

Threads

Threads Overview

- A thread is an execution.
- All programs have at least one thread of execution control.
- Java supports additional multiple, light-weight threads of execution.
- Running two separate programs at the same time is an example of a heavy-weight threads, each is a single program in execution, the internal *environments* of other heavy-weight threads are isolated from one another.
- A heavy-weight program thread can have multiple light-weight threads, each being switched to execute for a time then suspended while another thread executes.
- The prime distinction between heavy and light weight threads are that light threads share a portion of a common environment where heavy-weight threads are isolated.
- When light-weight threads are switched to allow one to execute there is less individual thread information to store and switch since much is common to all.

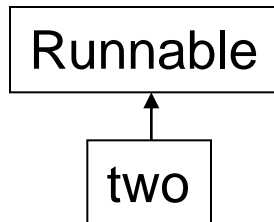
Execute one heavy thread.

```
public class one
{
    public static void main(String args[])
    {
        System.out.println("heavy one");
    }
}
```

Execute heavy and light thread.

```
public class one
{
    public static void main(String args[])
    {
        new two();
        System.out.println("heavy one");
    }
}
```

Thread Comparison



```
class two implements Runnable
{
    public two()
    {
        new Thread(this).start();
    }
    public void run()
    {
        System.out.println("light two");
    }
}
```

Java thread support

The key advantages of Java thread support are:

1. communication between threads is relatively simple
2. inexpensive in execution since light-weight, not necessary to save the entire program state when switching between threads
3. part of language, allows a program to handle multiple, concurrent operations in an operating system platform independent fashion.
4. Java on Windows and other *preemptive* operating systems ensure that all threads will get to execute occasionally by *preempting* thread execution (stopping the currently executing thread and running another) but this does not ensure that threads execute in any particular order.

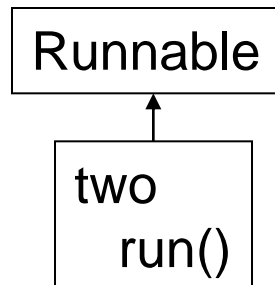
Thread Execution

1. Line 5 creates new *two* object.
2. Line 14 creates new *Thread* object.
3. Line 14 calls **start()** which calls **run()** and returns immediately.
4. Line 19 **run()** completes execution but does not return.

Possible execution sequences:

5, 14, 6, 18

5, 14, 18, 6



```
1. public class one
2. {
3.     public static void main(String args[])
4.     {
5.         new two();
6.         System.out.println("heavy one");
7.     }
8. }
9.
10. class two implements Runnable
11. {
12.     public two()
13.     {
14.         new Thread(this).start();
15.     }
16.     public void run()
17.     {
18.         System.out.println("light two");
19.     }
20. }
```

Execution Non-deterministic

```
1. class ST {
2.     public static void main(String args[]) {
3.         new Simple(1);
4.         new Simple(2);
5.         new Simple(3);
6.         new Simple(4);
7.     }
8.
9. class Simple implements Runnable {
10.    int n;
11.
12.    Simple(int n) {
13.        this.n = n;
14.        new Thread(this).start();
15.    }
16.
17.    public void run() {
18.        try { Thread.sleep(10); }
19.        catch(Exception e) {};
20.        System.out.println( n );
21.    }
22. }
```

4 Executions

>java ST

2

1

3

4

>java ST

2

1

3

4

> java ST

1

3

2

4

> java ST

1

2

4

3

Thread Race

Threads compete for the CPU. Below, no thread is able to complete before another obtains the CPU.

```
class Race {
    public static void main(String args[]) {
        new Simple("A");    new Simple("B");
        for (int i=1; i<=5; i++)
            System.out.println("main "+ i);
    }
}

class Simple implements Runnable {
    String name;
    Simple(String name) {
        this.name = name;
        new Thread(this).start();
    }
    public void run() {
        for (int i=1; i<=5; i++) {
            System.out.println( name+" "+i );
            try { Thread.sleep(10); }
            catch(Exception e) {};
        }
    }
}
```

