Object Oriented Programming using Java 11

Multithreading & Thread Synchronization

Chittaranjan Pradhan
School of Computer Engineering,
KIIT University

Multi-Tasking

- Two kinds of multi-tasking:
  - process-based multi-tasking
  - thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor
- Processes are heavyweight tasks:
  - that require their own address space
  - inter-process communication is expensive and limited
  - context-switching from one process to another is expensive and limited

Multi-Tasking...

- Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text
- Threads are lightweight tasks:
  - they share the same address space
  - they cooperatively share the same process
  - inter-thread communication is inexpensive
  - context-switching from one thread to another is low-cost
- A thread is a single sequential flow of control within a program
- ***Java multi-tasking is thread-based***

Need of Multi-Threading

- Multithreading is a process of executing multiple threads simultaneously
- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum
- Multithreading enables programs to have more than one execution paths which execute concurrently. Each such path of execution is a thread
- There is plenty of idle time for interactive, networked applications:
  - the transmission rate of data over a network is much slower than the rate at which the computer can process it
  - local file system resources can be read and written at a much slower rate than can be processed by the CPU
  - of course, user input is much slower than the computer

Single-Threading

- In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next
- Single-threaded systems use event loop with pooling:
  - a single thread of control runs in an infinite loop
  - the loop pools a single event queue to decide what to do next
  - the pooling mechanism returns an event
  - control is dispatched to the appropriate event handler
  - until this event handler returns, nothing else can happen

Threads: Model

- Thread exist in several states:
  - ready to run
  - running
  - a running thread can be suspended
  - a suspended thread can be resumed
  - a thread can be blocked when waiting for a resource
  - a thread can be terminated
- ***Once terminated, a thread cannot be resumed***

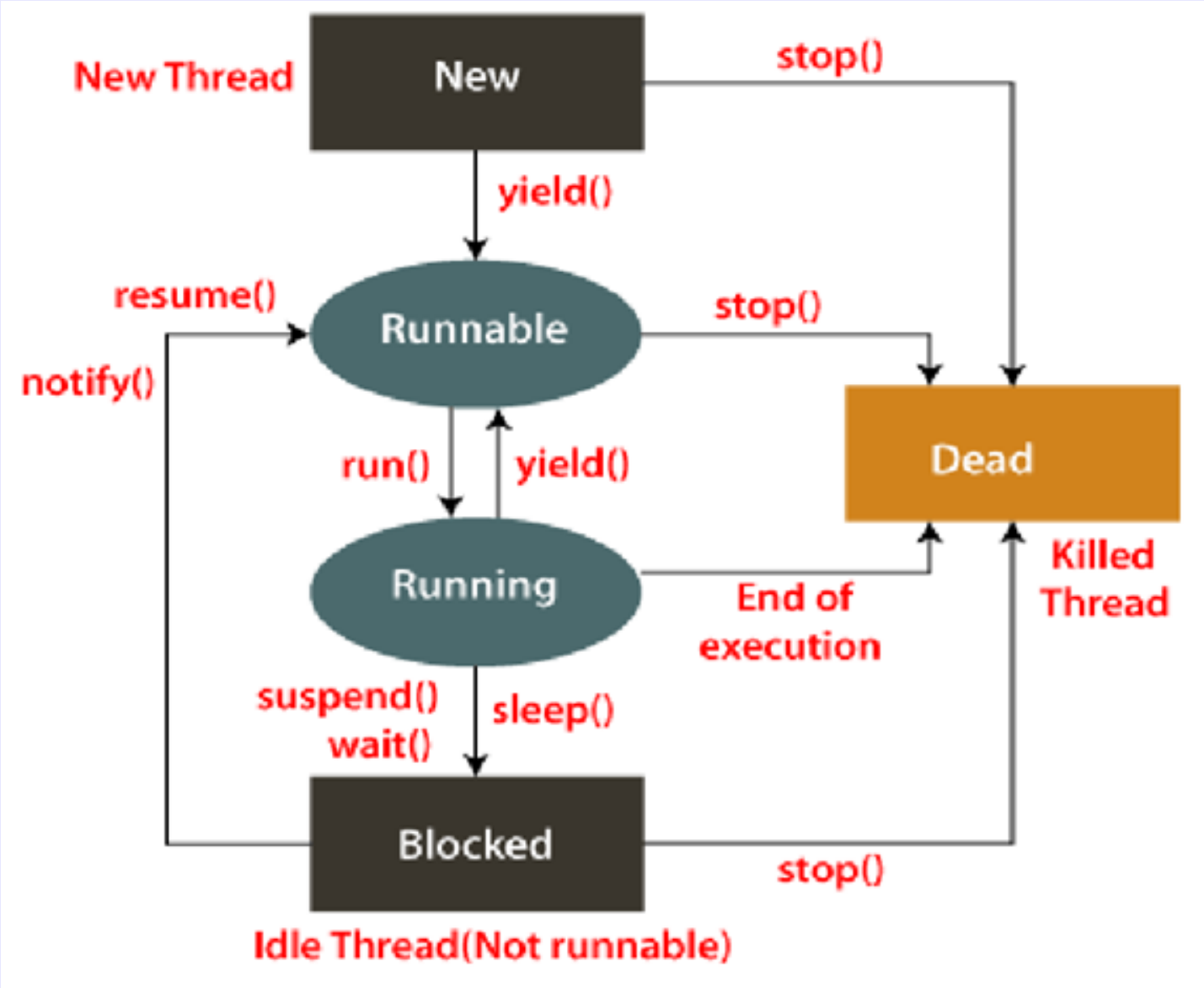
Life Cycle of a Thread (Thread States)

