Multi-threading

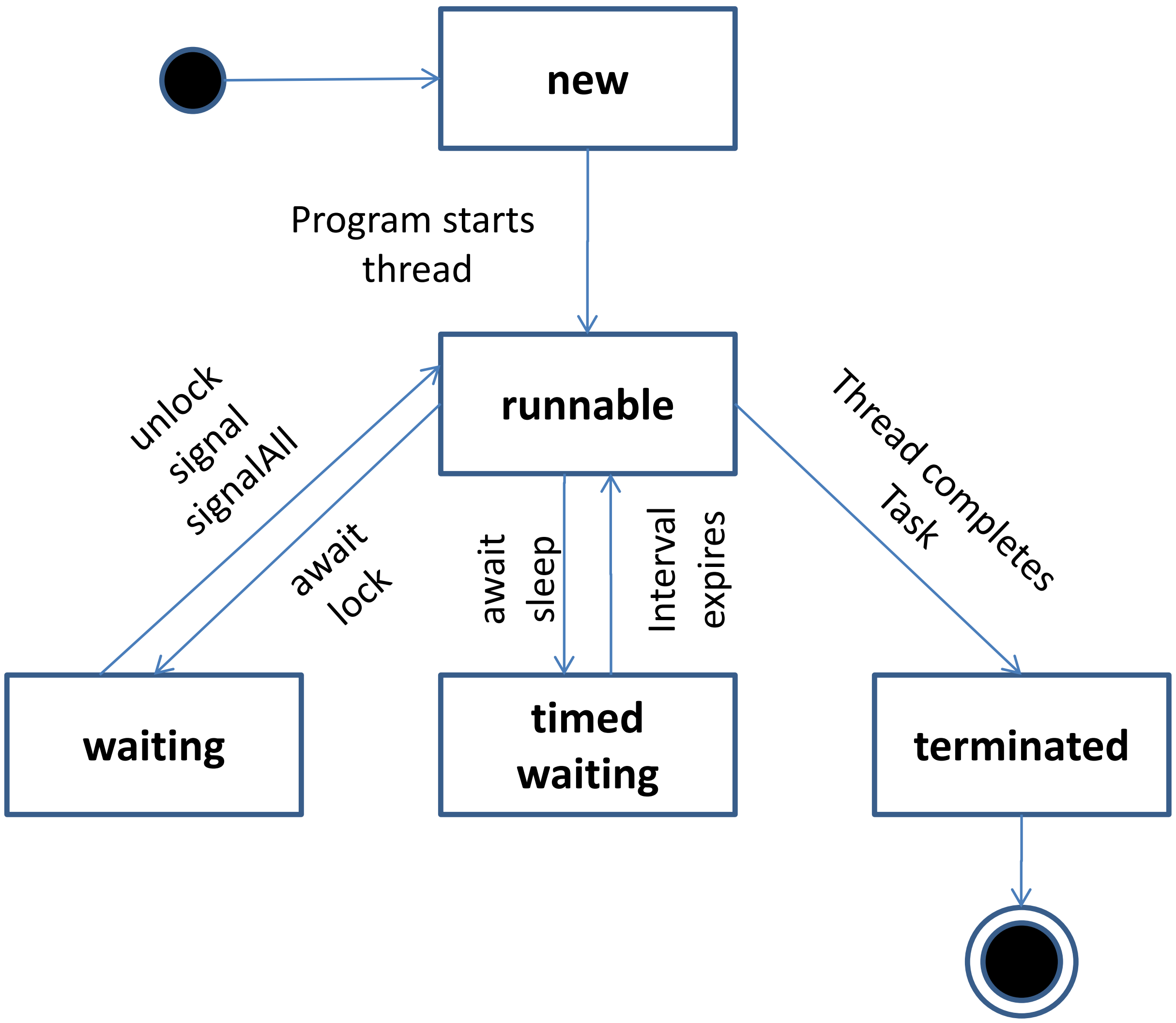
City University

...creating a culture of excellence

What is multithreading?

- A multithreaded program contains two or more parts that can run concurrently, Each part of such a program is called a **thread**.
- A **multithreading** is a specialized form of multitasking.
- Multitasking threads require less overhead than multitasking processes.
- A thread cannot exist on its own; it must be a part of a process.
- A process remains running until all of its child threads are done executing.

Life cycle of a Thread



Life cycle of a Thread (Cont.)

- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.
- There are **5** stages in the life cycle of the Thread
  - **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
  - **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
  - **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals waiting thread to continue.
  - **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
  - **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating a Thread in Java

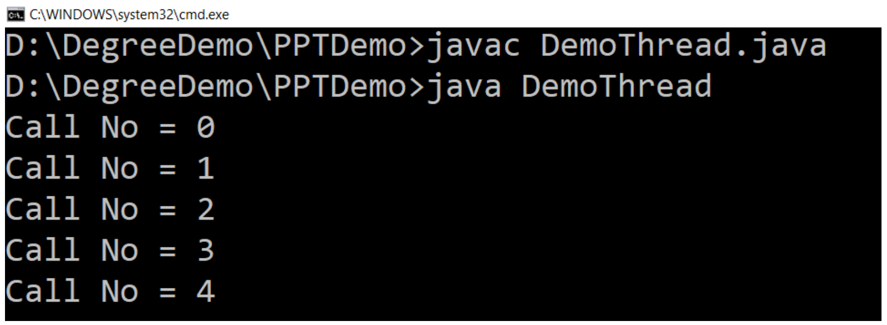
- There are two ways to create a Thread
  1. **extending** the **Thread class**
  2. **implementing** the **Runnable interface**

1) Extending Thread Class

- One way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run( ) method, which is the entry point for the new thread.
- It must also call start( ) to begin execution of the new thread.

Example with Thread Class

```java
class MyThread extends Thread {
    public void run(){
        for(int i=0;i<100;i++){
            System.out.println("Call No = "+ i);
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class DemoThread {
    public static void main(String[] ar){
        MyThread m = new MyThread();
        m.start();
    }
}
```


C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac DemoThread.java
D:\DegreeDemo\PPTDemo>java DemoThread
Call No = 0
Call No = 1
Call No = 2
Call No = 3
Call No = 4
```

2) Implementing Runnable Interface

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need to implement only a single method called run(), which is declared as follows:

```
public void run( )
```

- After creating a class that implements Runnable will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

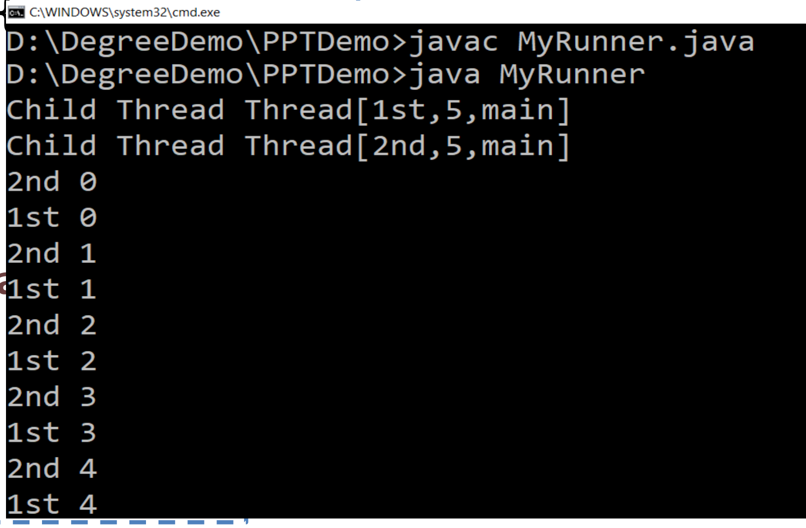
```
Thread(Runnable threadOb, String threadName);
```

- Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName.

Example with Runnable Interface

```
class MyRunnable implements Runnable {
    Thread t;
    public MyRunnable(String tName) {
        t = new Thread(this, tName);
        System.out.println("Child Thread " + t);
        t.start();
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                Thread.sleep(1000);
                System.out.println(t.getName() + " " + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class MyRunner {
    public static void main(String[] a
        new MyRunnable("1st");
        new MyRunnable("2nd");
    }
}
```



```
D:\DegreeDemo\PPTDemo>javac MyRunner.java
D:\DegreeDemo\PPTDemo>java MyRunner
Child Thread Thread[1st,5,main]
Child Thread Thread[2nd,5,main]
2nd 0
1st 0
2nd 1
1st 1
2nd 2
1st 2
2nd 3
1st 3
2nd 4
1st 4
```

Thread Priority

- Each thread have a priority. Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- 3 constants defined in Thread class:
  - public static int MIN_PRIORITY -> 1
  - public static int NORM_PRIORITY -> 5
  - public static int MAX_PRIORITY -> 10

Example Thread Priority

