**Multithreading**

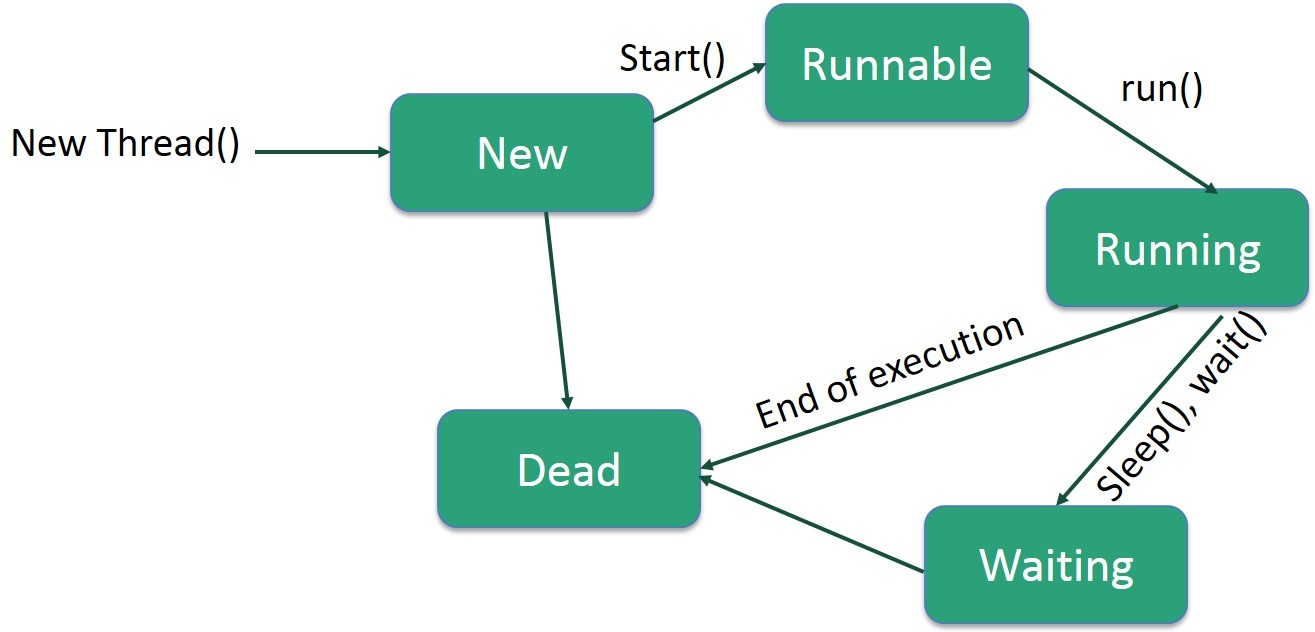
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources.

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

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

# **Life Cycle of a Thread:**

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



**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**.

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

**Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

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

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

# **Thread Priorities:**

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

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

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

**Creating a thread:**

There are two ways to create a thread.

1. By implementing a Runnable interface.
2. By extending thread class.

# **Create a Thread by Implementing a Runnable Interface:**

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

## **Step 1:**

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

**public void run( )**

## **Step 2**

As a second step, you will instantiate a **Thread** object using the following constructor −

