Thread in Java can be created in two ways

- **Create Thread by Implementing Runnable Interface (java.lang.Runnable) :**

You will need to follow three basic steps:

1. 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( )

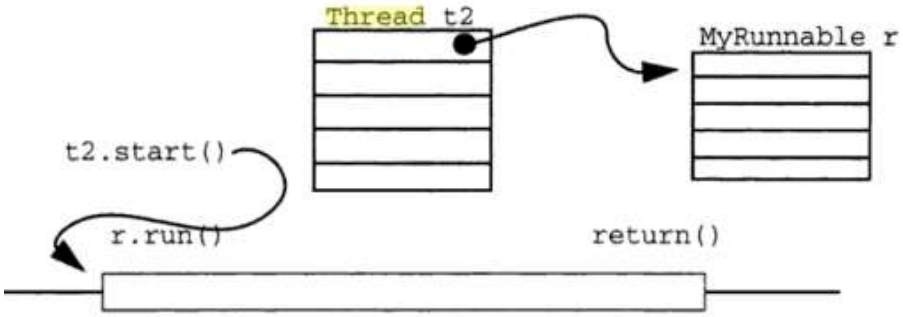
2. Instantiate a **Thread** object : Thread(Runnable threadObj, String threadName);

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

3. Once Thread object is created, you can start it by calling **start( )** method of thread, which executes a call to run( ) method. Simple syntax of start() method: void start( );

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



Creating a Thread via a Runnable

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 );   }
```

The **java.lang.Thread.sleep (long millis)** method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

```
  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.");
  }
//   This is start() method in the class to create and start the thread
  public void start ()
  {
    System.out.println("Starting " +  threadName );
    if (t == null)
    {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}
```

Every Thread has its own stack so local variable i is different for different threads.

It is executed in the main thread.

```
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();
  }
}
```

The start() method of thread causes this thread to begin execution; the JVM calls the run method of this thread.

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.
```

Output from Constructor of RunnableDemo object R1

Start method of R1

Output from Constructor of RunnableDemo object R2

Start method of R2

R1 start method subsequently call the run method

R2 start method subsequently call the run method

- **Create Thread by Extending Thread Class (java.lang.Thread) :**

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.

1. 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.
2. Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method.

Following is the declaration for java.lang.Thread class –
public class **Thread** extends Object implements Runnable

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 ();
    }
```

Output :

```
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.
```

InterruptedException is thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

start() method of Thread which will automatically call the run() method

```
  }
}
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();
  }
}
```

Common Pitfall: Calling run() instead of start()

When creating and starting a thread a common mistake is to call the run() method of the Thread instead of start(), like this:

```
Thread newThread = new Thread(MyRunnable());
thread.run();  //should be start();
```

At first you may not notice anything because the Runnable's run() method is executed like you expected. However, it is NOT executed by the new thread you just created. Instead the run() method is executed by the **thread that created the thread**. In other words, the thread that executed the above two lines of code. To have the run() method of the MyRunnable instance called by the new created thread, newThread, you MUST call the newThread.start() method.

Tip: Unlike class and instance field variables, threads cannot share local variables and parameters. The reason: Local variables and parameters allocate on a thread's method-call stack. As a result, each thread receives its own copy of those variables. In contrast, threads can share class fields and instance fields because those variables do not allocate on a thread's method-call stack. Instead, they allocate in shared heap memory—as part of classes (class fields) or objects (instance fields).

Thread class 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
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(String name)** Changes the name of the Thread object. getName() method for retrieving the name.

4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** 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.
9	**Thread.State getState()** This method returns the state of this thread. The java.lang.Thread.getState() method returns the state of this thread. **Return Value** : This method returns this thread's state.

```
import java.lang.*;

public class ThreadDemo implements Runnable {

  public void run() {
    // returns the state of this thread
    Thread.State state = Thread.currentThread().getState();
    System.out.println(Thread.currentThread().getName());
    System.out.println("state = " + state);
  }
  public static void main(String args[]) {
    Thread t = new Thread(new ThreadDemo());
    // this will call run() function
    t.start();
  }
}
```

currentThread() :
Returns a reference to the currently executing thread object.
public static Thread currentThread()

Default Thread name
when no name is set.
(Thraed-0, Thread-1 …)

Output :
Thread-0
state = RUNNABLE

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(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.
3	**public static boolean holdsLock(Object x)** 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.

Comparison between implementing Runnable and extending Thread

BASIS FOR COMPARISON	THREAD	RUNNABLE
Basic	Each thread creates a unique object and gets associated with it.	Multiple threads share the same object (share the same runnable instance).
Memory	As each thread is associated with a unique object when created by extending Thread class, more memory is required.	Each thread created by implementing Runnable interface shares same object space hence, it requires less memory.
Extending	In Java, multiple-inheritance is not allowed hence, after a class extends Thread class, it cannot extend any other class.	If a class defines thread implementing the Runnable interface it has a chance of extending one class.
Use	A user must extend thread class only if it wants to override the other methods in Thread class.	If you only want specialize run method then implementing Runnable is a better option.
Coupling	Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread	Implementing Runnable interface introduces loose coupling as the code of Thread is separate from the job of Threads.

Example of demonstrating some of the methods of the Thread class

Develop two classes as shown below one by implementing **Runnable** and another by extending Thread.

```
public class DisplayMessage implements Runnable
{
  private String message;
  public DisplayMessage(String message)
  {    this.message = message;  }

  public void run()
  {
    while(true) {
       System.out.println(message);
    }
  }
}
```

As infinite loop this **should run as daemon thread** so that JVM does not wait for this thread to finish its execution.

```
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);
```

Daemon thread in Java are those thread which runs in background and mostly created by JVM for performing background task like Garbage collection and other house- keeping tasks.

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

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

Runnable type is used as the class implements this interface

```
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...");
  }
}
```

Difference between Daemon and Non Daemon(User Threads) :

- as soon as all user thread finish execution java program or JVM terminates itself, JVM doesn't wait for daemon thread to finish its execution.

When the JVM halts any remaining daemon threads are abandoned:

- finally blocks are not executed,
- stacks are not unwound - the JVM just exits.

When you use join() , it makes sure that as soon as a thread calls join, the current thread (currently running thread, in this case main thread running the main method) will not execute unless the thread you have called join is finished. For example, thread3.join(); - it will make sure that the current running thread will run only after thread3 has finished running.

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

```
Starting hello thread...
Starting goodbye thread...
```

```
Hello
Hello
Hello
Hello
Hello
Hello
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
```

Use of yield() stop() and sleep() methods

```
Class ThreadA   extends Thread
{
    public void run() {
       for ( int i = 1; i<=5; i++ )
       {
          if ( i==1)   yield() ;
          System.out.println( "\tFrom ThreadA :
                                  i = " + i);
        }
        System.out.println( "Exit from ThreadA " );
    }
}
```

```
Class ThreadB   extends Thread
{
    public void run() {
       for ( int j = 1; j<=5; j++ )
       {
          System.out.println( "\tFrom ThreadB :
                                  j = " +  j);
          if ( j==3)   stop() ;
        }
        System.out.println( "Exit from ThreadB " );
    }
}
```

```
Class ThreadC   extends Thread
{
    public void run() {
       for ( int k = 1; j<=5; k++ )
       {
          System.out.println( "\tFrom ThreadC :
                                  k= " +  kj);
          if ( k==1)
             try  { sleep(1000) ;  }
             catch ( Exception e )   {   }
        }
        System.out.println( "Exit from ThreadC " );
    }
}
```

```
Class ThreadMethods   {
   public static  vaid  main ( String args [] )
   {
        ThreadA  thA = new ThreadA();
        ThreadB  thB = new ThreadB();
        ThreadC  thC = new ThreadC();
        System.out.println( "Start ThreadA ");
        thA.start();
        System.out.println( "Start ThreadB ");
        thB.start();
        System.out.println( "Start ThreadC ");
        thC.start();
        System.out.println( "End of main thread");
   }
}
```

```
Output :
Start ThreadA
Start ThreadB
Start ThreadC
```

- yield() method in ThreadA at iteration i=1 has relinquished its control although it start first. So thread scheduler schedule ThreadB and we

```
From ThreadB :  j =  1
From ThreadB :  j =  2
From ThreadA :  i =  1
From ThreadA :  i =  2
End of main thread
From ThreadC :  k =  1
From ThreadB :  j =  3
From ThreadA :  i =  3
From ThreadA :  i =  4
From ThreadA :  i =  5
Exit from ThreadA
From ThreadC :  k =  2
From ThreadC :  k =  3
From ThreadC :  k =  4
From ThreadC :  k =  5
Exit from ThreadC
```

are getting output from run method of this ThreadB.

- The stop() method in ThreadB has killed it after implementing the for loop only three times. So we are getting output up to j=3 and it has not reached end of run() method.
- ThreadC started sleeping after executing the for loop only once. When it woke up, the other two threads have already completed their runs and therefore was running alone.
- The main thread died much earlier than the other three threads. So other threads became **orphan**. If we use join() method appropriately it may not happen.

It is not a good practice to **create thread in constructor.** We are actually giving a reference (this) to our object before it is fully constructed. The thread will start before your constructor finishes. This can result in all kinds of unusual behaviors.

Use of join() method

```
// Using join() to wait for threads to finish.
public class ThreadEx3 implements Runnable {
   String name;    // name of thread
   Thread t;
   //  Constructor
   ThreadEx3 (String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start();       // Start the thread
   }

   // This is the entry point for thread.
   public void run() {
      try {
         for (int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
         System.out.println(name + " interrupted.");
      }
      System.out.println(name + " exiting.");
   }
}
```

```
public class ThreadDemoJoin {
   public static void main(String args[]) {
      //  Constructor of this object will create and
          start the thread
      ThreadEx3 ob1 = new ThreadEx3("One");
      ThreadEx3 ob2 = new ThreadEx3("Two");
      ThreadEx3 ob3 = new ThreadEx3("Three");
      System.out.println("Thread One is alive: " +  ob1.t.isAlive());
      System.out.println("Thread Two is alive: " +  ob2.t.isAlive());
      System.out.println("Thread Three is alive: " + ob3.t.isAlive());
```

When ob1 is created then the constructor is called where t.start() is written but still run() method is not executed rather main() method (i.e. main thread) continues execution. Calling start() doesn't mean run() will be called immediately, it depends on thread scheduler when it chooses to run your thread.

```
    // Wait for threads to finish
    try {
      System.out.println("Waiting for threads to
                                finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread
                Interrupted");
    }
    System.out.println("Thread One is alive: " +  ob1.t.isAlive());
    System.out.println("Thread Two is alive: " +  ob2.t.isAlive());
    System.out.println("Thread Three is alive: " +  ob3.t.isAlive());
    System.out.println("Main thread exiting.");
  }
}
```

So main thread waits for ob1.t, ob2.t, ob3.t threads to die(look into Thread#join doc). So all three threads execute successfully and main thread completes after that.

- When you use join() , it makes sure that as soon as a thread calls join, the current thread (currently running thread, in this case main thread running the main method) will not execute unless the thread you have called join is finished. For example, ob1.t.join(); - it will make sure that the current running thread will run only after t has finished running.

Suppose in a main method you have written following lines

```
t.start(); // Line 15
t.join();  // Line 16
t1.start();  // Line 17
```

When the main method (i.e. main thread) encounters line 15, thread t is available at thread scheduler. As soon as main thread comes to line 16, it will not execute unless thread t has finished(remember join() as the currently running thread will join to the end of the thread on which join is called). Hence main thread will come to line 17 only when thread t has finished. So it may appear that t.join will affect thread **t1**, but it is actually affecting main thread.

- In the above example , as for *join()*, it only guarantees that whatever is after it will only happen once the thread you are joining is completed execution. So when you have three *join()* calls in a row it doesn't mean the threads should end in a particular order. It simply means that you will wait for *ob1* first. Once *ob1* finishes, *ob2* and *ob3* may be still running or they may already be finished. If they are finished, your other *join()* calls will return immediately.

- join() method is used to wait until the thread on which it is called does not terminates, but here in output we see alternate outputs of the different thread why?

This is because of the Thread.sleep(1000) in your code. Remove that line and you will see ob1 finishes before ob2 which in turn finishes before ob3 (as expected with join()). Having said that it all depends on when ob1 ob2 and ob3 started. Calling sleep will pause thread execution for >= 1 second (in your code), giving scheduler a chance to call other threads waiting (same priority).

- And Join() statement is used **to prevent the child thread from becoming orphan.** Means if you did'nt call join() in your main class, then main thread will exit after its execution and child thread will be still there executing the statements. Join() will wait until all child thread complete its execution and then only main method will exit.

main thread group is created by JVM and main thread belongs to this group. All the thread created by main thread belong to this group.

Output of this program :

```
New thread: Thread[One,5,main]          // These 3 lines will be generated from the constructor
New thread: Thread[Two,5,main]          // It prints Thread Name, Priority and ThreadGroup
New thread: Thread[Three,5,main]
Thread One is alive: true               // Main thread generates this output
Thread Two is alive: true               // Main thread generates this output
Two: 5                                  // Thread Scheduler chooses thread Two to run, so its run method
                                           Generates this output and go for 1 sec sleep.
Thread Three is alive: true             // Main thread generates this output
Waiting for threads to finish.          // Main thread generates this output
One: 5
Three: 5
Three: 4
One: 4
Two: 4
Three: 3
One: 3
Two: 3
Three: 2
One: 2
Two: 2
Three: 1
Two: 1
One: 1
One exiting.
Two exiting.
Three exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

```
// If we remove the sleep statement it will generate following output
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
One: 4
One: 3
One: 2
Thread One is alive: true
One: 1
Thread Two is alive: true
One exiting.
Thread Three is alive: true
Three: 5
Two: 5
Two: 4
Two: 3
Three: 4
Waiting for threads to finish.
```

```
Three: 3
Two: 2
Two: 1
Two exiting.
Three: 2
Three: 1
Three exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```