Introduction to Thread:

Program in the state of execution is called as a process and the operating system which allows more than one process to execute at a same time is called as multi processing. **A piece or set of code in the program is called thread and the program which allows such blocks of statement to execute at the same time is called multi threading.** In short a "multithreaded program contains two or more parts that can run concurrently. Multithreading is a specialized form of multitasking".

Take an example of windows environment, which allows notepad, word, excel, etc to open up at the same time and allows execution of these program at the same time is called multi processing and the CPU which handle this is called heavy weight process execution.

Now take an example of video game, where process has to take care of movement of object, audio signals and increment of scores. Although these 3 things come in a single program, each task should be executed in its own environment and the user should feel that these entire tasks occur at the same time. This is an example of multi threading. **It is a light weight process**. Multithreading enables you to write efficient program and make maximum use of CPU time because ideal time can be kept minimum.

6.1 Creating Thread:

You can create thread in java using 2 ways:

1. By creating a class which **extends Thread** Class
2. By creating a class which **implements Runnable** Interface.

Choice of using either extends of Thread class or implementing Runnable interface depends upon requirement. if we require to extend another class then we have no choice but to implement Runnable interface because java can't have 2 super classes.

Thread class & Runnable interface overrides run() method. You can write necessary code required by the thread in this method.

```
Public void run()
{
// statement to implementing thread
}
```

Run() method can call other methods, use other classes & declare variable just like main thread.

Example:

Let's take an example of a simple program which prints time in following manner

```
hh:mm:ss
0:0:0
0:0:1
:
:
0:0:59
0:1:0
```

```
public class demo extends Thread
{
public static void main(String args[])
{
int hh,mm,ss;
try
{
        Thread t=new Thread("demo");
        for(hh=0;hh<24;hh++)
                {
                for(mm=0;mm<=59;mm++)
                        {
                        for(ss=0;ss<=59;ss++)
                                {
                                System.out.println(hh+":"+mm+":"+ss);
                                t.sleep(1000);
                                }//ss
                        }//mm
```

```
                }//hh
}//try
catch(Exception e)
{
System.out.println("Error:" + e);
}//catch

}//main
}//class
```

6.2.1 Implements Runnable:

Runnable interface has only one method run(). Once you declared any class which implements Runnable interface, you have to create object of thread type. There are several constructors to define thread but one of them is:
Thread(Runnable threadobject, String threadname)
Ex: Thread t = new Thread(this,"demo thread")

Example of implementing Runnable interface:

```
import java.lang.Thread;
class demo implements Runnable
{
        demo()
        {
        Thread t = new Thread(this,"demo thread");
        System.out.println("Child thread"+t);
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println("child thread:"+i);
                        Thread.sleep(500);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println("Child Interrupted");
                  }//catch
        System.out.println("Exit from child");

        }
```

```
};
class drdemo
{
public static void main(String a[])
{
demo d = new demo();// call to constructor.
//or
//new demo();// call to constructor.
try
        {
                for (int i =5;i>0;i--)
                        {
                        System.out.println("Main thread:"+i);
                        Thread.sleep(1000);
                        }
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch
        System.out.println("Exit from main");
}
};
```

We can not use super("demo thread") because Runnable is an interface & not a class.
We can not use only start() method because Runnable interface has one & only one method i.e. run().

6.2.2 Extending Thread:

To extends thread;

1. Create class extending Thread class
2. Implement run() method
3. Create an object & call the start method to initiate thread execution.

To create and run instance of thread class use following statments:
Demo d1 = new Demo(); ⟶[Equivalent to **new Demo();**]
d1.start();
OR
new Demo() . start(); (compact nature of above two statement)

Example of extending Thread class:

```
import java.lang.Thread;
```

```
class demo extends Thread
{
	demo()
	{
	super("demo thread");		//or Thread t = new Thread(this,"demo thread");
	System.out.println("Child thread"+this);// or System.out.println("Child thread"+t);
	start(); // or t.start();
	}

	public void run()
	{
		try{
			for (int i =5;i>0;i--)
			{
			System.out.println("child thread:"+i);
			Thread.sleep(500);
			}
		  }//try

		catch(InterruptedException e)
		  {
			System.out.println("Child Interrupted");
		  }//catch
	System.out.println("Exit from child");

	}
};
class dtdemo
{
public static void main(String a[])
{
demo d = new demo();// call to constructor.
//	OR
// new demo();// call to constructor.
try
	{
		for (int i =5;i>0;i--)
			{
			System.out.println("Main thread:"+i);
			Thread.sleep(1000);
			}
	}//try

	catch(InterruptedException e)
	{
```

```
System.out.println("Main Interrupted");
```

```
        }//catch
        System.out.println("Exit from main");
}
};
```