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

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

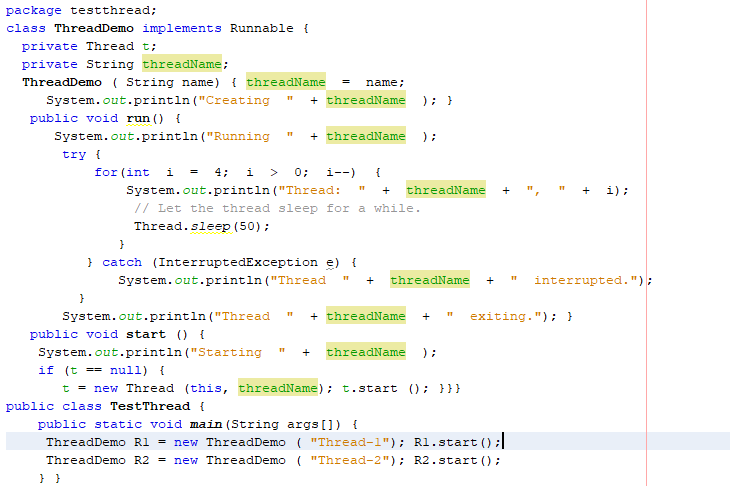
## **Step 3**

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

**void start ( );**

## **Example**

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



## **Output:**

## 

# **Create a Thread by Extending a Thread Class:**

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

## **Step 1**

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

**public void run( )**

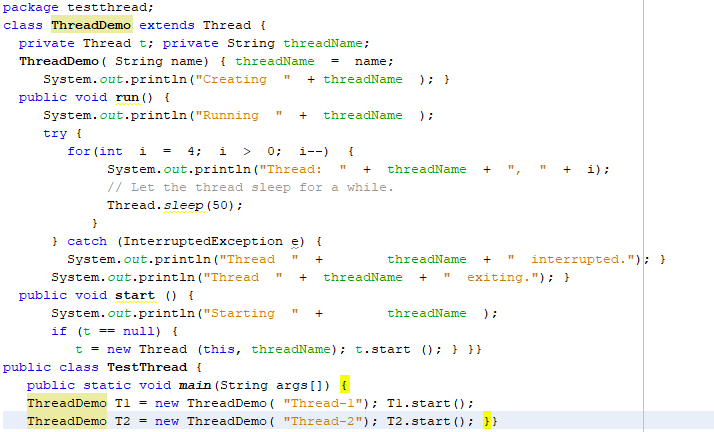
## **Step 2**

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

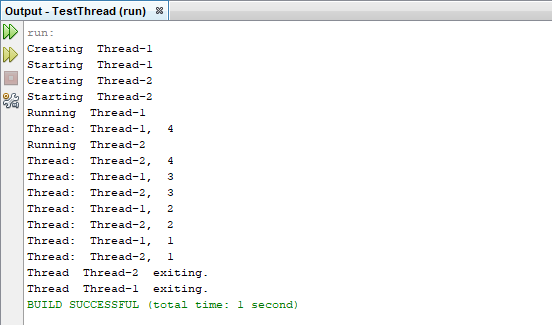
**void start( );**

## **Example:**

Here is the preceding program rewritten to extend the Thread.



## **Output:**



# **Thread Methods:**

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

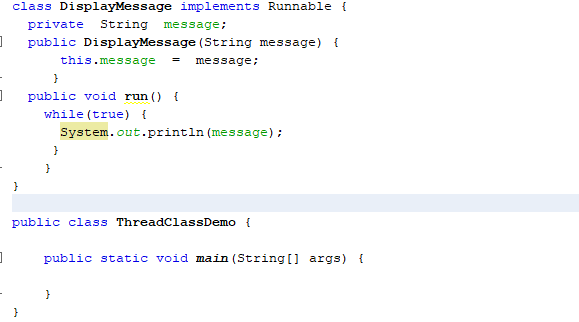
|  |  |
| --- | --- |
| **Sr.#** | **Method & Description** |
| **1** | **public void start()**   * Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| **2** | **public void run()**   * If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| **3** | **public final void setName(String name)**   * Changes the name of the Thread object. There is also a 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. * Daemon thread in java is low-pripority thread that runs in background to perform tasks such as garbage collection. Daemon thread in java is also a service provider thread that provides services to the user 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. |

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.

