UNIT I - JAVA FUNDAMENTALS

- Java Data types
- Class – Object
- I / O Streams
- File Handling concepts
- Threads
- Applets
- Swing Framework
- Reflection

Presented by,
B.Vijayalakshmi
Computer Centre
MIT Campus
Anna University

Thread

- It is **a line of execution**
- It is the **smallest unit of code** that is dispatched by the scheduler

Process

- It is a **program in execution**
- **A process can contain multiple threads** to execute its different sections.
- Ex. To calculate (b*b)-(4*a*c)
  - ➢ One thread can calculate b*b
  - ➢ Another can calculate 4*a*c
- Thus completion of this process is faster

Multithreading

- **The process of executing multiple threads simultaneously is known as multithreading.**
- Threads are independent because they all have separate path of execution that’s the reason if an exception occurs in one thread, it doesn’t affect the execution of other threads.
- All threads of a process share the common memory.
- **Two or more process running concurrently in a computer is called multitasking**

Multithreading Cont’d

- When a program requires user input, multithreading enable creation of a separate thread for this task alone. The main thread can continue with the execution of the rest of the program.
- Programs not using multithreading will have to wait until the user input a value for the continuation of the execution of the program
- Main thread executing a program is terminated after all its other threads have been destroyed

Advantages of using threads

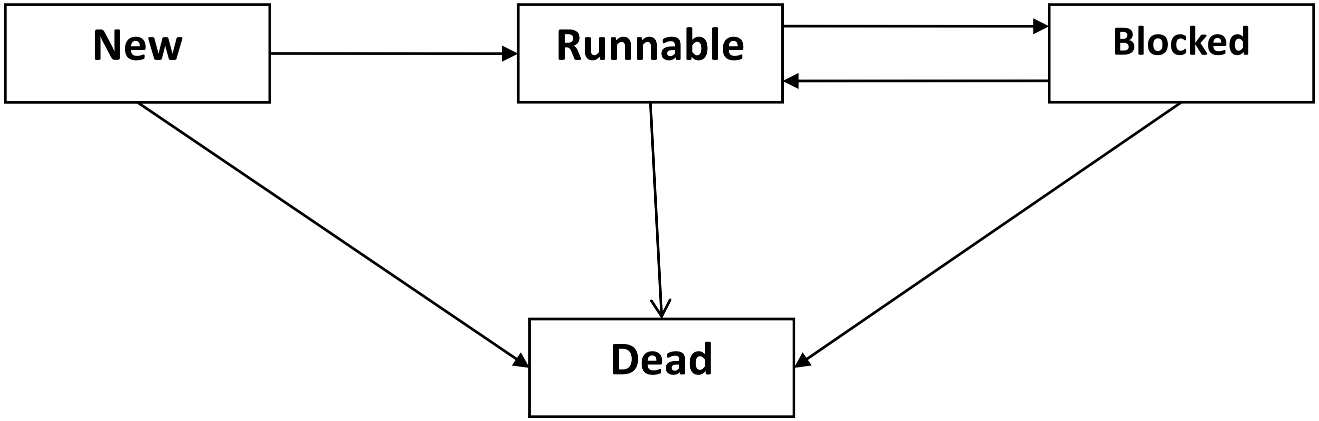
- Can be created faster
- Requires less overheads
- Interprocess communication is faster
- Context switching is faster
- Maximum use of CPU time
- Efficient use of system resources
- Minimize idle time of CPU

Multithreading in java

- Every program that we have been writing has at least one thread. i.e. the main thread.
- Whenever a program starts executing, the JVM is responsible for creating the main thread and calling the main() method from within the thread
- Alongside, many other invisible daemon threads responsible for supporting other activities of java runtime as finalization, garbage collection, etc also created
- Threads are executed by the processor according to the scheduling done by the java Runtime System by assigning priority to every thread,It simply means thread having high priority are given preference for getting executed over the threads having lower priority

States of thread

- There are 4 states,
  - New
  - Runnable
  - Dead
  - Blocked



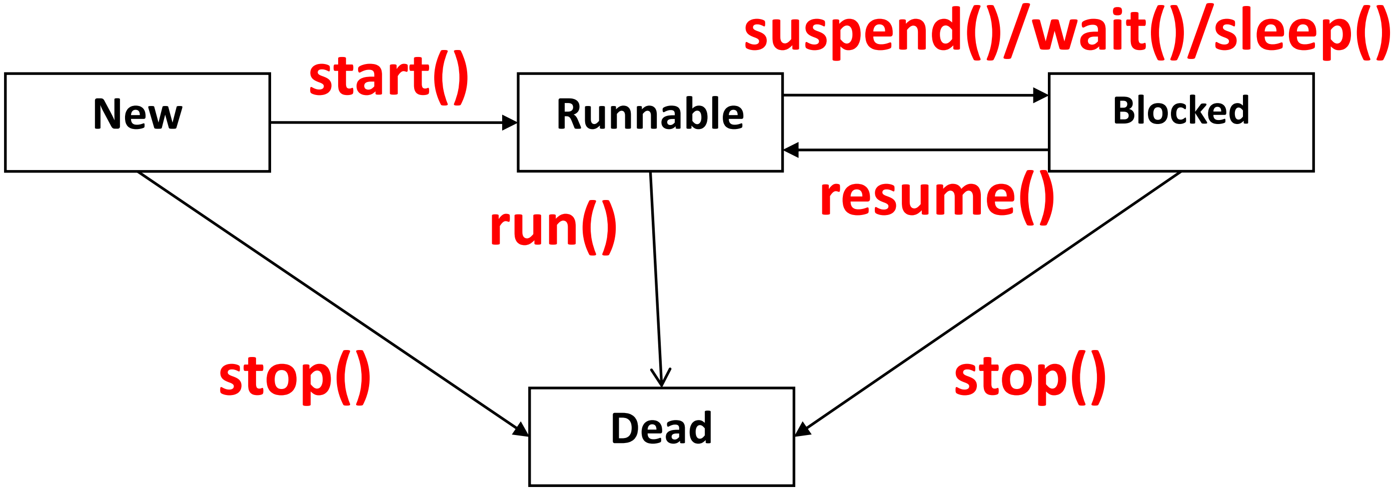
States of thread Cont'd

- **New**
  - When a **thread is created**, it is in the new state.
  - New implies that the **thread object has been created but it has not started** running.
  - It requires start() method to start it
- **Runnable**
  - **A thread** is said to be in runnable state, when it **is executing a set of instructions.**
  - The run() method contains the set of instructions
  - This method is called automatically after start() method.