Sequential Execution

```
A 1
A 2
A 3
A 4
A 5
B 1
B 2
B 3
B 4
B 5
main 1
main 2
main 3
main 4
main 5
```

Threaded Execution

```
main 1
B 1
A 1
main 2
B 2
A 2
B 3
A 3
main 3
B 4
A 4
B 5
main 4
A 5
main 5
```

Synchronization

- Thread execution can be:
 - independent of other threads (i.e. *parallel* or *asynchronous* execution)
 - dependent (i.e. *serialized* or *synchronous* execution) where one thread executes to the exclusion of the other threads
 - Object access is controlled in Java using the *synchronized* statement which limits object access to a single thread.
- The synchronization mechanism used by Java is termed a *monitor* (versus a semaphore, task, or other mechanism)
- A *monitor* allows only one thread to have access to an object at a time.
- The keyword *synchronized* defines a statement or method where one thread at a time has exclusive access to the object.

Race

```
1. class RT {
2.     public static void main(String args[]) {
3.         new Racer(1);
4.         new Racer(2);
5.         new Racer(3);
6.         new Racer(4);
7.     }
8. }
9. class Racer implements Runnable {
10.    int n;
11.
12.    Racer(int n) {
13.        this.n = n;
14.        new Thread(this).start();
15.    }
16.
17.    public void run() {
18.        System.out.print( "[" + n );
19.        try {Thread.sleep(10); }
20.        catch(Exception e) {}
21.        System.out.println( n + "]" );
22.    }
23. }
```

```
>java RT
[1[2[3[41]
3]
2 ]
4 ]
```

```
>java RT
[1[2[3[44]
3]
1]
2]
```

```
>java RT
[1[3[2[42]
1]
3]
4]
```

Non-cooperating Synchronized Example

```
1. class ST {
2.     public static void main(String args[]) {
3.         new Racer(1);
4.         new Racer(2);
5.         new Racer(3);
6.         new Racer(4);
7.     }
8. }
9. class Racer implements Runnable {
10.    static String common="common";
11.    int n;
12.    Racer(int n) {
13.        this.n = n;
14.        new Thread(this).start();
15.    }
16.    public void run() {
17.        synchronized( common ) {
18.            System.out.print( "[" + n );
19.            try {Thread.sleep(10); }
20.            catch(Exception e) {};
21.            System.out.println( n + "]" );
22.        }
23.    }
```

>java ST
C:\d
[11]
[22]
[44]
[33]

>java ST
[11]
[33]
[22]
[44]

Example

- In the example above the object *common* is accessible to the threads 1, 2, 3 and 4.
- In the *run* method access to *common* is *synchronized* so that only one thread can execute the code block:

```
class Racer implements Runnable {
    static String common="common";
    int n;

    public void run() {
        synchronized( common ) {
            System.out.print( "[" + n );
            try {Thread.sleep(10); }
            catch(Exception e) {};
            System.out.println( n + "]" );
        }
    }
}
```

Non-Cooperating Threads

- Example of three independent threads:
 - the main heavy-weight
 - *Producer* that puts onto a common queue
 - *Consumer* that gets from the common queue
- *Consumer* thread may *get* from the queue before the *Producer* thread *puts* onto the queue
- Any thread execution may be preempted at any time
- No guarantee a *put* or *get* finished before another thread runs
- Most obvious when the printed output is intermixed as in:
 - $[put: 1 \{get:0\}]$ rather than:
 - $[put: 1]$
 $\{get: 0\}$

Exercise 6

1. The *Producer* and *Consumer* are both threads, each having a *run()* method. Starting in the *run()* of each, list the sequence of lines executed that would print:

```
[put: 1{get:0}
}
```

2. Again, starting in the *run()* of each thread, list the sequence that would print:

```
{get:-1 [put: 0]
}
```

```
1. public class ProducerConsumer extends UserInterface{
2.     public void onClick(char c) {
3.         switch(c) {
4.             case 'S' : Q q = new Q( );
5.                     new Consumer(q);
6.                     new Producer(q);
7.                     break;
8.             case 'C' : Display.clear();
9.         }
10.    }
11. }
```

```
12. class Q {
13.     int n = 0;
14.     void put() {
15.         n = n + 1;
16.         Display.print("[put:  " + n);
17.         Display.println("]");
18.     }
19.     void get() {
20.         n = n - 1;
21.         Display.print("{get:  " + n);
22.         Display.println("}");
23.     }
24. }
```

