Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

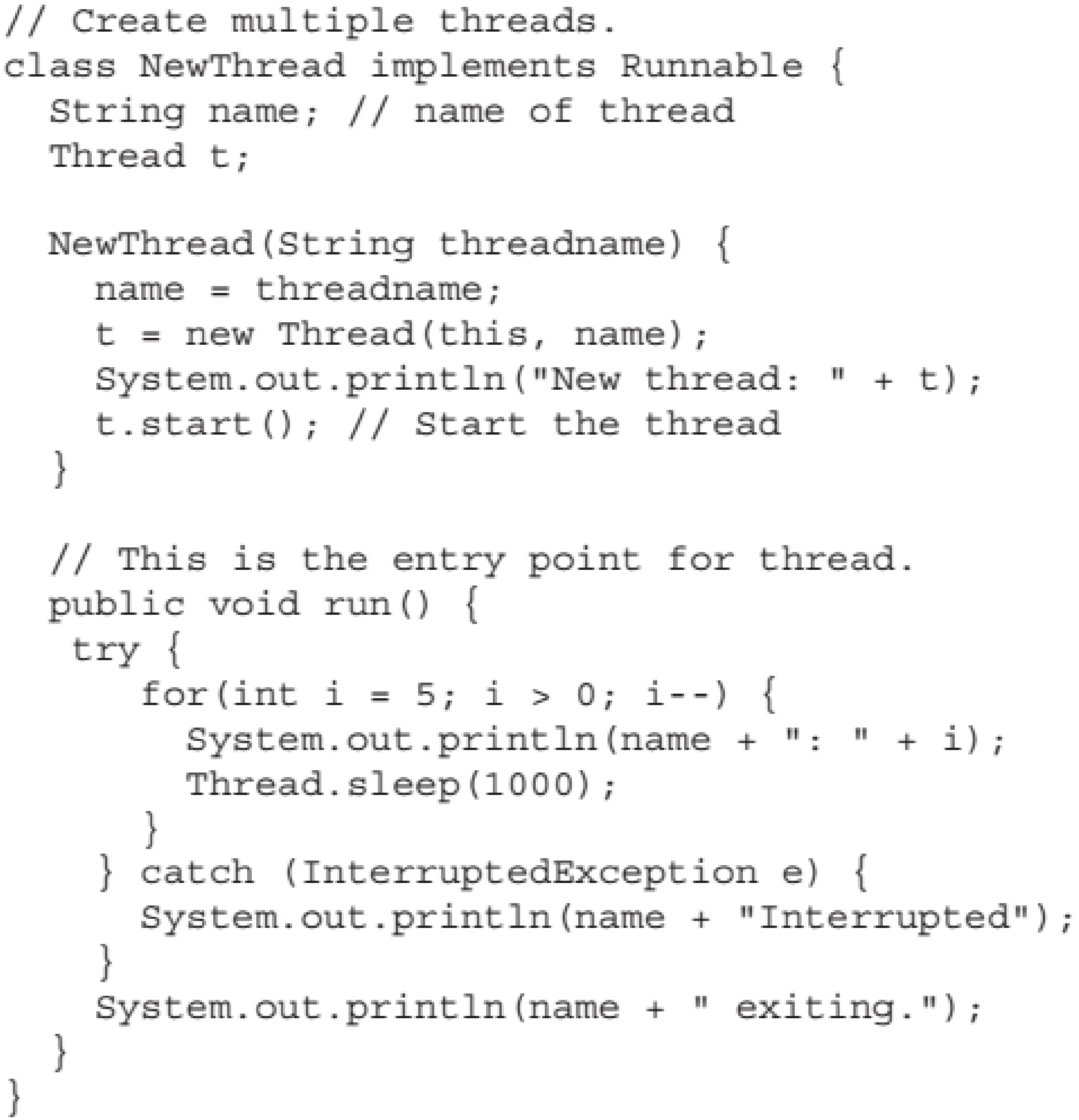
Lecture 13: Multi Threading

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

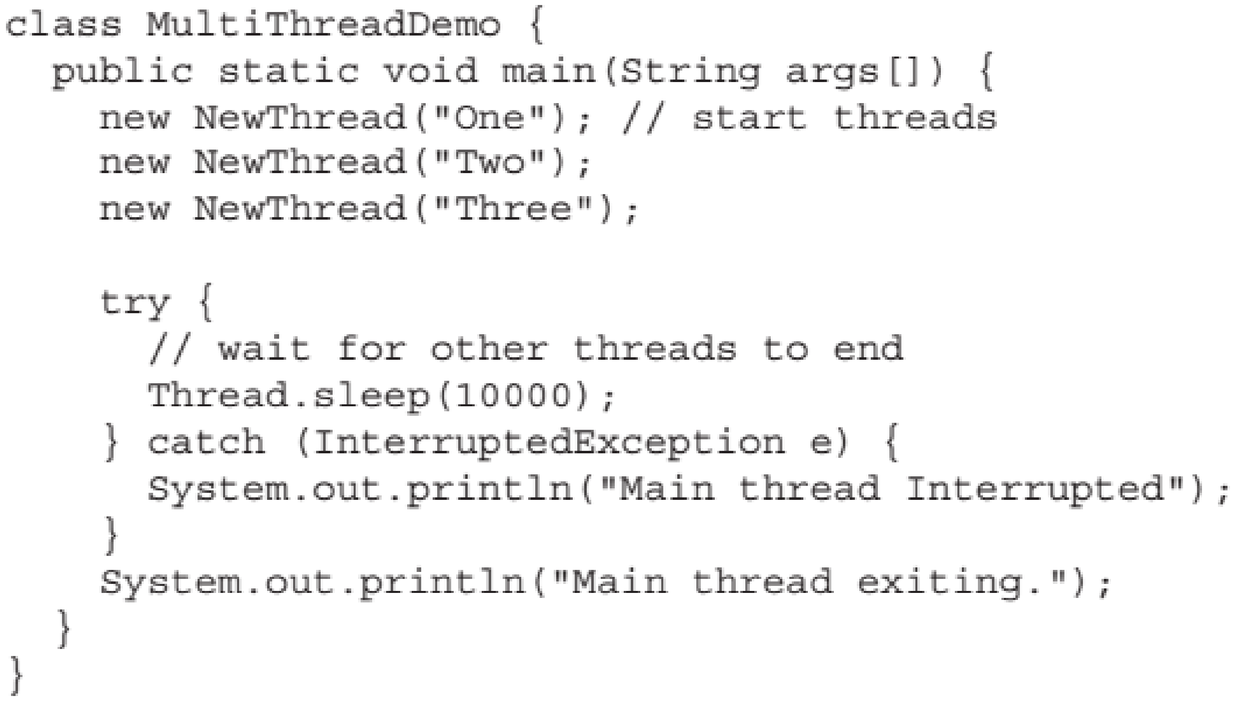
Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Creating Multiple Threads