Life Cycle of a Thread (Thread States)

- **New**: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution
- **Active**: When a thread invokes the start() method, it moves from new state to active state. Active state contains two states within it: one is runnable, & other is running
- **Runnable**: A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state
- **Running**: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable
- **Blocked or Waiting**: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state

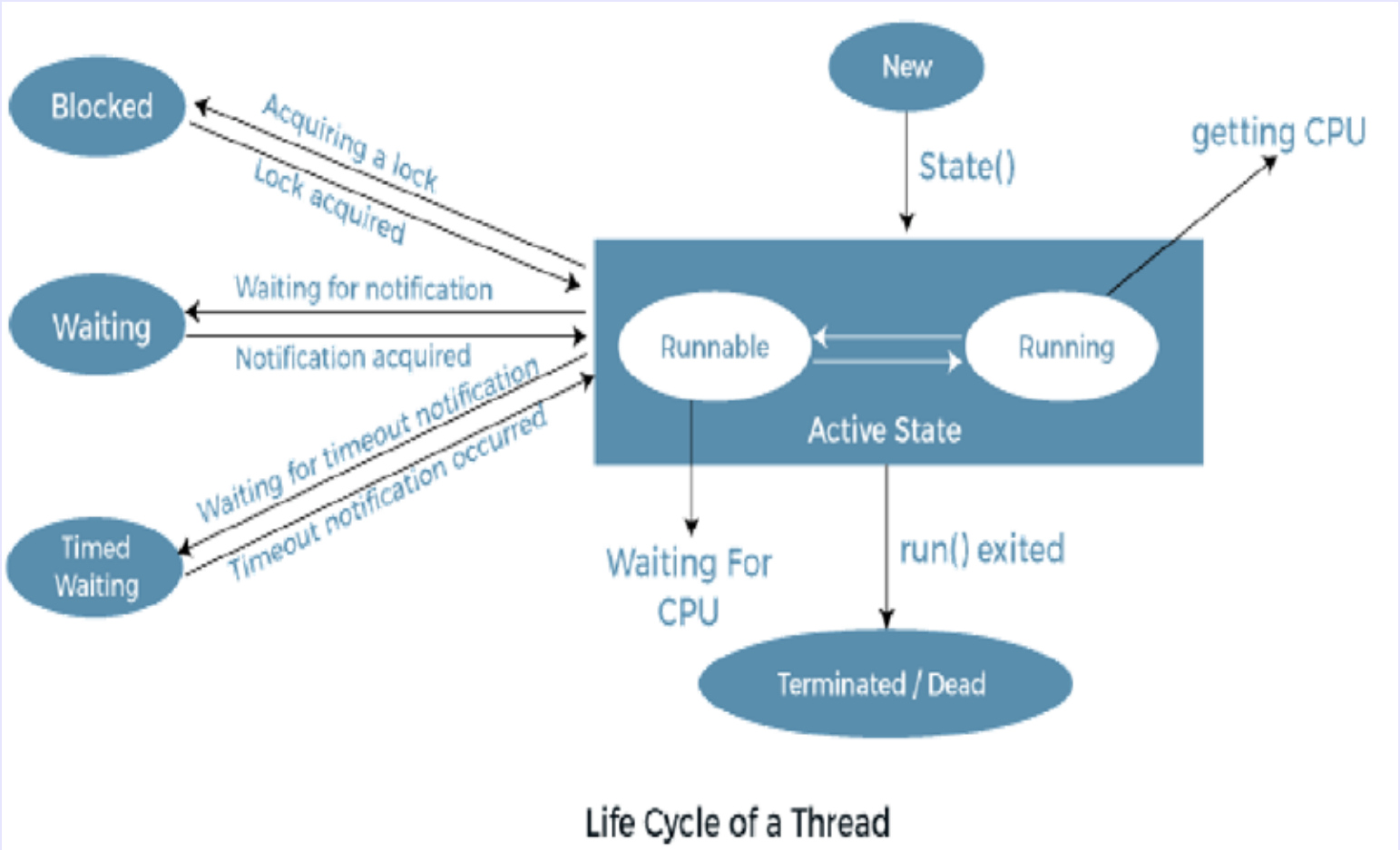
Life Cycle of a Thread (Thread States)...

