Threads

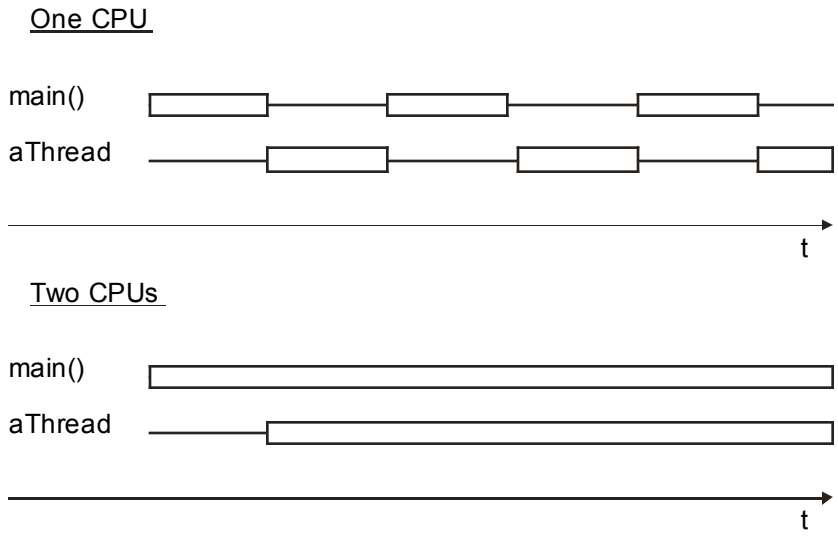
- What are Threads?
- Thread Lifecycle
- Synchronization

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What are Threads?

- A thread is a unit of execution within a Java program
- All Java programs have at least one thread on which main() runs, and the program can start multiple additional threads



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use Threads?

- If you have an application with opportunities for parallel processing, threads can reduce overall execution time
- Java GUI application must use background threads for lengthy tasks so the main thread can remain responsive to user input
- Server-based programs, e.g. Web servers and JSP/Servlet containers use threads to handle multiple simultaneous requests

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Not Use Threads?

- Threaded programs are hard to debug
- Threading introduces complexity and issues such as **deadlock** and **race conditions**
- Java threading potentially works differently on different host platforms

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

An Aside: Using Timers

- If all you need to do is to run a task at scheduled intervals, you can use the *timer* facility rather than manually creating a thread

```
1    import java.util.Timer;
2    import java.util.TimerTask;
3
4    public class Timers
5    {
6        private Timer timer;
7
8        public class MyTimerTask extends TimerTask
9        {
10           private int count = 0;
```


Timers are useful for cases where you need to do a one-time scheduled task or if you want to do a scheduled task repeatedly.

To use a timer, you subclass the java.util.TimerTask class and override the run() method. Then you create a java.util.Timer object and call its schedule() method, passing the reference to the TimerTask, the initial delay in milliseconds, and the repeat interval in milliseconds (if you want to do a one-time task, omit the initial delay argument).

Here we show using an inner class that extends TimerTask.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

An Aside: Using Timers, cont'd

```
11              public void run()
12              {
13                  System.out.println("Timer clicked!");
14                  count++;
15                  if (count == 5)
16                      timer.cancel();
17              }
18          }
19          public Timers()
20          {
21              MyTimerTask mtt = new MyTimerTask();
22              timer = new Timer();
23              timer.schedule(mtt, 0, 3000 );
24          }
25          public static void main(String[] args)
26          {
27              Timers t = new Timers();
28          }
29  }
```

Continuing the timer listing, here we show the inner TimerTask subclass overriding the run() method, which executes when the timer fires. In this simple example, we increment a count and then cancel (terminate) the timer after five intervals.

Note that timers internally use threads, so the code in the run() method should be thread-safe as we will discuss later in this chapter. The cancel() method essentially terminates the timer thread.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