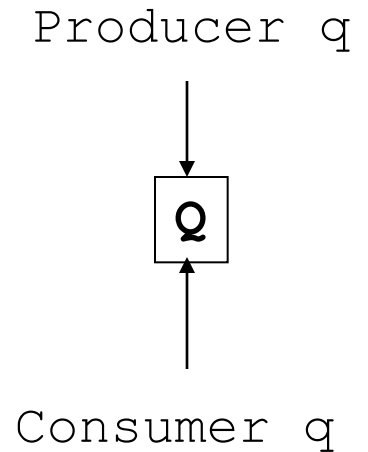
```
25. class Producer extends Thread {
26.     Q q;
27.     Producer( Q q ) {
28.         this.q = q;
29.         new Thread(this).start();
30.     }
31.     public void run() {
32.         for (int i=1; i<=10; i++)
33.             q.put(); // Produce
34.     }
35. }
```

```
36. class Consumer extends Thread {
37.     Q q;
38.     Consumer( Q q ) {
39.         this.q = q;
40.         new Thread(this).start();
41.     }
42.     public void run() {
43.         for (int i=1; i<=10; i++)
44.             q.get(); // Consume
45.     }
46. }
```

```

1. public class ProducerConsumer extends UserInterface {
2.     public void onClick(char c) {
3.         switch(c) {
4.             case 'S' : Q q = new Q( );
5.                 new Consumer(q);
6.                 new Producer(q);
7.             break;
8.             case 'C' : Display.clear();
9.         }
10.    }
11.}

```



```

12.class Q {
13.    int n = 0;
14.    void put() {
15.        n = n + 1;
16.        Display.print("[put: " + n);    Display.println("]");
17.    }
18.    void get() {
19.        n = n - 1;
20.        Display.print("{get: " + n);    Display.println("}");
21.    }
22.}

```

```

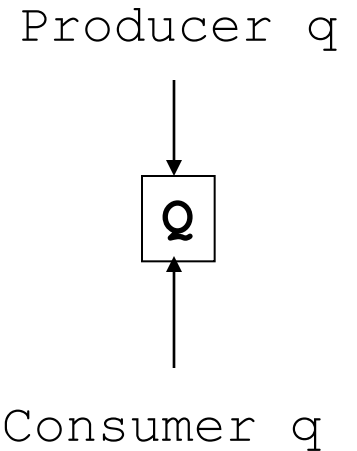
23.class Producer extends Thread {
24.    Q q;
25.    Producer( Q q ) {
26.        this.q = q;
27.        new Thread(this).start();
28.    }
29.    public void run() {
30.        for (int i=1; i<=10; i++) q.put(); // Produce
31.    }
32.}

```

```

33.class Consumer extends Thread {
34.    Q q;
35.    Consumer( Q q ) {
36.        this.q = q;
37.        new Thread(this).start();
38.    }
39.    public void run() {
40.        for (int i=1; i<=10; i++) q.get(); // Consume
41.    }
42.}

```



- Threads interact when simultaneously accessing common resource.
- Java supports a control mechanism called a *monitor* that allows only one thread at a time to execute a *synchronized* method on an object.
- By adding *synchronized* to methods, a thread entering such a method has exclusive access to that object in *synchronized* class methods.
- When a *synchronized* method completes, other threads may enter.
- For a *queue* object, the synchronized *put* and *get* method ensures either completes before another thread enters either, for an object.
- With two queue objects, one thread could have access to one queue object while another thread accessed the other queue object.
- This does not ensure that something has been put into the queue before a thread attempts to get it, that is another problem.
- Beyond cosmetics, the only change is the addition of:

```
synchronized void put ( );  
synchronized void get ( );
```

