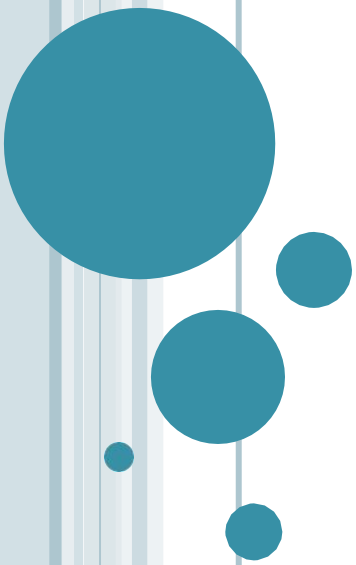
MULTITHREADING



MULTITHREADING

- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

MULTITHREADING

- **Multitasking**

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

  - Process-based Multitasking (Multiprocessing)
  - Thread-based Multitasking (Multithreading)

MULTITHREADING

- **1) Process-based Multitasking (Multiprocessing)**

- Each process has an address in memory. In other words,
- each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some
- time for saving and loading registers, memory maps,
- updating lists, etc.

MULTITHREADING

2) Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

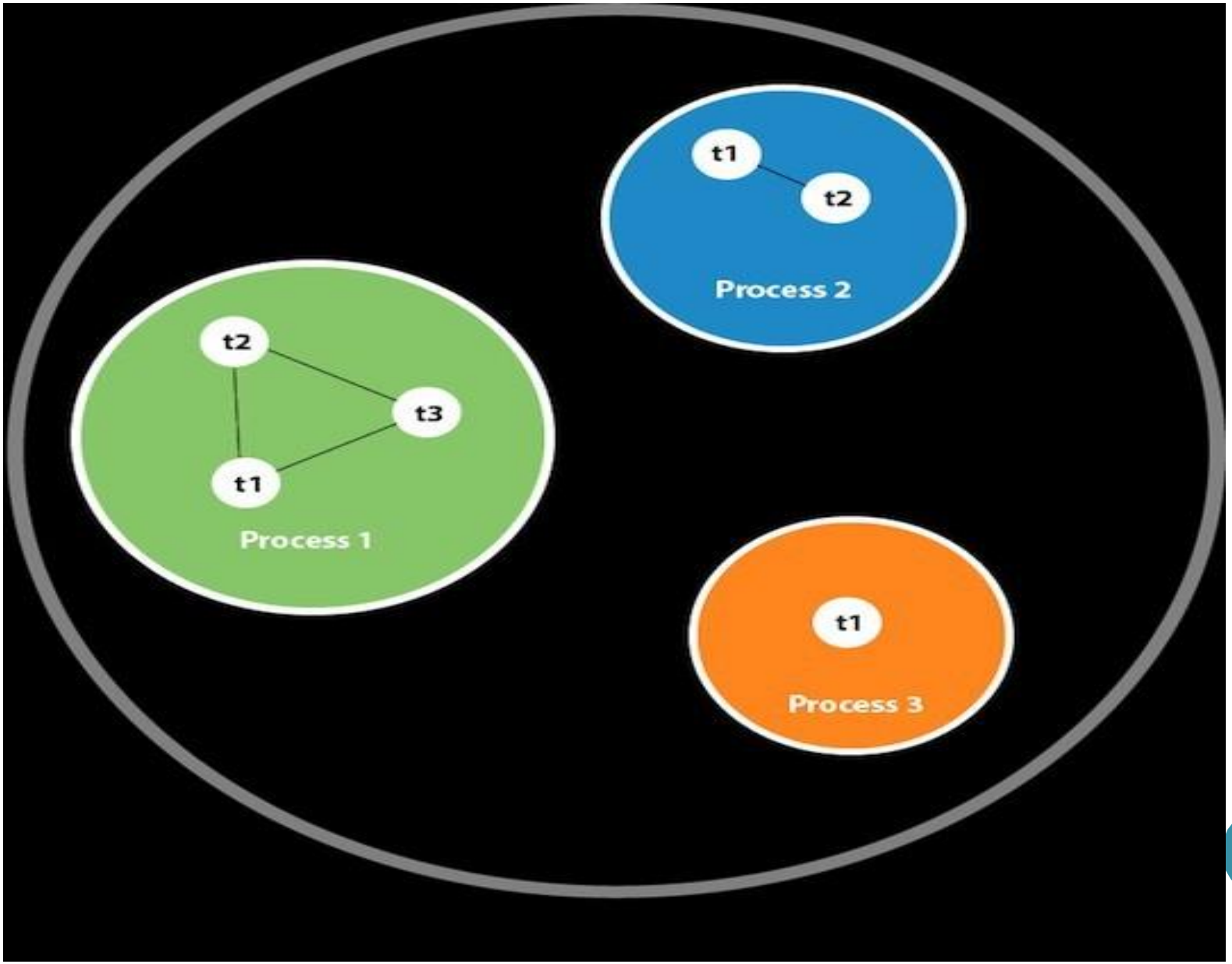
MULTITHREADING

- **What is Thread in java?**

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

MULTITHREADING



MULTITHREADING

- **Java Thread class**
- Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread.

MULTITHREADING

- **Java Thread Methods**

1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.