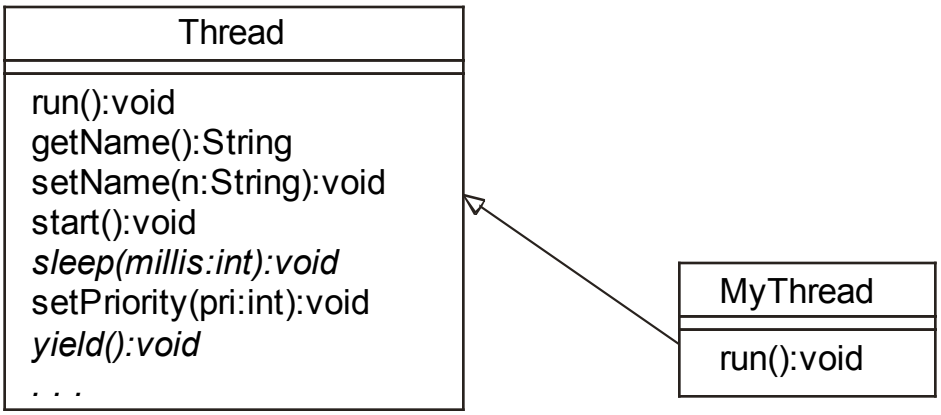
Host Platform Issues

- Some host platforms support threads where the operating system can stop any running thread to run another -- this is referred to as *preemption*
- Some host platforms do not support preemption -- instead, each thread must voluntarily yield the processor
- Host platform support for *thread priority* varies

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Defining a Thread: Technique 1

- One way to define a thread is to extend the java.lang. Thread class and override the **run** method
- Remember that a Java class can only extend one superclass



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Technique 1, cont'd

```
1    public class Threads2
2    {
3        public static class MyThread1 extends Thread
4        {
5            public MyThread1(String name)
6            {
7                super(name);
8            }
9
10           public void run()
11           {
12               while(true)
13               {
14                   System.out.println(
15                      this.getName());
16               }
17           }
18       }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

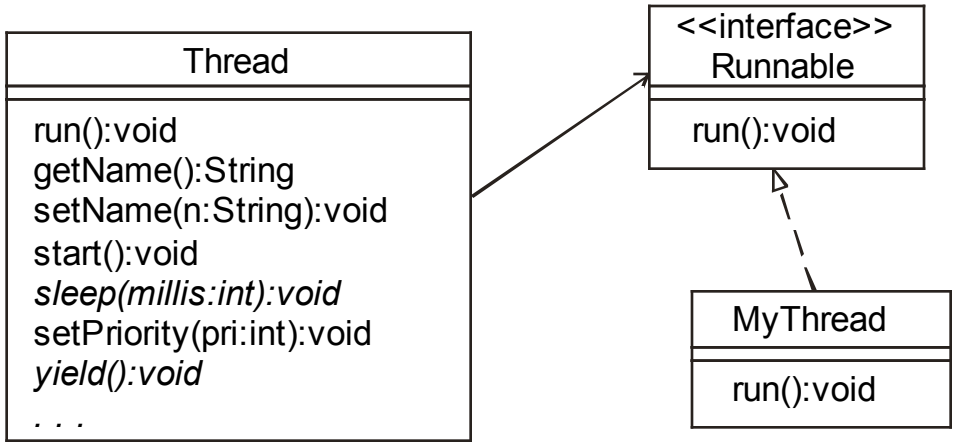
Technique 1, cont'd

```
19        public static void main(String[] args)
20        {
21            MyThread1 t1 = new MyThread1("thread1");
22            t1.start();
23        }
24    }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Defining a Thread: Technique 2

- Another way to define a thread is to implement that java.lang.Runnable interface and provide the **run** method
- This technique is useful for applets, which must extend javax.swing.Applet or java.applet.Applet



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Technique 2, cont'd

```
1    public class Threads2
2    {
3        public static class MyThread2
4           implements Runnable
5        {
6            public void run()
7            {
8                while(true)
9                {
10                   String name =
11                     Thread.currentThread().getName();
12                   System.out.println(name);
13               }
14           }
15       }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Technique 2, cont'd

