**A Simple Thread Example**

The simple example shown in full on the first page of this lesson defines two classes: SimpleThread and TwoThreadsTest. Let's begin our exploration of the application with the SimpleThread class--a subclass of the [Thread](http://java.sun.com/products/jdk/1.1/api/java.lang.Thread.html) class, which is provided by the java.lang package:

class SimpleThread extends Thread {

public SimpleThread(String str) {

super(str);

}

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println(i + " " + getName());

try {

sleep((int)(Math.random() \* 1000));

} catch (InterruptedException e) {}

}

System.out.println("DONE! " + getName());

}

}

The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the SimpleThread class is the run method. The run method is the heart of any Thread and where the action of the Thread takes place. The run method of theSimpleThread class contains a for loop that iterates ten times. In each iteration the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second. After the loop has finished, the run method prints DONE! along with the name of the thread. That's it for the SimpleThread class.

The TwoThreadsTest class provides a main method that creates two SimpleThread threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide--go to the island whose thread prints "DONE!" first.)

class TwoThreadsTest {

public static void main (String[] args) {

new SimpleThread("Jamaica").start();

new SimpleThread("Fiji").start();

}

}

The main method also starts each thread immediately following its construction by calling the start method. To save you from typing in this program, click here for the source code to the [SimpleThread](http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/example/SimpleThread.java) class and here for the source code to the [TwoThreadsTest](http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/example/TwoThreadsTest.java) program. Compile and run the program and watch your vacation fate unfold. You should see output similar to the following:

0 Jamaica

0 Fiji

1 Fiji

1 Jamaica

2 Jamaica

2 Fiji

3 Fiji

3 Jamaica

4 Jamaica

4 Fiji

5 Jamaica

5 Fiji

6 Fiji

6 Jamaica

7 Jamaica

7 Fiji

8 Fiji

9 Fiji

8 Jamaica

DONE! Fiji

9 Jamaica

DONE! Jamaica

(Looks like I'm going to Fiji!!) Notice how the output from each thread is intermingled with the output from the other. This is because both SimpleThread threads are running concurrently. Thus, both run methods are running at the same time and each thread is displaying its output at the same time as the other.

**Try This:** Change the main program so that it creates a third thread with the name "Bora Bora". Compile and run the program again. Does this change the island of choice for your vacation? Here's the code for the new main program, which is now named [ThreeThreadsTest](http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/example/ThreeThreadsTest.java).

**Keep Going**

This page glosses over many of the details of threads such as the start and sleep methods. Don't worry, the next several pages of this lesson explain these concepts and others in detail. The important thing to understand from this page is that a Java program can have many threads, and that those threads can run *concurrently*.

# Java Thread Example

APRIL 2, 2018 BY [PANKAJ](https://www.journaldev.com/author/pankaj) [15 COMMENTS](https://www.journaldev.com/1016/java-thread-example#comments)

Welcome to the Java Thread Example. **Process** and **Thread** are two basic units of execution. [Concurrency](https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers)programming is more concerned with java threads.

**Table of Contents**[[hide](https://www.journaldev.com/1016/java-thread-example)]

* + [0.1 Process](https://www.journaldev.com/1016/java-thread-example#process)
  + [0.2 Thread](https://www.journaldev.com/1016/java-thread-example#thread)
* [1 Java Thread Example](https://www.journaldev.com/1016/java-thread-example#java-thread-example)
  + [1.1 Java Thread Benefits](https://www.journaldev.com/1016/java-thread-example#java-thread-benefits)
  + [1.2 Java Thread Example – implementing Runnable interface](https://www.journaldev.com/1016/java-thread-example#java-thread-example-8211-implementing-runnable-interface)
  + [1.3 Java Thread Example – extending Thread class](https://www.journaldev.com/1016/java-thread-example#java-thread-example-8211-extending-thread-class)
  + [1.4 Runnable vs Thread](https://www.journaldev.com/1016/java-thread-example#runnable-vs-thread)

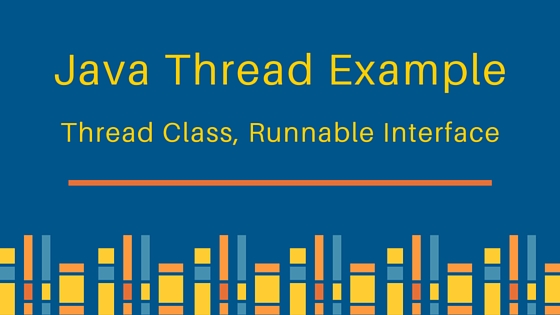
### Process

A process is a self contained execution environment and it can be seen as a program or application. However a program itself contains multiple processes inside it. Java runtime environment runs as a single process which contains different classes and programs as processes.

### Thread

Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

## Java Thread Example

[](https://cdn.journaldev.com/wp-content/uploads/2012/12/java-thread-example.jpg)  
Every java application has at least one thread – [main thread](https://www.journaldev.com/611/exception-in-thread-main-java). Although there are so many other java threads running in background like memory management, system management, signal processing etc. But from application point of view – main is the first java thread and we can create multiple threads from it.