Monitor Behavior

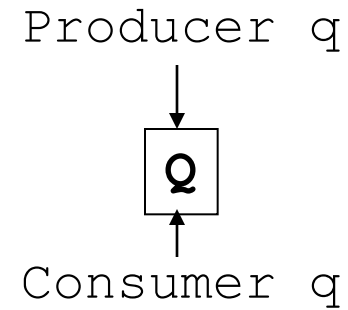
- Monitor controls access to a *synchronized* object.
- Monitor allows only one thread to access object; all other threads are blocked.
- When thread exits *synchronized* method other threads can access *synchronized* object.
- Blocked threads are automatically allowed to attempt to access *synchronized* object again.
- Threads can still access object through *unsynchronized* methods.
- Methods are not synchronized, objects are.
- No conflict when threads access different objects.

Monitor

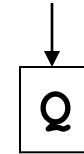
```
1. class Q {
2.   int n = 0;
3.   synchronized void put( ) {
4.     n = n + 1;
5.     Display.print("[put:  " + n);
6.     Display.println("]");
7.   }
8.   synchronized void get( ) {
9.     n = n - 1;
10.    Display.print("{get:  " + n);
11.    Display.println("}");
12.  }
13.}
```

```
Producer p = new Producer(q);
:
q.put();

Consumer c = new Consumer(q);
:
q.get();
```



Entered
Producer using q



Consumer blocked for q

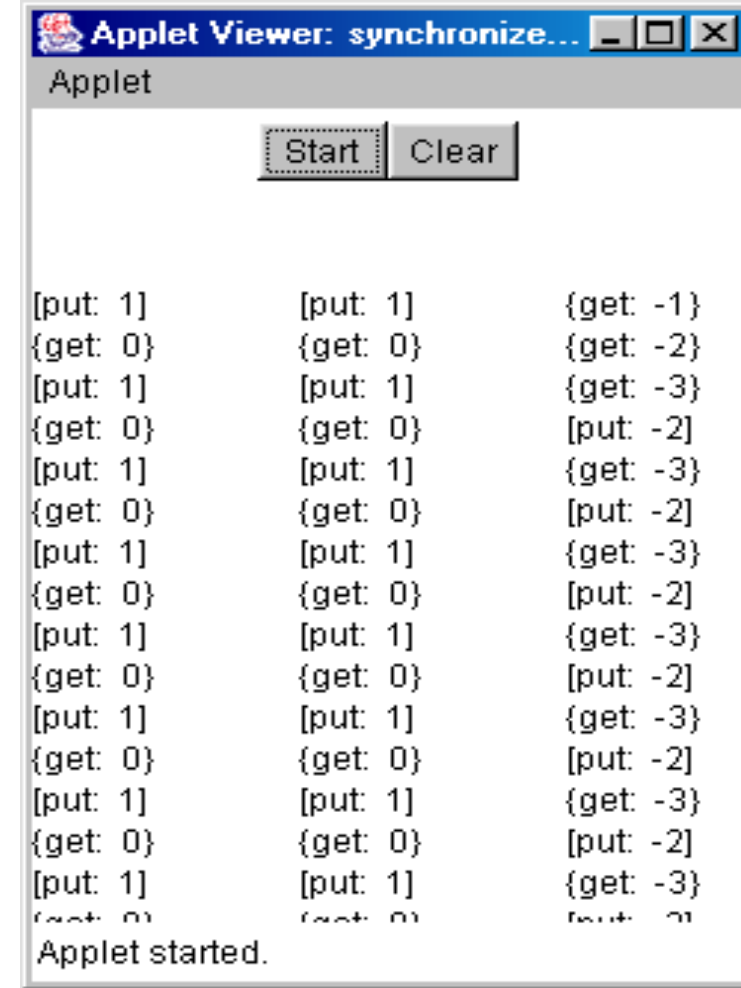
Consumer blocked
when common Q
object in use by
Producer

Synchronized Queue

```
1. class Q {
2.     int n = 0;
3.     synchronized void put( ) {
4.         n = n + 1;
5.         Display.print("[put:  " + n);
6.         Display.println("]");
7.     }
8.     synchronized void get( ) {
9.         n = n - 1;
10.        Display.print("{get:  " + n);
11.        Display.println("}");
12.    }
13. }
```