States of thread Cont'd

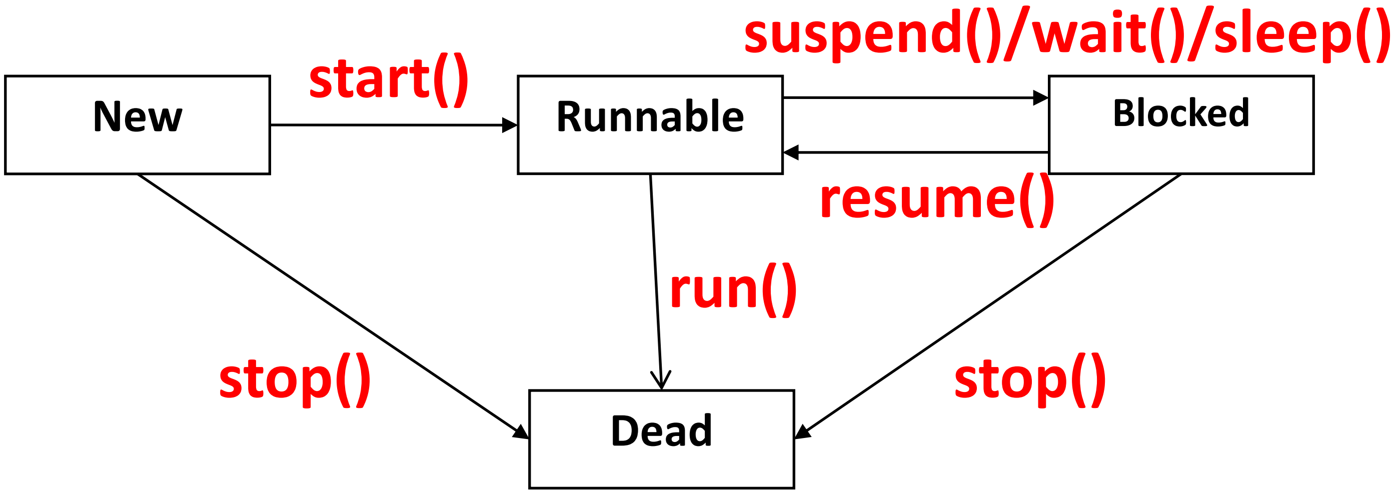
- **Dead**
  - The normal way for a thread to die is by returning from its run() method
  - We can also call stop()
- **Blocked**
  - The thread could run but there is something that prevents it
  - While a thread is in the blocked state, the scheduler will simply skip over it and not give it any CPU time until a thread re-enter the runnable state it will not perform any operations

Common thread methods



- **start()**
  - ➢ **Start the execution** of the invoking object.
  - ➢ It can throw an IllegalThreadStateException if the thread was already started
- **stop()**
  - ➢ **Terminates** the invoking object
- **suspend()**
  - ➢ It **suspends** the invoking object
  - ➢ The thread will become runnable again if it gets the resume method

Common thread methods Cont’d



- **sleep()**
  - ➢ This method **suspends execution of the executing thread for the specified number of milliseconds** it can throw an InterruptedException
    1. public static void sleep(long ms)
    2. public static void sleep(long ms, int ns)
- **resume()**
  - ➢ **Restarts the suspended thread** at the point at which it was halted
  - ➢ Resume() method is called by some thread outside the suspended one, there is a separate class called Resume which does just that

Creation of a thread

- Creation of new threads can be done as follows,
  - By inheriting Thread class
  - By implementing Runnable interface
- **Thread class**
  - This class belongs to java.lang package
  - It provide constructors and methods to create and perform operations on a thread.
  - Thread class extends object class and implements Runnable interface

public class Thread extends Object implements Runnable

Thread class Constructors

- **Thread()**: Allocates a new Thread object
- **Thread(Runnable target)**: Allocates a new Thread object
- **Thread(Runnable target, String name)**: Allocates a new Thread object
- **Thread(String name)**: Allocates a new Thread object
- **Thread(ThreadGroup group, Runnable target)**: Allocates a new Thread object
- **Thread(ThreadGroup group, Runnable target, String name)**: Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group
- **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**: Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size
- **Thread(ThreadGroup group, String name)**: CAllocates a new Thread object

Methods of Thread class

1. **public void run()**→used to perform action for a thread
2. **public void start()**→starts the execution of the thread. JVM calls the run() method on the thread
3. **public void sleep(long milliseconds)**→causes the currently executing thread to sleep (temporarily leave execution) for the specified number of milliseconds
4. **public void join()**→wait for a thread to die
5. **public void join(long milliseconds)**→wait for a thread to die for the specified milliseconds
6. **public int getPriority()**→ returns the priority of the thread
7. **public setPriority(int priority)**→changes the priority of the thread
8. **public String getName()**→returns the name of the thread
9. **public void setName(String name)**→changed the name of the thread

Methods of Thread class Cont’d

10. **public Thread currentThread()**→returns the reference of currently executing thread
11. **public int getId()**→returns the id of the thread
12. **public Thread.State getState()**→ returns the state of the thread
13. **public boolean isAlive()**→test if the thread is alive
14. **public void yield()**→ causes the currently executing thread object to temporarily pause and allow threads to execute
15. **public void suspend()**→is used to suspend the thread(depreciated)
16. **public void resume()**→is used to resume the suspended thread (depreciated)
17. **public void stop()**→is used to stop the thread (depreciated)

Methods of Thread class Cont’d

18. **public boolean isDaemon()**→test if the thread is a daemon thread Or user thread
19. **public void setDaemon(boolean b)**→marks the thread as daemon
20. **public void interrupt()**→interrupts the thread
21. **public boolean isInterrupted()**→ tests if the thread has been interrupted
22. **public static boolean interrupted()**→tests if the current thread has been interrupted.
23. **public void notify()**→ It is used to give the notification for only one thread which is waiting for a particular object.
24. **public void notifyAll()**→ It is used to give the notification to all waiting threads of a particular object.
25. **public void destroy()**→ It is used to destroy the thread group and all of its subgroups.
26. **public static int activeCount()**→ It returns the number of active threads in the current thread's thread group.

