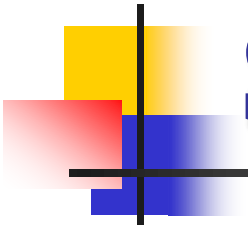
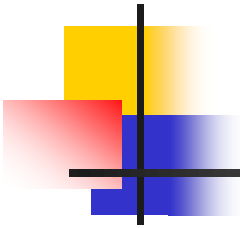


Session 7. Thread in Java

- Introduction to Java threads
- To create a thread
- extending Thread class
- implementing Runnable interface
- Daemon thread, join()
- Thread Synchronization
- wait(), notify() and notifyAll()

Edited by Trinh Thi Van Anh - PTIT



Introduction to Java threads

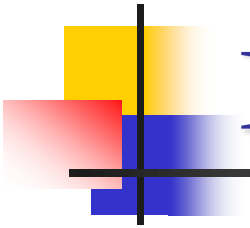
- A thread, in the context of Java, is the **path** followed when executing a program. All Java programs have at least one thread, known as the **main** thread, which is created by the JVM at the program's start, when the **main()** method is invoked with the main thread.
- When a thread is created, it is assigned a priority. The thread with higher priority is executed first, followed by lower-priority threads.

The Thread class and Runnable Interface

- A **thread** can be created in **two** ways:
  - By **extending Thread** class
  - By **implementing Runnable** interface.

Thread creation by extending Thread class

- One way of creating a thread is to create a thread. Here we need to create a new class that **extends** the **Thread** class.
- The class should override the **run()** method which is the **entry point** for the new thread as described above.
- Call **start()** method to start the **execution** of a thread.



Example (threaddemo)

- **HelloMain** is a class, that has a **main** method, it is a **main thread**.
- **HelloThread** is sub class, that extends from a **Thread class**.

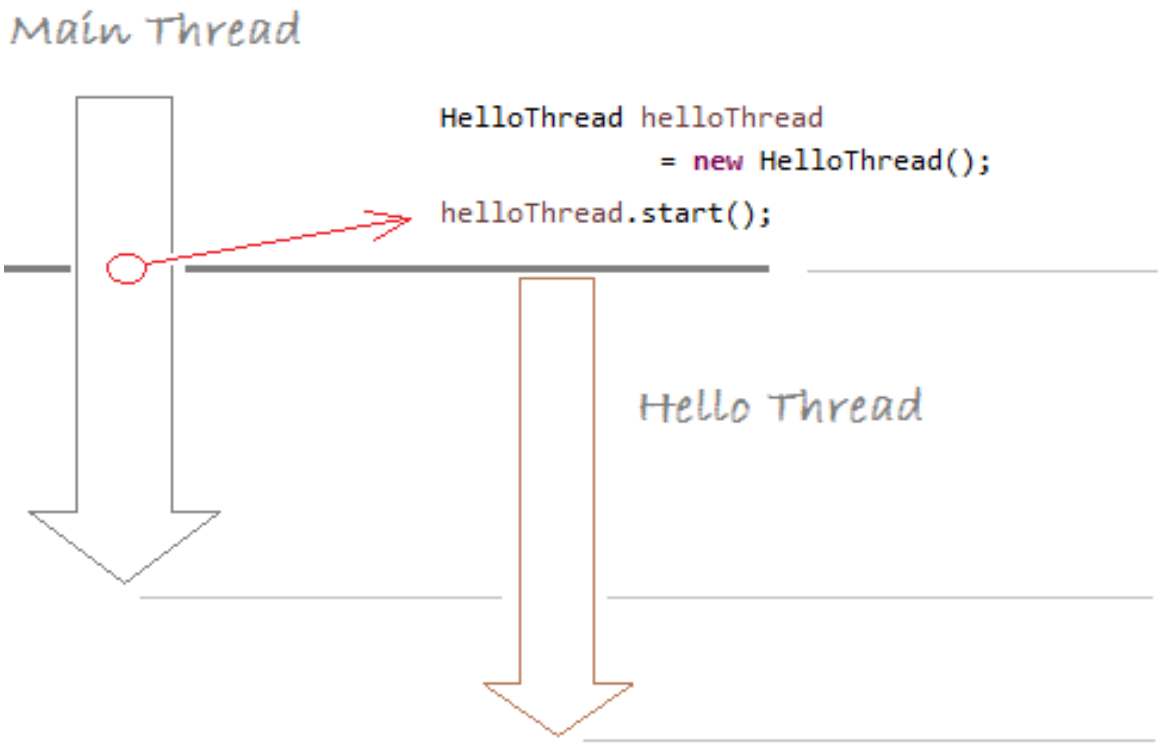
HelloThread.java

```
1.  public class HelloThread extends Thread{
2.     @Override
3.     public void run() {
4.        int index = 1;
5.        for (int i = 0; i < 10; i++) {
6.           System.out.println("  - HelloThread running " +
    index++);
7.           try {
8.              // sleep 1000 milli seconds.
9.              Thread.sleep(1000);
10.          } catch (InterruptedException e) {
11.          } }
12.       System.out.println("  - ==> HelloThread stopped");  }}
```

HelloMain.java

```
1.  public class HelloMain {
2.      public static void main(String[] args) throws
    InterruptedException {
3.          int idx = 1;
4.          for (int i = 0; i < 2; i++) {
5.              System.out.println("Main thread running " + idx++);
6.              Thread.sleep(2000);  }
7.          HelloThread   helloThread = new HelloThread();
8.          helloThread.start();// Chạy thread
9.          for (int i = 0; i < 3; i++) {
10.             System.out.println("Main thread running " + idx++);
11.             Thread.sleep(2000); }
12.         System.out.println("==> Main thread stopped"); }}
```

