Multithreading in Java

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

AD

2) Thread-based Multitasking (Multithreading)

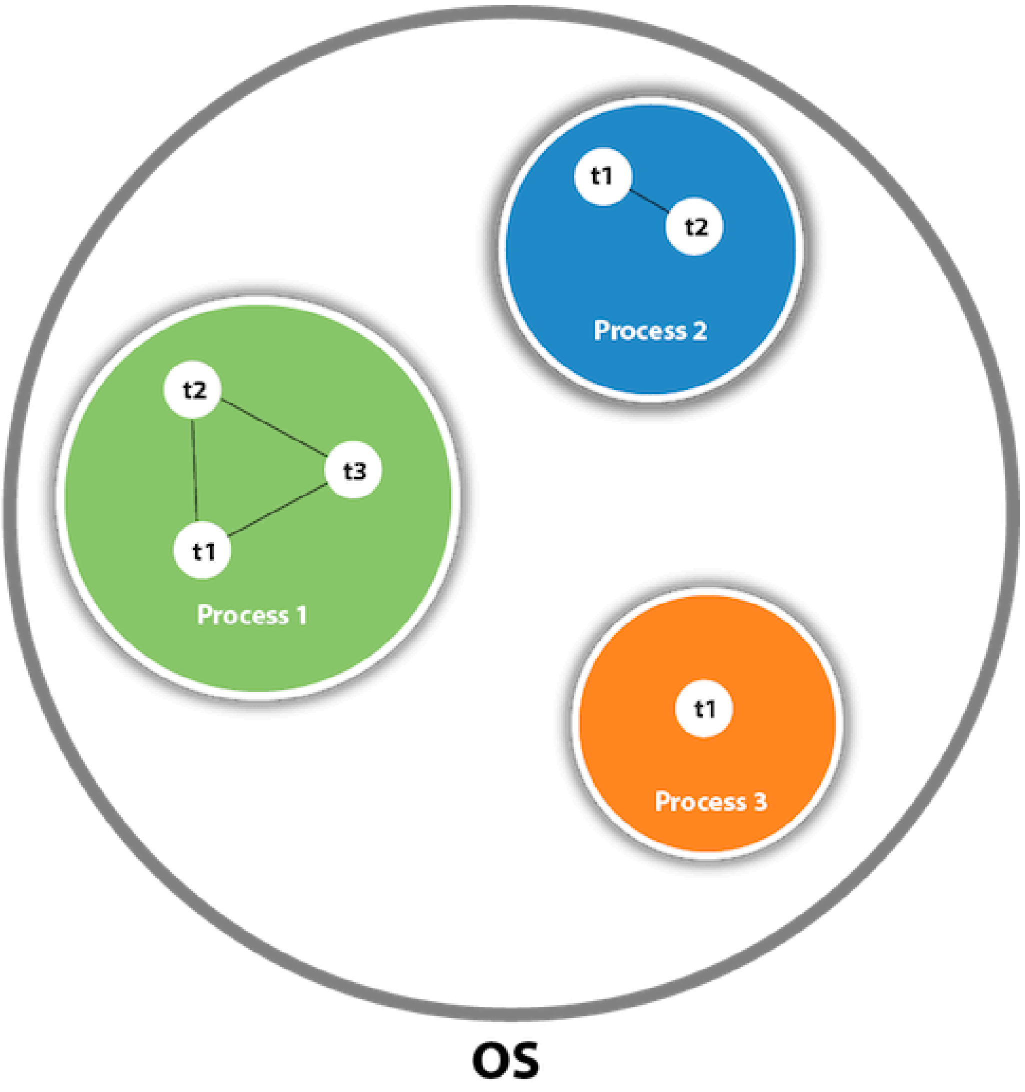
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

> Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

> Note: At a time one thread is executed only.

Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

AD

Java Thread Methods

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.

10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.

18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.

24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.

29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.

34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception.

Do You Know

- How to perform two tasks by two threads?
- How to perform multithreading by anonymous class?
- What is the Thread Scheduler and what is the difference between preemptive scheduling and time slicing?
- What happens if we start a thread twice?
- What happens if we call the run() method instead of start() method?
- What is the purpose of join method?
- Why JVM terminates the daemon thread if no user threads are remaining?
- What is the shutdown hook?
- What is garbage collection?

- What is the purpose of finalize() method?
- What does the gc() method?
- What is synchronization and why use synchronization?
- What is the difference between synchronized method and synchronized block?
- What are the two ways to perform static synchronization?
- What is deadlock and when it can occur?
- What is interthread-communication or cooperation?

