eg#1.

```
class MyThread extends Thread
{
	@Override
	public void run()
	{
		for (int i=0;i<10 ;i++ )
		{
			System.out.println("child thread");
		}
	}
}
public class Test
{
	//JVM ---> main thread
	public static void main(String[] args)throws Exception{

		//instantiation of a thread
		MyThread t = new MyThread();

		//starting a thread
		t.start();

		//job of main thread
		for (int i=0;i<5 ; i++)
		{
			System.out.println("main thread");
			Thread.sleep(1000);
		}
	}
}
```

Behind the scenes

1. Main thread is created automatically by JVM.
2. Main thread creates child thread and starts the child thread.

Case1:ThreadScheduler

==========

If multiple threads are waiting to execute,then which thread will execute 1st is decided by ThreadScheduler which is part of JVM.

In case of MultiThreading we can't predict the exact output only possible output we can expect.

Since jobs of threads are important,we are not interested in the order of execution it should just execute such that performance should be improved.

case2: diff b/w t.start() and t.run()

if we call t.start() and seperate thread will be created which is responsible to execute run() method.

if we call t.run(), no seperate thread will be created rather the method will be called just like normal method by main thread.

if we replace t.start() with t.run() then the output of the program would be

```
child thread
child thread
child thread
child thread
child thread
```

```
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

case3:: Importance of Thread class start() method
For every thread, required mandatory activities like registering the thread with Threadscheduler will be taken care by Thread class
start() method and programmer is responsible of just doing the job of the Thread inside run() method.
start() acts like an assistance to programmer.

```
   start()
   {
      register thread with ThreadScheduler
        All other mandatory low level activities
      invoke or calling run() method.
   }
```

We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java.
Due to this start() is considered as heart of MultiThreading.

case4:: If we are not overriding run() method
If we are not Overriding run() method then Thread class run() method will be executed which has empty implementation and
hence we wont get any output.

eg::

```
class MyThread extends Thread{}
class ThreadDemo{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
      }
}
```

It is highly recomended to override run() method, otherwise don't go for MultiThreading concept.

case5:Overloading of run() method
we can overload run() method but Thread class start() will always call run() with zero argument.
if we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed
just like normal method.
eg::