1. Because Thread is a class, we can use "super" keyword to access Thread class's property. here super("any thread name") will call Thread class constructor & allocates specific name to current thread. Ex. super("demo thread") allocates "demo thread" name to current thread & which is similar to Thread t = new Thread(this,"demo thread");
2. To access current thread we can make use of "this" keyword Ex. System.out.println("Child thread"+this);
3. Once we have directly access thread without any object, we can directly call start methd Ex. start();

6.2.3 Multi threading:

An example which demonstrates, more than 2 threads running at the same time.

Example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread"+t);
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(name+":"+i);
                        Thread.sleep(1000);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println(name+"Interrupted");
                  }//catch
        System.out.println(name+" Exiting");
```

```
        }
};

class mrdemo
{
public static void main(String a[])
{
new demo("one"); // demo d1 = new demo("one");
new demo("two"); // demo d2 = new demo("two");
new demo("three"); // demo d3 = new demo("three");

        try
        {
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

6.2 Methods

The Thread class defines several methods that help manage threads. The table below displays the same:

Method	Meaning
getName	It returns the name of the thread.
getPriority	It returns the priority of the thread.
isAlive	It determines if a thread is still running
Join	Wait for a thread to terminate
Run	Entry point for the thread
Sleep	Suspend a thread for a period of time
Start	Start a thread by calling its run method
setPriority	It changes the priority of the thread.
yield()	It causes current thread on halt and other threads to execute.
setName	It changes the name of the thread

Note: Rather than above methods, there is one more thread which work as a service provider thread name as Daemon thread.

Daemon thread: daemon thread is a service provider thread that provides services to the user thread. It is a low priority thread. Its life depend on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically. It has no role in life than to serve user threads. It provides services to user threads for background supporting tasks. There are many java daemon threads running automatically e.g. gc, finalizer etc.

Daemon thread methods by Thread class

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

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

getName & setName example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread:"+t.getName());//one
        t.setName("Thread:"+name);
        System.out.println("New Name of thread is:"+t.getName());//Thread:one
        t.start();
        }
        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(t.getName()+":"+i);
                        Thread.sleep(1000);
                        }
                  }//try

                catch(InterruptedException e)
                  {
                        System.out.println(t.getName()+"Interrupted");
                  }//catch
```

```
        System.out.println(name+" Exiting");

        }
};
class mrgetset
{
public static void main(String a[])
{
new demo("one"); // demo d1 = new demo("one");
new demo("two"); // demo d2 = new demo("two");
new demo("three"); // demo d3 = new demo("three");

try
        {
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

Using isAlive() & join():

We generally used sleep() within main to finish the main thread at last. This is used to ensure that all child thread s terminate prior to the main thread. However it will hardly work in predictable way & it will raise a question: how one thread know when another thread has ended?
However java (thread) provides solution for this question.
There is 2 way to determine whether thread has finished.
First is isAlive() & second is join() method.

isAlive():

This method return true if the thread upon which it is called is still running, otherwise it will return false.
General form of this method is:
Final boolean isAlive()

Join():

This method waits until thread on which it is called terminates. Its name comes from the concept of calling thread waiting until the specified thread joins it.

Additional form of join() allows you to specify maximum amount of time on specified thread to terminate. General form of this method is:
Final void join() throws InterruptedException

join Example:

```
import java.lang.Thread;
class callme
{
        void call(String msg)
        {
        System.out.println("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted:");
                }
        System.out.println("]");
        }
}
class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;
public demo(callme targ,String s)
        {
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
                trgt.call(msg);
        }
}// class demo

class join
{
public static void main(String a[])
{
```

```
callme target = new callme();
demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");
try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}
catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//main
};
```

Here is the o/p produce by this program:
[hello[synchronized[world]]] // it may be different.