Exercise 7

1. Only one thread executes any synchronized method on a single object. Assume *Consumer* thread is executing Line 9 when *Producer* thread executes Line 3; list the lines then executed in the above.
2. Is synchronizing the *get* method but not the *put* method sufficient to prevent intermixed output?
3. Are there still problems between the *Consumer* and *Producer*?



wait, notify and notifyAll

- **wait** suspends thread until **notify** or **notifyAll** executed by another thread
- **notify** sends signal to one waiting thread
- Which thread notified is non-deterministic
- **notifyAll** sends signal to all waiting threads
- Only one notified thread can execute any *synchronized* method on an object at a time
- A notified thread continues from point where **wait** executed
- Thread enters a dead state when **run** completes
- The monitor controls access to a *synchronized* object
- No conflict when threads access different objects

Monitor Behavior

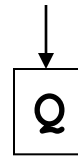
- Monitor controls access to *synchronized* object.
- Difference between threads *blocked* because monitor busy with another thread and threads that explicitly called **wait**
 - When *synchronized* method completes, blocked threads automatically can re-attempt object access
 - Threads that called **wait** can only proceed by another thread calling **notify** or **notifyAll**

Monitor

```
1. class Q {
2.     int n = 0;
3.     synchronized void put( ) {
4.         n = n + 1;
5.         Display.print("[put:" + n);
6.         Display.println("]");
7.         notify( );
8.     }
9.     synchronized void get( ) {
10.        while ( n==0 ) {
11.            try { wait( ); }
12.            catch (Exception e) { }
13.        }
14.        n = n - 1;
15.        Display.print("{get:  " + n);
16.        Display.println("}");
17.        notify( );
18.    }
19.}
```

1. Consumer enters get.
2. Consumer places itself on wait.
3. Producer enters put.
4. Producer sends notify().
5. Consumer released from wait.

Producer using q



Consumer waits for q

Consumer waits
when n==0

```
Producer p = new Producer(q); ... q.put();
Consumer c = new Consumer(q); ... q.get();
```

Cooperating Producer Consumer

- **wait()**; suspends thread until **notify()**; executed by another thread
- **notify()**; sends signal to waiting thread
- Continues from point where **wait()**; executed.

Exercise 8

1. **while (n==0)** is needed. Why?
2. What happens without line 7?
3. Without line 17?

```
1. class Q {
2.     int n = 0;
3.     synchronized void put( ) {
4.         n = n + 1;
5.         Display.print("[put:  " + n);
6.         Display.println("]");
7.         notify( );
8.     }
9.     synchronized void get( ) {
10.        while ( n==0 ) {
11.            try { wait(); }
12.            catch (Exception e) { }
13.        }
14.        n = n - 1;
15.        Display.print("{get:  " + n);
16.        Display.println("}");
17.        notify( );
18.    }
19. }
```

Cooperating Producer Consumer Busy Wait

Exercise 8

**3. Line 17 not
needed.**

Exercise 9

**4. This seems to
prevent
consuming
before
producing. How?**

**5. What is the
problem?**

```
1. class Q {
2.     int n = 0;
3.     synchronized void put( ) {
4.         n = n + 1;
5.         Display.print("[put:  " + n);
6.         Display.println("]");
7.
8.     }
9.     synchronized void get( ) {
10.        while ( n==0 );
11.
12.
13.
14.        n = n - 1;
15.        Display.print("{get:  " + n);
16.        Display.println("}");
17.
18.    }
19. }
```

Multiple inheritance: Thread class versus Runnable interface

```
class MultiThread      extends OneThread
                       implements Runnable
```

- At the start of our thread discussion we had a class **OneThread** that bounced a one ball around the screen. To bounce lots of balls we needed a class just like **OneThread** but that also supported *threads*.
- We could rework **OneThread** to do threads but better solution is to inherit from **OneThread** to bounce Balls, and a *thread* class; multiple inheritance.
- Java does not directly support multiple inheritance.
- Java solution is *interface*.
- Class **MultiThread** is a subclass of **Thread** by *implementing Runnable*.
- Implementing **Runnable** requires that class **MultiThread** must define method:

```
    public void run()
```
- **Thread** class implements **Runnable**.

Examples: Thread and Runnable

```
public class example {
    public static void main(String args[])
    {   new Simple();   }
}

class Simple implements Runnable {
    public Simple()
    {
        new Thread(this).start();
    }
    public void run()
    {   System.out.println("Simple");   }
}
```

```
public class example {
    public static void main(String args[])
    {   new Simple().start();   }
}

class Simple extends Thread
{
    public void run()
    {   System.out.println("Simple");   }
}
```

A simple example illustrates the syntactic differences between implementing Runnable versus inheriting Thread.

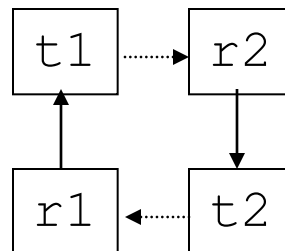
Thread inheritance is simpler to use but prohibits other inheritance.

Runnable allows inheritance (and implementing other interfaces) but is a little more complicated.

Both require a method:
public void run()

Deadlock

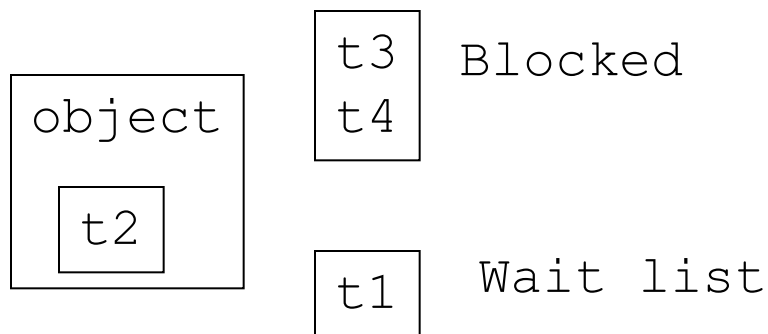
- Deadlock occurs when one thread cannot complete.
- One example is deadly embrace (or circular wait) where two threads each hold a resource required by the other.
 - t1 and t2 threads require both resources r1 and r2 to complete.
 - t1 thread holds r1 resource; t2 thread holds r2 resource.
 - t1 and t2 both deadlocked.



Threads

Deadlock

- t1,t2,t3, t4 – Execute synchronized method on common object.
- t1 executes wait();
- t3, t4 blocked while t2 holds object.
- notify() not executed.
- t1 never completes.



```

1.  waitObject wo = new waitObject();
2.  for(int i=0;i<10;i++) new ExampleThread( wo );

3.  class ExampleThread extends Thread
4.  { waitObject wo;
5.    ExampleThread( waitObject wo )
6.    {      this.wo = wo;
7.          new Thread(this).start();
8.    }
9.    public void run()
10.   {      wo.exit();
11.         wo.enter();
12.   }
13. }
14. class waitObject
15. { synchronized void enter( )
16.   { System.out.println("enter");
17.     notify( );
18.   }
19.   synchronized void exit( )
20.   { try { wait(); } catch(Exception e) {}
21.     System.out.println("exit");
22.   }
23. }

```

Deadlock

Exercise 9

1. This always deadlocks. Why?
2. Changing to the following still deadlocks. Why?

```

public void run()
{      wo.enter();
        wo.exit();
}

```

3. Would *notify()* at line 22 prevent deadlock? *notifyAll()*?

Consider with only one thread.

Multi-thread Summary

- Thread creation - Extend **Thread** class or implement **Runnable** interface
- Serialization – Monitor automatically limits execution of synchronized method on a shared object to one thread
- Basic thread control
 - Wait – Places a thread on wait list for object.
 - Notify – Releases an arbitrary thread from wait list for object.
 - NotifyAll – Releases all threads from wait list for object.
 - Sleep – Suspends thread execution for a minimum specified time.