Runnable interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

Starting a thread

- **start() method** of Thread class is used to start a newly created thread.
- It performs following tasks:
  - ➢A new thread starts(with new callstack).
  - ➢The thread moves from New state to the Runnable state.
  - ➢**When the thread gets a chance to execute, its target run() method will run.**

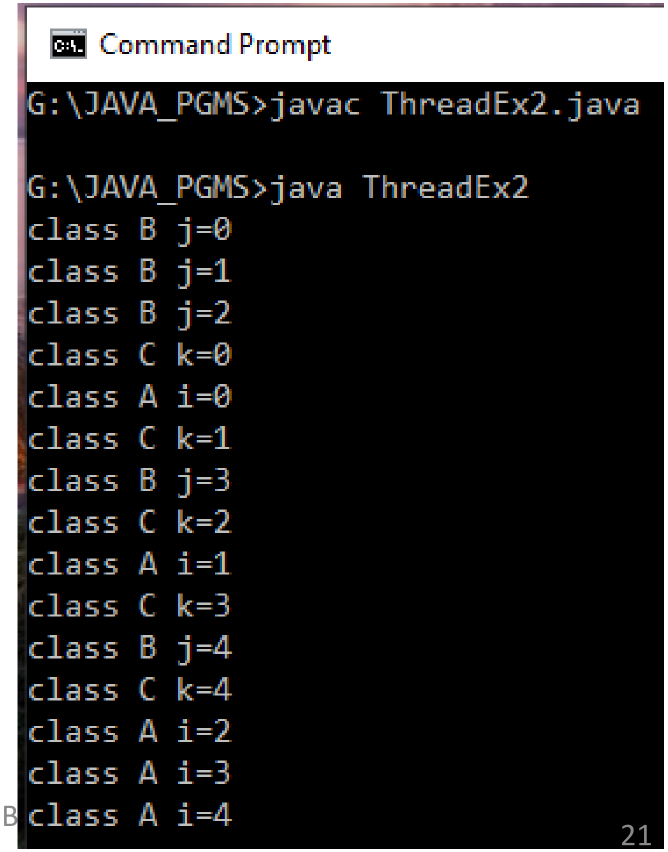
Thread Scheduler in Java

- Thread scheduler in java is the **part of the JVM that decides which thread should run**.
- There **is no guarantee that which runnable thread will be chosen** to run by the thread scheduler.
- **Only one thread at a time can run in a single process.**
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Java Thread using Thread class - Example

```
class A extends Thread
{
        public void run()
        {
                int i;
                for(i=0;i<5;i++)
                        System.out.println("class A i="+i);
        }

}
class B extends Thread
{
        public void run()
        {
                int j;
                for(j=0;j<5;j++)
                        System.out.println("class B j="+j);
        }

}
```

```
class C extends Thread
{
        public void run()
        {
                int k;
                for(k=0;k<5;k++)
                        System.out.println("class C k="+k);
        }

}
class ThreadEx2 extends Thread
{
        public static void main(String args[])
        {
                A objA=new A();
                B objB=new B();
                C objC=new C();
                objA.start();
                objB.start();
                objC.start();
        }
}
```

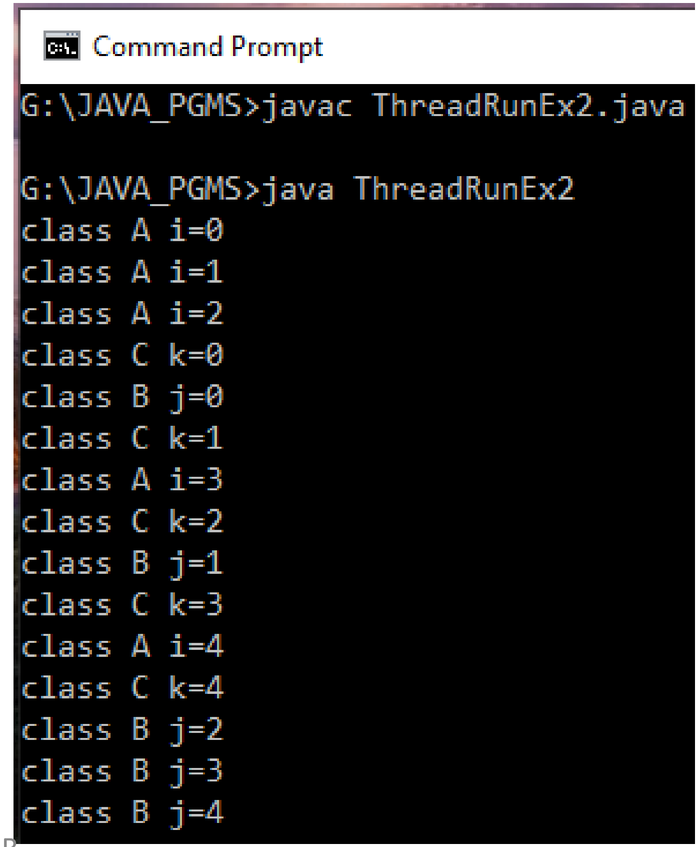


```
G:\JAVA_PGMS>javac ThreadEx2.java

G:\JAVA_PGMS>java ThreadEx2
class B j=0
class B j=1
class B j=2
class C k=0
class A i=0
class C k=1
class B j=3
class C k=2
class A i=1
class C k=3
class B j=4
class C k=4
class A i=2
class A i=3
class A i=4
```

Java Thread by implementing Runnable interface- Example

```
class A implements Runnable
{
        public void run()
        {
                int i;
                for(i=0;i<5;i++)
                        System.out.println("class A i="+i);
        }

}
class B implements Runnable
{
        public void run()
        {
                int j;
                for(j=0;j<5;j++)
                        System.out.println("class B j="+j);
        }

}
```

