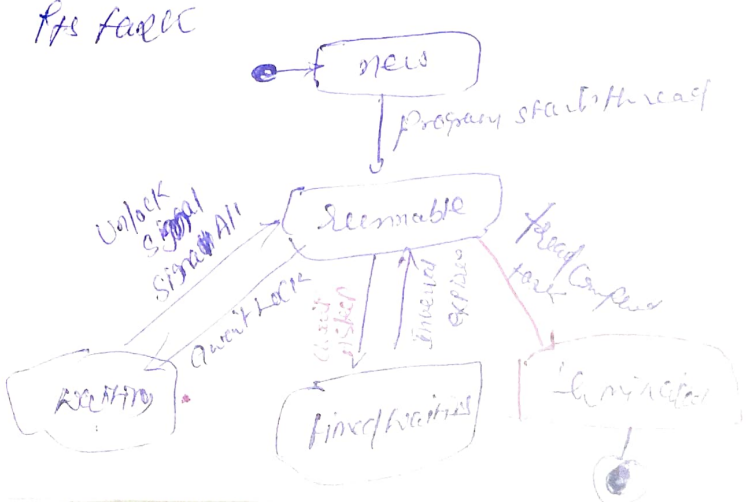
Multithreading :-

- Provide application with multiple threads of execution
- Allows program to perform task concurrently.
- Often requires programmer to synchronize threads to function

Thread States

- New state
  - New threads begin its life cycle in the new state.
  - Remain in this state until program starts the thread, placing it in the runnable state.
- Runnable state
  - A thread in this state is executing its task
- Waiting state
  - A thread transitions to this state to wait for another thread to perform a task.
- Time waiting state
  - A threads enters this state to wait for another thread or for an amount of time to elapse.
  - A thread in this state return to the runnable state when it is signaled by another thread or when the timed interval expires.
- Terminated state
  - A runnable thread enters this state when it completes its task



Dijkstra Algorithm

Thread Priorities And Thread Scheduling

- Priorities
  - Every Java thread has a priority
  - Java priorities are in the range between MIN_PRIO. (a constant 1) and MAX_PRIORITY (a constant 10)
  - Threads with a higher priority are more important and will be allocated a processor before threads with a lower priority.
  - Default priority is NORM_PRIORITY (a constant 95)

Thread scheduler

- Determine which thread runs next
- Simple implementation uses equal-priority threads in a round-robin fashion.
- Higher priority threads can preempt the currently running thread.
- In some cases, higher priority threads can indefinitely postpone lower-priority threads which is also known as starvation.

Creating and Executing Threads:-

- Runnable Interface-
  - Preferred means of creating a multithreaded app
  - Declares method run
  - Executed by an object that implements the Runnable interface.
- Executor Interface-
  - Declares methods to execute
  - Creates and manages a group of threads called a thread pool

- ExecutorService Interface:
  - Sub interface of Executor that declares other methods for managing the life cycle of an executor.
  - can be created using static methods of class Executors.
  - method shutdown ends when tasks are completed

- Executors class — ThreadPool is collection of thread that is used to perform multiple tasks in the background
  - method newFixedThreadPool Creates a pool consisting of a fixed number of threads.
  - method newCachedThreadPool Creates a pool that creates new threads as they are needed.

Thread Synchronization
- Provide to the programmer with mutual exclusion
  - exclusive access to a shared object
- implemented in Java using locks

Lock Interface — mutual exclusion
- lock method obtains lock, enforcing
- unlock method release the lock
- class ReentrantLock implements the Lock interface

(condition variable)
- If a thread holding the lock cannot continue with its task until a condition is satisfied, the thread can wait on a condition variable
- Create by calling lock method newCondition
- Represented by an object that implements the condition interface (state)

Condition Interface

- Declares methods (await) to make a thread wait, Signal, to wake up a waiting thread, and SignalAll, to wake up waiting threads

Producer/Consumer Relationship without Synchro

- Producer/Consumer relationship

- Producer Generates data and stores it in shared memo
- Consumer reads data from shared memory
- Shared memory is called the buffer.

Program

"Producer/consumer Relationship"

- this example uses locks and Conditions to implement synchronization.

Pro

Circular Buffer:-

- Provides extra buffer space into which Producer can place values and Consumer can read values

ArrayBlockingQueue

- Fully implemented version of the Circular buffer
- Implements the BlockingQueue interface
- Declares methods put and take to write and read data from the buffer respectively.

Other Classes and Interfaces in java.util.concurrent

Callable Interface

- Declare method call
- method call allows a concurrent task to return a value or throw an exception
- ExecutorService method submit takes a Callable and returns a Future representing the result of the task.

Future Interface

- Declares method get
- method get returns the result of the task represented by the future.

Monitors and Monitor Locks

Monitors

- Every object in java has a monitor
- allows one thread at a time to execute inside a synchronized statement
- Thread waiting to acquire the monitor lock are placed in the blocked state
- object method wait places a thread in the waiting state
  " " notify wake up a waiting thread
  " " notifyAll wake up all waiting threads

It is an error if a thread issues a wait, a notify, or a notifyAll on an object without having acquired a lock for it. This causes an IllegalMonitorStateException.

Creating Threads

```
Public void run ()
    {
      ---  (Statements for implementing thread)
      ---
    }
```

Extending the THREAD CLASS

The Thread class can be extended as follows

```
class myThread extends Thread
    {
      ---
      ---
    }
```

Starting new Thread

```
MyThread aThread = new myThread();
aThread.start(); // invokes run method
```

Threads

```
class A extends Thread
    {
    p v run()
    {
    for (int i=1; i<=5; i++)
      {
      s.o.p("It from thread A: i=" +i);
      }
    S.O.P ("Exit from A");
    }}

class B extends Thread
    { p v run()
      {
```

```
for (int j=1; j<=5; j++)
{
System.out.println("From Thread B : j=" + j);
}
System.out.println("Exit from B");
}}
class C extends Thread
{
public void run()
{
for (int k=1; k<=5; k++)
{
System.out.println("From Thread C : k=" + k);
}
System.out.println("Exit from C");
}
}
class ThreadTest
{
public static void main(String args[])
{
new A().start();
new B().start();
new C().start();
}}
```

Multitasking

Multitasking, in an operating system, is allowing a user to perform more than one computer task (such as the operation of an application program) at a time. The operating system is able to keep track of where you are in these tasks and go from one to the other without losing information. Microsoft Windows 2000, IBM's OS/390, and Linux are examples of operating systems that can do multitasking (almost all of today's operating systems can). When you open your Web browser and then open Word at the same time, you are causing the operating system to do multitasking.

Multithreading

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. Each user request for a program or system service (and here a user can also be another program) is kept track of as a thread with a separate identity. As programs work on behalf of the initial request for that thread and are interrupted by other requests, the status of work on behalf of that thread is kept track of until the work is completed.

Multiprogramming

Multiprogramming is a rudimentary form of parallel processing in which several programs are run at the same time on a uniprocessor. Since there is only one processor, there can be no true simultaneous execution of different programs. Instead, the operating system executes part of one program, then part of another, and so on. To the user it appears that all programs are executing at the same time.

Multithreading in Java

1. Multithreading
2. Multitasking
3. Process-based multitasking
4. Thread-based multitasking
5. What is Thread

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

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

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

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

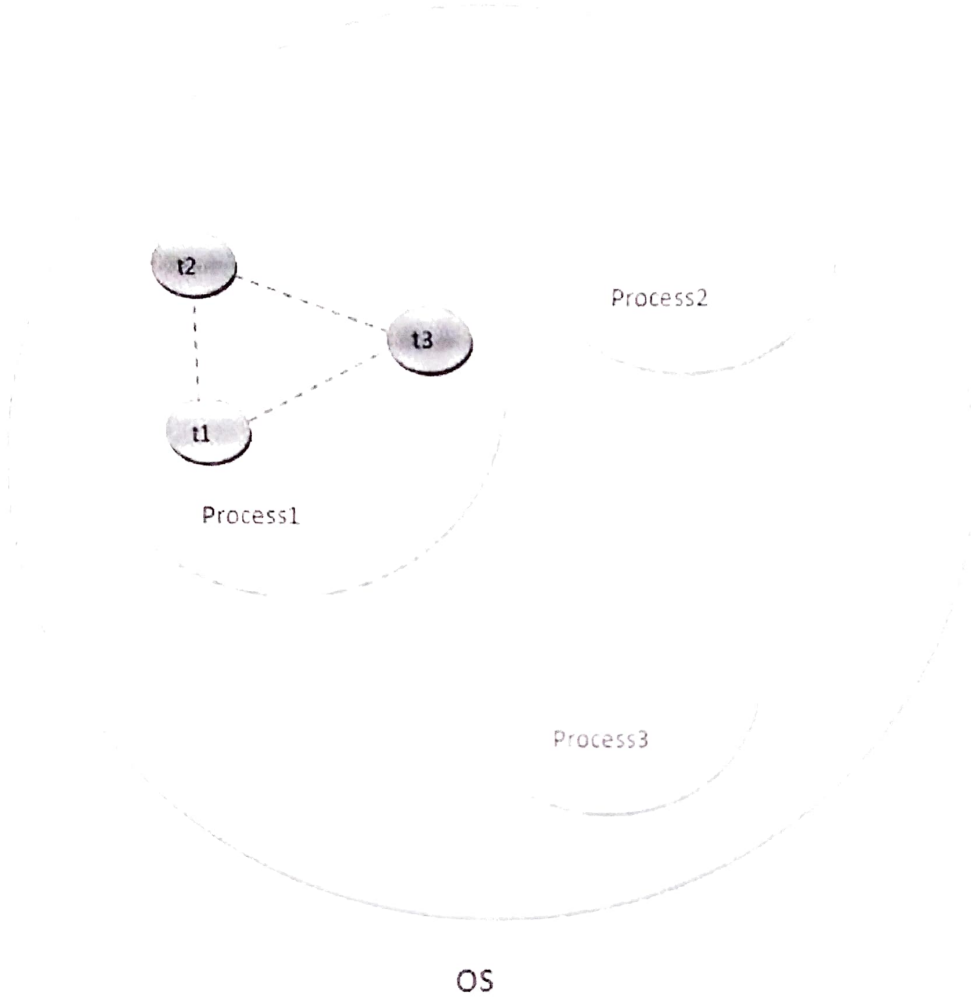
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight sub process, a 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 shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.

Do You Know

- How to perform two tasks by two threads ?
- How to perform multithreading by annonymous class ?
- What is the Thread Schedular and what is the difference between preemptive scheduling and time slicing ?
- What happens if we start a thread twice ?

JAVA - MULTITHREADING

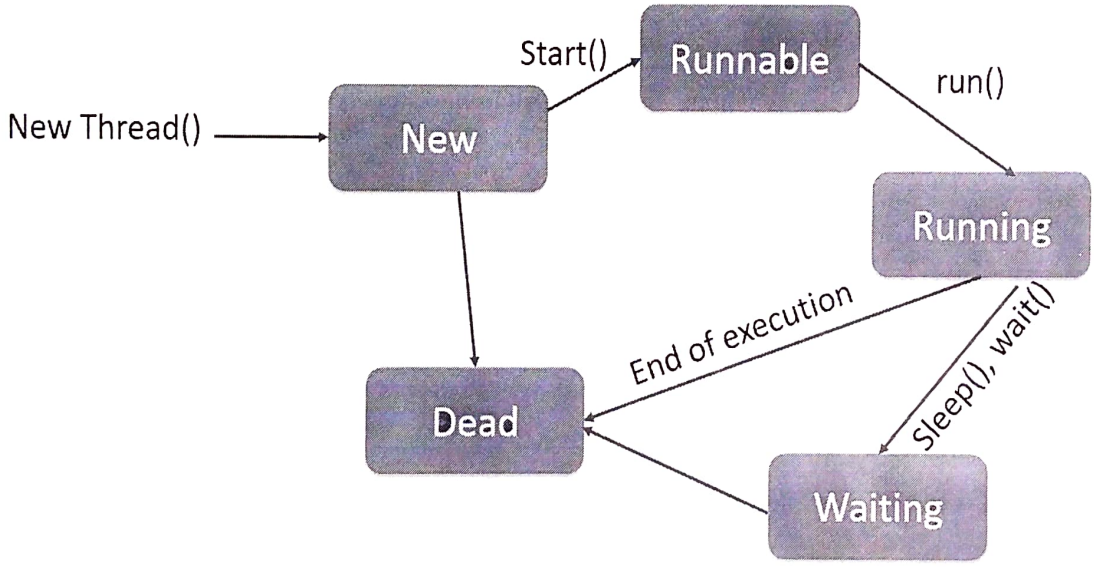
Java is a*multi threaded programming language* which means we can develop multi threaded program using Java. A multi threaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multi threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multi threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

- **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 the waiting thread to continue executing.
- **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** ***Dead*****:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY *aconstantof*1 and MAX_PRIORITY *aconstantof*10. By default, every thread is given priority NORM_PRIORITY *aconstantof*5.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a run method provided by **Runnable** interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

```
public void run( )
```

Step 2:

At second step you will instantiate a **Thread** object using the following constructor:

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

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling **start** method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
```

```
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start** method, which executes a call to run method. Following is simple syntax of start method:

```
void start( );
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      ThreadDemo T1 = new ThreadDemo( "Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo( "Thread-2");
      T2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

Thread Methods:

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	**public void start** Starts the thread in a separate path of execution, then invokes the run method on this Thread object.
2	**public void run** If this Thread object was instantiated using a separate Runnable target, the run method is invoked on that Runnable object.
3	**public final void setName***Stringname* Changes the name of the Thread object. There is also a getName method for retrieving the name.
4	**public final void setPriority***intpriority* Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon***booleanon* A parameter of true denotes this Thread as a daemon thread.
6	**public final void join***longmillisec* The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	**public static void yield** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep***longmillisec*

Causes the currently running thread to block for at least the specified number of milliseconds.

3 **public static boolean holdsLock*Objectx***

Returns true if the current thread holds the lock on the given Object.

4 **public static Thread currentThread**

Returns a reference to the currently running thread, which is the thread that invokes this method.

5 **public static void dumpStack**

Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable**:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
      {
         System.out.println(message);
      }
   }
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                             + " guesses " + guess);
          counter++;
      }while(guess != number);
```

```
      System.out.println("** Correct! " + this.getName()
                         + " in " + counter + " guesses.**");
   }
}
```

Following is the main program which makes use of above defined classes:

```
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(bye);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

    thread4.start();
      System.out.println("main() is ending...");
   }
}
```

This would produce the following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.......
```

Major Java Multithreading Concepts:

While doing Multithreading programming in Java, you would need to have the following concepts very handy:

1. **public void run():** is 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 miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int 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):** changes the name of the thread.
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():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
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.

I. Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
1. Class Table{
2.
3. void printTable(int n){//method not synchronized
4.   for(int i=1;i<=5;i++){
5.     System.out.println(n*i);
6.     try{
7.      Thread.sleep(400);
8.     }catch(Exception e){System.out.println(e);}
9.    }
10.
11. }
12. }
13.
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. class TestSynchronization1{
35. public static void main(String args[]){
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }
```

```
Output: 5
        100
        10
        200
        15
        300
        20
```

400

25

500

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
1.  //example of java synchronized method
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.  }
12. }
13.
14. class MyThread1 extends Thread{
15. Table t;
16. MyThread1(Table t){
17. this.t=t;
18. }
19. public void run(){
20. t.printTable(5);
21. }
22.
23. }
24. class MyThread2 extends Thread{
25. Table t;
26. MyThread2(Table t){
27. this.t=t;
28. }
29. public void run(){
30. t.printTable(100);
31. }
32. }
33.
34. public class TestSynchronization2{
35. public static void main(String args[]){
36. Table obj = new Table();//only one object
37. MyThread1 t1=new MyThread1(obj);
38. MyThread2 t2=new MyThread2(obj);
```

```
39. t1.start();
40. t2.start();
41. }
42. }
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

```
1.  //Program of synchronized method by using annonymous class
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11. }
12. }
13.
14. public class TestSynchronization3{
15. public static void main(String args[]){
16. final Table obj = new Table();//only one object
17.
18. Thread t1=new Thread(){
19. public void run(){
20. obj.printTable(5);
21. }
22. };
23. Thread t2=new Thread(){
24. public void run(){
25. obj.printTable(100);
26. }
27. };
28.
29. t1.start();
```

Annonymous class is an inner class without any name and for which only a single object is created

```
30. t2.start();
31. }
32. }
    Output: 5
            10
            15
            20
            25
            100
            200
            300
            400
            500
```

Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
1. synchronized (object reference expression) {
2.   //code block
3. }
```

II. INTER-THREAD COMMUNICATION IN JAVA

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.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

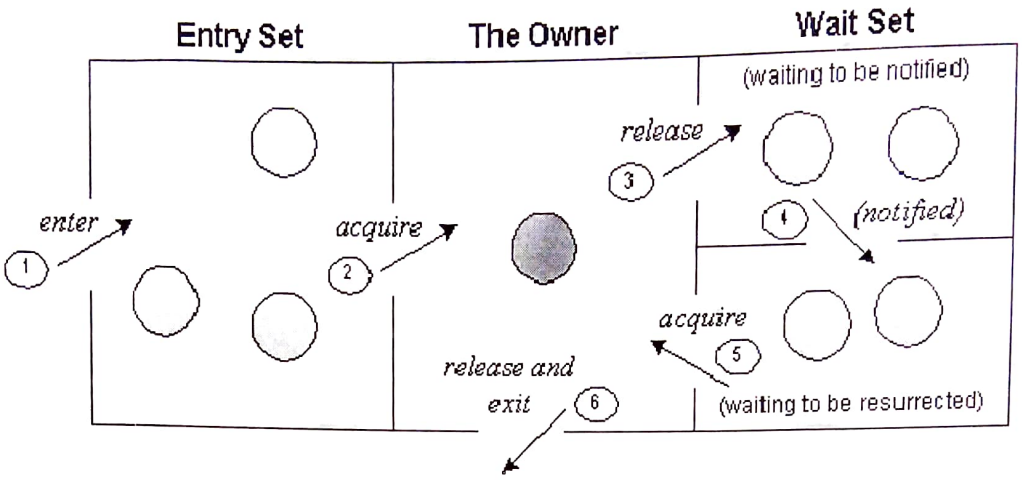
public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method

is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
1. class Customer{
2. int amount=10000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(this.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10. }
11. this.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
14.
15. synchronized void deposit(int amount){
16. System.out.println("going to deposit...");
17. this.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
22.
23. class Test{
24. public static void main(String args[]){
25. final Customer c=new Customer();
26. new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29. new Thread(){
30. public void run(){c.deposit(10000);}
31. }.start();
32.
33. }}
```

```
Output: going to withdraw...
        Less balance; waiting for deposit...
        going to deposit...
        deposit completed...
        withdraw completed
```

III. NTERRUPTING A THREAD:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

- **public void interrupt()**
- **public static boolean interrupted()**
- **public boolean isInterrupted()**

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

```
1. class TestInterruptingThread1 extends Thread{
2. public void run(){
3. try{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }catch(InterruptedException e){
7. throw new RuntimeException("Thread interrupted..."+e);
8. }
9.
10. }
11.
12. public static void main(String args[]){
13. TestInterruptingThread1 t1=new TestInterruptingThread1();
14. t1.start();
15. try{
16. t1.interrupt();
17. }catch(Exception e){System.out.println("Exception handled "+e);}
18.
19. }
20. }
```

Test it Now

download this example

```
Output:Exception in thread-0
        java.lang.RuntimeException: Thread interrupted...
```

```
java.lang.InterruptedException: sleep interrupted
at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

```
1. class TestInterruptingThread2 extends Thread{
2. public void run(){
3. try{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }catch(InterruptedException e){
7. System.out.println("Exception handled "+e);
8. }
9. System.out.println("thread is running...");
10. }
11.
12. public static void main(String args[]){
13. TestInterruptingThread2 t1=new TestInterruptingThread2();
14. t1.start();
15.
16. t1.interrupt();
17.
18. }
19. }
```

Test it Now

download this example

```
Output:Exception handled
       java.lang.InterruptedException: sleep interrupted
       thread is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1. class TestInterruptingThread3 extends Thread{
2.
3. public void run(){
4. for(int i=1;i<=5;i++)
5. System.out.println(i);
6. }
7.
8. public static void main(String args[]){
9. TestInterruptingThread3 t1=new TestInterruptingThread3();
10. t1.start();
11.
12. t1.interrupt();
```

```
13.
14. }
15. }
```

Test it Now

```
Output:1
       2
       3
       4
       5
```

What about isInterrupted and interrupted method?

The isInterrupted() method returns the interrupted flag either true or false. The static interrupted method returns the interrupted flag afterthat it sets the flag to false if it is true.

```
1.  public class TestInterruptingThread4 extends Thread{
2.
3.  public void run(){
4.  for(int i=1;i<=2;i++){
5.  if(Thread.interrupted()){
6.  System.out.println("code for interrupted thread");
7.  }
8.  else{
9.  System.out.println("code for normal thread");
10. }
11.
12. }//end of for loop
13. }
14.
15. public static void main(String args[]){
16.
17. TestInterruptingThread4 t1=new TestInterruptingThread4();
18. TestInterruptingThread4 t2=new TestInterruptingThread4();
19.
20. t1.start();
21. t1.interrupt();
22.
23. t2.start();
24.
25. }
26. }
```

Test it Now

```
Output:Code for interrupted thread
       code for normal thread
       code for normal thread
       code for normal thread
```

IV. REENTRANT MONITOR IN JAVA

According to Sun Microsystems, **Java monitors are reentrant** means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

Let's understand the java reentrant monitor by the example given below:

```
1. class Reentrant {
2.    public synchronized void m() {
3.    n();
4.    System.out.println("this is m() method");
5.    }
6.    public synchronized void n() {
7.    System.out.println("this is n() method");
8.    }
9. }
```

In this class, m and n are the synchronized methods. The m() method internally calls the n() method.

Now let's call the m() method on a thread. In the class given below, we are creating thread using annonymous class.

```
1. public class ReentrantExample{
2. public static void main(String args[]){
3. final ReentrantExample re=new ReentrantExample();
4.
5. Thread t1=new Thread(){
6. public void run(){
7. re.m();//calling method of Reentrant class
8. }
9. };
10. t1.start();
11. }}
```

Test it Now

```
Output: this is n() method
this is m() method
```

v. THE 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.

Syntax:

```
public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException
```

Example of join() method

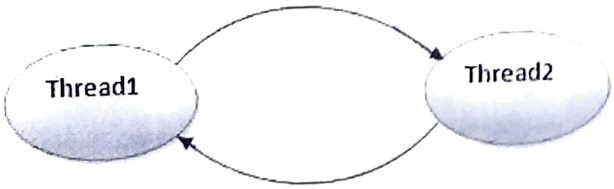
```
1.  class TestJoinMethod1 extends Thread{
2.   public void run(){
3.    for(int i=1;i<=5;i++){
4.     try{
5.      Thread.sleep(500);
6.     }catch(Exception e){System.out.println(e);}
7.    System.out.println(i);
8.    }
9.   }
10. public static void main(String args[]){
11.  TestJoinMethod1 t1=new TestJoinMethod1();
12.  TestJoinMethod1 t2=new TestJoinMethod1();
13.  TestJoinMethod1 t3=new TestJoinMethod1();
14.  t1.start();
15.  try{
16.   t1.join();
17.  }catch(Exception e){System.out.println(e);}
18.
19.  t2.start();
20.  t3.start();
21.  }
22. }
```

Test it Now

```
Output:1
       2
       3
       4
       5
       1
       1
       2
       2
       3
       3
       4
       4
```

VI. DEADLOCK IN JAVA

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in java

```
1.  public class TestDeadlockExample1 {
2.    public static void main(String[] args) {
3.      final String resource1 = "ratan jaiswal";
4.      final String resource2 = "vimal jaiswal";
5.      // t1 tries to lock resource1 then resource2
6.      Thread t1 = new Thread() {
7.        public void run() {
8.          synchronized (resource1) {
9.           System.out.println("Thread 1: locked resource 1");
10.
11.          try { Thread.sleep(100);} catch (Exception e) {}
12.
13.          synchronized (resource2) {
14.           System.out.println("Thread 1: locked resource 2");
15.          }
16.        }
17.      }
18.    };
19.
20.    // t2 tries to lock resource2 then resource1
21.    Thread t2 = new Thread() {
22.      public void run() {
23.        synchronized (resource2) {
24.          System.out.println("Thread 2: locked resource 2");
25.
26.          try { Thread.sleep(100);} catch (Exception e) {}
27.
```

```
28.         synchronized (resource1) {
29.           System.out.println("Thread 2: locked resource 1");
30.         }
31.       }
32.     }
33.   };
34.
35.
36.   t1.start();
37.   t2.start();
38.  }
39. }
40.
```

```
Output: Thread 1: locked resource 1
        Thread 2: locked resource 2
```

VII. DAEMON THREAD IN JAVA

Daemon thread in java is a service provide, 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.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

Simple example of Daemon thread in java

File: MyThread.java

```
1. public class TestDaemonThread1 extends Thread{
2.  public void run(){
3.  if(Thread.currentThread().isDaemon()){//checking for daemon thread
4.   System.out.println("daemon thread work");
5.  }
6.  else{
7.  System.out.println("user thread work");
8. }
9. }
10. public static void main(String[] args){
11.  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
12.  TestDaemonThread1 t2=new TestDaemonThread1();
13.  TestDaemonThread1 t3=new TestDaemonThread1();
14.
15.  t1.setDaemon(true);//now t1 is daemon thread
16.
17.  t1.start();//starting threads
18.  t2.start();
19.  t3.start();
20. }
21. }
```

Test it Now

Output

```
daemon thread work
user thread work
user thread work
```

JAVA - 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 issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite 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. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized block, then it prints counter very much in sequence for both the threads.

Multithreading example without Synchronization:

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces different result based on CPU availability to a thread.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;

   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
        PD = pd;
   }
   public void run() {
     PD.printCount();
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
```

```
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

This produces different result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

Multithreading example with Synchronization:

Here is the same example which prints counter value in sequence and every time we run it, it produces same result.

```
class PrintDemo {
   public void printCount(){
    try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Counter   ---   "  + i );
         }
     } catch (Exception e) {
         System.out.println("Thread  interrupted.");
     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;
   PrintDemo  PD;
```

```
   ThreadDemo( String name,  PrintDemo pd){
       threadName = name;
       PD = pd;
   }
   public void run() {
     synchronized(PD) {
        PD.printCount();
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      PrintDemo PD = new PrintDemo();

      ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
      ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

      T1.start();
      T2.start();

      // wait for threads to end
      try {
         T1.join();
         T2.join();
      } catch( Exception e) {
         System.out.println("Interrupted");
      }
   }
}
```

This produces same result every time you run this program:

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 1  exiting.
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```