```
16        public static void main(String[] args)
17        {
18            MyThread2 mt = new MyThread2();
19            Thread t2 = new Thread(mt);
20            t2.setName("thread2");
21            t2.start();
22        }
23    }
```

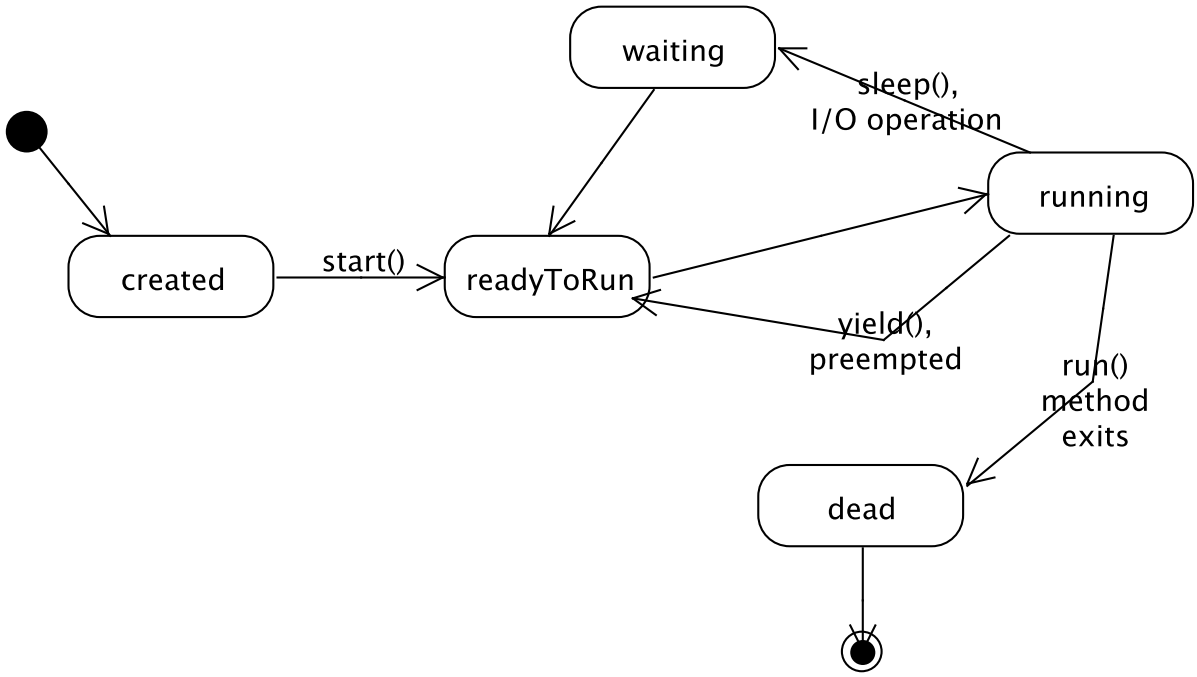

Note that in Java 8 or later, since Runnable is a functional interface, you can use a lambda to implement Runnable:

```
Thread myThread = new Thread(() ->
{
  while (true)
  {
    String name = Thread.currentThread().getName();
    System.out.println(name);
  }
});

myThread.setName("thread2");
myThread.start();
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thread Lifecycle

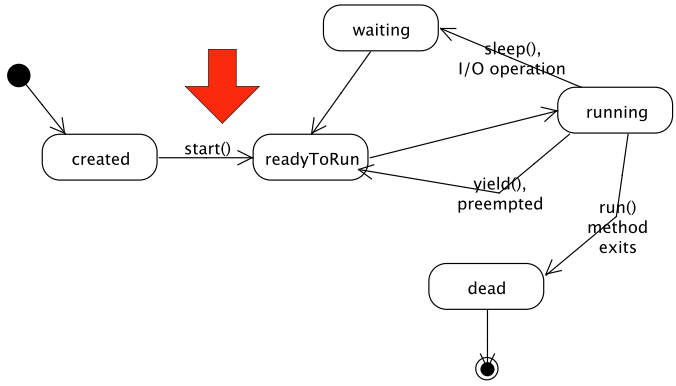


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Starting a Thread

- After you've create an instance of the class containing the **run** method, you can schedule the thread by calling the **start** method
- Note that the thread may or may not run immediately

```
1    public static void main(String[] args)
2    {
3        SomeThread t1 = new SomeThread();
4        t1.start();
5    }
```

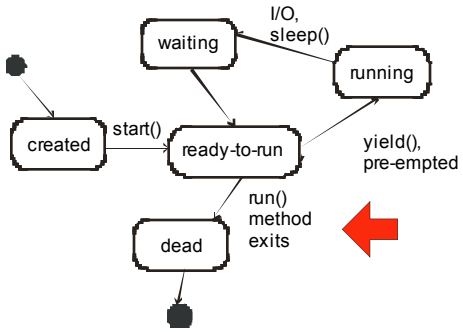


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Stopping a Thread

- Threads terminate themselves by reaching the end of the **run** method
- The **stop** method is considered unsafe and is deprecated

```
1    public class SomeThread extends Thread
2    {
3        private int count = 0;
4        public void run()
5        {
6            while(true)
7            {
8                if (++count == 5)
9                    return;
10           }
11       }
12   }
```



Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Relinquishing the Processor

- A thread can relinquish its CPU time by calling **sleep**, **yield** or by calling an I/O function that waits

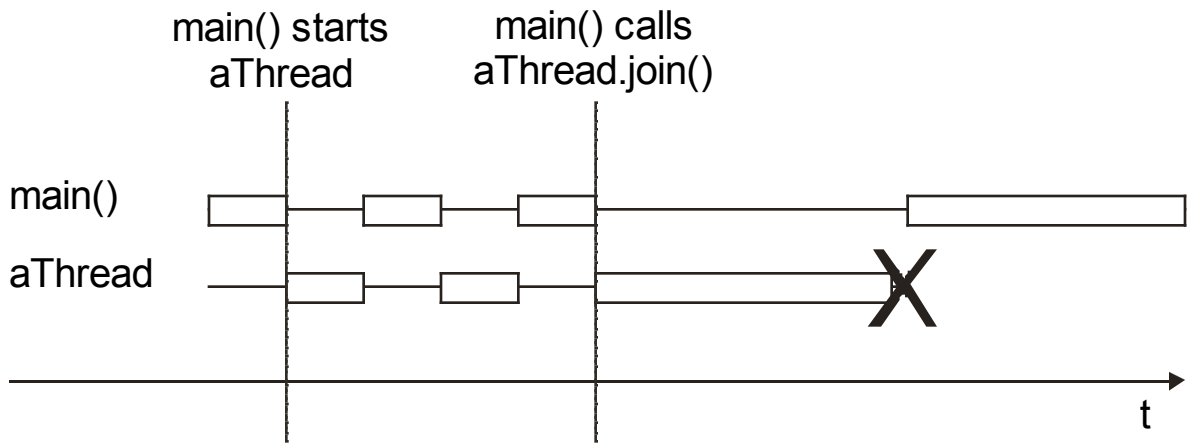
```
1    public class SomeThread extends Thread
2    {
3        public void run()
4        {
5           try
6           {
7              Thread.sleep(1000);
8           }
9           catch (InterruptedException e)
10          {
11          }
12       }
13   }
```

Sometimes, especially on host systems that do not pre-empt, a thread may voluntarily relinquish its CPU time. The sleep() method lets a thread "go away" for a specified number of milliseconds, while the yield() method lets a thread to pause and let other threads run.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using join()

- Using **thread joins**, a thread can wait for the termination of another thread



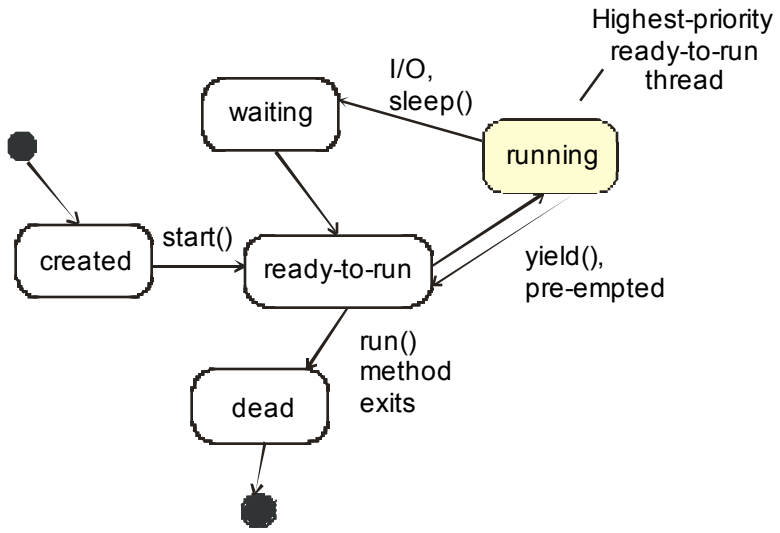
Having a thread wait for the termination of another thread is a useful technique. For example, the main thread could start a background thread to perform some lengthy operation or I/O and do other work in parallel. Then, when the main thread needs the results of the lengthy operation, main can call join() on the thread object -- join() with no arguments waits until the thread has terminated or returns immediately if the thread has already died.