Output: class **HelloMain**



Console

<terminated> HelloMain [Java Applica

```
Main thread running 1
Main thread running 2
Main thread running 3
   - HelloThread running 1
   - HelloThread running 2
   - HelloThread running 3
Main thread running 4
   - HelloThread running 4
   - HelloThread running 5
Main thread running 5
   - HelloThread running 6
   - HelloThread running 7
==> Main thread stopped
   - HelloThread running 8
   - HelloThread running 9
   - HelloThread running 10
   - ==> HelloThread stopped
```

Exaple (clockdemo)

```
1.  static class Clock extends Thread{
2.   public Clock(){     }
3.   public void run(){
4.     while(true){
5.        SimpleDateFormat sdf = new
    SimpleDateFormat("hh:mm:ss");
6.        Calendar calendar= Calendar.getInstance();
7.        String str;
8.        str= sdf.format(calendar.getTime());
9.        lb.setText(str);
10.       try{
11.          sleep(1000);
12.       } catch(Exception e){
13.         System.out.println(e);
14.         }} } }
```

```
1.  public class ClockDemo extends JFrame{
2.     JFrame frame = new JFrame();
3.     static JLabel lb= new JLabel("",JLabel.CENTER);
4.     ClockDemo(){
5.  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
6.     setLayout(new FlowLayout(FlowLayout.CENTER));
7.     add(lb);
8.     setSize(200,100);
9.     setVisible(true);
10.    show();
11.    setLocationRelativeTo(null);
12.       }
13.       public static void main(String[] args) {
14.            new ClockDemo();
15.            Clock clock= new Clock();
16.            clock.start(); }
```

Thread creation by implementing Runnable Interface

- This is the second way of creating a class that **implements** the **Runnable** interface. We must need to give the **definition** of **run()** method.
- This run method is the entry point for the thread and thread will be alive till run method finishes its execution.
- Once the thread is created it will start running when **start()** method gets called.
  Basically start() method calls run() method implicitly.

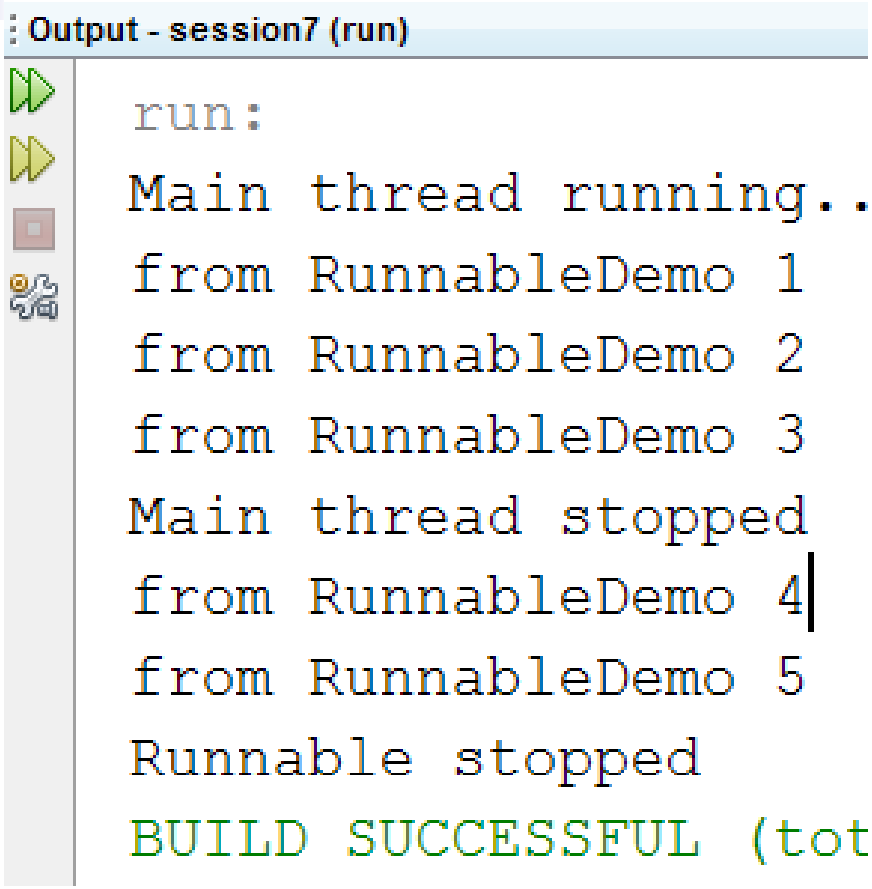
Example: threadrunnable (1)

```
public class RunnableDemo implements Runnable{
    @Override
   public void run() {
      int idx = 1;
      for (int i = 0; i < 5; i++) {
   System.out.println("from RunnableDemo " +
  idx++);
      try {
        Thread.sleep(2000);
          } catch (InterruptedException e) {
          }
        }}}
```

Example: (2)

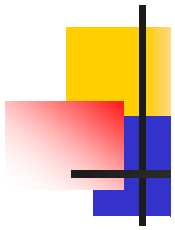
```
public class RunnableTest {
   public static void main(String[] args)
                throws InterruptedException {
 System.out.println("Main thread running..");
        // create thread from Runnable.
1. Thread thread = new Thread(new
   RunnableDemo());
     thread.start();
     Thread.sleep(5000);
     System.out.println("Main thread stopped");
    }
}
```