What will we learn in Multithreading

- Multithreading
- Life Cycle of a Thread
- Two ways to create a Thread
- How to perform multiple tasks by multiple threads
- Thread Scheduler
- Sleeping a thread
- Can we start a thread twice?
- What happens if we call the run() method instead of start() method?
- Joining a thread
- Naming a thread
- Priority of a thread
- Daemon Thread
- ShutdownHook
- Garbage collection
- Synchronization with synchronized method
- Synchronized block
- Static synchronization
- Deadlock
- Inter-thread communication

← Prev Next →

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

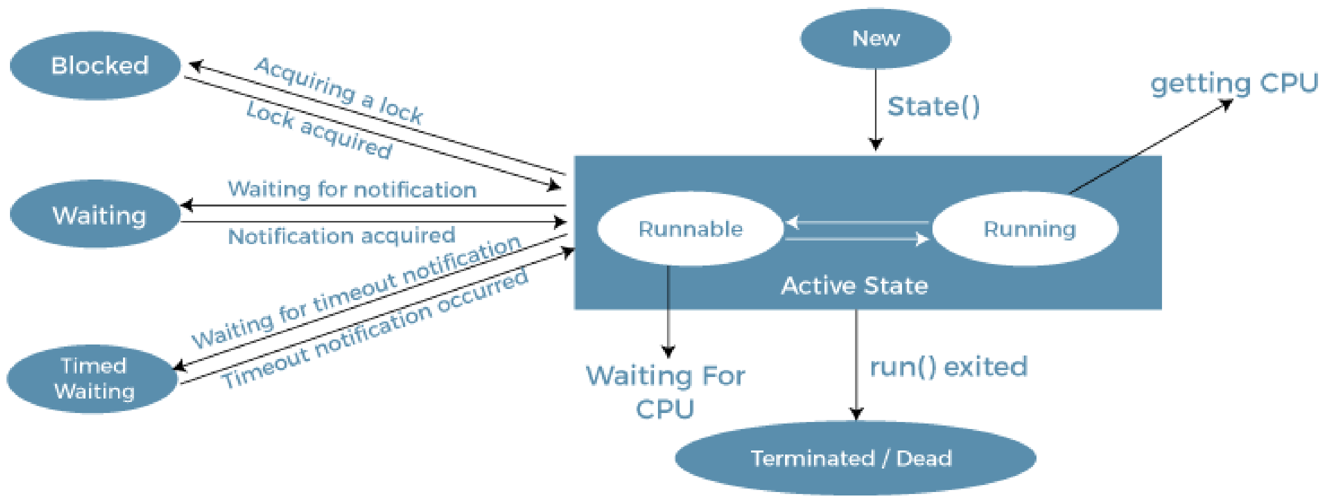
Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.



Life Cycle of a Thread

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

```
public static final Thread.State NEW
```

It represents the first state of a thread that is the NEW state.

```
public static final Thread.State RUNNABLE
```

It represents the runnable state.It means a thread is waiting in the queue to run.

```
public static final Thread.State BLOCKED
```

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

```
public static final Thread.State WAITING
```

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

```
public static final Thread.State TIMED_WAITING
```

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

- sleep
- join with timeout
- wait with timeout

- parkUntil
- parkNanos

```
public static final Thread.State TERMINATED
```

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

The following Java program shows some of the states of a thread defined above.

FileName: ThreadState.java

```
// ABC class implements the interface Runnable
class ABC implements Runnable
{
public void run()
{

// try-catch block
try
{
// moving thread t2 to the state timed waiting
Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}



System.out.println("The state of thread t1 while it invoked the method join() on thread t2 -"+ ThreadState.t1.getState());

// try-catch block
try
{
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}

// ThreadState class implements the interface Runnable
public class ThreadState implements Runnable
{
public static Thread t1;
public static ThreadState obj;

// main method
public static void main(String argvs[])
{
// creating an object of the class ThreadState
obj = new ThreadState();
t1 = new Thread(obj);

// thread t1 is spawned
// The thread t1 is currently in the NEW state.
System.out.println("The state of thread t1 after spawning it - " + t1.getState());

// invoking the start() method on
// the thread t1
t1.start();

// thread t1 is moved to the Runnable state
System.out.println("The state of thread t1 after invoking the method start() on it - " + t1.getState());
}

public void run()
{
ABC myObj = new ABC();
Thread t2 = new Thread(myObj);

// thread t2 is created and is currently in the NEW state.
```

```
System.out.println("The state of thread t2 after spawning it - "+ t2.getState());
t2.start();

// thread t2 is moved to the runnable state
System.out.println("the state of thread t2 after calling the method start() on it - " + t2.getState());

// try-catch block for the smooth flow of the  program
try
{
// moving the thread t1 to the state timed waiting
Thread.sleep(200);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}

System.out.println("The state of thread t2 after invoking the method sleep() on it - "+ t2.getState() );

// try-catch block for the smooth flow of the  program
try
{
// waiting for thread t2 to complete its execution
t2.join();
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println("The state of thread t2 when it has completed it's execution - " + t2.getState());
}


}
```