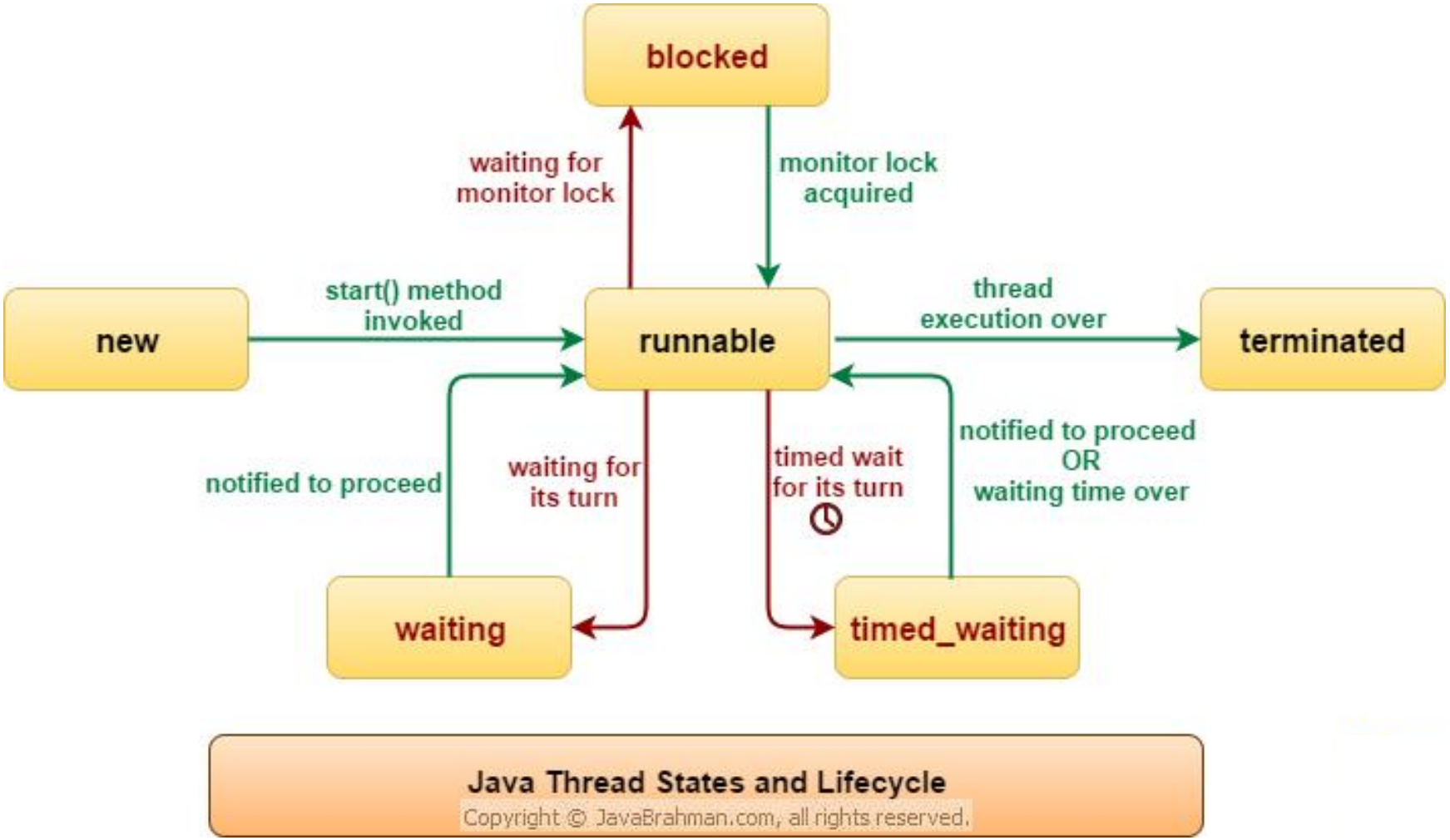
MULTITHREADING

6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.

MULTITHREADING

- **Life cycle of a Thread (Thread States)**
- In Java, a thread always exists in any one of the following states. These states are:

- New
- Active
- Blocked / Waiting
- Timed Waiting
- Terminated


blocked
waiting for monitor lock
monitor lock acquired
new
start() method invoked
runnable
thread execution over
terminated
notified to proceed
waiting for its turn
timed wait for its turn
notified to proceed OR waiting time over
waiting
timed_waiting
Java Thread States and Lifecycle
Copyright © JavaBrahman.com, all rights reserved.

MULTITHREADING

- **New:** Whenever a new thread is created, it is always in the new state.
- **Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.
- **Runnable:** A thread, that is ready to run is then moved to the runnable 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.

MULTITHREADING

- **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.
- **Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation.
- **Terminated:** When a thread has finished its job, then it exists or terminates normally.

MULTITHREADING

- **How to create a thread**
- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface

MULTITHREADING

- **Thread class:**

- **Thread class provide constructors and methods to create and perform operations on a thread.**
- **Commonly used methods of Thread class:**
- public **void run():** is used to perform action for a thread.
- public **void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

MULTITHREADING

- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- public int getPriority(): returns the priority of the thread.
- public int setPriority(int priority): changes the priority of the thread.
- public String getName(): returns the name of the thread.
- public int getId(): returns the id of the thread.
- public Thread.State getState(): returns the state of the thread.