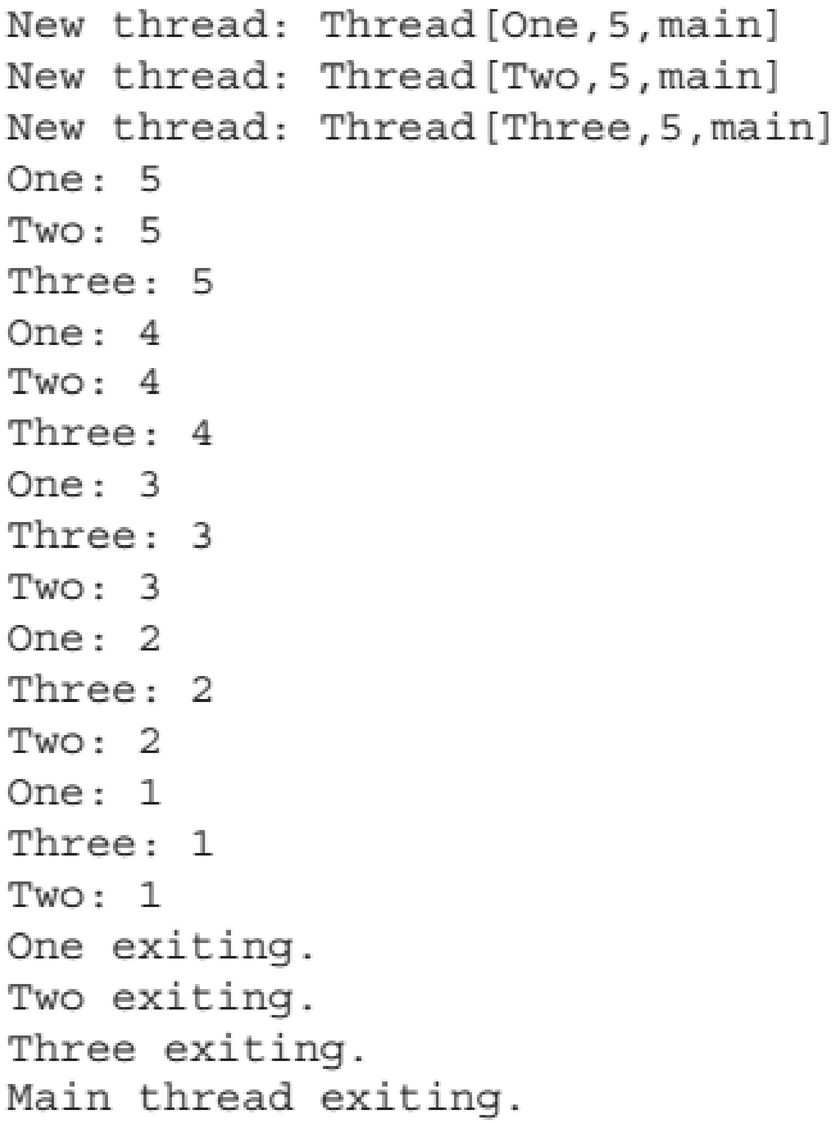


```
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
   try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}
```



```
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");

    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Creating Multiple Threads