Example: (3)

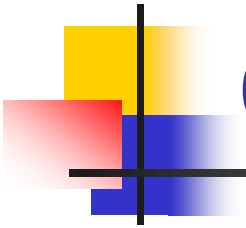

Output - session7 (run)
run:
Main thread running..
from RunnableDemo 1
from RunnableDemo 2
from RunnableDemo 3
Main thread stopped
from RunnableDemo 4
from RunnableDemo 5
Runnable stopped
BUILD SUCCESSFUL (tot

What will be the output of the program?

```
class s1 implements Runnable {
    int x = 0, y = 0;
    int addX() {x++; return x;}
    int addY() {y++; return y;}
    public void run() {
        for(int i = 0; i < 10; i++)
  System.out.println(addX() + " " +
  addY()+" ");
}
```



```
public static void main(String
  args[]){
     s1 run1 = new s1();
     s1 run2 = new s1();
     Thread t1 = new Thread(run1);
     Thread t2 = new Thread(run2);
     t1.start();
     t2.start();
     }
}
```



Output????

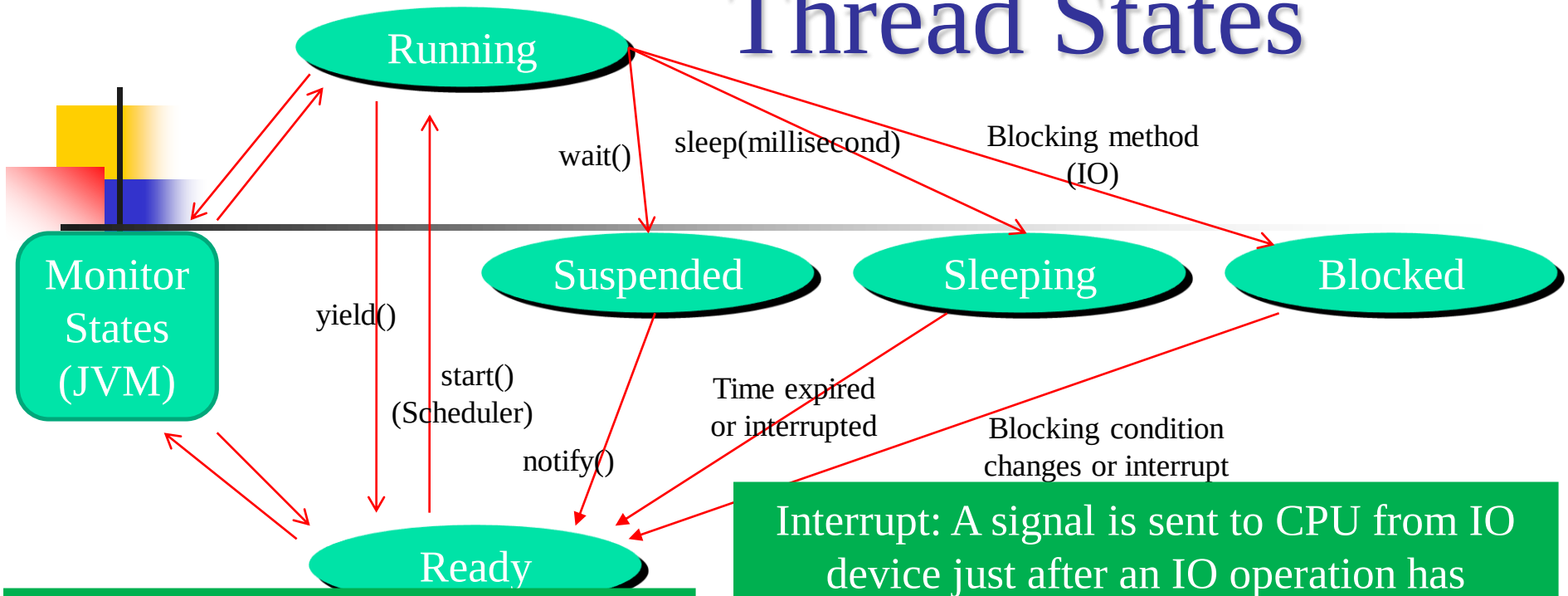
A. Compile time Error: There is no start() method

B. Will print in this order: `1 1 2 2 3 3 4 4 5 5...`

C. Will print but not exactly in an order (e.g: `1 1 2 2 1 1 3 3...`)

D. Will print in this order: `1 2 3 4 5 6... 1 2 3 4 5 6...`

Thread States



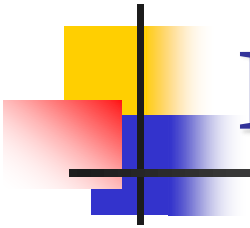
Interrupt: A signal is sent to CPU from IO device just after an IO operation has terminated.

Only one of ready threads will be chosen by the JVM scheduler at a time.

Ready: As soon as it is created , it can enter the **running** state when JVM's processor is assigned to it.

Running: It get full attention of JVM's processor which executes the thread's **run()** method

Dead: When the **run()** method terminates.



Daemon thread in Java (1)

- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.