- public boolean isAlive(): tests if the thread is alive.

MULTITHREADING

- **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.

- The Main Thread
- When a Java program starts up, one thread begins running immediately.
- This is usually called the main thread of your program, because it is the one that is executed when your program begins
- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
- To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread

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

Thread[main,5,main] 5 4 3 2 1

MULTITHREADING

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
  }
}
```

Threaddemo1.java

Output

```
Child thread: Thread[Demo Thread,5,main]
Child Thread: 5
Main Thread: 5
Child Thread: 4
Child Thread: 3
Main Thread: 4
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

start( ) is called, which starts the thread of execution beginning at the run( ) method.
This causes the child thread's for loop to begin.
After calling start( ), NewThread's constructor returns to main( ).
When the main thread resumes, it enters its for loop.
Both threads continue running, sharing the CPU in singlecore systems

MULTITHREADING

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
  // the thread will sleep for the 500 milli seconds
    try{Thread.sleep(500);}
    catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

MULTITHREADING

- **Starting a thread:**

- The start() method of Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts.
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

MULTITHREADING Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need only implement a single method called run( ),
- which is declared like this: public void run( )
- Inside run( ), you will define the code that constitutes the new thread.
- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
- Thread(Runnable threadOb, String threadName)
- After the new thread is created, execute the start( ) call to run( ).