```
class MyThread extends Thread{
      public void run(){
```

```
            System.out.println("no arg method");
      }
      public void run(int i){
            System.out.println("zero arg method");
      }
}
class ThreadDemo{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
      }
}
```
Output:: NO arg method.

Case6::Overriding of start() method
If we override start() then our start() method will be executed just like normal method,but no
new Thread will be created and no new Thread will be started.

eg#1.
```
class MyThread extends Thread{
      public void run(){
            System.out.println("no arg method");
      }
      public void start(){
            System.out.println("start arg method");
      }
}
class ThreadDemo{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
      }
}
```
Output:: start arg method
It is never recomended to override start() method.

case7::
```
class MyThread extends Thread{
      public void run(){
            System.out.println("run method");
      }
      public void start(){
            System.out.println("start  method");
      }
}
class ThreadDemo{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
            System.out.println("Main method");
      }
}
```
Output::
MainThread
  a. Main method
  b. start method.

eg#2.

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

}
class ThreadDemo{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
            System.out.println("Main method");
      }
}
Output::
MainThread
  a. Main method
  b. start method
UserDefinedThread
  a. run method

case8:: Life cycle of a Thread
MyThread t=new MyThread(); // Thread is in born state
      t.start();  //Thread is in ready/runnable state
if Thread scheduler allocates CPU time then we say thread entered into Running
state.
if run() is completed by thread then we say thread entered into dead state.
=> Once we created a Thread object then the Thread is said to be in new state or
born state.
=> Once we call start() method then the Thread will be entered into Ready or
Runnable state.
=> If Thread Scheduler allocates CPU then the Thread will be entered into running
state.
=> Once run() method completes then the Thread will entered into dead state.


case9::
After starting the Thread, we are not supposed to start the same Thread again,then
we say Thread is in "IllegalThreadStateException".
 MyThread t=new MyThread(); // Thread is in born state
      t.start();  //Thread is in ready state
       ....
       ....
      t.start(); //IllegalThreadStateException


Defining a Thread by implementing Runnable Interface
====================================================
public interface Runnable{
    public abstract void run();
}
public class Thread implements Runnable{
      public void start(){
            1. register Thread with ThreadScheduler
            2. All other mandatory low level activites
            3. invoke run()
```

```
	}
	public void run(){
		//empty implementation
	}
}

eg::1
 class MyRunnable implements Runnable{
	@Override
	public void run(){
		for(int i=1;i<=10;i++)
			System.out.println("child thread");
	}
 }
public class ThreadDemo{
	public static void main(String... args){
		MyRunnable r=new MyRunnable();
		Thread t=new Thread(r);//call MyRunnable run()
		t.start();
		for(int i=1;i<=10;i++)
			System.out.println("main thread");
	}
}
Output::
 MainThread
   a. main thread
       ....
       ....
 ChildThread
   a. child thread
      ...
      ...
        ...

Case study
=========
 MyRunnable r=new MyRunnable();
 Thread t1=new Thread();
 Thread t2=new Thread(r);



case1:: t1.start();
 A new thread will be created and it will execute Thread class run() which has
empty implementation.

Output::
  MainThread
      a. main thread
       ....
       ....
  ChildThread
      a. no output

case2:: t1.run()
 No new Thread will be created, rather run() of Thread class will be executed just
like normal method
Output::
  MainThread
```

```
      a. main thread
          ...
          ...


case3:: t2.start()
A new thread will be created and it will execute MyRunnable class run() which has
specific job
Output::
  MainThread
      a. main thread
       ....
       ....
  ChildThread
      a. child thread
       ...
       ...

case4:: t2.run()
No new Thread will be created, rather run() of MyRunnable class will be executed
just like normal
 method
Output::
  MainThread
      a. main thread
          ...
          ...
      b. child thread
          ...
          ...

case5:: r.start()
      It results in compile time error at MyRunnable class
      symbol:method start()
        location:MyRunnable

case6:: r.run()
No new Thread will be created, rather run() of MyRunnable class will be executed
just like normal method
Output::
  MainThread
      a. main thread
          ...
          ...
      b. child thread
          ...
          ...

In which of the above cases a new Thread will be created which is responsible for
the execution of MyRunnable run() method ?
      Ans.t2.start()

In which of the above cases a new Thread will be created ?
      Ans.t1.start();
          t2.start();


In which of the above cases MyRunnable class run() will be executed ?
      Ans.t2.start();//In multithreading environment
```

```
          t2.run();
          r.run();
```

Different approach for creating a Thread?
 A. extending Thread class
 B. implementing Runnable interface

Which approach is the best approach?
 a. implements Runnable interface is recomended becoz our class can extend other class through
     which inheritance benift can brought in to our class.
     Internally performance and memory level is also good when we work with interface.

 b. if we work with extends feature then we will miss out inheritance benefit becoz already our
     class has inherited the feature from "Thread class", so we normally we don't prefere
     extends approach rather implements approach is used in real time for working with "MultiThreading".

Various Constructors available in Thread class
==========
a. Thread t=new Thread()
b. Thread t=new Thread(Runnable r)
c. Thread t=new Thread(String name)
d. Thread t=new Thread(Runnable r,String name)
e. Thread t=new Thread(ThreadGroup g, String name);
f. Thread t=new Thread(ThreadGroup g, Runnable r);
g. Thread t=new Thread(ThreadGroup g, Runnable r,String name);
h. Thread t=new Thread(ThreadGroup g, Runnable r,String name,long stackSize);

Alternate approach to define a Thread(not recomended)
==========
```
class MyThread extends Thread{
      public void run(){
            System.out.println("child thread");
      }
}
class ThreadDemo {
      public static void main(String... args){
            MyThread t=new MyThread();
            Thread t1=new Thread(t);
            t1.start();
            System.out.println("main thread");
      }
}
```

Output::2 threads are created
 MainThread
    main thread
 ChildThread
    child thread

Output::
 ChildThread
    child thread
 MainThread
    main thread

```
internally related
==================
Runnable
  ^
  |
Thread
  ^
  |
MyThread


Names of the Thread
==================
 Internally for every thread,there would be a name for the thread.
   a. name given by jvm
   b. name given by the user.

eg::
class MyThread extends Thread{

}
public class TestApp{
      public static void main(String... args){
            System.out.println(Thread.currentThread().getName());//main

            MyThread t=new MyThread();
            t.start();
            System.out.println(t.getName());//Thread-0

            Thread.currentThread().setName("Yash");//Yash
            System.out.println(Thread.currentThread().getName());//Yash

            System.out.println(10/0);
                //Exception in thread "yash" java.lang.ArithmeticException:/by zero
                                TestApp.main()

      }
}
It is also possible to change the name of the Thread using setName().
It is possible to get the name of the Thread using getName().
methods
  public final String getName();
  public final void setName(String name);

eg#2.
class MyThread extends Thread{
      @Override
      public void run(){
           System.out.println("run() executed by Thread  ::
"+Thread.currentThread().getName());
      }
}
public class TestApp{
      public static void main(String... args){
            MyThread t=new MyThread();
            t.start();
          System.out.println("main() executed by Thread ::
"+Thread.currentThread().getName());
```

```
      }
}
Output:: run()  executed  by  Thread:: Thread-0
                main() executed  by  Thread:: main


ThreadPriorities
================
 For every Thread in java has some priority.
 valid range of priority is 1 to 10,it is not 0 to 10.
 if we try to give a differnt value the it would result in
"IllegalArgumentException".
 Thread.MIN_PRIORITY = 1
 Thread.MAX_PRIORITY = 10
 Thread.NORM_PRIORITY = 5
 Thread class does not have priorities is Thread.LOW_PRIORITY,Thread.HIGH_PRIORITY.
 Thread scheduler allocates cpu time based on "Priority".
 If both the threads have the same priority then which thread will get a chance as
a pgm we can't
 predict becoz it is vendor dependent.
 We can set and get priority values of the thread using the following methods
   a. public final void setPriority(int priorityNumber)
   b. public final int getPriority()
 The allowed priorityNumber is from 1 to 10,if we try to give other values it woudl
result in
 "IllegalArgumentException".

System.out.println(Thread.currentThread().setPriority(100);//IllegalArgumentExcepti
on.

DefaultPriority
==============
The default priority for only main thread is "5",where as for other threads
priority will be
inherited from parent to child.
Parent Thread priority will be given as Child Thread Priority.

eg#1.
class MyThread extends Thread{}
public class TestApp{
      public static void main(String... args){
            System.out.println(Thread.currentThread().getPriority());//5
            Thread.currentThread().setPriority(7);
            MyThread t= new MyThread();
            System.out.println(Thread.currentThread().getPriority());//7
      }
}

reference
=========
Thread
  ^
  |extends
  |
MyThread

MyThread is creating by "mainThread", so priority of "mainThread"  will be shared
as a priority
for "MyThread".
```

eg#2.

```
class MyThread extends Thread{
	@Override
	public void run(){
		for (int i=1;i<=5 ;i++ ){
			System.out.println("child thread");
		}
	}
}
public class TestApp{
	public static void main(String... args){

		MyThread t= new MyThread();
		t.setPriority(7);//line -1
		t.start();
		for (int i=1; i<=5; i++){
			System.out.println("main thread");
		}

	}
}
```

Since priority of child thread is more than main thread, jvm will execute child thread first
whereas for the parent thread priority is 5 so it will get last chance.
if we comment line-1, then we can't predict the order of execution becoz both the threads have
same priority.

Some platform won't provide proper support for Thread priorities.

eg:: windows7,windows10,...

Next session is on Friday :: 7.30PM IST