[Multithreading](https://www.journaldev.com/1079/multithreading-in-java) refers to two or more threads executing concurrently in a single program. A computer single core processor can execute only one thread at a time and **time slicing** is the OS feature to share processor time between different processes and threads.

### Java Thread Benefits

1. Java Threads are lightweight compared to processes, it takes less time and resource to create a thread.
2. Threads share their parent process data and code
3. Context switching between threads is usually less expensive than between processes.
4. Thread intercommunication is relatively easy than process communication.

Java provides two ways to create a thread programmatically.

1. Implementing the **java.lang.Runnable** interface.
2. Extending the **java.lang.Thread** class.

### Java Thread Example – implementing Runnable interface

To make a class runnable, we can implement java.lang.Runnable interface and provide implementation in public void run() method. To use this class as Thread, we need to create a Thread object by passing object of this runnable class and then call start() method to execute the run() method in a separate thread.

Here is a java thread example by implementing Runnable interface.

package com.journaldev.threads;

public class HeavyWorkRunnable implements Runnable {

@Override

public void run() {

System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

### Java Thread Example – extending Thread class

We can extend **java.lang.Thread** class to create our own java thread class and override run() method. Then we can create it’s object and call start() method to execute our custom java thread class run method.

Here is a simple java thread example showing how to extend Thread class.

package com.journaldev.threads;

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

System.out.println("MyThread - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("MyThread - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

Here is a test program showing how to create a java thread and execute it.

package com.journaldev.threads;

public class ThreadRunExample {

public static void main(String[] args){

Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");

Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");

System.out.println("Starting Runnable threads");

t1.start();

t2.start();

System.out.println("Runnable Threads has been started");

Thread t3 = new MyThread("t3");

Thread t4 = new MyThread("t4");

System.out.println("Starting MyThreads");

t3.start();

t4.start();

System.out.println("MyThreads has been started");

}

}

Output of the above java thread example program is:

Starting Runnable threads

Runnable Threads has been started

Doing heavy processing - START t1

Doing heavy processing - START t2

Starting MyThreads

MyThread - START Thread-0

MyThreads has been started

MyThread - START Thread-1

Doing heavy processing - END t2

MyThread - END Thread-1

MyThread - END Thread-0

Doing heavy processing - END t1

Once we start any thread, it’s execution depends on the OS implementation of time slicing and we can’t control their execution. However we can set threads priority but even then it doesn’t guarantee that higher priority thread will be executed first.

Run the above program multiple times and you will see that there is no pattern of threads start and end.

### Runnable vs Thread

If your class provides more functionality rather than just running as Thread, you should implement Runnable interface to provide a way to run it as Thread. If your class only goal is to run as Thread, you can extend Thread class.

Implementing Runnable is preferred because java supports implementing multiple interfaces. If you extend Thread class, you can’t extend any other classes.

**Tip**: As you have noticed that thread doesn’t return any value but what if we want our thread to do some processing and then return the result to our client program, check our [**Java Callable Future**](https://www.journaldev.com/1090/java-callable-future-example).

**Update**: From [Java 8](https://www.journaldev.com/2389/java-8-features-with-examples) onwards, Runnable is a functional interface and we can use lambda expressions to provide it’s implementation rather than using anonymous class. For more details, check out [Java 8 Functional Interfaces](https://www.journaldev.com/2763/java-8-functional-interfaces).

# **Java - Multithreading**

Advertisements

[Previous Page](https://www.tutorialspoint.com/java/java_sending_email.htm)

[Next Page](https://www.tutorialspoint.com/java/java_applet_basics.htm)

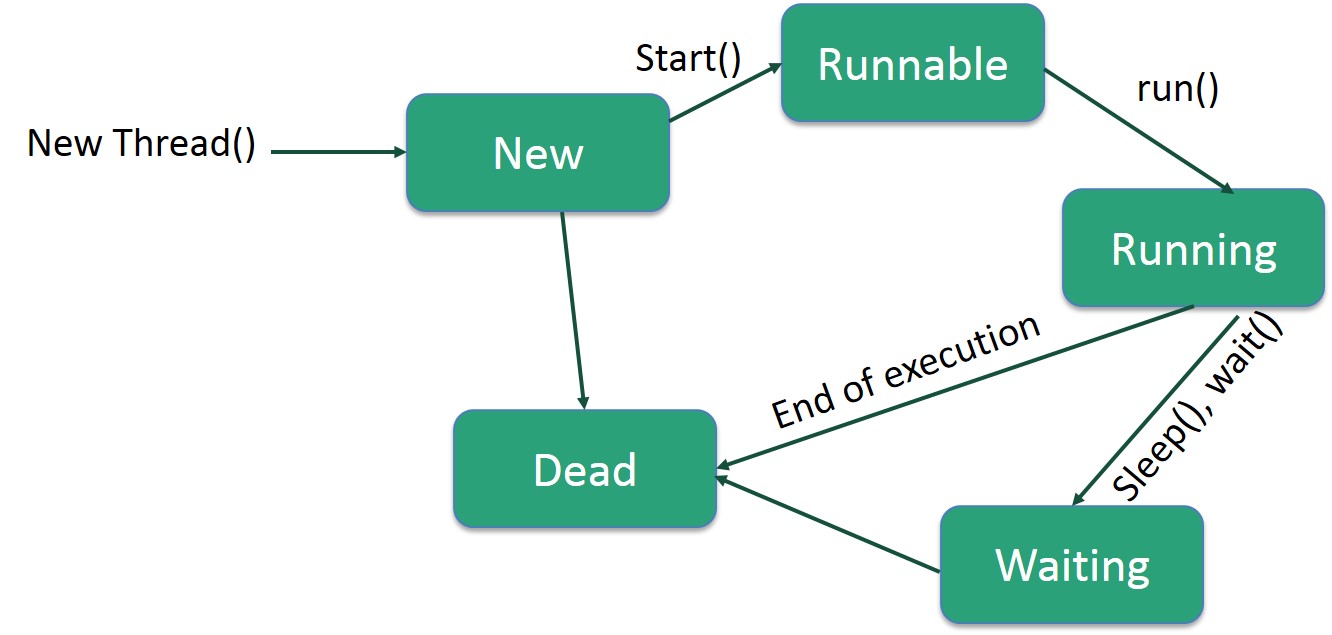
Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially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