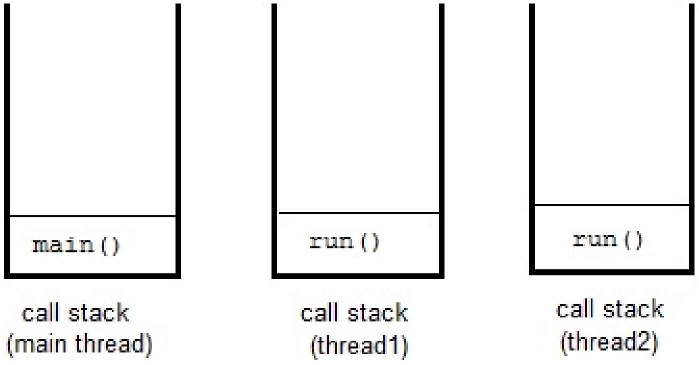
```
class C implements Runnable
{
        public void run()
        {
                int k;
                for(k=0;k<5;k++)
                        System.out.println("class C k="+k);
        }

}
class ThreadRunEx2
{
        public static void main(String args[])
        {
                A objA=new A();
                B objB=new B();
                C objC=new C();
                Thread t=new Thread(objA);
                t.start();
                t=new Thread(objB);
                t.start();
                t=new Thread(objC);
                t.start();
        }
}
```



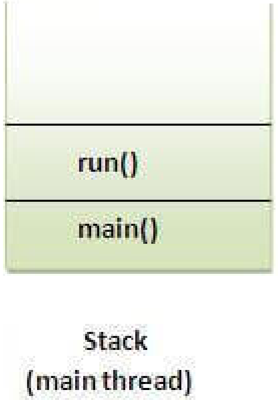
```
G:\JAVA_PGMS>javac ThreadRunEx2.java

G:\JAVA_PGMS>java ThreadRunEx2
class A i=0
class A i=1
class A i=2
class C k=0
class B j=0
class C k=1
class A i=3
class C k=2
class B j=1
class C k=3
class A i=4
class C k=4
class B j=2
class B j=3
class B j=4
```

Can we call run() method directly to start a new thread?

- **No**, we can not directly call run method to start a thread. we need to call start method to create a new thread.
- In java each thread has its own call stack.

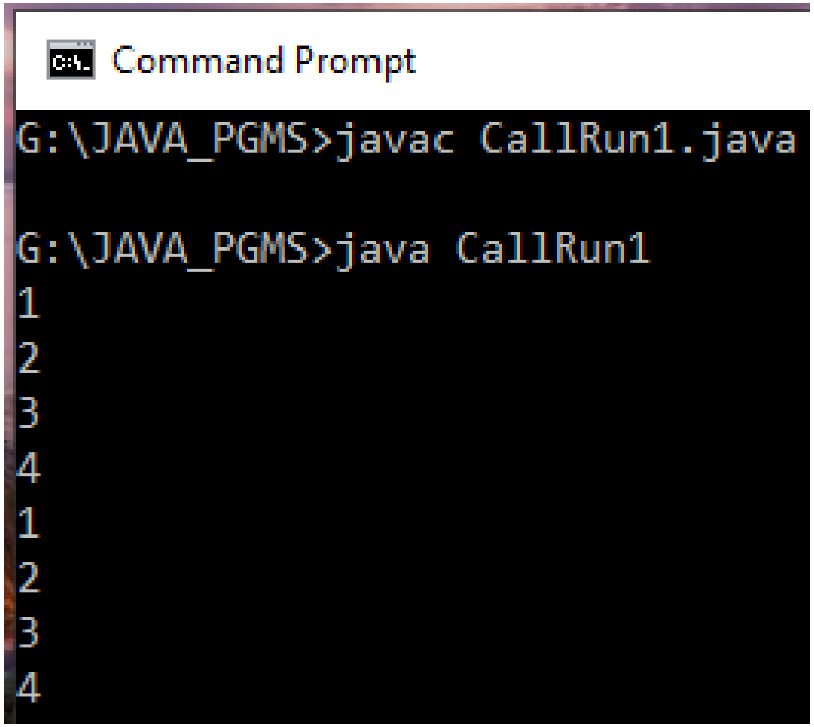


- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.



```
class CallRun1 extends Thread
{
 public void run()
  {
   for(int i=1;i<5;i++)
   {
     try
     {
        Thread.sleep(500);
     } catch(InterruptedException e)
       {
            System.out.println(e);
        }
    System.out.println(i);
   }
  }
 public static void main(String args[])
 {
   CallRun1 t1=new CallRun1();
   CallRun1 t2=new CallRun1();
   t1.run();
   t2.run();
 }
}
```

call run() method directly

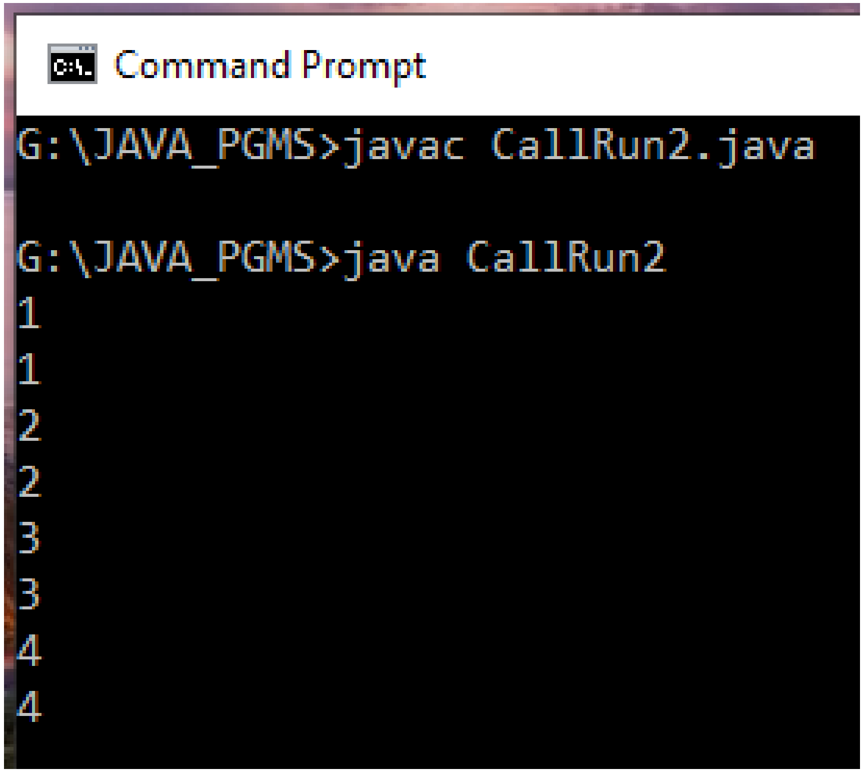


```
G:\JAVA_PGMS>javac CallRun1.java

G:\JAVA_PGMS>java CallRun1
1
2
3
4
1
2
3
4
```