getName/setName/isAlive/join example:

```
import java.lang.Thread;
class demo implements Runnable
{
String name;
Thread t;
        demo(String tname)
        {
        name=tname;
        t = new Thread(this,name);
        System.out.println("New Child thread:"+t.getName());
        t.setName("Thread:"+name);
        System.out.println("New Name of thread is:"+t.getName());
        t.start();
        }

        public void run()
        {
                try{
                        for (int i =5;i>0;i--)
                        {
                        System.out.println(t.getName()+":"+i);
                        Thread.sleep(1000);
                        }
```

```
                }//try
                catch(InterruptedException e)
                {
                        System.out.println(t.getName()+"Interrupted");
                }//catch
        System.out.println(name+" Exiting");

        }
};
class mrgsaj
{
public static void main(String a[])
{
 demo d1 = new demo("one");
 demo d2 = new demo("two");
 demo d3 = new demo("three");

System.out.println(d1.t.isAlive());
System.out.println(d2.t.isAlive());
System.out.println(d3.t.isAlive());

        try
        {
                d1.t.join();
                d2.t.join();
                d3.t.join();
                Thread.sleep(10000);
        }//try


        catch(InterruptedException e)
        {
                        System.out.println("Main Interrupted");
        }//catch

        System.out.println("Main Exiting");
}
};
```

6.3 Race condition:

Race condition in **Java** occurs in a multi-threaded environment when more than one thread try to access a shared resource (modify, write) at the same time.

Let's say, When more than one thread want to complete it task in that process then it simply snatches the control of the CPU so intern you can't specify which process will execute first?, because all threads are in hurry to complete its process which is called as race condition.

6.4 Synchronization:

When 2 or more threads need to access a shared resources they need some way to ensure that the resources will be used by **only 1 thread at a time**. The process by which it is achieved is called synchronization.

Key to synchronization is the concept of the monitor (also called semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All the other threads attempting to enter a locked monitor will be suspended until the first thread exists the monitor. These other threads are said to be waiting for the monitor, so here instead of getting the garbage output on monitor. By giving time slice to each thread, java helpsto implement the synchronization (which allows to have mutex). A thread that owns a monitor can reenter the same monitor if it so desire.

You can synchronize your code in either of 2 ways. Both involve the use of the synchronized keyword.

1. By using synchronized method
2. By using synchronized statement

By using synchronized method:

Earlier example produce o/p like: [hello[synchronized[world]]]. This can be achieve by calling sleep(), the call() method allows execution to switch to another thread. This result in mixed o/p. to fix the o/p of preceding program you must serialize access to call(). That is, you must restrict its access to only 1 thread at a time. To do so, you simply need to precede call()'s definition with the keyword synchronized.

Ex:
```
class callme {
        synchronized void call(String msg){ ..............}
}
```

Example:

```
import java.lang.Thread;
class callme
{
    synchronized void call(String msg)
    {
    System.out.println("["+msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted:");
        }
    System.out.println("]");
```

```
        }
}

class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;

public demo(callme targ,String s)
        {
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
                trgt.call(msg);
        }
}// class demo

class synchro
{
public static void main(String a[])
{
callme target = new callme();
demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");
try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}
catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//psvm
};
```

After synchronized has been added to call(), the o/p of the program is as follows:

[hello][synchronized][world]

By using synchronized statement:

Using synchronized keyword to methods will not work in all the cases. Consider the case, that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Further this class was not created by you, but by the third party & you don't know how to access the source code. Thus you can't add synchronized to the methods of that class. How to synchronize access to an object of this class? But java provides solution to this problem: you put calls to the methods defined by this class inside synchronized block.

Syntax: synchronized(object){ ……. }

```
Ex: public void run()
        {
            synchronized(trgt){
                                        trgt.call(msg);
                                    }
        }
```

Example:

```
import java.lang.Thread;
class callme
{
        void call(String msg)
        {
        System.out.println("["+msg);
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println("Interrupted:");
                }
        System.out.println("]");
        }

}
class demo implements Runnable
{
String  msg;
callme trgt;
Thread t;
public demo(callme targ,String s)
        {
```

```
        trgt=targ;
        msg=s;
        t=new Thread(this);
        t.start();
        }
public void run()
        {
             synchronized(trgt){
                                        trgt.call(msg);
                                  }
        }

}// class demo

class synchro
{
public static void main(String a[])
{
callme target = new callme();

demo d1 = new demo(target,"Hello");
demo d2 = new demo(target,"Synchronized");
demo d3 = new demo(target,"World");

try
{
        d1.t.join();
        d2.t.join();
        d3.t.join();
}

catch(InterruptedException e)
{
        System.out.println("Interrupted");
}//catch
}//psvm
};
```

This program will produce the same o/p like:
[hello][synchronized][world]

6.5 Thread priority:

Thread priority are used by scheduler to decide when each thread should be allowed to run. In theory, higher priority thread gets more CPU time than low priority thread but in practice the amount of CPU time gets by any thread depends upon several factors.