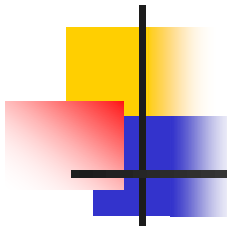
Daemon thread in Java (2)

- ***setDeamon(boolean)***

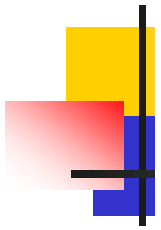
```
Thread thread = new MyThread();
thread.setDeamon(true);
thread.setDeamon(false);
```

Example (deamonthread)

```
public class NoneDeamonThread extends Thread{
    @Override
  public void run() {
   int i = 0;
   while (i < 10) {
      System.out.println("  - Hello from None Deamon
  Thread " + i++);
      try {
         Thread.sleep(1000);
         } catch (InterruptedException e) {} }
      System.out.println("\n==> None Deamon Thread
  ending\n");
  }}
```

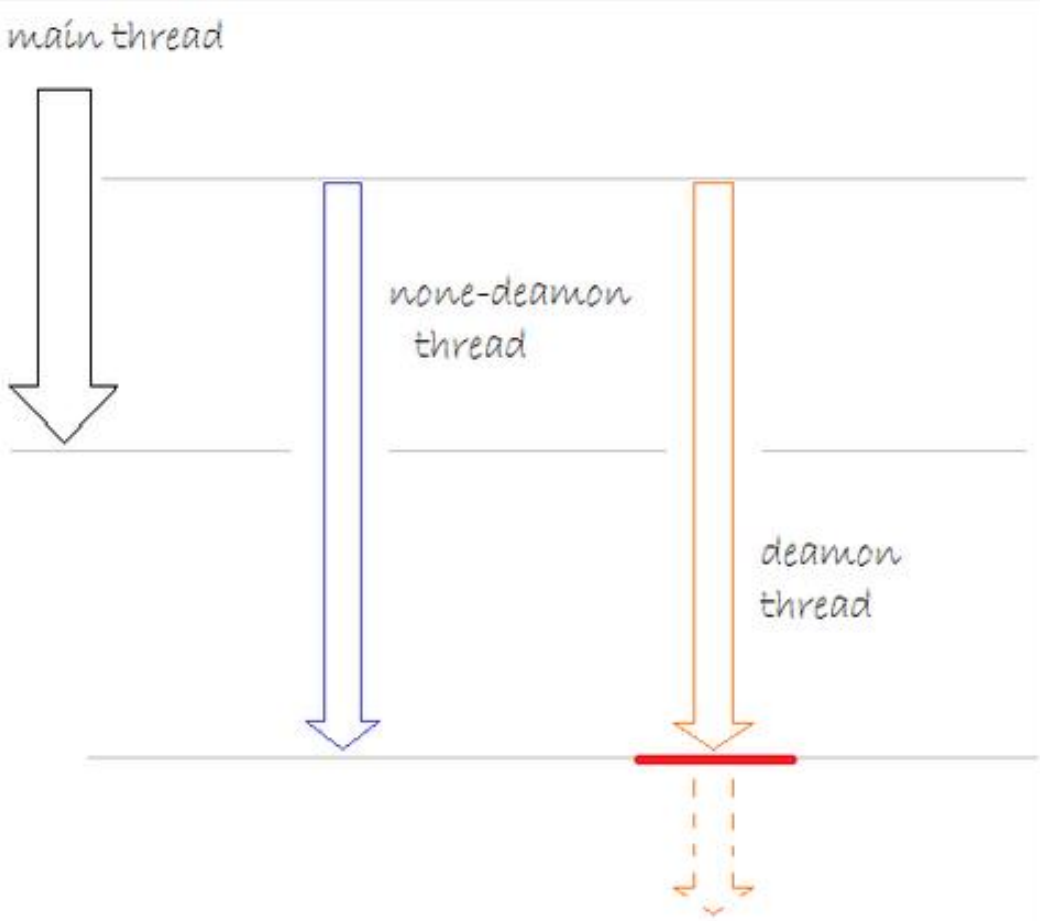


```
public class DeamonThread extends
  Thread{
     @Override
   public void run() {
    int count = 0;
    while (true) {
       System.out.println("+ Hello from
  Deamon Thread " + count++);
       try {
          sleep(2000);
     } catch (InterruptedException e) {
         }}}}
```



```
public class DaemonTest {
   public static void main(String[] args) {
     System.out.println("==> Main Thread
  running..\n");
     Thread deamonThread = new DeamonThread();
     deamonThread.setDaemon(true);
     deamonThread.start();
     new NoneDeamonThread().start();
       try {
           Thread.sleep(5000);
       } catch (InterruptedException e) {
       }
    System.out.println("\n==> Main Thread
  ending\n");
   }}
```

Output: class **DeamonTest**



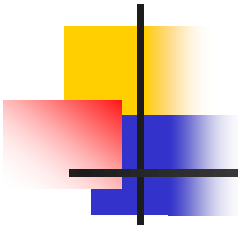
Console

<terminated> DaemonTest [Java Application] D:\D

```
==> Main Thread running..

+ Hello from Deamon Thread 0
  - Hello from None Deamon Thread 0
  - Hello from None Deamon Thread 1
+ Hello from Deamon Thread 1
  - Hello from None Deamon Thread 2
  - Hello from None Deamon Thread 3
+ Hello from Deamon Thread 2
  - Hello from None Deamon Thread 4

==> Main Thread ending

  - Hello from None Deamon Thread 5
+ Hello from Deamon Thread 3
  - Hello from None Deamon Thread 6
  - Hello from None Deamon Thread 7
  - Hello from None Deamon Thread 8
+ Hello from Deamon Thread 4
  - Hello from None Deamon Thread 9

==> None Demon Thread ending

+ Hello from Deamon Thread 5
```

join() method

- The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.
- **join()**
  - public final void join()throws InterruptedException
  - Public final void join(**long milliseconds**)throws InterruptedException