```
public class ThreadPriorityDemo extends Thread {
    public ThreadPriorityDemo(String tName) {
        super(tName);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
              sleep(200);
              System.out.println("Call of " + this.getName() + i);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] ar) {
        ThreadPriorityDemo t1 = new ThreadPriorityDemo("Low");
        t1.setPriority(Thread.MIN_PRIORITY); //1
        ThreadPriorityDemo t2 = new ThreadPriorityDemo("High");
        t2.setPriority(Thread.MAX_PRIORITY); // 10
        t1.start();
        t2.start();
    }
}
```

Thread Synchronization

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.
- Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized methods & synchronized blocks**.

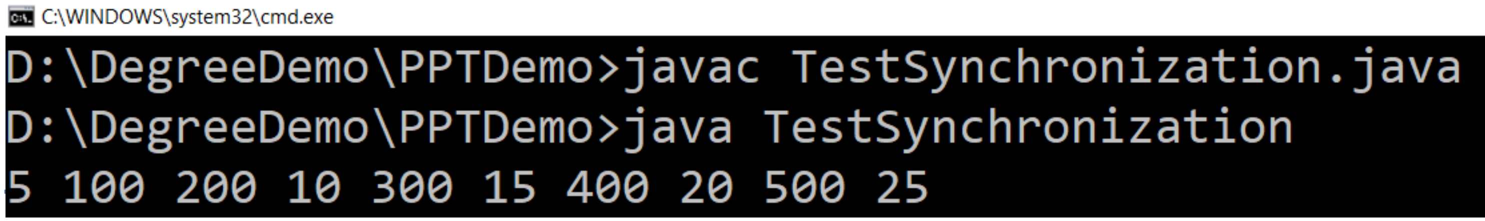
Problem without synchronization (Example)

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```



C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 100 200 10 300 15 400 20 500 25
```

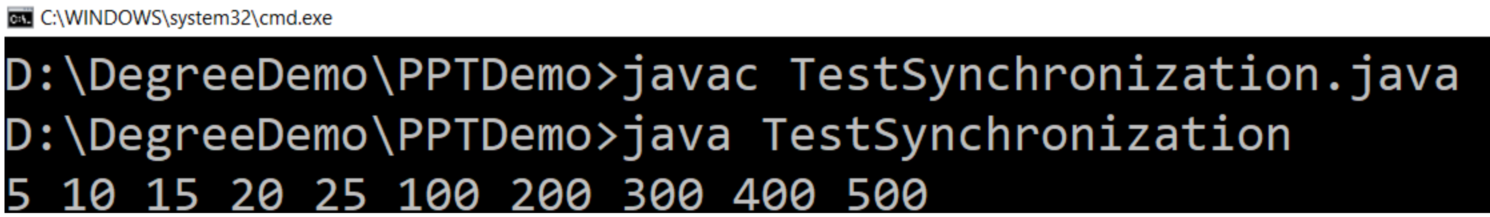
Solution with synchronized method

```java
class Table {
  synchronized void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```java
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(5);
  }
}
```

```java
class MyThread2 extends Thread {
  Table t;
  MyThread2(Table t) {
    this.t = t;
  }
  public void run() {
    t.printTable(100);
  }
}
```

```java
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```



C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```

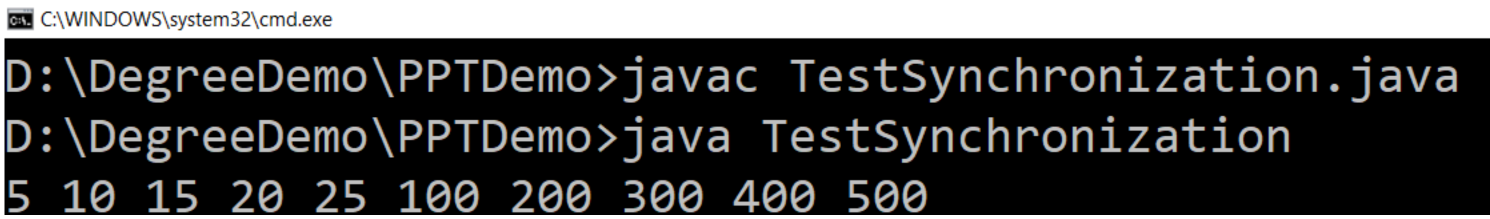
Solution with synchronized blocks

```
class Table {
  void printTable(int n) {
    for (int i = 1; i <= 5; i++) {
      System.out.print(n * i + " ");
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(5);
    }
  }
}
```

```
public class TestSynchronization {
  public static void main(String args[]){
    Table obj = new Table();
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

```
class MyThread1 extends Thread {
  Table t;
  MyThread1(Table t) {
    this.t = t;
  }
  public void run() {
    synchronized (t) {
      t.printTable(100);
```



C:\WINDOWS\system32\cmd.exe

```
D:\DegreeDemo\PPTDemo>javac TestSynchronization.java
D:\DegreeDemo\PPTDemo>java TestSynchronization
5 10 15 20 25 100 200 300 400 500
```

Interprocess communication mechanism

- To avoid polling, Java includes an elegant interprocess communication mechanism via the wait(), notify(), and notifyAll() methods.
- These methods are implemented as final methods in Object, so all classes have them.
- All three methods can be called only from within a **synchronized** context.
- 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.