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.

## 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 −

[Live Demo](http://tpcg.io/6iqMVy)

class RunnableDemo implements Runnable {

private Thread t;

private String threadName;

RunnableDemo( 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 ();

}

}

}

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();

}

}

This will produce the following result −

### **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.

## 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 −

[Live Demo](http://tpcg.io/rVxO3k)

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 ();

}

}

}

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();

}

}

This will produce the following result −

### **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.

## Thread Methods

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

|  |  |
| --- | --- |
| **Sr.No.** | **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. |
| 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.No.** | **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. |
| 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** −

// File Name : DisplayMessage.java

// Create a thread to implement Runnable

public class DisplayMessage implements Runnable {

private String message;

public DisplayMessage(String message) {

this.message = message;

}

public void run() {

while(true) {

System.out.println(message);

}

}

}

Following is another class which extends the Thread class −

// File Name : GuessANumber.java

// Create a thread to extentd Thread

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);

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

}

}

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

// File Name : ThreadClassDemo.java

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...");

}

}

This will produce the following result. You can try this example again and again and you will get a different result every time.

### **Output**

Starting hello thread...

Starting goodbye thread...

Hello

Hello

Hello

Hello

Hello

Hello

Goodbye

Goodbye

Goodbye

Goodbye

Goodbye

.......

## Major Java Multithreading Concepts

While doing Multithreading programming in Java, you would need to have the following concepts very handy −

* [What is thread synchronization?](https://www.tutorialspoint.com/java/java_thread_synchronization.htm)
* [Handling interthread communication](https://www.tutorialspoint.com/java/java_thread_communication.htm)
* [Handling thread deadlock](https://www.tutorialspoint.com/java/java_thread_deadlock.htm)
* [Major thread operations](https://www.tutorialspoint.com/java/java_thread_control.htm)

# **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.  
  
Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface  
  
   
**Thread creation by extending the Thread class**  
  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

|  |
| --- |
| // Java code for thread creation by extending  // the Thread class  class MultithreadingDemo extends Thread  {      public void run()      {          try          {              // Displaying the thread that is running              System.out.println ("Thread " +                    Thread.currentThread().getId() +                    " is running");            }          catch (Exception e)          {              // Throwing an exception              System.out.println ("Exception is caught");          }      }  }    // Main Class  public class Multithread  {      public static void main(String[] args)      {          int n = 8; // Number of threads          for (int i=0; i<8; i++)          {              MultithreadingDemo object = new MultithreadingDemo();              object.start();          }      }  } |

Run on IDE

Output :

Thread 8 is running

Thread 9 is running

Thread 10 is running

Thread 11 is running

Thread 12 is running

Thread 13 is running

Thread 14 is running

Thread 15 is running

**Thread creation by implementing the Runnable Interface**  
  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

|  |
| --- |
| // Java code for thread creation by implementing  // the Runnable Interface  class MultithreadingDemo implements Runnable  {      public void run()      {          try          {              // Displaying the thread that is running              System.out.println ("Thread " +                                  Thread.currentThread().getId() +                                  " is running");            }          catch (Exception e)          {              // Throwing an exception              System.out.println ("Exception is caught");          }      }  }    // Main Class  class Multithread  {      public static void main(String[] args)      {          int n = 8; // Number of threads          for (int i=0; i<8; i++)          {              Thread object = new Thread(new MultithreadingDemo());              object.start();          }      }  } |

Run on IDE

Output :

Thread 8 is running

Thread 9 is running

Thread 10 is running

Thread 11 is running

Thread 12 is running

Thread 13 is running

Thread 14 is running

Thread 15 is running

**Thread Class vs Runnable Interface**  
  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.  
  
   
This article is contributed by Mehak Narang. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above