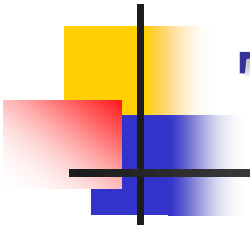


Example

```
class TestJoin extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
```

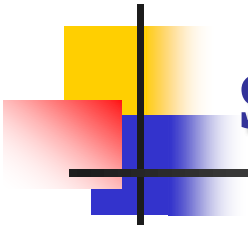
```
public static void main(String args[]){
    TestJoin t1 = new TestJoin();
    TestJoin t2 = new TestJoin();
    TestJoin t3 = new TestJoin();
    t1.start();
    try {
       t1.join();
    } catch (Exception e) {
        System.out.println(e);
    }
         t2.start();
         t3.start();
     }
}
```

```
public static void main(String args[]){
    TestJoin t1 = new TestJoin();
    TestJoin t2 = new TestJoin();
    TestJoin t3 = new TestJoin();
    t1.start();
    try {
       t1.join(1500);
    } catch (Exception e) {
        System.out.println(e);
    }
        t2.start();
        t3.start();
     }
}
```



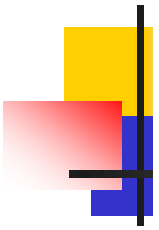
Thread Synchronization

- Thread synchronization is the concurrent execution of two or more threads that share critical resources. Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.



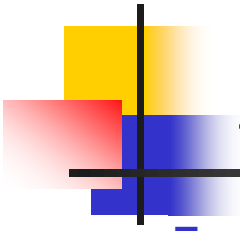
synchronization in Java

- Let’s suppose we have an online banking system, where people can log in and access their account information. Whenever someone logs in to their account online, they receive a separate and unique thread so that different bank account holders can access the central system simultaneously.



```
public class BankAccount {
   int accountNumber;
   double accountBalance;
public boolean transfer (double
   amount){
   double newAccountBalance;
   if( amount > accountBalance) {
      return false; }
   else {
      newAccountBalance = accountBalance
   - amount;
      accountBalance = newAccountBalance;
      return true;
} }
```

```
public boolean deposit(double amount) {
   double newAccountBalance;
   if( amount < 0.0) {
        return false;
   } else {
   newAccountBalance = accountBalance +
    amount;
   accountBalance = newAccountBalance;
   return true;
} }
```



A race condition

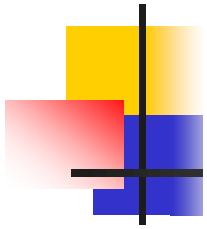
- Let’s say that there’s a husband and wife - Jack and Jill - who share a joint account. They currently have $1,000 in their account. They both log in to their online bank account at the same time, but from different locations.
- They both decide to deposit $200 each into their account at the **same time**.
- So, the total account balance after these 2 deposits should be $1,000 + ($200 * 2), which equals $1,400.
- Let’s say Jill’s transaction goes through first, but Jill's thread of execution is switched out

Synchronization fixes race conditions

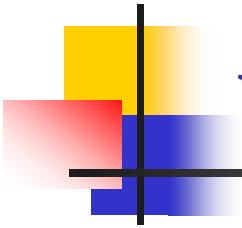
- In the code below, all we do is add the **synchronized** keyword to the transfer and deposit methods
  - public **synchronized** boolean transfer (double amount){}
  - public **synchronized** boolean deposit(double amount){}
- This means that only **one thread** can **execute** those functions **at a time**

```
public class SynMethod {
    public static void print(String s){
        String
  name=Thread.currentThread().getName();
        System.out.println(name+" - "+s);
    }
    public void takeaPen(){
        print("Take a pen");
        print("be writing");
        try{
            Thread.sleep(2000);
        }catch(Exception e){
            e.fillInStackTrace();
        }
        print("finish writing!");
    }
```

```
public static void main(String[] agrs){
    final SynMethod bb=new SynMethod();
    Runnable runA = new Runnable(){
    public void run(){
        bb.takeaPen();
            }};
    Thread threadA = new
  Thread(runA,"threadA");
    threadA.start();
    try{
        Thread.sleep(200);
    }catch(Exception e){
        e.fillInStackTrace();}
```

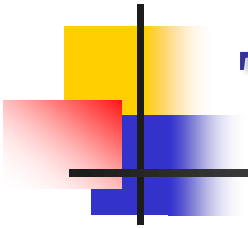


```
Runnable runB = new Runnable(){
    public void run(){
        bb.takeaPen();
            }
          };
  Thread threadB = new
  Thread(runB,"threadB");
    threadB.start();
    }
}
```



wait(), notify() and notifyAll()

- Multithreading replaces event loop programming by dividing your tasks into discrete and logical units.
- Three methods: **wait( )**, **notify( )**, and **notifyAll( ),** these methods are implemented as **final** methods in **Object** and can be called only from within a **synchronized** method.
  - final void wait( ) throws InterruptedException
  - final void notify( )
  - final void notifyAll( )

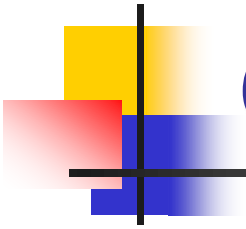


The rules for using three methods

- **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( )**.
- **notify( )** wakes up the first thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

The sample program incorrectly implements (usenotify)

- It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.