In the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object

call run() method indirectly through start()

```
class CallRun2 extends Thread
{
  public void run()
   {
     for(int i=1;i<5;i++)
     {
        try
        {
            Thread.sleep(500);
        } catch(InterruptedException e)
           {
                 System.out.println(e);
            }
       System.out.println(i);
     }
   }
  public static void main(String args[])
  {
     CallRun2 t1=new CallRun2();
     CallRun2 t2=new CallRun2();
     t1.start();
     t2.start();
  }
}
```

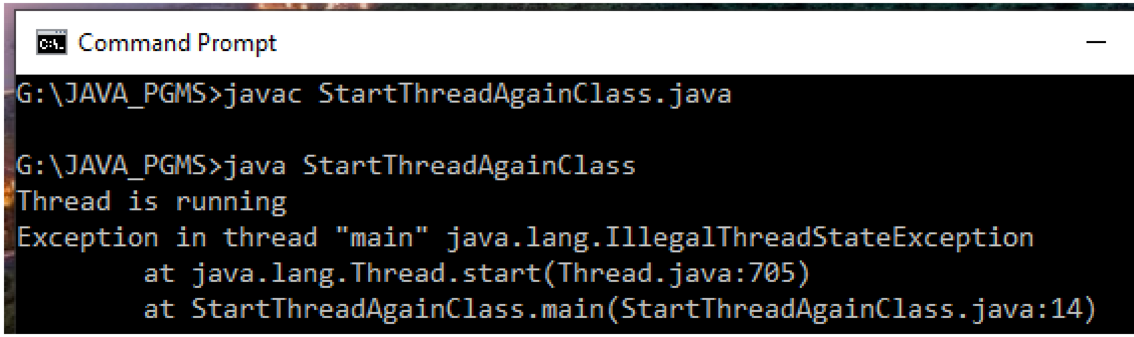


```
G:\JAVA_PGMS>javac CallRun2.java

G:\JAVA_PGMS>java CallRun2
1
1
2
2
3
3
4
4
```

Can we start a thread twice?

- **No, Once we have started a thread, it can not be started again**
- If we try to start thread again , it will throw IllegalThreadStateException.
- In such case, thread will run once but for second time, it will throw exception.

```
class FirstThread extends Thread
{
 public void run()
 {
  System.out.println("Thread is running");
 }
}
public class StartThreadAgainClass
{
 public static void main(String[] args)
{
  FirstThread ft = new FirstThread();
  ft.start();
  ft.start();
 }
}
```



```
G:\JAVA_PGMS>javac StartThreadAgainClass.java

G:\JAVA_PGMS>java StartThreadAgainClass
Thread is running
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.start(Thread.java:705)
        at StartThreadAgainClass.main(StartThreadAgainClass.java:14)
```

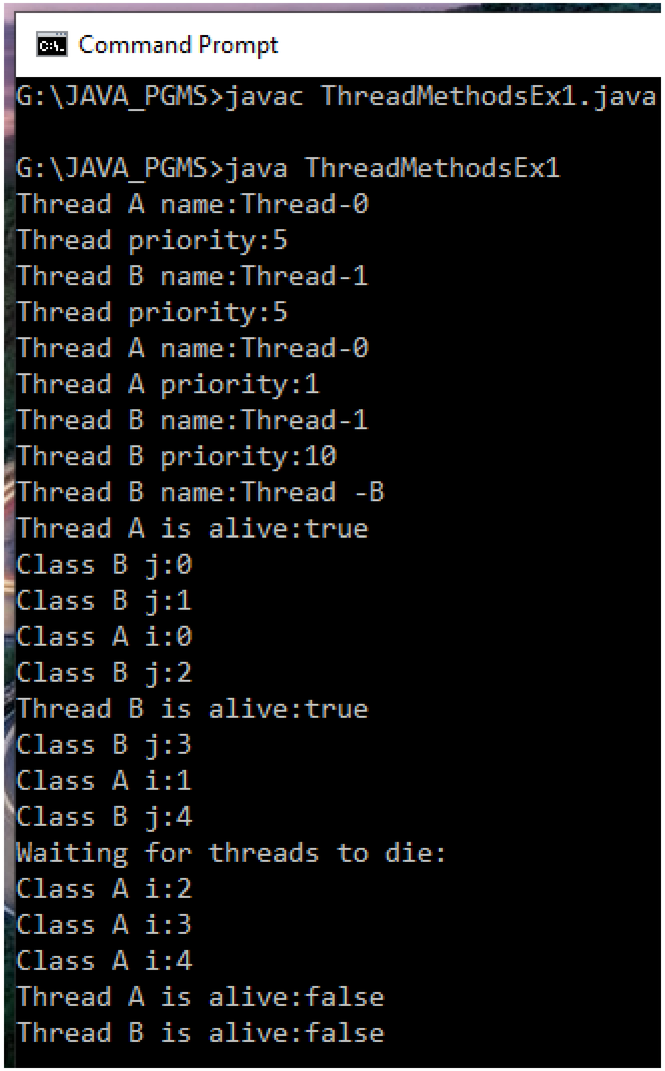
Thread Priority

- In java, **each thread is assigned a priority** which affects the order in which it is scheduled for running
- Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- Threads of same priority are given equal treatment by java scheduler and share the processor on a first-come first-serve.
- **priority constants:**
  - MIN_PRIORITY =1
  - NORM_PRIORITY=5 (default)
  - MAX_PRIORITY=10
- **To set priority:** ThreadName.setPriority(int number);