MULTITHREADING

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);    // Using the constructor Thread
t1.start();
 }
}
```

runnabledemo.java - multithreading

Threaddemo2.java - multithreading

MULTITHREADING

// testing of getName()

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

MULTITHREADING

join() method

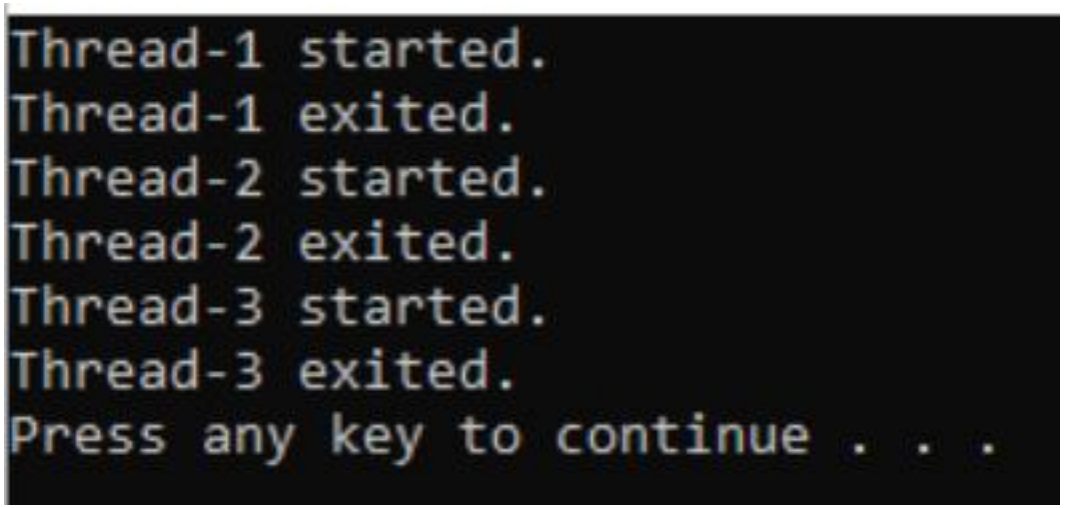
The join() method in Java is provided by the **java.lang.Thread** class that **permits one thread to wait until the other thread to finish its execution.**

Suppose th be the object the class Thread whose thread is doing its execution currently, then the th.join(); statement ensures that th is finished before the program does the execution of the next statement.

e.G all child threads terminate prior to the main thread

```
class CustomThread implements Runnable {
   public void run() {
      System.out.println(Thread.currentThread().getName() + " started.");
      try {
         Thread.sleep(500);
      } catch (InterruptedException e) {
         System.out.println(Thread.currentThread().getName() + " interrupted.");
      }
      System.out.println(Thread.currentThread().getName() + " exited.");
   }
}
public class Tester {
   public static void main(String args[]) throws InterruptedException {
      Thread t1 = new Thread( new CustomThread(), "Thread-1");
      t1.start();
      //main thread class the join on t1
      //and once t1 is finish then only t2 can start
      t1.join();
      Thread t2 = new Thread( new CustomThread(), "Thread-2");
      t2.start();
      //main thread class the join on t2
      //and once t2 is finish then only t3 can start
      t2.join();
      Thread t3 = new Thread( new CustomThread(), "Thread-3");
      t3.start();
   }
}
```

MULTITHREADING



MULTITHREADING

Thread Priority

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling).

MULTITHREADING

Setter & Getter Method of Thread Priority

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority.

```
import java.lang.*;

public class ThreadPriorityExample extends Thread
{
public void run()
{
    System.out.println("Inside the run() method");
}
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

}
}
```

MULTITHREADING

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th3 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Press any key to continue . . .
```

THREAD SYNCHRONIZATION

When two or more 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

The process by which this synchronization is achieved is called *thread synchronization*

“The synchronized keyword in Java creates a block of code referred to as a critical section.
Every Java object with a critical section of code gets a lock associated with the object.
To enter a critical section, a thread needs to obtain the corresponding object's lock”

CONT..

Threads are synchronized in Java through the use of a monitor.

Think of a monitor as an object that enables a thread to access a resource

Only one thread can use a monitor at any one time period. Programmers say that the thread *owns* the monitor for that period of time. The monitor is also called a *semaphore*

A thread can own a monitor only if no other thread owns the monitor

If the monitor is available, a thread can own the monitor and have exclusive access to the resource associated with the monitor

CONT..