The higher priority thread can also preempt lower priority thread, when lower priority thread is running & higher priority thread resumes, it will preempt lower priority thread.

In theory, threads of equal priority get equal access to the CPU. But you need to be careful because java designed to works in wide environment. Threads that shares same priority should yield (surrender/give up) control once in while. And this situation may leads to race condition & your program may behave differently because of this kind of situation. You can control behavior through setting priority to threads.

To set thread's priority, use the **setPriority()** method, which is a member of thread. General form of this method is:

final void setPriority(int level)

Here level specifies the new priority for the calling thread. The value for the level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently these values are 1and 10 respectively. To return thread to default priority use NORM_PRIORITY, which is currently 5.You can obtained the current priority setting by calling the **getPriority()** method of Thread. General form of this method is:

final int getPriority()

6.6 Interthread communication:

Multithreading replaces event loop programming by dividing your task into discrete & logical unit. Threads also provide secondary benefit to do away with polling. Polling is usually implemented by loop. i.e. Used to check some condition repeatedly. Once the condition is true appropriate action is taken. This wastes CPU time. This situation is undesirable.

[For example…… Let us consider producer & consumer problem. Now suppose that producer has to wait for generating data until consumer finishes his job. In polling system, consumer wasting many CPU cycle while it waited for the producer to produce. & once producer was finished, it would start polling **(asking/questioning)**, wasting more CPU cycle waiting for consumer to finish & so…on.]

To avoid polling, java includes an interprocess communication mechanism through **wait(),notify(),notifyAll().** These methods are the final methods in object (so all class have them) & should be call only within the synchronized method.

1. wait(): tells the calling thread to give up the monitor & go to sleep until some other thread enters the same monitor & calls notify().
   final void wait() throws Interrupted Exception
2. notify(): wakes up the first thread that called wait() on the same object.
   final void notify()
3. notify All(): wakes up all the threads that called wait() on the same object.
   final void notifyAll()

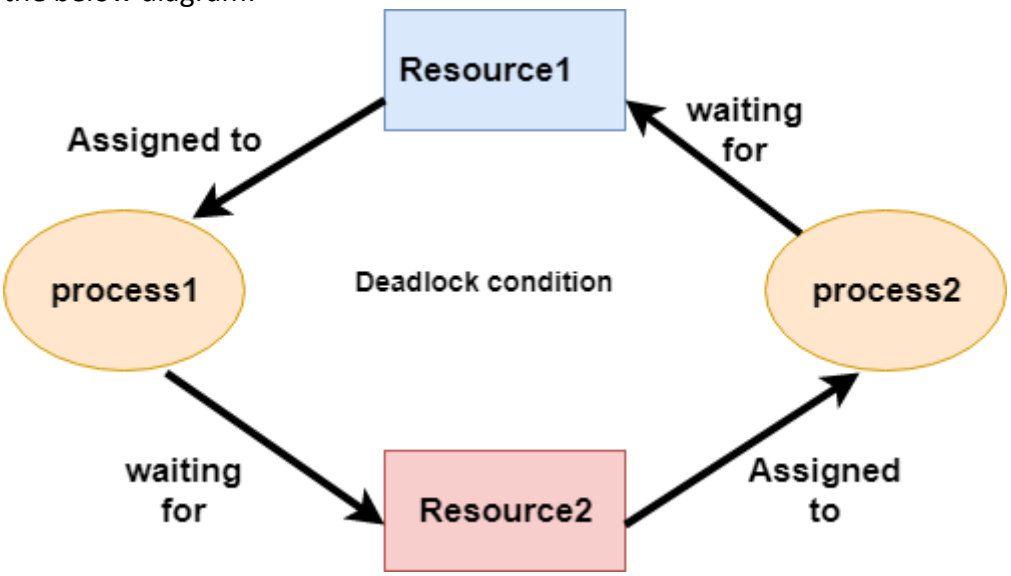
Additional forms of wait() exist which allows you to specify a period of time to wait.