Thread class methods Example 1

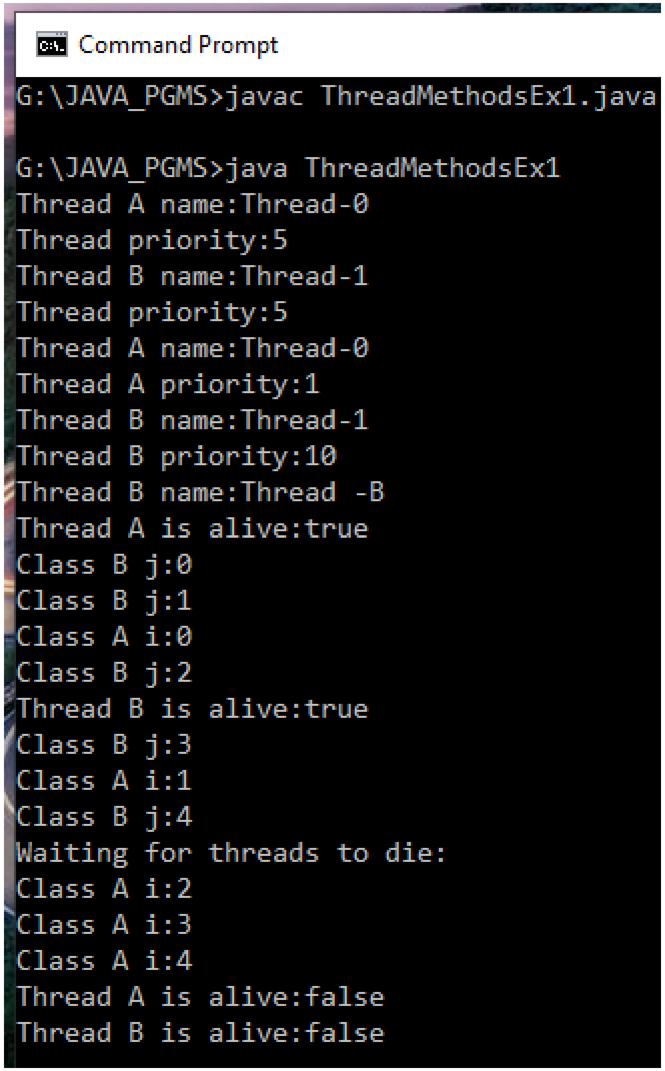
```
class A extends Thread
{
        public void run()
        {
                int i=0;
                for(i=0;i<5;i++)
                {
                        System.out.println("Class A i:"+i);
                }
        }
}
class B extends Thread
{
        public void run()
        {
                int j=0;
                for(j=0;j<5;j++)
                {
                        System.out.println("Class B j:"+j);
                }
        }
}
```

```
class ThreadMethodsEx1
{
   public static void main(String a[])
  {
     A objA=new A();
    System.out.println("Thread A name:"+objA.getName());
    System.out.println("Thread priority:"+objA.getPriority());
    B objB=new B();
    System.out.println("Thread B name:"+objB.getName());
    System.out.println("Thread priority:"+objB.getPriority());
    objA.setPriority(Thread.MIN_PRIORITY);
    objB.setPriority(Thread.MAX_PRIORITY);
    System.out.println("Thread A name:"+objA.getName());
    System.out.println("Thread A priority:"+objA.getPriority());
    System.out.println("Thread B name:"+objB.getName());
    System.out.println("Thread B priority:"+objB.getPriority());
    objB.setName("Thread -B");
    System.out.println("Thread B name:"+objB.getName());
   objA.start();
   objB.start();
   System.out.println("Thread A is alive:"+objA.isAlive());
   System.out.println("Thread B is alive:"+objB.isAlive());
```



```
G:\JAVA_PGMS>javac ThreadMethodsEx1.java

G:\JAVA_PGMS>java ThreadMethodsEx1
Thread A name:Thread-0
Thread priority:5
Thread B name:Thread-1
Thread priority:5
Thread A name:Thread-0
Thread A priority:1
Thread B name:Thread-1
Thread B priority:10
Thread B name:Thread -B
Thread A is alive:true
Class B j:0
Class B j:1
Class A i:0
Class B j:2
Thread B is alive:true
Class B j:3
Class A i:1
Class B j:4
Waiting for threads to die:
Class A i:2
Class A i:3
Class A i:4
Thread A is alive:false
Thread B is alive:false
```

```
try
{
   System.out.println("Waiting for threads to die:");
   objA.join();
   objB.join();
  }catch(Exception e){}
  System.out.println("Thread A is alive:"+objA.isAlive());
  System.out.println("Thread B is alive:"+objB.isAlive());
 }
}
```



```
G:\JAVA_PGMS>javac ThreadMethodsEx1.java

G:\JAVA_PGMS>java ThreadMethodsEx1
Thread A name:Thread-0
Thread priority:5
Thread B name:Thread-1
Thread priority:5
Thread A name:Thread-0
Thread A priority:1
Thread B name:Thread-1
Thread B priority:10
Thread B name:Thread -B
Thread A is alive:true
Class B j:0
Class B j:1
Class A i:0
Class B j:2
Thread B is alive:true
Class B j:3
Class A i:1
Class B j:4
Waiting for threads to die:
Class A i:2
Class A i:3
Class A i:4
Thread A is alive:false
Thread B is alive:false
```

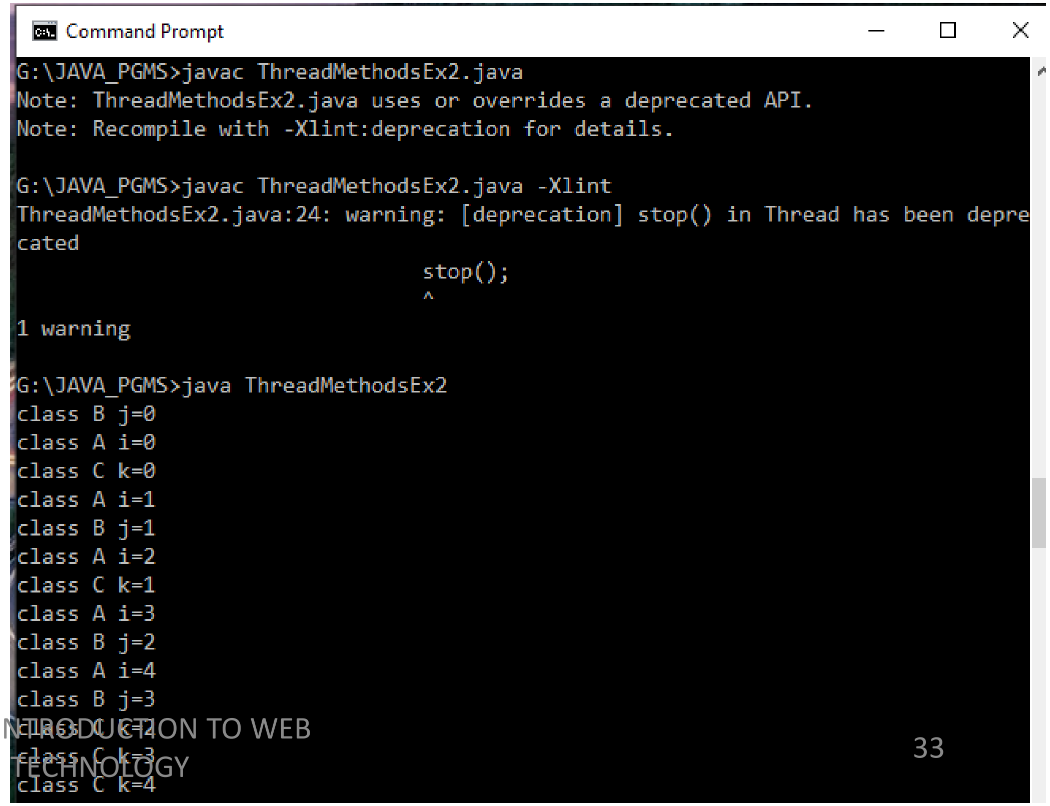
Thread class methods Example 1

```
class A extends Thread
{
        public void run()
        {
                int i;
                for(i=0;i<5;i++)
                {
                        if(i==1)
                                yield(); // This method causes the currently executing
                        //thread object to pause temporarily and allow other threads to run
                        System.out.println("class A i="+i);
                }
        }
 }
class B extends Thread
{
        public void run()
        {
                int j;
                for(j=0;j<5;j++)
                {
                        System.out.println("class B j="+j);
                        if(j==3)
                                stop(); //can use it to halt the thread. [deprecated]
                }
        }
 }
```

Note: Sun has deprecated a variety of Thread methods, such as suspend(), resume() and stop() because they can lock up our programs or damage objects. As a result, we should not call them in your code.

```
class C extends Thread
{
  public void run()
  {
     int k;
     for(k=0;k<5;k++)
    {
      System.out.println("class C k="+k);
      if(k==1)
       try
       {
          sleep(1000); // It suspends a thread for the specified period.
       }catch(Exception e) {}
    }
  }
}
class ThreadMethodsEx2
{
  public static void main(String args[])
   {
     A objA=new A();
     B objB=new B();
     C objC=new C();
     objA.start();
    objB.start();
    objC.start();
   }
}
```

Note: If any other thread interrupts this sleeping thread, it will throw an InterruptedException. So it is suggested to enclose the sleep() method in the try block.



```
G:\JAVA_PGMS>javac ThreadMethodsEx2.java
Note: ThreadMethodsEx2.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

G:\JAVA_PGMS>javac ThreadMethodsEx2.java -Xlint
ThreadMethodsEx2.java:24: warning: [deprecation] stop() in Thread has been depre
cated
                                stop();
                                ^

1 warning

G:\JAVA_PGMS>java ThreadMethodsEx2
class B j=0
class A i=0
class C k=0
class A i=1
class B j=1
class A i=2
class C k=1
class A i=3
class B j=2
class A i=4
class B j=3
class C k=2
class C k=3
class C k=4
```

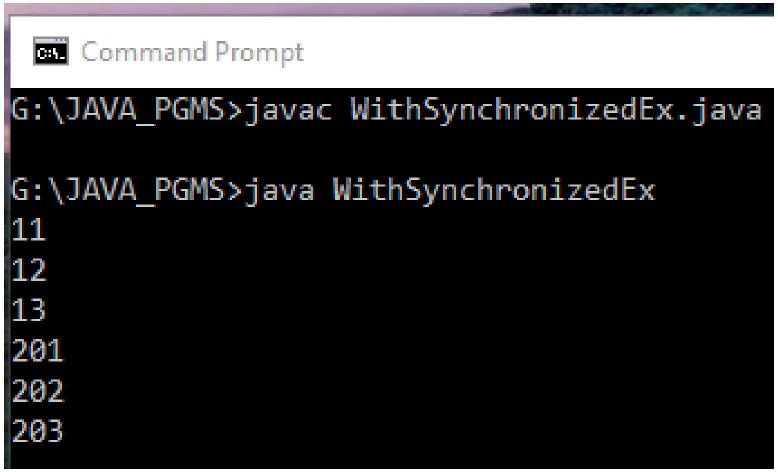
Synchronization

- **If two or more threads access the same data simultaneously, it may lead to loss of data integrity**.
- Example: Simultaneous read and write on a file.
- Java uses the concept of **monitor or a semaphore** to enable this
- A **monitor is an object used as a mutually exclusive lock**
- At a time only one thread can access the monitor
- Second thread cannot access the monitor until the first one comes out
- Till such time the other is said to be waiting
- The **keyword synchronized** is used in the code to enable synchronization
- The word synchronized can be **used along with a method or within a block**

Synchronization keyword example

```
class Print
{
    synchronized public static void printValues(int x)
    {
        int i;
        for(i=1;i<=3;i++)
        {
            System.out.println(x+i);
            try
            {
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }
}
class A extends Thread
{
    public void run()
    {
        Print.printValues(10);
    }
 }
```

```
class B extends Thread
{
        public void run()
        {
                Print.printValues(200);
        }
 }
class WithSynchronizedEx
{
        public static void main(String args[])
        {
                A objA=new A();
                B objB=new B();
                objA.start();
                objB.start();
        }
}
```