Life Cycle of a Thread (Thread States)...



Life Cycle of a Thread (Thread States)...

Life Cycle of a Thread (Thread States)...



Life Cycle of a Thread

Life Cycle of a Thread (Thread States)...

Threads: Priorities

- Every thread is assigned priority - an integer number to decide when to switch from one running thread to the next (context-switching)
- Rules for context switching:
  - a thread can voluntarily relinquish control (sleeping, blocking on I/O, etc.), then the highest-priority ready to run thread is given the CPU
  - a thread can be preempted by a higher-priority thread - a lower-priority thread is suspended
- *When two equal-priority threads are competing for CPU time, which one is chosen depends on the operating system*

Thread Class

Thread Class

- To create a new thread a program will:
  - extend the Thread class, or
  - implement the Runnable interface
- **Thread** class encapsulates a thread of execution
- The whole Java multithreading environment is based on the **Thread** class

Method	Description
currentThread()	returns the reference of current thread
getName()	obtain a thread's name
getPriority()	obtain a thread's priority
isAlive()	determine if a thread is still running
join()	wait for a thread to terminate
run()	entry-point for a thread
setName()	changes the name of the thread
setPriority()	changes the priority of the thread
sleep()	suspend a thread for a period of time
start()	start a thread by calling its run method

Main Thread

- The main thread is a thread that begins as soon as a program starts
- The main thread:
  - is invoked automatically
  - is the first to start and the last to finish
  - is the thread from which other childthreads will be spawned
- It can be obtained through the

  public static Thread currentThread() method of *Thread*

Main Thread...

```
class CurrentThreadDemo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                try {
                        for (int n = 5; n > 0; n--) {
                                System.out.println(n);
                                Thread.sleep(1000);
                                }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted");
                        }
                }
        }
```

Main Thread...

- ***static void sleep(long milliseconds) throws InterruptedException***: Causes the thread from which it is executed to suspend execution for the specified number of milliseconds
- ***final String getName()***: Allows to obtain the name of the current thread
- ***final void setName(String threadName)***: Sets the name of the current thread

Creating a Thread

Creating a Thread

Two methods to create a new thread:

- by implementing the Runnable interface
- by extending the Thread class

New Thread: Runnable

To create a new thread by implementing the Runnable interface:

- create a class that implements the run method (inside this method, we define the code that constitutes the new thread):
  public void run()
- instantiate a Thread object within that class, a possible constructor is:
  Thread(Runnable threadOb, String threadName)
- call the start method on this object (*start calls run*):
  void start()

New Thread: Runnable...

```
class NewThread implements Runnable {
        Thread t;
        NewThread() {
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start();
                }
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                                }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
                System.out.println("Exiting child thread.");
                }
}
```

New Thread: Runnable...