If the monitor is not available, the thread is suspended until the monitor becomes available. Programmers say that the thread is *waiting* for the monitor

You have two ways in which you can synchronize threads:

- You can use the synchronized method or
- The synchronized statement.

SYNCHRONIZED STATEMENT

This is the general form of the synchronized statement:

- synchronized(object) { // statements to be synchronized}

Here, object is a reference to the object being synchronized

A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor

Calls to the methods contained in the synchronized block happen only after the thread enters the monitor of the object

CONT..

Synchronizing a method is the best way to restrict the use of a method one thread at a time

However, there will be occasions when you won't be able to synchronize a method, such as when you use a class that is provided to you by a third party

Although you can call methods within a synchronized block, the method declaration must be made outside a synchronized block

CONT..

```
class Table{
void printTable(int n){//method not synchronized
    for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
    }

  }
}
```

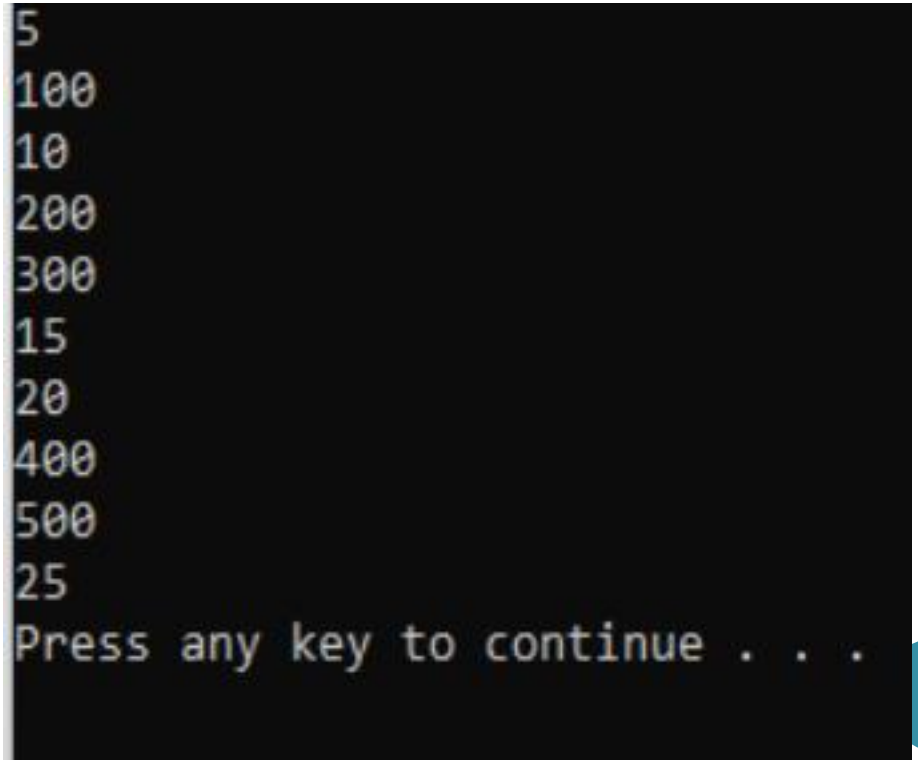
CONT..

```
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);
}
}
```

CONT..

```
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```



JAVA SYNCHRONIZED METHOD

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

 }
}
```

JAVA SYNCHRONIZED METHOD

```
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);
}
}
```

JAVA SYNCHRONIZED METHOD

```
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
5
10
15
20
25
100
200
300
400
500
Press any key to continue . . .
```

JAVA SYNCHRONIZED METHOD

```
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
5
10
15
20
25
100
200
300
400
500
Press any key to continue . . .
```

Interthread Communication

g, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object,

. All three methods can be called only from within a synchronized context.

•

- **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( ) or notifyAll( ).
- **notify( )** wakes up a thread that called wait( ) on the same object.
- **notifyAll( )** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )

Demosync.java