Note that there are overloaded versions of join() that accept a timeout value -- these essentially let you poll for the thread termination event.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thread Priority

- To give the scheduler a hint about the relative importance of threads, you can assign thread priority
- The actual usage of thread priority depends on how the host operating system implements it



You can use thread priority to adjust the relative importance of threads. However, you should note that the way priority works is highly dependent on the host system, so it's tough to use thread priority in a truly portable fashion.

In Java, a thread's priority is represented by an integer, with three pre-defined values: Thread.MIN_PRIORITY, Thread.MAX_PRIORITY and Thread.NORM_PRIORITY.

Note that even though Java supports an integer range of priority values, most host operating systems have a much more limited number of priority levels. That's one of the things that makes using priority in a portable manner difficult. For best portability, it's a good idea to only use the three pre-defined values, which are likely to work on most systems.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thread Priority, cont'd

```
1    public class Priorities
2    {
3        public static class MyThread1 extends Thread
4        {
5            private char ch;
6            public MyThread1(char ch)
7            {
8                this.ch = ch;
9            }
10           public void run()
11           {
12               while(true)
13               {
14                   System.out.print(ch);
15               }
16           }
17       }
```


Here we show a thread implemented as an inner class with a constructor that accepts a character, which we store in an instance variable. The thread's run() method goes into an infinite loop, printing the character assigned when the thread object is constructed.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thread Priority, cont'd

```
18        public static void main(String[] args)
19        {
20            MyThread1 t1 = new MyThread1('a');
21            MyThread1 t2 = new MyThread1('b');
22            t1.setPriority(Thread.MIN_PRIORITY);
23            t2.setPriority(Thread.MAX_PRIORITY);
24            t1.start();
25            t2.start();
26        }
27    }
```

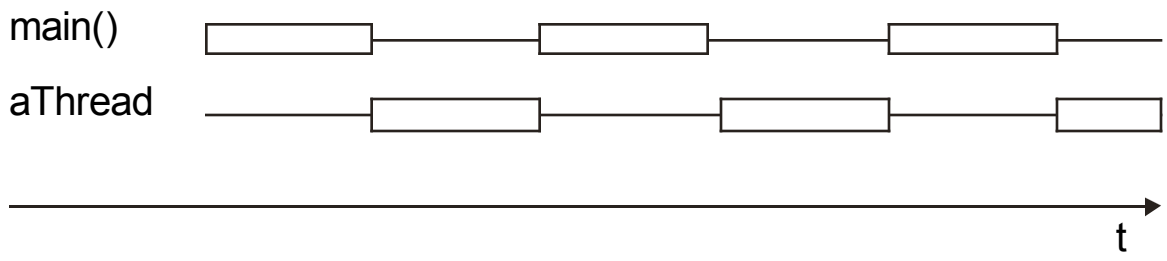

Continuing the listing, the main() method creates two Thread1 objects, using a different character. Main() then sets the priority of each thread and then makes them available to run.

If you run this program on Microsoft Windows XP, you will see very, very few 'a' characters. That's because Windows schedules the highest priority running thread almost exclusively, and since the thread never relinquishes CPU time, the higher-priority thread gets the vast majority of CPU cycles.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Time Slicing

- On some host platforms, if multiple ready-to-run threads have the same priority, they will each get equal chunks of CPU time

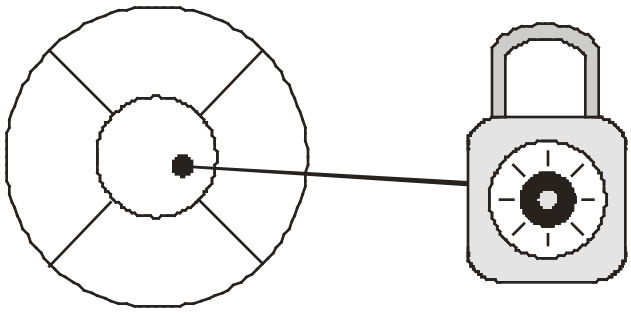


If you don't explicitly assign a thread priority, Java assigns Thread.NORM_PRIORITY.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Synchronization

- In a multithreaded program, you may need to ensure that multiple threads don't access a **serially re-usable resource** at the same time
- Java provides each object with a **lock** that a thread can acquire exclusively -- subsequent threads wait until the original thread releases the lock
- You can use the **synchronized** keyword to acquire the lock



On a host system that supports preemption (or a multi-CPU system), then it's possible that multiple threads can access a resource, such as a field, an open file, etc. at the same time. If the resource is not designed to be "thread safe", then this concurrent access can lead to problems that are difficult to diagnose and debug. Such problems are sometimes referred to as "race conditions".