6.7 Deadlock:

Deadlock in java is a part of multithreading, which occurs when two threads have a circular dependency on a pair of synchronized objects.

Deadlock in Java is a situation that occurs 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 the first thread.

Since both threads are waiting for each other to release the lock simultaneously, this condition is called deadlock in Java. The object locks acquired by both threads are not released until their execution is not completed. For example Process1 is holding Resource1 and waiting for resource2, which is acquired by process2, and process2 is waiting for resource1 is explained in the below diagram:



In above diagram process1 and process2 are two threads that are in a deadlock. The process1 holds the lock for the resource R1 and waits for resource R2 that is acquired by the process2. At the same time, the process holds the lock for the resource R2 and waits for R1 resource that is acquired by the process1. But process2 cannot release the lock for resource R2 until it gets hold of resource R1.

Since both threads are waiting for each other to unlock resources R1 and R2, therefore, these mutually exclusive conditions are called deadlock in Java.

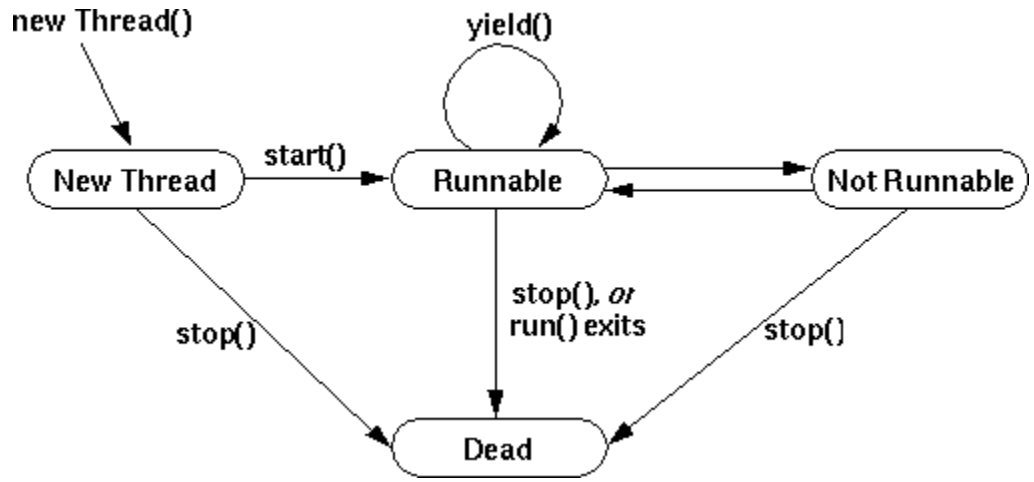
How to avoid/release deadlock:

1. A deadlock in a program can be prevented if any of the above four conditions are not met.
2. It can be achieved if a thread is to force to hold only one resource at a time. If it needs another resource, it must first release that resource that is held by it and then requests another.
3. It can be achieved by acquiring resources (locks) in a specific order and releasing them in reverse order so that a thread can only continue to acquire a resource if it held the other one.

Deadlock is a difficult error to debug for two reasons:

1. In general, it occurs only rarely, when the two threads time-slice in just the right way.
2. It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

6.8 Thread State:



New Thread:

The following statement creates a new thread but does not start it, thereby leaving the thread in the "New Thread" state.

Thread t = new Thread();

It is an empty thread. No system resources have been allocated for it yet. So you can only start or stop this thread.

Runnable:

Consider the statement t.start()in following 2 lines of code. It will start the thread and put thread in runnable state.

Thread t = new Thread();

t.start();

The start() method allocates system resources, schedules the thread, and calls the threads's run() method and not actually run the method. When the thread actually *runs* is determined by the **scheduler**

Not Runnable:

A thread enters the "Not Runnable" state when one of these four events occurs:

- Someone invokes its sleep() method.
- Someone invokes its suspend() method.
- The thread uses its wait() method to wait on a condition variable.
- The thread is blocking on I/O.

Dead:

- a thread enters in the **dead** state when it's run() method completes.
- An **interrupt** does not kill a thread.
- The **destroy ()** method kills a thread dead but does not release any of it's object locks.
- The stop () method causes a sudden termination of a Thread's run() method. For ex:t.stop();