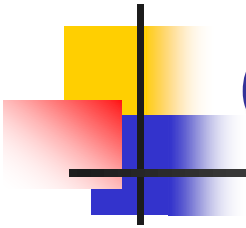
Class Q

```
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/ Consumer

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

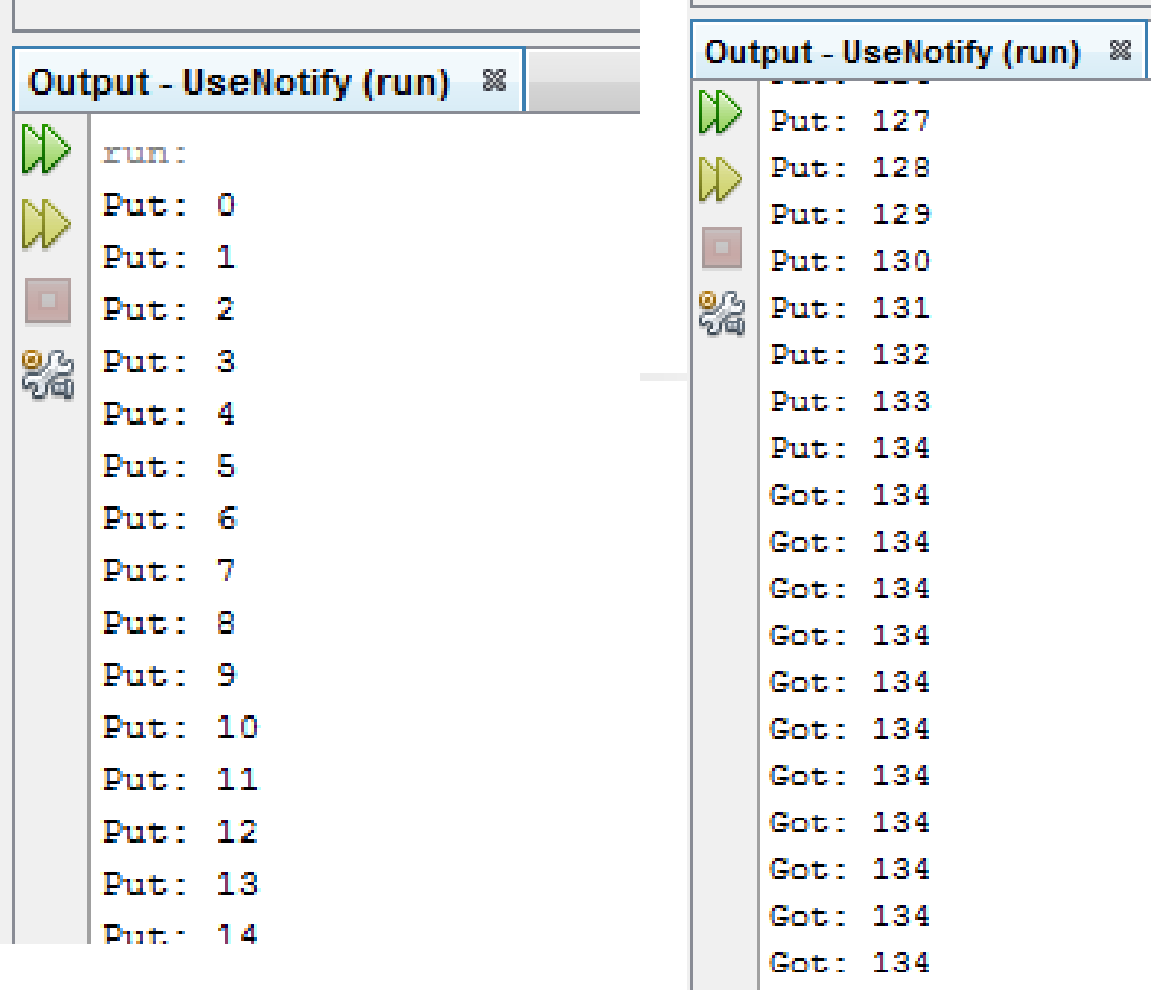
```
class Consumer
  implements Runnable
  {
  Q q;
  Consumer(Q q) {
  this.q = q;
  new Thread(this,
  "Consumer").start();
  }
  public void run() {
  while(true) {
  q.get();
  }
  }
  }
```



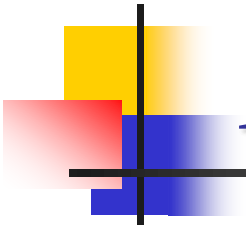
Class PC

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

Running (PC)



- Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice.

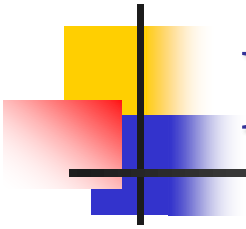


A correct implementation

- The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions
- Inside **get( )**, **wait( )** is called.
- After the data has been obtained, **get( )** calls **notify( )**.