These object thread locks as also known as "monitors".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Non-Thread-Safe Objects

- A *race condition* can occur if multiple threads read and write an object's fields in a non-atomic fashion
- Classes such as **String** are thread safe since they are immutable

```
1    public class Test
2    {
3       private int count;
4       public void someMethod()
5       {
6          count = 12;
7          . . .
8          count++;
9          . . .
10         System.out.println(count);
11      }
12   }
```


This code shows how multi-threading could lead to trouble. Looking at the code and assuming that the ". . ." chunks do not modify "count", you would think that the value 13 should be printed.

However, if this object is accessed by multiple threads, the method might instead print 12! Assume that there are two threads, and the first thread runs and sets count to 12, increments it, and then is preempted. Assume now that the second thread runs and sets count back to 12 and then itself is preempted in favor of the first thread (preemption can happen at any time). The first thread then executes System.out.println() with the current value of count being 12.

The real issue is that a given object has only one copy of its fields, regardless of how many threads access the object (i.e. the fields are shared across threads).

Also note that if the fields are read-only, then there's no problem. That's why we say that immutable object classes like String are thread safe.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Thread-Safe Objects

- The *synchronized* keyword acquires a lock on an object
- If you use **synchronized** on an instance method definition, Java uses the implicit *this* object to obtain the lock - the entire method is the lock boundary

```
1    public class Test
2    {
3       private int count;
4       public synchronized void someMethod()
5       {
6          count = 12;
7          . . .
8          count++;
9          . . .
10         System.out.println(count);
11      }
12   }
```

This code removes the race condition shown on the previous pag, using the "synchronized" keyword to ensure that only one thread at a time runs the method. In other words, when the first thread enters the synchronized method, it obtains the object's lock and if another thread tries to execute the method, the second thread will be blocked until the first thread exits the method, thus releasing the lock.

Using synchronized in this fashion uses the "this" implicit object reference.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Quiz: What Does This Program Print?

```
1    public class Test
2    {
3      static int a = 0;
4
5      public static class MyThread extends Thread
6      {
7        public void run()
8        {
9          System.out.println(a);
10       }
11     }
12
13     public static void main(String[] args)
14     {
15       new MyThread().start();
16       a = 1;
17     }
18   }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Synchronizing Blocks

- To mutually exclude threads for a portion of a method, use a **syncnronized block**

```
1    public class Test
2    {
3       private int count;
4       public void someMethod()
5       {
6          . . .
7          synchronized(this)
8          {
9             count = 12;
10            . . .
11            count++;
12            . . .
13            System.out.println(count);
14         }
15         . . .
16      }
17   }
```


If you have a lengthy method (or a method that performs lengthy processing), you might not want to lock an object for the entire method. You can use a so-called "synchronized block" to lock during a subset of the method's execution.

Note that the synchronized block requires an object reference. Here we use "this", but you can use any object reference.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Synchronizing Static Methods

- You can use the **synchronized** keyword on static methods - Java uses the so-called *class object* to provide the lock
- You can also supply the *class object* for synchronized blocks

```
1    public class Test
2    {
3      public synchronized static void method1()
4      {
5      }
6      public static void method2()
7      {
8         synchronized(Test.class)
9         {
10        }
11     }
12   }
```

Since static fields can, in general, be both read and written, you will need to protect them if you have multiple threads accessing the static fields.

The syntax for statics is much the same as for instances -- the primary difference is that for a synchronized block, you specify a reference to the runtime existence of the class, rather than an object reference.

Note also that you can write so-called "static blocks", typically to initialize static fields. In a static block, you can use synchronized blocks to protect.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Re-Entrant Locks

- If a thread already holds a lock on an object, calling another method on the same object does not need to re-acquire the lock
- This re-entrant locking scheme prevents deadlocks

```
1    public class Test
2    {
3      public synchronized void method1()
4      {
5         method2();
6      }
7      public synchronized void method2()
8      {
9      }
10   }
```

It's common for one method to call other methods -- even if each of the methods are synchronized, the invocation of subsequent methods will not require the thread to wait. That's because Java keeps track of the which thread owns the lock on an object.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- What is a race condition, and how can you protect against them?
- How does thread priority affect how a thread is scheduled?

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- About the benefits and dangers of writing multithreaded programs
- How to start, stop and synchronize threads

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum