Multithreading

Multithreading: Java.lang.Thread, the main Thread, creation of new Threads, multiple Threads, using isAlive() and join(),Thread priority, Synchronization, Communication between Threads, suspending and resuming Threads.

1.Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

(OR)

- ✓ **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.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the 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 an exception occurs in a single thread.

Types of 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)

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.

2) Thread-based Multitasking (Multithreading) ✓

Threads share the same address space.

- ✓ A thread is lightweight.
- ✓ Cost of communication between the thread is low.

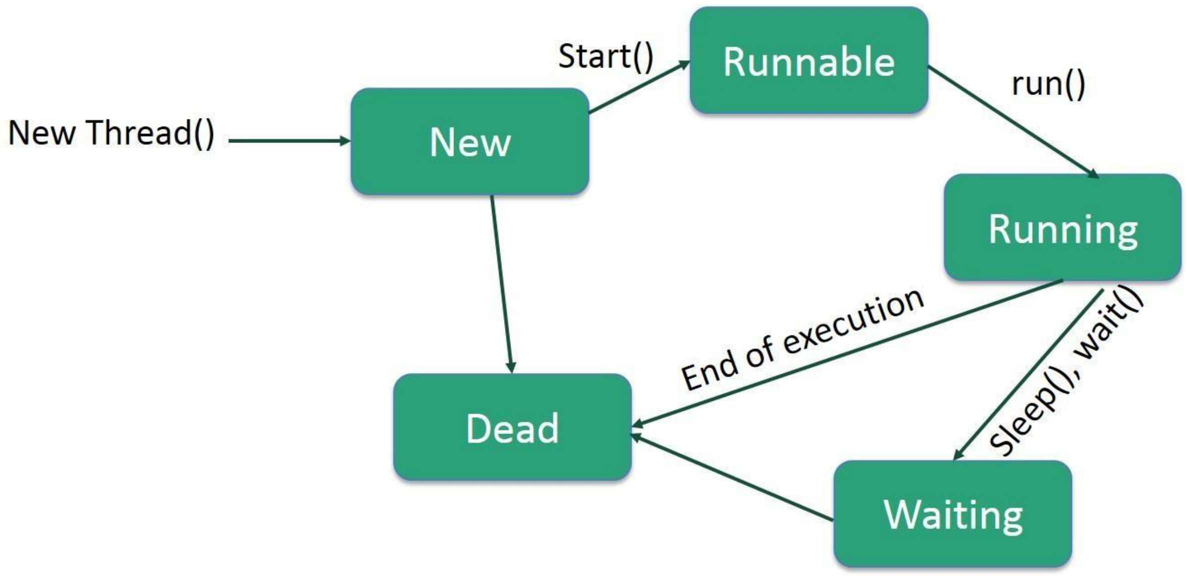
Note: At least one process is required for each thread.

Threads can be created by using two mechanisms :
1. Extending the Thread class
2. Implementing the Runnable Interface.

2.Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
But for better understanding the threads, we are explaining it in the 5 states.
The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)/Waiting
5. Terminated/Dead



Following are the stages of the life cycle −

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**.

A newly created thread object instance on which the **start()** method has not yet been invoked is in the **new** state.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

A thread in **new** state enters the **runnable** state when the **Thread.start()** method is invoked on it. There are 2 important points to note regarding the **runnable** state

- Although the thread enters the **runnable** state immediately on invoking the **start()** method, but it is not necessary that the thread immediately starts executing. A thread runs when the logic it holds in its **run()** method can be executed by the processor. In case the thread logic needs any resource which is not available then the thread waits for the resource to become available.
- Secondly, a thread in **runnable** state may run for some time and then get blocked for a monitor lock, or enter the **waiting/timed_waiting** states as it waits for the opportunity/time to enter **runnable** state again.

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.

A thread enters the **waiting** state when it is made to wait for a go-ahead signal to proceed. The go-ahead in this case is given by another thread and can be given in the following 3 scenarios –

1. **Thread waiting due to Thread.wait() method being called on it:** The other thread can use **Thread.notify()** or **Thread.notifyAll()** to give the go-ahead to the **waiting** thread.
2. **Thread waiting as it itself has asked for joining another thread using Thread.join():** The **waiting** thread gets a go-ahead when the thread its waiting for ends.
3. **Thread waiting due to LockSupport.park()method being invoked on it:** The waiting thread resumes when **LockSupport.unPark()** is called with the parked thread object as the parameter.

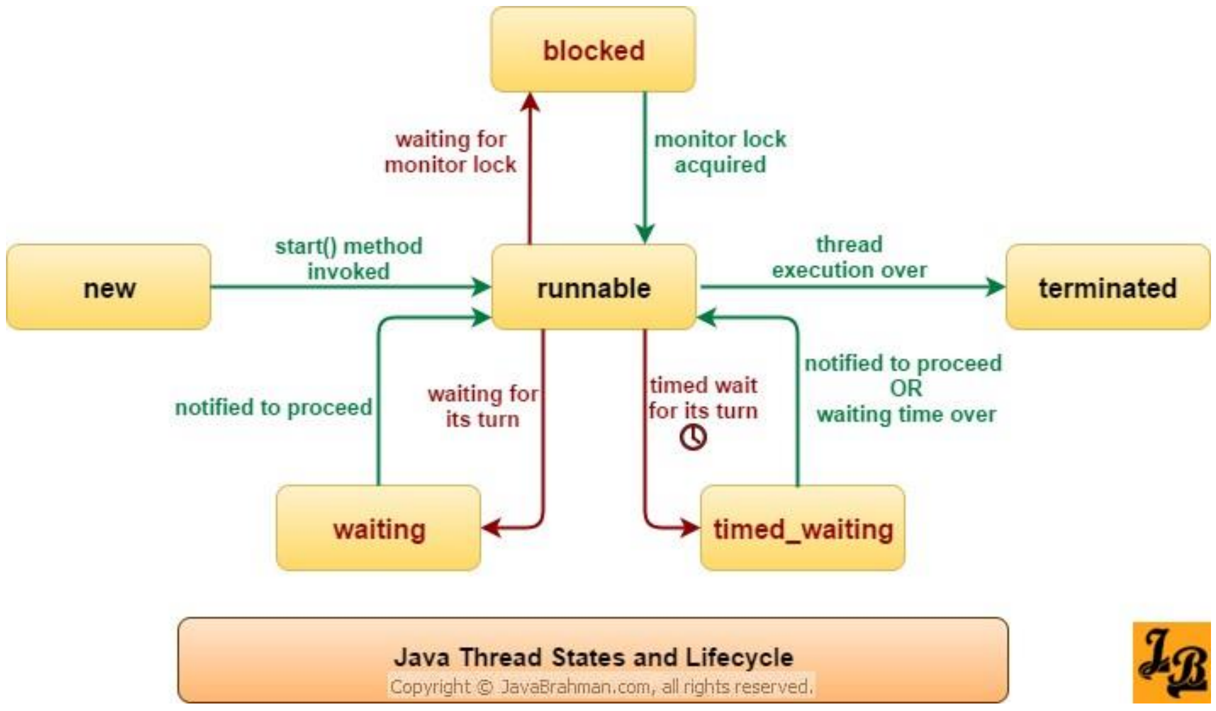
Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- A thread can be made to wait for a pre-determined amount of time in the following ways – 1. Thread made to wait using **Thread.sleep()** method.

2. Threads being asked to wait for a permit for a specified amount of time using **LockSuport.parkNanos()** and **LockSupport.parkUntil()** methods.
3. Threads being made to wait for a fixed amount of time using **Thread.wait(long millis)** or **Thread.join(long millis, int nanos)**. **]**

Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

- A thread enters its 'final resting' state or **terminated** state when it has finished executing the logic specified in its **run()** method. ==



Above diagram shows how a typical thread moves through the different stages of its life cycle. Let us look at these states one-by-one and understand their place in the life of a Java thread

===

3.Creation of new Threads

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.

There are two ways to create a thread:

1. By extending Thread class(Java.lang.Thread)
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- ○ Thread() ○ Thread(String name) ○ Thread(Runnable r)
- ○ Thread(Runnable r,String name)

Commonly used methods of Thread class:

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.

Example1:Java Thread Example by extending Thread class

```
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();
 }
}
```

```
Output:thread is running...
```

Example2:Real time example shows multiple tasks at one time

```
class hi extends Thread
{
          public void run()
          {
                    for(int i=0;i<=5;i++)
                    {
                              System.out.println("hi");
          try{Thread.sleep(1000);}
          catch(Exception e){}
                    }
          }
}

class hello extends Thread
{
          public void run()
          {
                     for(int i=0;i<=5;i++)
                     {
                              System.out.println("hello");
          try{Thread.sleep(1000);         }
                              catch(Exception e){}
                    }
          }
}

public class Main
{
public static void main(String arg[])
{
          hi obj1=new hi();
          hello obj2=new
hello();
          obj1.start();
                              try{Thread.sleep(1000);}
          catch(Exception e){}
          obj2.start();
 }
}
```

Output:

```
hi hi
hell
o hi
hell
```

```
o hi
hell
o hi
hell
o
hell
o hi
hell
o
```

in the above programs two tasks executes simultaneously. The two classes are extended Thread class and Sleep() is used to stop the task for specified time. There is a chance of getting exception that's why we included try-catch block also.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be

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

Output:thread is running...

executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Example 1:Java Thread Example by implementing Runnable interface

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Example 2:

```
class hi implements Runnable
{
  public void run()
         {
```

```
    for(int i=0;i<=5;i++)
    {
    System.out.println("hi");
                    try{Thread.sleep(1000);}
                    catch(Exception e){}
     }
```

```
     }
}

class hello implements Runnable
{
        public void run()
   {
   for(int i=0;i<=5;i++)
   {
   System.out.println("hello");
        try{Thread.sleep(1000);         }
        catch(Exception e){}
   }
        }
}

public class Main
{
public static void main(String arg[])
{
        hi obj1=new hi();
        hello obj2=new hello();

  Thread t1= new Thread(obj1);
  Thread t2= new Thread(obj2);
        t1.start();
                        try{Thread.sleep(1000);}
                        catch(Exception e){}
        t2.start();
 }
}
```

Output:

```
hi hi
hell
o hi
hell
o hi
```

hell
o hi
hell
o
hell
o hi
hell
o in
the
abov
e
prog
ram
s
two
task
s
exec
utes
simu
ltan
eous
ly.
The
two
class
es
are
impl
eme
nted
runn
able
inter
face
and
Slee
p() is
used
to
stop
the
task
for
spec

ified
time
.
Ther
e is
a
chan
ce of
getti
ng
exce
ptio
n
that'
s
why
we
inclu
ded
try-
catc
h
bloc
k
also.

4.Java.lang.Thread class in Java

Thread a line of execution within a program. Each program can have multiple associated threads. Each thread has a priority which is used by thread scheduler to determine which thread must run first. Java provides a thread class that has various method calls inorder to manage the behaviour of threads.

Note: Every class that is used as thread must implement Runnable interface and over ride it's run method.

OR

The **java.lang.Thread** class is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Following are the important points about Thread –

- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority
- Each thread may or may not also be marked as a daemon.
- There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread and,

- the other way to create a thread is to declare a class that implements the Runnable interface

Declaration:

```
public class Thread
extends Object
implements Runnable
```

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:(alredy we discussed in above section with topic name Commonly used methods of Thread class:)

➢ **activeCount():** java.lang.Thread.activeCount() Returns an estimate of the number of active threads in the current thread’s thread group and its subgroups Syntax:

public static int activeCount() Returns: an estimate of the number of active threads in the current thread's thread group and in any other thread group that has the current thread's thread group as an ancestor

➢ **checkAccess():** java.lang.Thread.checkAccess() Determines if the currently running thread has permission to modify this thread Syntax:

public final void checkAccess() **Throws:**
SecurityException - if the current thread is not allowed to access this thread.

➢ **clone():** java.lang.Thread.clone() Throws CloneNotSupportedException as a Thread can not be meaningfully cloned Syntax:

protected Object clone() throws CloneNotSupportedException Throws:
CloneNotSupportedException - always **Returns:**
a clone of this instance

- **currentThread():** java.lang.Thread.currentThread() Returns a reference to the currently executing thread object Syntax:

public static Thread currentThread()
Returns: the currently executing thread

- **dumpStack():** java.lang.Thread.dumpStack() Prints a stack trace of the current thread to the standard error stream Syntax:

public static void dumpStack() Description:
Prints a stack trace of the current thread to the standard error stream. This method is used only for debugging

- **enumerate(Thread[] tarray):** java.lang.Thread.enumerate(Thread[] tarray) Copies into the specified array every active thread in the current thread's thread group and its subgroups Syntax:

public static int enumerate(Thread[] tarray) **Parameters:**
tarray - an array into which to put the list of threads

Returns:
the number of threads put into the array **Throws:**
SecurityException - if ThreadGroup.checkAccess() determines that the current thread cannot access its thread group

- **getAllStackTraces():** java.lang.Thread.getAllStackTraces() Returns a map of stack traces for all live threads

Syntax:
public static Map getAllStackTraces() **Returns:** a Map from Thread to an array of StackTraceElement that represents the stack trace of the corresponding thread **Throws:**
SecurityException - if a security manager exists and its checkPermission method doesn't allow getting the stack trace of thread

- **getContextClassLoader():** java.lang.Thread.getContextClassLoader() Returns the context ClassLoader for this Thread Syntax:

public ClassLoader getContextClassLoader() **Returns:** the context ClassLoader for this Thread, or null indicating the system class loader (or, failing that, the bootstrap class loader)
Throws:
SecurityException - if the current thread cannot get the context ClassLoader

- **getDefaultUncaughtExceptionHandler():** java.lang.Thread.getDefaultUncaughtExceptionHandler() Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception Syntax:

public static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()
Returns: the default uncaught exception handler for all threads

- **getId():** java.lang.Thread.getId() Returns the identifier of this Thread Syntax:

public long getId() **Returns:**
this thread's ID

- **getName():** java.lang.Thread.getName() Returns this thread's name Syntax:

public final String getName()
Returns: this thread's name

- **getPriority():** java.lang.Thread.getPriority() Returns this thread's priority Syntax:

public final int getPriority() **Returns:**
this thread's priority

Like that 35 methods are available.

Methods inherited from class java.lang.Object

- equals
- finalize
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

5.Main thread in Java

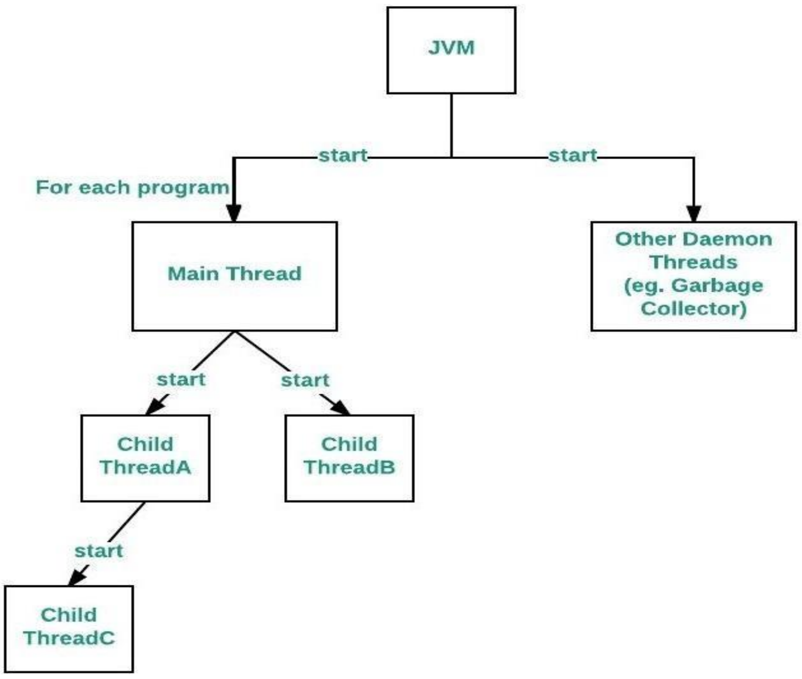
Java provides built-in support for multithreaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program, because it is the one that is executed when our program begins.

Properties :

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions **Flow diagram :**



How to control Main thread

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method *currentThread( )* which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

```
// Java program to control the Main Thread  public
class Test extends Thread
{
    public static void main(String[] args)
    {
            // getting reference to Main thread
            Thread t = Thread.currentThread();

            // getting name of Main thread
            System.out.println("Current thread: " + t.getName());

                // changing the name of Main thread
        t.setName("Geeks");
            System.out.println("After name change: " + t.getName());

            // getting priority of Main thread
            System.out.println("Main thread priority: "+ t.getPriority());

                // setting priority of Main thread to MAX(10)
        t.setPriority(MAX_PRIORITY);

 System.out.println("Main thread new priority: "+ t.getPriority());    for (int i =
0; i < 5; i++)
            {
                    System.out.println("Main thread");
            }

            // Main thread creating a child thread
            ChildThread ct = new ChildThread();

            // getting priority of child thread
            // which will be inherited from Main thread
            // as it is created by Main thread
            System.out.println("Child thread priority: "+ ct.getPriority());

            // setting priority of Main thread to MIN(1)
            ct.setPriority(MIN_PRIORITY);

             System.out.println("Child thread new priority: "+ ct.getPriority());
            // starting child thread
            ct.start();
      }
}
// Child Thread class
class ChildThread extends Thread
{
      @Override
      public void run()
      {
            for (int i = 0; i < 5; i++)
            {
                  System.out.println("Child thread");
            }
      } }
```

Output:

```
Current thread: main
After name change: Geeks
Main thread priority: 5
Main thread new priority: 10
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread priority: 10
Child thread new priority: 1
Child thread
Child thread
Child thread
Child thread
Child thread
```

Relation between the main() method and main thread in Java

For each program, a Main thread is created by JVM(Java Virtual Machine). The "Main" thread first verifies the existence of the main() method, and then it initializes the class. Note that from JDK 6, main() method is mandatory in a standalone java application.

Deadlocking with use of Main Thread(only single thread)

We can create a deadlock by just using Main thread, i.e. by just using a single thread. The following java program demonstrate this.

```
// Java program to demonstrate deadlock
// using Main thread
public class Test
{
          public static void main(String[] args)
       {                  try
                    {
                              System.out.println("Entering into Deadlock");

                              Thread.currentThread().join();

                              // the following statement will never execute
                              System.out.println("This statement will never execute");

                    }

                    catch (InterruptedException e)
                    {
                              e.printStackTrace();
                    }
          }
}
```

Output:

```
Entering into Deadlock
```

Explanation **:**
The statement “Thread.currentThread().join()”, will tell Main thread to wait for this thread(i.e. wait for itself) to die. Thus Main thread wait for itself to die, which is nothing but a deadlock.

6.Multiple Threads

Multithreading in java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications are where two or more threads run concurrently; hence it is also known as Concurrency in Java. This multitasking is done, when multiple processes share common resources like CPU, memory, etc. Each thread runs parallel to each other. Threads don't allocate separate memory area; hence it saves memory. Also, context switching between threads takes less time.

Example of Multi thread:

```
package demotest;
public class GuruThread1 implements Runnable
{
       public static void main(String[] args) {
        Thread guruThread1 = new
Thread("Guru1");        Thread guruThread2 =
new Thread("Guru2");        guruThread1.start();
guruThread2.start();
        System.out.println("Thread names are following:");
        System.out.println(guruThread1.getName());
System.out.println(guruThread2.getName());
    }

    public void run() {
    }
}
Output:
Thread names are following:
Guru1
Guru2
```

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times
- As such the threads are independent, the other threads won't get affected if one thread meets an exception.

7.Java Thread isAlive() method

The **isAlive()** method of thread class tests if the thread is alive. A thread is considered alive when the start() method of thread class has been called and the thread is not yet dead.

This method returns true if the thread is still running and not finished. **Or**
Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

Syntax:

```
final boolean isAlive()
```

Java isAlive method example:

Lets take an example and see how the isAlive() method works. It returns true if thread status is live, false otherwise.

```
public class MyThread extends Thread
{
		public void run()
		{
				System.out.println("r1 ");
				try {
				Thread.sleep(500);
				}
		catch(InterruptedException ie)
		{
				// do something
		}
		System.out.println("r2 ");
		}
		public static void main(String[] args)
		{
				MyThread t1=new MyThread();
				MyThread t2=new MyThread();
				t1.start();
				t2.start();
				System.out.println(t1.isAlive());
				System.out.println(t2.isAlive());
		}
}
```

Output:

```
r1
true
true
r1
r2
r2
```

8.Thread join() method in Java with example

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution).

Why we use join() method?

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.

Example of thread without join() method

If we run a thread without using join() method then the execution of thread cannot be predict. Thread scheduler schedules the execution of thread.

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
                }
                catch(InterruptedException ie){ }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
        }
}
```

```
r1
r1
r2
r2
```

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 ms. At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like `r1 r1 r2 r2`

Example of thread with join() method

In this example, we are using join() method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

```
public class MyThread extends Thread
{
        public void run()
        {
          System.out.println("r1 ");                    try {
               Thread.sleep(500);
               }catch(InterruptedException ie){ }
               System.out.println("r2 ");
     }

     public static void main(String[] args)
     {
               MyThread t1=new MyThread();
               MyThread t2=new MyThread();
               t1.start();

               try{
               t1.join(); //Waiting for t1 to finish
               }catch(InterruptedException ie){}

               t2.start();
     }
}
```

```
r1
r2
r1
r2
```

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using **join()** with **t1**, then **t1** will execute for that time, and then **t2** will join it.

```
public class MyThread extends Thread
{
            MyThread(String str){
         super(str);
              }

     public void run()
     {
          System.out.println(Thread.currentThread().getName());
     }
     public static void main(String[] args)
     {
          MyThread t1=new MyThread("first thread");
MyThread t2=new MyThread("second thread");
t1.start();

          try{
               t1.join(1500);     //Waiting for t1 to finish
          }catch(InterruptedException ie){
           System.out.println(ie);
          }

          t2.start();
try{
            t2.join(1500);     //Waiting for t2 to finish
          }catch(InterruptedException ie){
           System.out.println(ie);
          }
     } }
```

Doing so, initially t1 will execute for 1.5 seconds, after which t2 will join it.

9.Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10. **Example of priority of a Thread:**

```
class TestMultiPriority1 extends Thread{
public void run(){
System.out.println("running thread name is:"+Thread.currentThread().getName());
System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

 }

public static void main(String args[]){
 TestMultiPriority1 m1=new TestMultiPriority1();
TestMultiPriority1 m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
 m2.start();
  } }
```

Output:
```
running thread name is:Thread-
0       running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

Example: Fetch Thread Priority

If we don't set thread priority of a thread then by default it is set by the JVM. In this example, we are getting thread's default priority by using the getPriority() method.

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }

        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                MyThread p3 = new MyThread();
                p1.start();
        System.out.println("P1 thread priority : " + p1.getPriority());
        System.out.println("P2 thread priority : " + p2.getPriority());
        System.out.println("P3 thread priority : " + p3.getPriority());

        }
}
```

```
P1 thread priority : 5
Thread Running...
P2 thread priority : 5
P3 thread priority : 5
```

Example: Thread Constants/MIN,MAX,NORM priorities

We can fetch priority of a thread by using some predefined constants provided by the Thread class. these constants returns the max, min and normal priority of a thread.

```
class MyThread extends Thread
{
		public void run()
		{
			System.out.println("Thread Running...");
		}

		public static void main(String[]args)
		{
			MyThread p1 = new MyThread();
			p1.start();
System.out.println("max thread priority : " + p1.MAX_PRIORITY);
System.out.println("min thread priority : " + p1.MIN_PRIORITY);
System.out.println("normal thread priority : " + p1.NORM_PRIORITY);

		}
}
```

```
Thread Running...
max thread priority : 10
```

```
min thread priority : 1
normal thread priority : 5
```

Example : Set Priority

To set priority of a thread, setPriority() method of thread class is used. It takes an integer argument that must be between 1 and 10. see the below example.

```
class MyThread extends Thread
{
		public void run()
		{
			System.out.println("Thread Running...");
		}

		public static void main(String[]args)
		{
			MyThread p1 = new MyThread();
			// Starting thread
			p1.start();
			// Setting priority
			p1.setPriority(2);
			// Getting priority
			int p = p1.getPriority();

			System.out.println("thread priority : " + p);

		}
}
```

```
thread priority : 2
Thread Running...
```

Example:

In this example, we are setting priority of two thread and running them to see the effect of thread priority. Does setting higher priority thread get CPU first. See the below example.

```
class MyThread extends Thread
{
		public void run()
		{
		System.out.println("Thread Running... "+Thread.currentThread().getName());
		}

		public static void main(String[]args)
```

```
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                // Starting thread
                p1.start();
                p2.start();
                // Setting priority
                p1.setPriority(2);
                // Getting -priority
                p2.setPriority(1);
                int p = p1.getPriority();
                int p22 = p2.getPriority();

                System.out.println("first thread priority : " + p);
                System.out.println("second thread priority : " + p2);

        }
}
```

```
Thread Running... Thread-0
first thread priority : 5
second thread priority : 1
Thread Running... Thread-1
```

Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

10.Synchronization in Java

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. 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.

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.

Thread Synchronization

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

1. **Mutual Exclusive**
    1. Synchronized method.
    2. Synchronized block.
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

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:

```
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);}
  }

 }
}

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);
}
}

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();
}
}
```

```
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.

```
//example of 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);}
   }
 }
}

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 TestSynchronization2{
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();
}
}
```

```
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

```
synchronized (object reference expression) {
  //code block
}
```

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
System.out.println(n*i);        try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the 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);
}
}

public class TestSynchronizedBlock1{
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();
}
}
```

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

11.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 time.

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.

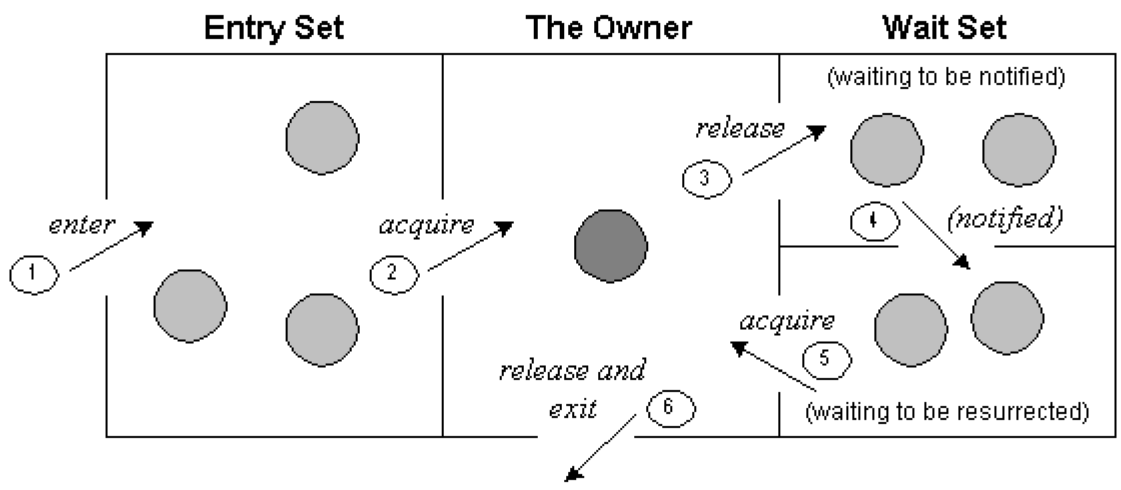
Syntax: public final void notify()

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.

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

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

12.Java Thread suspend() method

The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Syntax

```
public final void suspend()
```

Return

This method does not return any value.

Exception

SecurityException: If the current thread cannot modify the thread.

Example

```
public class JavaSuspendExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
try
{
            // thread to sleep for 500 milliseconds
sleep(500);
             System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      // creating three threads
      JavaSuspendExp t1=new JavaSuspendExp ();
      JavaSuspendExp t2=new JavaSuspendExp ();
      JavaSuspendExp t3=new JavaSuspendExp ();
      // call run() method
      t1.start();
t2.start();
      // suspend t2 thread
t2.suspend();
      // call run() method
t3.start();
   }
}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

If we remove t2.suspend();   then the output of above program as followes

```
Thread-2
Thread-1
Thread-0
1
1
1
Thread-0
Thread-1
2
Thread-2
2
2
Thread-1
3
Thread-0
3
Thread-2
3
Thread-1
4
Thread-0
4
Thread-2
```

13.Java Thread resume() method

The **resume()** method of thread class is only used with suspend() method. This method is used to resume a thread which was suspended using suspend() method. This method allows the suspended thread to start again. **Syntax**

public final void resume()

Return value

This method does not return any value.

Exception

SecurityException: If the current thread cannot modify the thread.

Example

```
public class JavaResumeExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
try
{
            // thread to sleep for 500 milliseconds
sleep(500);
             System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      // creating three threads
      JavaResumeExp t1=new JavaResumeExp ();
      JavaResumeExp t2=new JavaResumeExp ();
      JavaResumeExp t3=new JavaResumeExp ();
      // call run() method
      t1.start();
t2.start();
      t2.suspend(); // suspend t2 thread
      // call run() method
      t3.start();
      t2.resume(); // resume t2 thread
   }
}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-1
1
Thread-0
2
Thread-2
2
Thread-1
2
Thread-0
3
Thread-2
3
Thread-1
3
Thread-0
4
Thread-2
4
Thread-1
4
```