Inside **get( )**

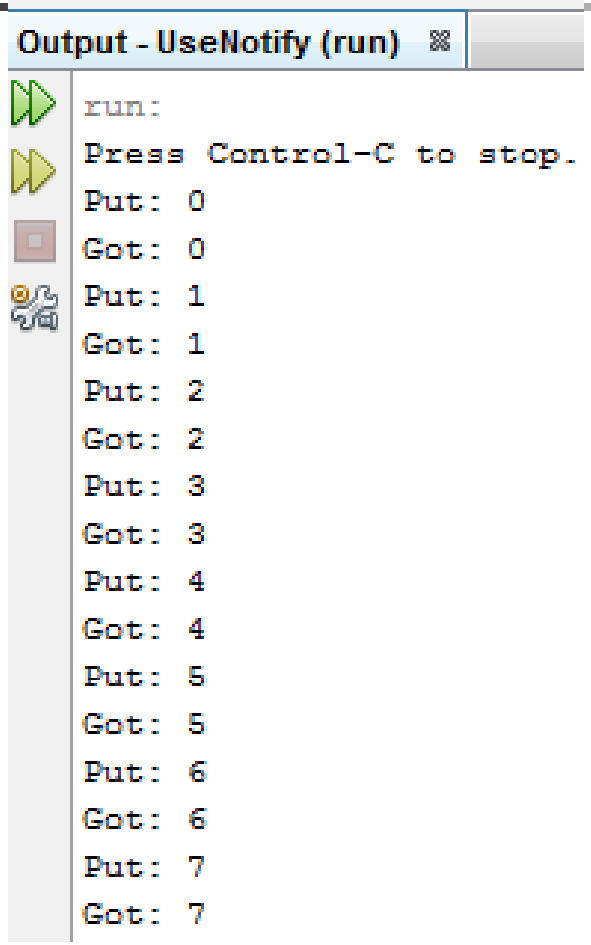
```
synchronized int get() {
try {
notify();
wait();
} catch (InterruptedException e) {
System.out.println("InterruptedException
  caught");}
System.out.println("Got: " + n);
return n;}
```



Inside **put( )**

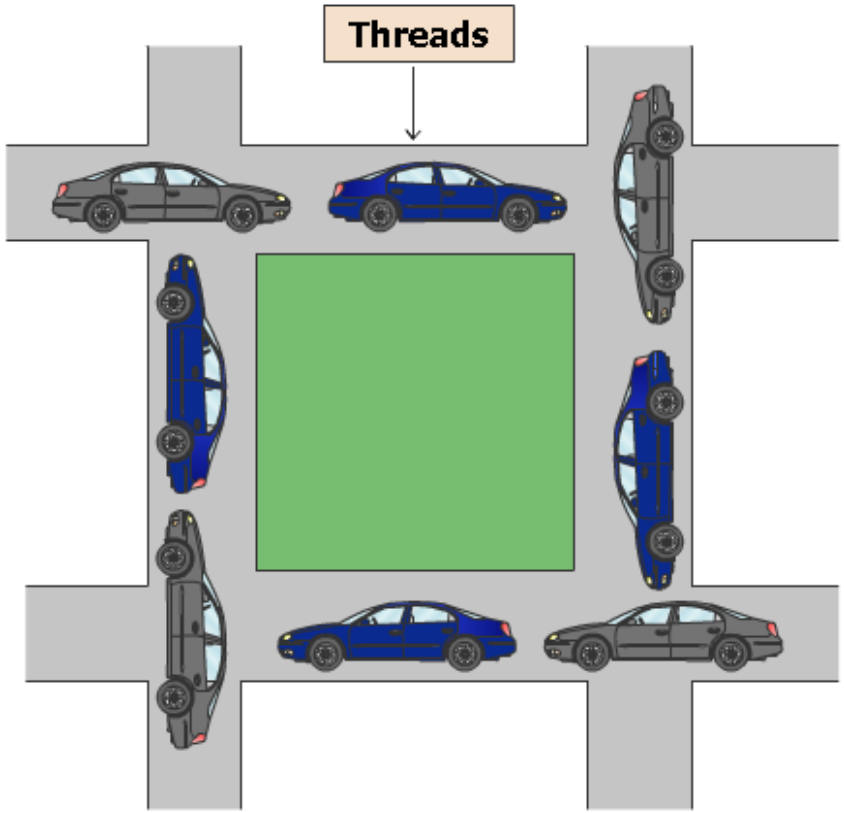
```
synchronized void put(int n) {
try {
notify();
wait();
} catch (InterruptedException e) {
System.out.println("InterruptedExcepti
  on caught");}
this.n = n;
System.out.println("Put: " + n);}
```

Running (PCFixed)



Deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other → All threads in a group halt.
- <u>When does deadlock occur?</u>
- There exists a circular wait the lock that is held by other thread.



Nothing can ensure that DEADLOCK do not occur.

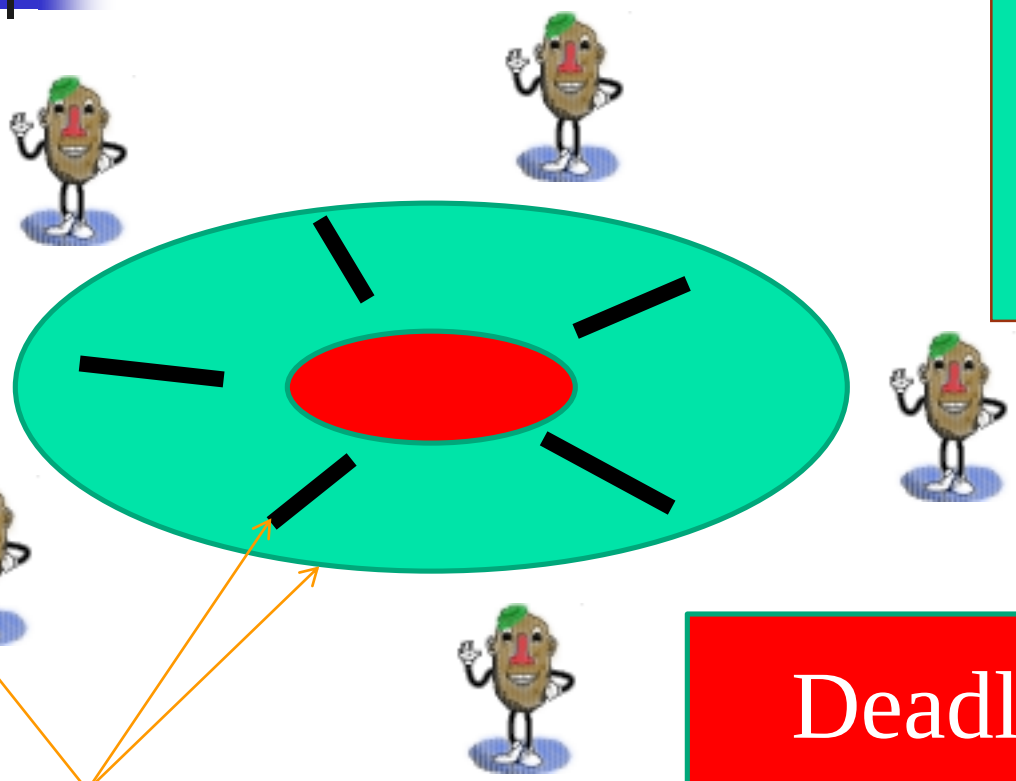
Deadlock Demo.