New Thread: Runnable...

```
class ThreadDemo {
        public static void main(String args[]) {
                new NewThread();
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                                }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                        }
                System.out.println("Main thread exiting.");
                }
        }
```

New Thread: Extend Thread

- The second way to create a new thread:
  - create a new class that extends Thread
  - create an instance of that class
- Thread provides both run and start methods:
  - the extending class must override **run**
  - it must also call the **start** method

New Thread: Extend Thread...

New Thread: Extend Thread...

```
class NewThread extends Thread {
        NewThread() {
                super("Demo Thread");
                System.out.println("Child thread: " + this);
                start();
                }
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                                }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
        System.out.println("Exiting child thread.");
    }       }
```

New Thread: Extend Thread...

```
class ExtendThread {
        public static void main(String args[]) {
                new NewThread();
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                                }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                        }
        System.out.println("Main thread exiting.");
        }
    }
```

New Thread: Which Approach?

New Thread: Which Approach?

- implement Runnable if only run() is overridden
- extend Thread if other methods are also overridden

Multiple Threads

```
class MyThread1 extends Thread{
    public void run(){
        System.out.println("Thread1 running");
        }
    }
class MyThread2 extends Thread{
    public void run(){
        System.out.println("Thread2 running");
        }
    }
class MultiThread1{
    public static void main(String []args){
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        t1.start();
        t2.start();
    }
}
```

Multiple Threads...

Multiple Threads...

```
class MyThread1 extends Thread{
    public void run(){
        for(int i=5;i>0;i--)
            System.out.println("Thread1 "+i);
        }
    }
class MyThread2 extends Thread{
    public void run(){
        for(int i=5;i>0;i--)
            System.out.println("Thread2 "+i);
        }
    }
class MultiThread2{
    public static void main(String []args){
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        t1.start();
        t2.start();
    }
}
```

Multiple Threads...

Multiple Threads...

```
class NewThread implements Runnable {
        String name;
        Thread t;
        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start();
                }
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println(name + ": " + i);
                                Thread.sleep(1000);
                                }
                        }
```

Multiple Threads...

```
        catch (InterruptedException e) {
                        System.out.println(name + "Interrupted");
                        }
                System.out.println(name + " exiting.");
        }       }
class MultiThreadDemo {
        public static void main(String args[]) {
                new NewThread("One");
                new NewThread("Two");
                new NewThread("Three");
                try {
                        Thread.sleep(1000 );
                } catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                        }
                System.out.println("Main thread exiting.");
        }       }
```

Using isAlive and join Methods

Using isAlive and join Methods

- How can one thread know when another thread has ended?
- **final boolean isAlive()**: returns true if the thread upon which it is called is still running and false otherwise
- **final void join() throws InterruptedException**: waits until the thread on which it is called terminates

Using isAlive and join Methods...

Using isAlive and join Methods...

```
class NewThread implements Runnable {
        String name;
        Thread t;
        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start();
                }
        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.");
        }       }
```

Using isAlive and join Methods...

Using isAlive and join Methods...

```
class MultiThreadDemo {
        public static void main(String args[]) {
                NewThread ob1=new NewThread("One");
                NewThread ob2=new NewThread("Two");
                NewThread ob3=new NewThread("Three");
                System.out.println(ob1.t.isAlive());
                System.out.println(ob2.t.isAlive());
                System.out.println(ob3.t.isAlive());
                try {
                        System.out.println("Waiting to finish.");
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                } catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                        }
                System.out.println(ob1.t.isAlive());
                System.out.println(ob2.t.isAlive());
                System.out.println(ob3.t.isAlive());
                System.out.println("Main thread exiting.");
        }       }
```

Thread Priorities

- Priority is used by the scheduler to decide when each thread should run
- In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU
- In practice, the amount of CPU time that a thread gets depends on several factors besides its priority
- Setting thread's priority:
  final void setPriority(int level)
  where level specifies the new priority setting between:
  - MIN_PRIORITY (1)
  - MAX_PRIORITY (10)
  - NORM_PRIORITY (5)
- Obtain the current priority setting:
  final int getPriority()