```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive( ) and join( )

- How can one thread know when another thread has ended without Sleep function?
- Two ways exist to determine whether a thread has finished.
- First, you can call **isAlive( )** on the thread.
  - This method is defined by **Thread**, and its general form is shown here:

    ```
    final boolean isAlive( )
    ```
  - The **isAlive( )** method returns **true** if the thread upon which it is called is still running.
  - It returns **false** otherwise.
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

  ```
  final void join( ) throws InterruptedException
  ```
- This method waits until the thread on which it is called terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");
    NewThread ob3 = new NewThread("Three");

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());
    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + ob1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + ob2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + ob3.t.isAlive());

    System.out.println("Main thread exiting.");
  }
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Thread Priorities

- To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.
- This is its general form:

  final void setPriority(int *level*)
- *level* specifies the new priority setting for the calling thread.
- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**.
- Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.
- These priorities are defined as **static final** variables within **Thread**.
- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

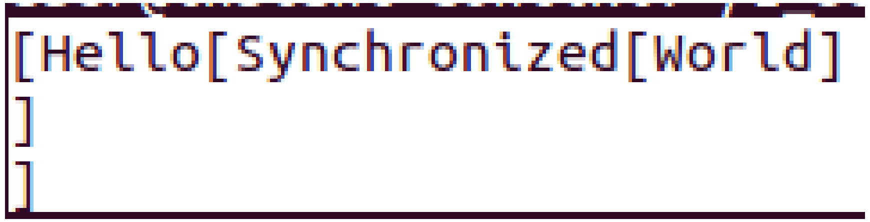
  final int getPriority( )

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called *synchronization*.
- Key to synchronization is the concept of the monitor.
- A *monitor* is an object that is used as a mutually exclusive lock.
- Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor.
- These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword

```java
// This program is not synchronized.
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }

 }

 class Caller implements Runnable {
   String msg;
   Callme target;
   Thread t;

   public Caller(Callme targ, String s) {
     target = targ;
     msg = s;
     t = new Thread(this);
     t.start();
   }
   public void run() {
     target.call(msg);
   }
 }
```

```java
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```



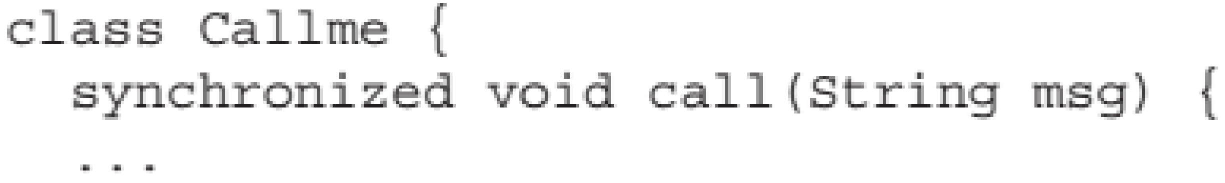
```
[Hello[Synchronized[World]
]
]
```

Synchronization

- As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread.
- This results in the mixed-up output of the three message strings.
- In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time.
- This is known as a *race condition*, because the three threads are racing each
  other to complete the method.
- This example used **sleep( )** to make the effects repeatable and obvious.
- In most situations, a race condition is more subtle and less predictable, because you can’t be sure when the context switch will occur.
- This can cause a program to run right one time and wrong the next.

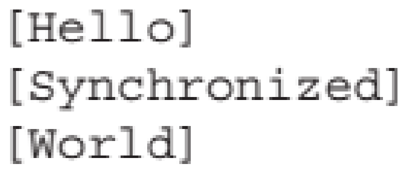
Synchronization

- To fix the preceding program, you must *serialize* access to **call( )**.
- That is, you must restrict its access to only one thread at a time.
- To do this, you simply need to precede **call( )**’s definition with the keyword **synchronized**, as shown here:



```
class Callme {
  synchronized void call(String msg) {
  ...
```

- This prevents other threads from entering **call( )** while another thread is using it.
- After **synchronized** has been added to **call( )**, the output of the program is as follows:



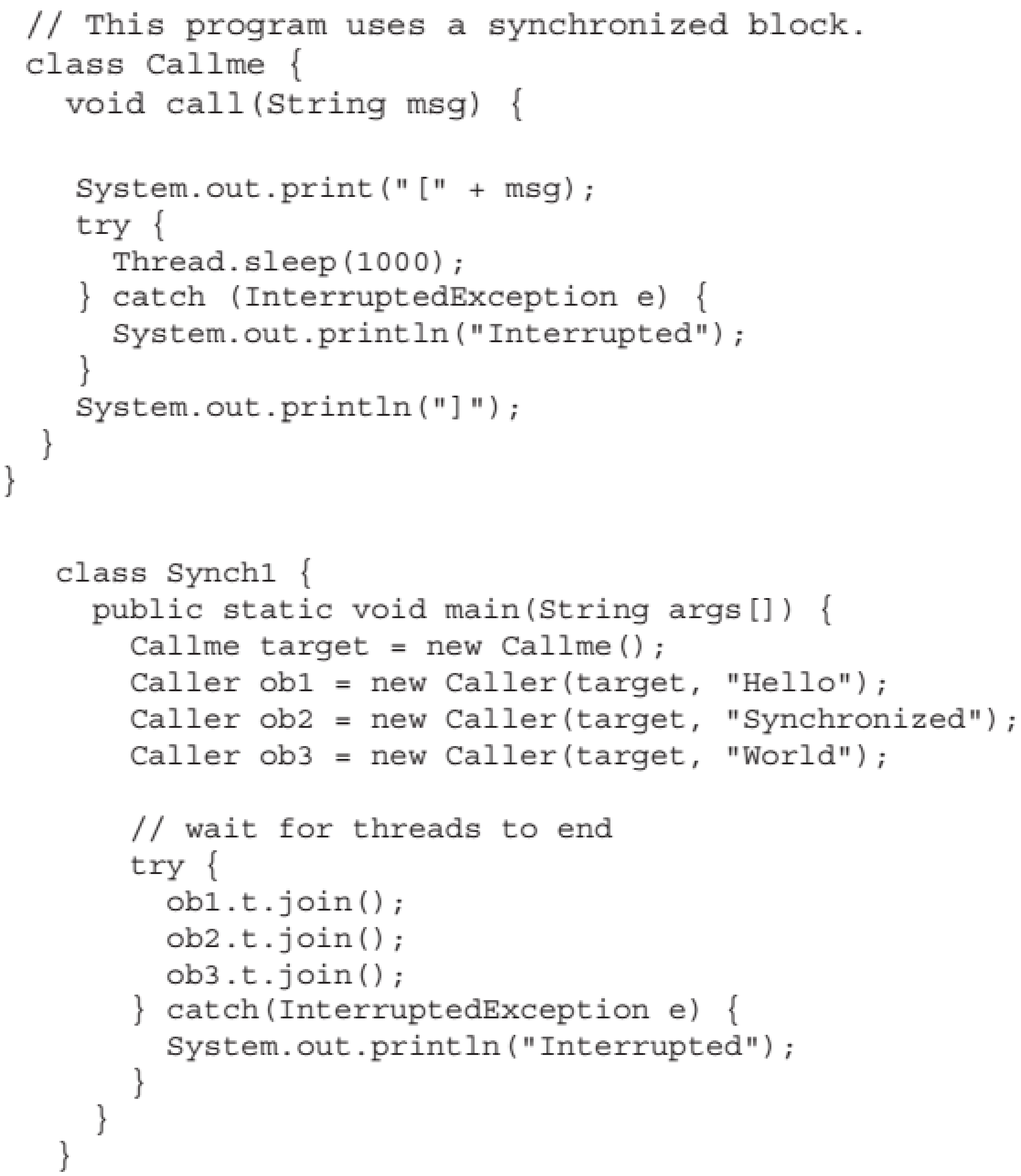
```
[Hello]
[Synchronized]
[World]
```

The synchronized Statement

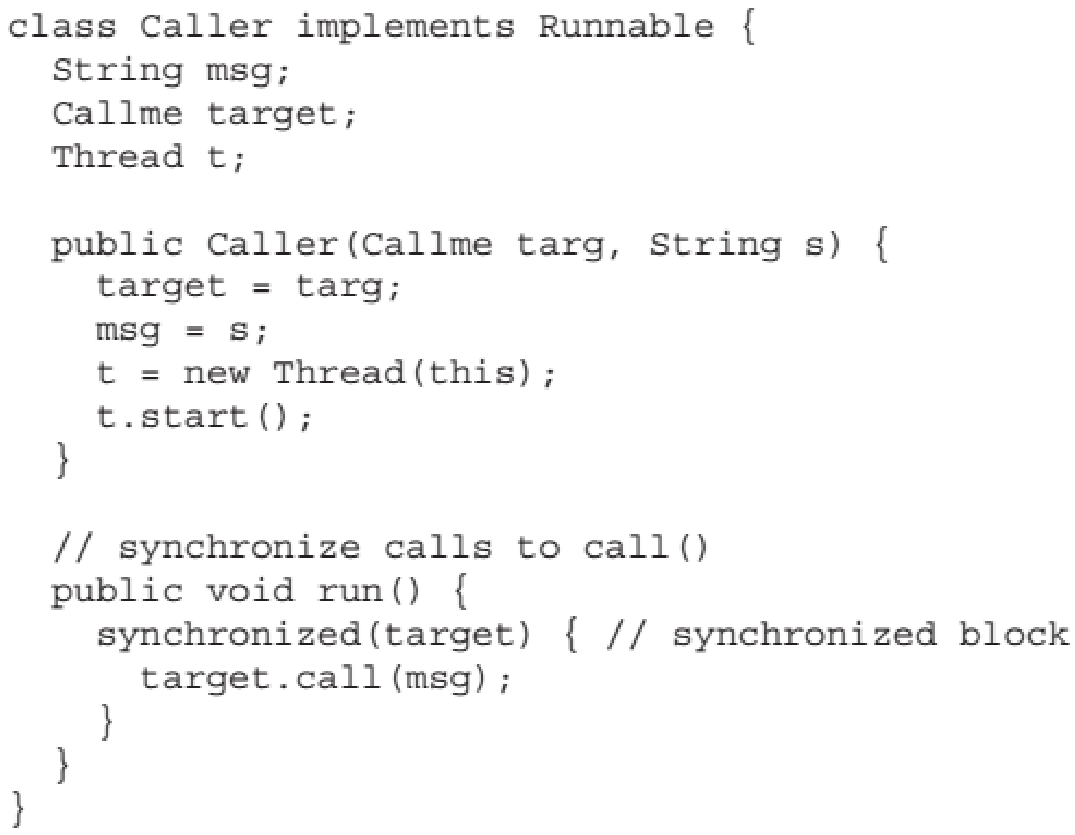
- While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization.
- This is the general form of the **synchronized** statement:

```
synchronized(objRef) {
        // statements to be synchronized
}
```

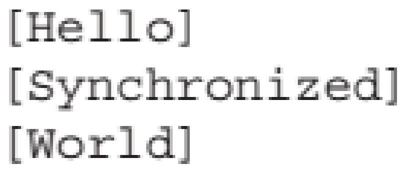
- Here, *objRef* is a reference to the object being synchronized.
- A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.



```
// This program uses a synchronized block.
class Callme {
  void call(String msg) {

    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}


class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target, "Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
  }
}
```



```
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

  // synchronize calls to call()
  public void run() {
    synchronized(target) { // synchronized block
      target.call(msg);
    }
  }
}
```



```
[Hello]
[Synchronized]
[World]
```

Interthread Communication

- Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.
- These methods are implemented as **final** methods in **Object**, so all classes have them.
- All three methods can be called only from within a **synchronized** context.
- Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:
  - **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( )**.
  - **notify( )** wakes up a thread that called **wait( )** on the same object.
  - **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.
- These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

- Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

Producer/Consumer problem.

- Consider the classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- Clearly, this situation is undesirable.

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}
```

```
class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}
```

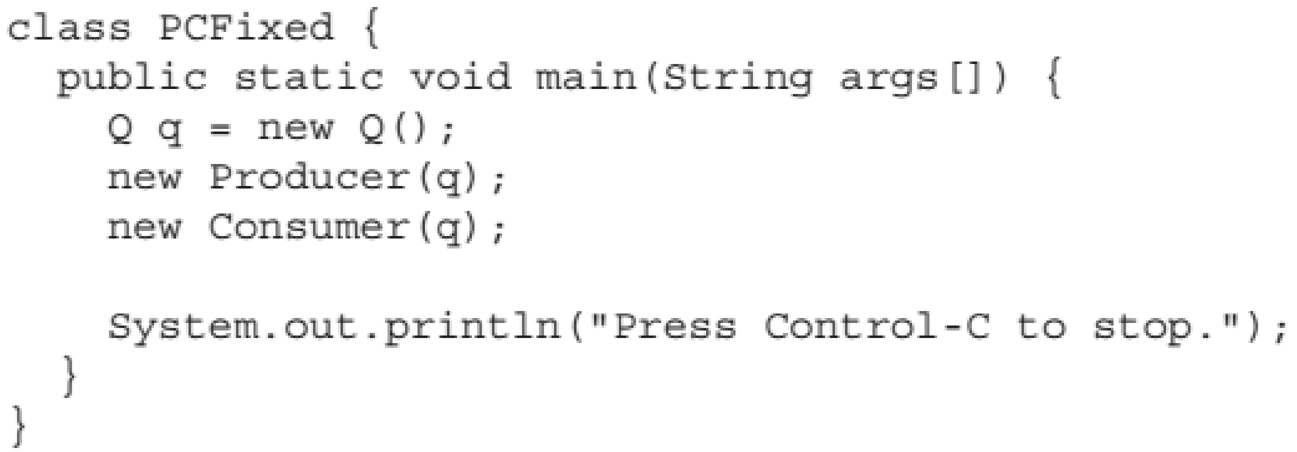
```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

```
class PC {
  public static void main(String args[]) {

    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

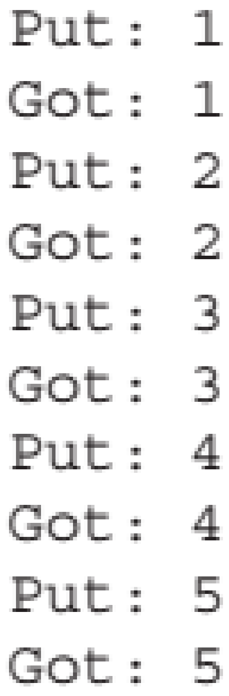
```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
  }

  synchronized void put(int n) {

    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
  }
}
```

```
class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    int i = 0;

    while(true) {
      q.put(i++);
    }
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```



```
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```



```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

Figure - 1

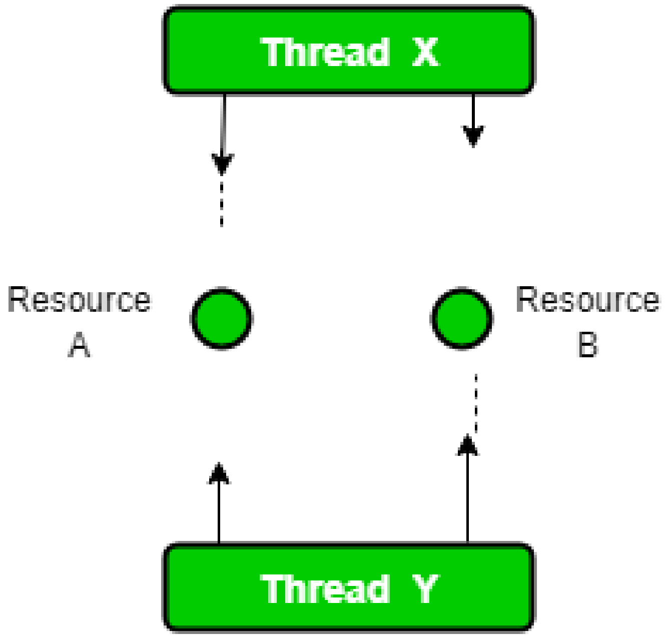
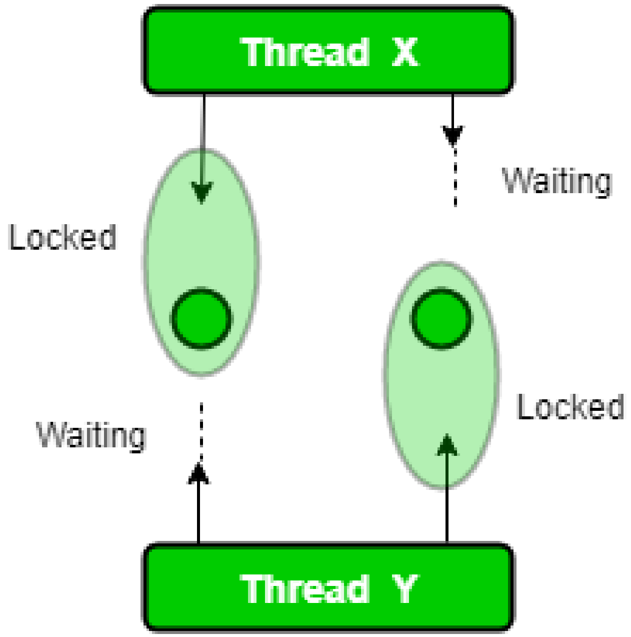


Figure - 2



```
// An example of deadlock.
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();

    System.out.println(name + " entered A.foo");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");

    }

    System.out.println(name + " trying to call B.last()");
    b.last();
  }

  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
```

```
class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entered B.bar");

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("B Interrupted");
    }

    System.out.println(name + " trying to call A.last()");
    a.last();
  }

  synchronized void last() {
    System.out.println("Inside A.last");
  }
}
```

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();

  Deadlock() {
    Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();

    a.foo(b); // get lock on a in this thread.
    System.out.println("Back in main thread");
  }

  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");
  }

  public static void main(String args[]) {
    new Deadlock();
  }
}
```

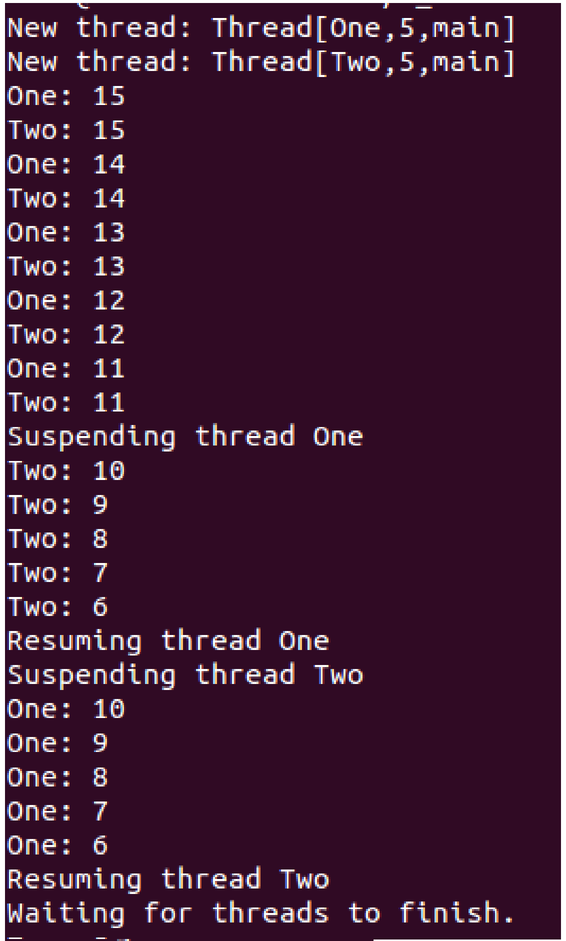
```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Suspending, Resuming, and Stopping Threads

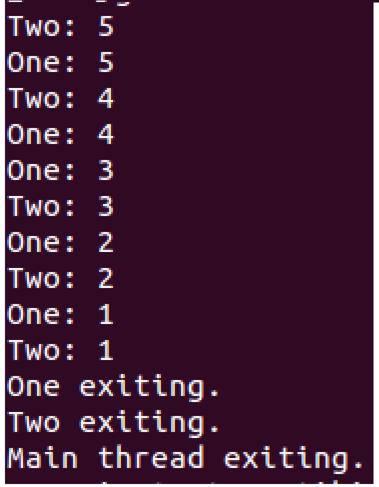
- A program used **suspend( )**, **resume( )**, and **stop( )**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread.
-

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    suspendFlag = false;
    t.start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }

    System.out.println(name + " exiting.");
  }

  synchronized void mysuspend() {
      suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}
```

```java
class SuspendResume {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Resuming thread One");
      ob2.mysuspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
      ob2.myresume();
      System.out.println("Resuming thread Two");
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

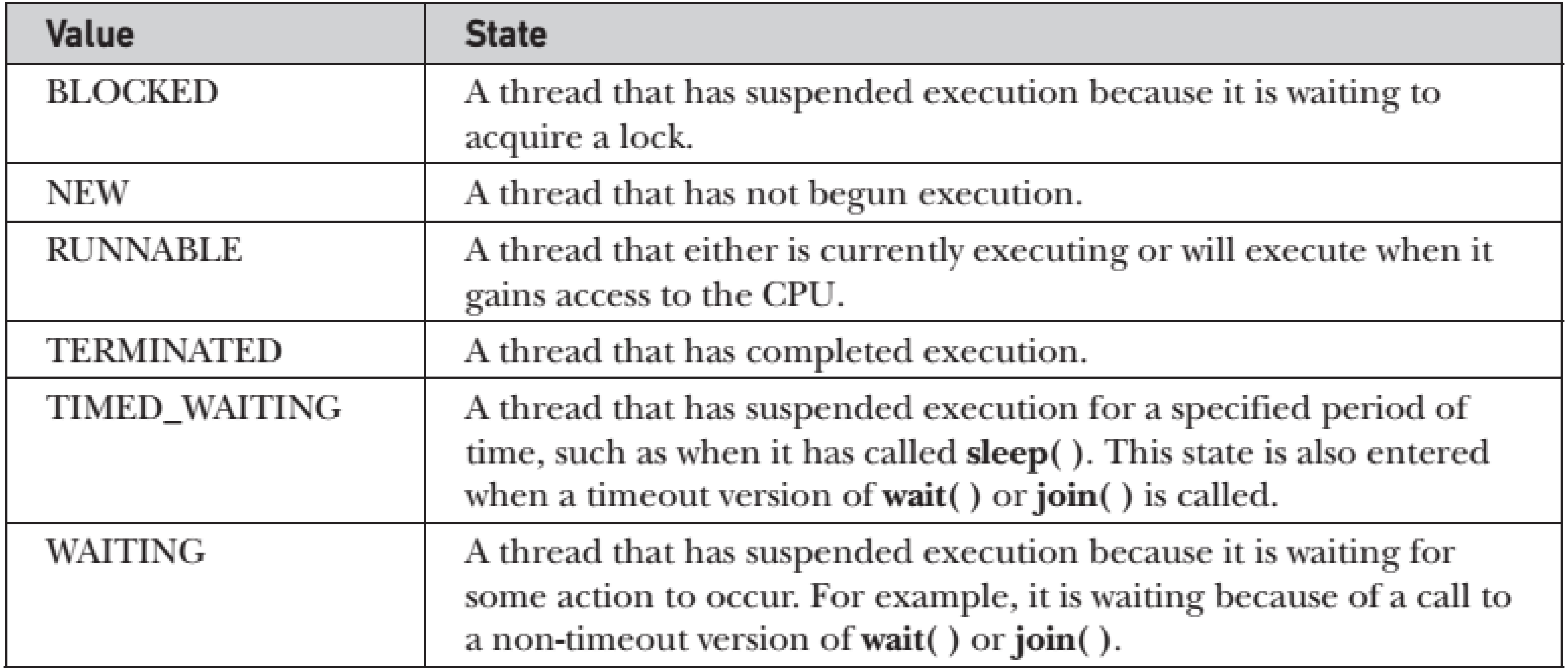


```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 15
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
```



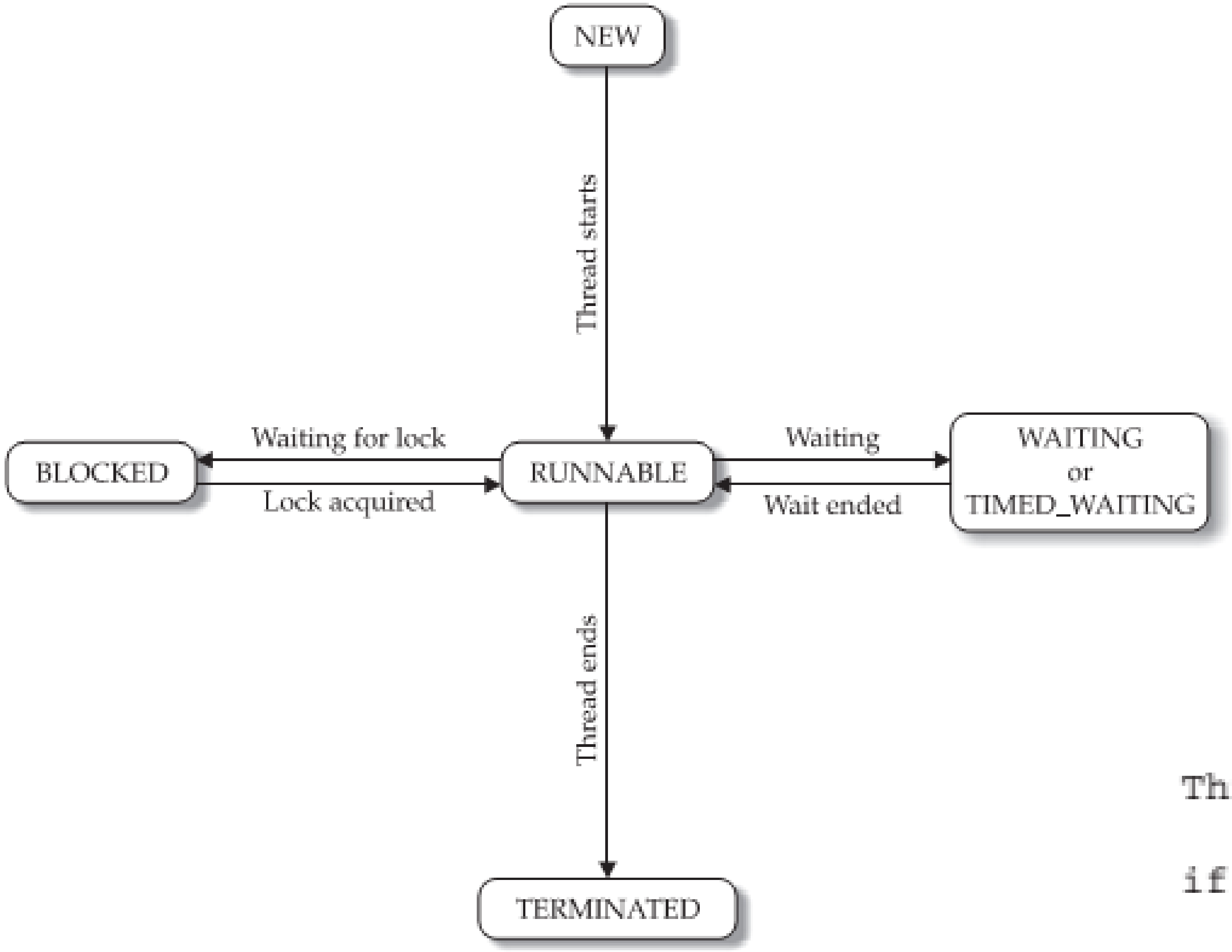
```
Two: 5
One: 5
Two: 4
One: 4
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One exiting.
Two exiting.
Main thread exiting.
```

Obtaining A Thread’s State



Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

Obtaining A Thread’s State



```
Thread.State ts = thrd.getState();

if(ts == Thread.State.RUNNABLE) // ...
```