```
public class DeadLockDemo implements Runnable  {
   DeadLockDemo assistance=null; // giám đốc có trợ lý
   int a=100, b=200;
   public synchronized void changeValues(){
    try{ Thread.sleep(500); a++; b++;  }
    catch(Exception e) { }
   }
   public synchronized void run(){
    while (true)
    {  try { System.out.println(Thread.currentThread().getName());
          System.out.println("a=" + a);
          System.out.println("b=" + b);
          Thread.sleep(500);
       }
       catch(Exception e) { }
       assistance.changeValues();
     }
   }
```

```
Output - ThreadDemo (run-single) #2
init:
deps-jar:
compile-single:
run-single:
Thread-2
a=100
b=200
Thread-1
a=100
b=200
```

```
public static void main(String args[]) {
   DeadLockDemo person1= new DeadLockDemo();
   DeadLockDemo person2= new DeadLockDemo();
   person1.assistance= person2;  // hai giám đốc
   person2.assistance= person1;  // lại là trợ lý của nhau
   Thread t1= new Thread(person1,"Thread-1");
   Thread t2= new Thread(person2,"Thread-2");
   t1.start();
   t2.start();
   try {
        t1.join();// t1 will be executed to the end
        t2.join();// t2 will be executed to the end
    }
    catch(Exception e) { }
}
}
```

The Philosophers Problem



Wait-Notify Mechanism, a way helps preventing deadlocks

Deadlock!

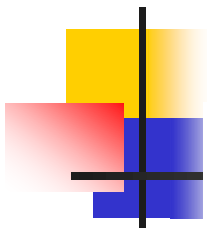
3 classes

```
package threadpkg;
public class ChopStick {
    boolean ready;
    ChopStick(){
        ready=true;
    }
    public synchronized void getUp()
      { while (!ready)
          { try {
              System.out.println("A philosopher is waiting for a chopstick.");
               wait();
             }
            catch (InterruptedException e){
               System.out.println ("An error occured!");
             }
          }
        ready=false;
      }
    public synchronized void getDown
    {  ready= true;
       notify();
    }
}
```

Thread table

Thread	Code Addr	Duration (mili sec)	CPU	State
Thread 1	10320	15	1	Suspended →Ready
Thread 2	40154	17	2	Suspended
Thread 3	80166	22	1	Suspended

```
package threadpkg;
public class Philosopher extends Thread{
   ChopStick leftStick, rightStick;  // He/she needs 2 chop sticks
   int position;                     // His/her position at the dinner table
   Philosopher(int pos, ChopStick lStick, ChopStick rStick)
   { position=pos ; leftStick=lStick; rightStick=rStick;
   }
   public void eat()
   { leftStick.getUp();  rightStick.getUp();
     System.out.println("The " + position +"(th) philosopher is eating");
   }
   public void think()
   { leftStick.getDown(); rightStick.getDown();
     System.out.println("The " + position +"(th) philosopher is thinking\"");
   }
```



Philosopher.java

```
    public void run()
    { while (true)
      { eat();
        try   {  sleep(1000); }
        catch (InterruptedException e)
        { System.out.println("An error occurred!");
        }
        think();
        try {  sleep(1000); }
        catch (InterruptedException e)
        { System.out.println("An error occurred!");
        }
      }
    }
}
```

```
package threadpkg;
public class DinnerTable {
  static int n;
  static ChopStick[] sticks = new ChopStick[5];
  static Philosopher[] philosopers = new Philosopher[5];

  public static void main (String args[])
  { n=5;
    int i;
    for (i=0;i<n;++i) sticks[i]=new ChopStick();
    for (i=0;i<n;++i) philosopers[i] =
          new Philosopher (i,sticks[i],sticks[(i+1)%5]);
    for (i=0;i<n;++i) philosopers[i].start();
  }
}
```

Output - DJA_P1 (run)

```
The 0(th) philosopher is eating
The 1(th) philosopher is thinking"
The 3(th) philosopher is thinking"
The 2(th) philosopher is eating
A philosopher is waiting for a chopstick.
The 0(th) philosopher is thinking"
A philosopher is waiting for a chopstick.
The 4(th) philosopher is eating
The 2(th) philosopher is thinking"
A philosopher is waiting for a chopstick.
The 1(th) philosopher is eating
```