a multithreaded 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.

Assignment 4 (Problem 4.24) (20 points)

An interesting way of calculating π is to use a technique known as *Monte Carlo*, which involves randomization. This technique works as follows:

Suppose you have a circle inscribed within a square, as shown in Figure 4.25. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.
- Next, estimate π by performing the following calculation:

$$\pi = 4\times (\textit{number of points in circle}) / (\textit{total number of points})$$

Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result in a global variable. When this thread has exited, **the parent thread will calculate and output the estimated value of π**. It is worth experimenting with the number of random points generated. As a general rule, the greater the number of points, the closer the approximation to π.

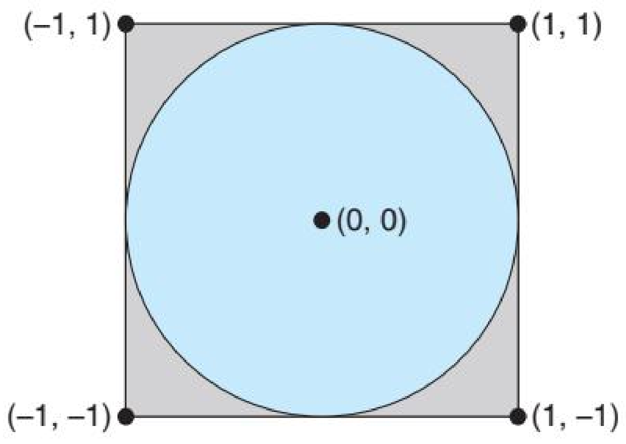


Figure 4.25 Monte Carlo technique for calculating π.

Assignment 5 (Problem 4.27) (20 points)

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

$fib_0 = 0$

$fib_1 = 1$

$fib_\mathrm{n} = fib_{n-1} + fib_{n-2}$

Write a multithreaded program that generates the Fibonacci sequence. This program should work as follows: On the command line, the user will enter 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, the parent thread will have to wait for the child thread to finish.