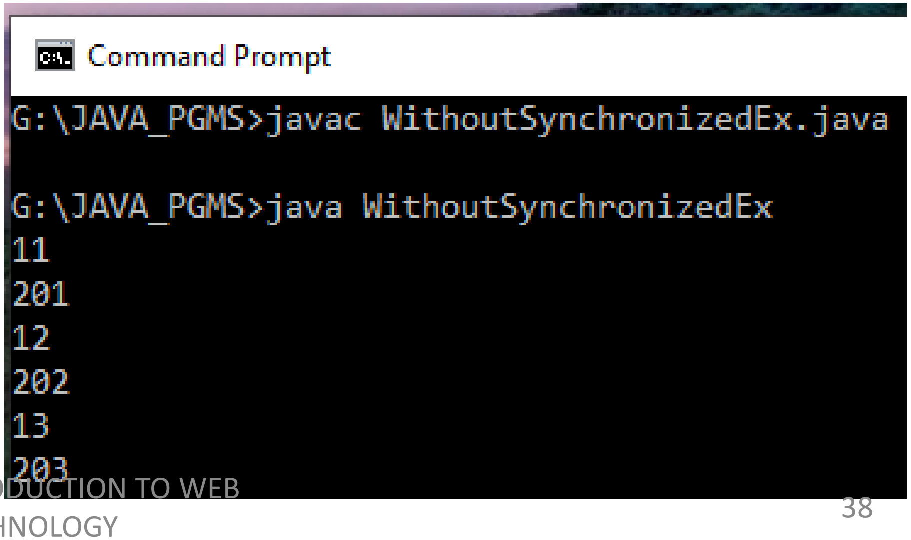


```
G:\JAVA_PGMS>javac WithSynchronizedEx.java

G:\JAVA_PGMS>java WithSynchronizedEx
11
12
13
201
202
203
```

Without using Synchronization keyword example

```
class Print
{
        public static void printValues(int x)
        {
                int i;
                for(i=1;i<=3;i++)
                {
                        System.out.println(x+i);
                        try
                        {
                                Thread.sleep(400);
                        }catch(Exception e){System.out.println(e);}
                }
        }
}
class A extends Thread
{
        public void run()
        {
                Print.printValues(10);
        }
 }
```

```
class B extends Thread
{
        public void run()
        {
                Print.printValues(200);
        }
 }
class WithoutSynchronizedEx
{
        public static void main(String args[])
        {
                A objA=new A();
                B objB=new B();
                objA.start();
                objB.start();
        }
}
```



```
G:\JAVA_PGMS>javac WithoutSynchronizedEx.java

G:\JAVA_PGMS>java WithoutSynchronizedEx
11
201
12
202
13
203
```

Deadlock

- Assume that a thread A must access method1 before it can release method2, but the thread B cannot release method1 until it gets hold of method2.
- Because they are mutually exclusive conditions, a deadlock occurs.

```
Thread A
          synchronized method2()
          {
                    synchronized method1()
                    {
                    }
          }
Thread B
          synchronized method1()
          {
                    synchronized method2()
                    {
                    }
          }
```

Inter-thread Communication

- There may be cases where one of the threads requires data from another.
- It becomes essential to check every few seconds, whether the second thread has produced the required information (or) not.
- Till such time , the first thread waits, wasting CPU time
- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Inter-thread Communication Cont’d

- Java offers inter-process communication through the usage of,
  1. wait()
  2. notify()
  3. notifyAll()

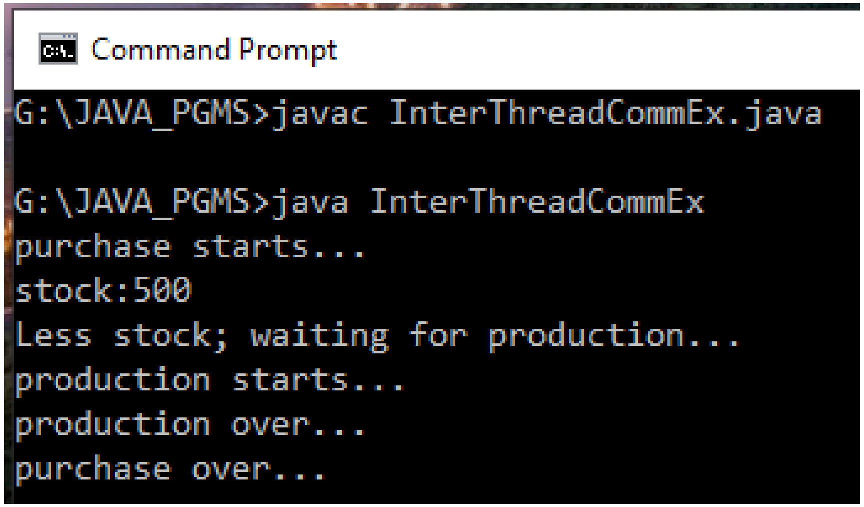
Methods of Object class and all are synchronized methods

- **wait()**→This method waits indefinitely on another thread of execution until it receives a notify() or notifyAll() message
- **notify()**→This method wakes up a single thread waiting on the objects monitor.
- **notifyAll()**→This method wakes up all the threads waiting on the object’s monitor

```
class Inventory
{
        static int stock=500;
        synchronized void purchase(int qt)
        {       System.out.println("purchase starts...");
                if(stock<qt)
                {
                        System.out.println("stock:"+stock);
                        System.out.println("Less stock; waiting for production...");
                        try
                        {
                                wait();
                        }catch(Exception e){}
                }
                stock=stock-qt;
                System.out.println("purchase over...");
        }
        synchronized void production(int qt)
        {
                System.out.println("production starts...");
                stock=stock+qt;
                System.out.println("production over... ");
                notify();
        }
}
```

```
class Customer extends Thread
{
        Inventory i;
        Customer(Inventory i)
        {
                this.i=i;
        }
        public void run()
        {
                i.purchase(600);
        }
}
class Producer extends Thread
{
        Inventory i;
        Producer(Inventory i)
        {
                this.i=i;
        }
        public void run()
        {
                i.production(200);
        }
}
```

```
class InterThreadCommEx
{
        public static void main(String args[])
        {
                Inventory i=new Inventory();
                Customer c=new Customer(i);
                Producer p=new Producer(i);
                c.start();
                p.start();

        }
}
```



```
G:\JAVA_PGMS>javac InterThreadCommEx.java

G:\JAVA_PGMS>java InterThreadCommEx
purchase starts...
stock:500
Less stock; waiting for production...
production starts...
production over...
purchase over...
```