Thread Priorities...

```
class ThreadDemo extends Thread{
   ThreadDemo(){ }
   public void run(){
      System.out.println("Thread Name: " + Thread.currentThread().getName()
         + ", Thread Priority: " +Thread.currentThread().getPriority());
      for(int i = 4; i > 0; i--){
         System.out.println("Thread: " + Thread.currentThread().getName() + ", " + i);
      }
      try{
         Thread.sleep(50);
      }
      catch (InterruptedException e){
         e.printStackTrace();
      }
   }
   public void start(){
      super.start();
   }
}
public class TestThread{
   public static void main(String args[]){
      ThreadDemo thread1 = new ThreadDemo();
      ThreadDemo thread2 = new ThreadDemo();
      ThreadDemo thread3 = new ThreadDemo();
      thread1.setPriority(Thread.MAX_PRIORITY);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread3.setPriority(Thread.NORM_PRIORITY);
      thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

Synchronization

- When several 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. This way is called synchronization

- Synchronization uses the concept of monitors:
  - only one thread can enter a monitor at any one time
  - other threads have to wait until the threat exits the monitor

- Java implements synchronization in two ways:
  - through the synchronized methods
  - through the synchronized statement

Synchronized Method

- All objects have their own implicit monitor associated with them
- To enter an object's monitor, call this object's synchronized method
- While a thread is inside a monitor, all threads that try to call this or any other synchronized method on this object have to wait
- To exit the monitor, it is enough to return from the synchronized method

Synchronized Method...

No Synchronization

```
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);
        }       }
```

Synchronized Method...

No Synchronization...

```
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");
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                  } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                        }
                }
        }
```

Synchronized Method...

Synchronization

- Output from the earlier program:
  [Hello[World[Synchronized]
  ]
  ]
- By pausing for one second, the call method allows execution to switch to another thread. A mix-up of the outputs from of the three message strings
- Here, nothing exists to stop all three threads from calling the same method on the same object at the same time
- Thus, we must serialize the access to call:
  class Callme {
  synchronized void call(String msg) {
  ... } }
- This prevents other threads from entering call while another thread is using it. The output is now as follows:
  [Hello]
  [World]
  [Synchronized]

Synchronized Statement

Synchronized Statement

- How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?
- Put calls to the methods of this class inside the synchronized block:

```
public void run() {
synchronized(target) {
target.call(msg);
} }
```

- This ensures that a call to a method that is a member of the *object* occurs only after the current thread has successfully entered the *object's* monitor

- The result is:
[Synchronized]
[World]
[Hello]

Inter-Thread Communication

Inter-thread communication relies on three methods in the Object class:

- **final void wait() throws InterruptedException**: tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify()
- **final void notify()**: wakes up the first thread that called wait() on the same object
- **final void notifyAll()**: wakes up all the threads that called wait() on the same object; the highest-priority thread will run first
- *All three must be called from within a **synchronized** context*

Incorrect Producer-Consumer problem

```
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++);
                        }
        }       }
```

Incorrect Producer-Consumer problem...

Incorrect Producer-Consumer problem...

```
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.");
                }
        }
```

Incorrect Producer-Consumer problem...

- Here is the output:
  Put: 1
  Got: 1
  Got: 1
  Put: 2
  Put: 3
  Get: 3
  ...

- *Nothing stops the producer from overrunning the consumer, nor the consumer from consuming the same data twice*

Correct Producer-Consumer problem

Correct Producer-Consumer problem

```
class Q {
        int n;
        boolean valueSet = false;
        synchronized int get() {
                if (!valueSet){
                        try{
                                wait();
                                }
                        catch(InterruptedException e) {
                                System.out.println("InterruptedException");
                                }
                        }
                System.out.println("Got: " + n);
                valueSet = false;
                notify();
                return n;
                }
```

Correct Producer-Consumer problem...

Correct Producer-Consumer problem...

```
        synchronized void put(int n) {
                if (valueSet){
                        try {
                                wait();
                                }
                        catch(InterruptedException e) {
                                System.out.println("InterruptedException");
                                }
                        }
                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++);
                }
  }     }
```

Correct Producer-Consumer problem...

Correct Producer-Consumer problem...

```
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.");
                }
        }
```