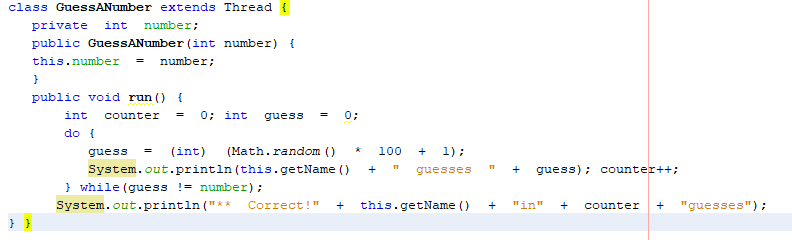
|  |  |
| --- | --- |
| **Sr. #** | **Method & Description** |
| 1 | **public static void yield()**   * Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. * It can stop the currently executing thread and will give a chance to other waiting threads of same priority. |
| 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. |

## **Example:**

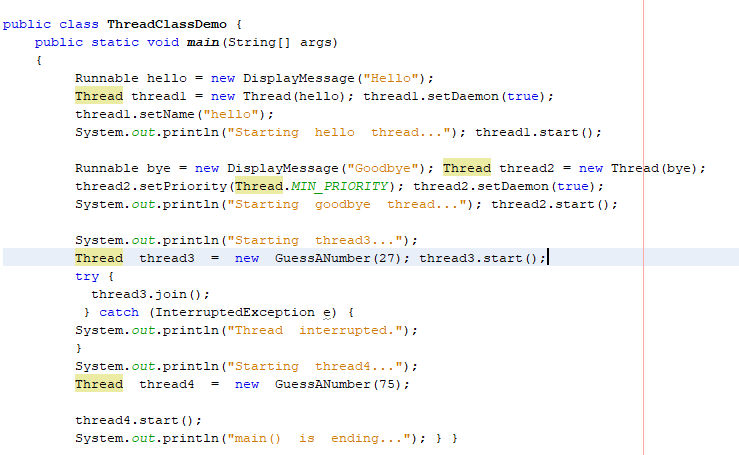
The following ThreadClassDemo program demonstrates some of these methods of the Thread class. Consider a class **DisplayMessage** which implements **Runnable.**



**Following is another class which extends the Thread class:**



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



**Thread Synchronization:**

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency 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.

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

### **Syntax**

**Synchronized (objectidentifier) {**

**// Access shared variables and other shared resources**

**}**

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

## **Multithreading Example without Synchronization:**

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

class PrintDemo {

public void printCount() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Counter --- " + i );

}

} catch (Exception e) {

System.out.println("Thread interrupted.");

}}}

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

PrintDemo PD;

ThreadDemo( String name, PrintDemo pd) {

threadName = name;

PD = pd;

}

public void run() {

PD.printCount();

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start (); }}}

public class TestThread {

public static void main(String args[]) {

PrintDemo PD = new PrintDemo();

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

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

T1.start();

T2.start();

// wait for threads to end

try {

T1.join();

T2.join();

} catch ( Exception e) {

System.out.println("Interrupted");

}

}

}

This produces a different result every time you run this program.

## **Multithreading Example with Synchronization**

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

### **Example**

class PrintDemo {

public void printCount() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Counter --- " + i );

}

} catch (Exception e) {

System.out.println("Thread interrupted.");

}

}

}

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

PrintDemo PD;

ThreadDemo( String name, PrintDemo pd) {

threadName = name;

PD = pd;

}

public void run() {

synchronized(PD) {

PD.printCount();

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

PrintDemo PD = new PrintDemo();

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

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

T1.start();

T2.start();

// wait for threads to end

try {

T1.join();

T2.join();

} catch ( Exception e) {

System.out.println("Interrupted");

}

}

}

**Lab Exercise:**

1. Write a program that creates a user interface to perform integer division. The user enters two numbers in the text fields, Num1 and Num2. The division of Num1 and Num2 is displayed in the Result field when the Divide button is clicked. If Num1 and Num2 were not an integer, the program would throw a Number Format Exception. If Num2 were Zero, the program would throw an Arithmetic Exception Display the exception in a message dialog box.
2. Write Java Program that implements a multithread application that has three threads. First thread generates random integer for every second and if the value is even, second thread computes the cube of number and prints. If the value is odd, the third thread will print the value of square of number.
3. Write a java program in which thread sleep for 3 seconds and change the name of thread.
4. Write a java program for multithread in which user thread and thread started from main method invoked at a time each thread sleep for 5 seconds
5. Give your own code example for thread Synchronization.