Output:

```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it - RUNNABLE
The state of thread t2 after spawning it - NEW
```

```
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it - TIMED_WAITING
The state of thread t2 when it has completed it's execution - TERMINATED
```

Explanation: Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

← Prev Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Threads | How to create a thread in Java

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.

13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

The **start() method** of Thread class is used to start a newly created thread. It performs the 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.

1) Java Thread Example by extending Thread class

FileName: Multi.java

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

```
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

```
thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

FileName: MyThread1.java

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

Observe the following program.

FileName: MyThread2.java

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}
// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

```
My new thread
Now the thread is running ...
```

← Prev Next →

Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

Priority: Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

Time of Arrival: Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

Thread Scheduler Algorithms

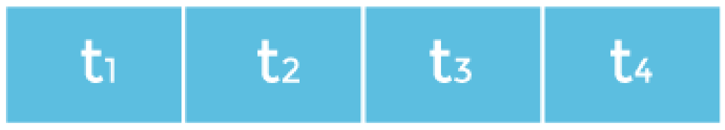
On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads thar arrive first in the runnable queue. Observe the following table:

Threads	Time of Arrival
t1	0
t2	1
t3	2
t4	3

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

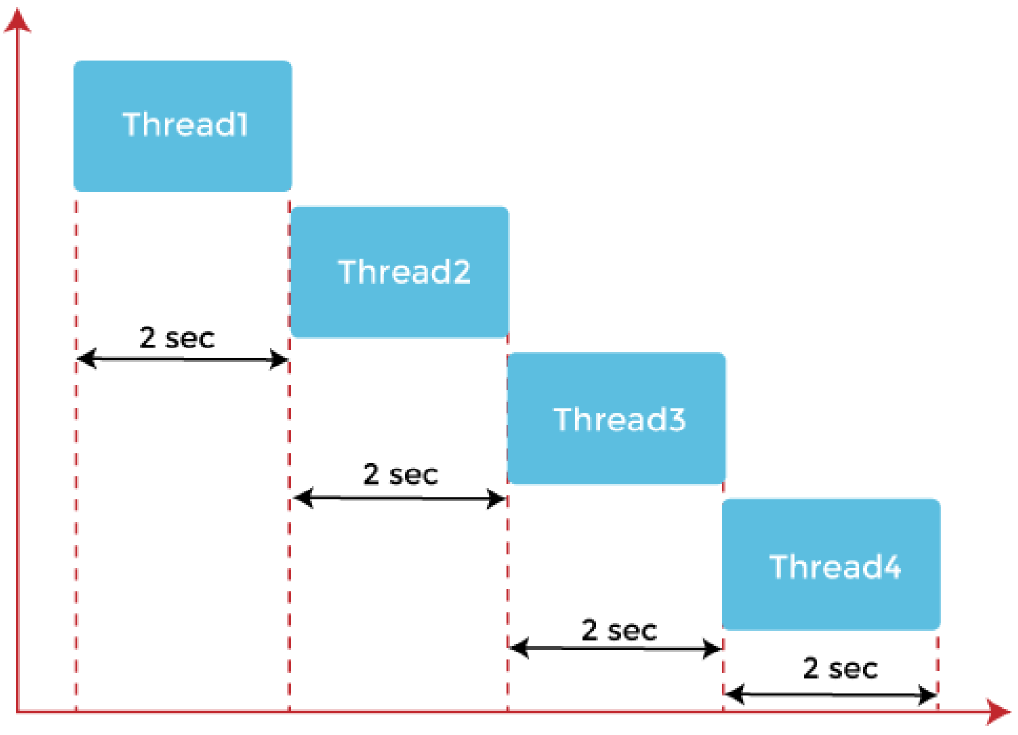


First Come First Serve Scheduling

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

Time-slicing scheduling:

Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

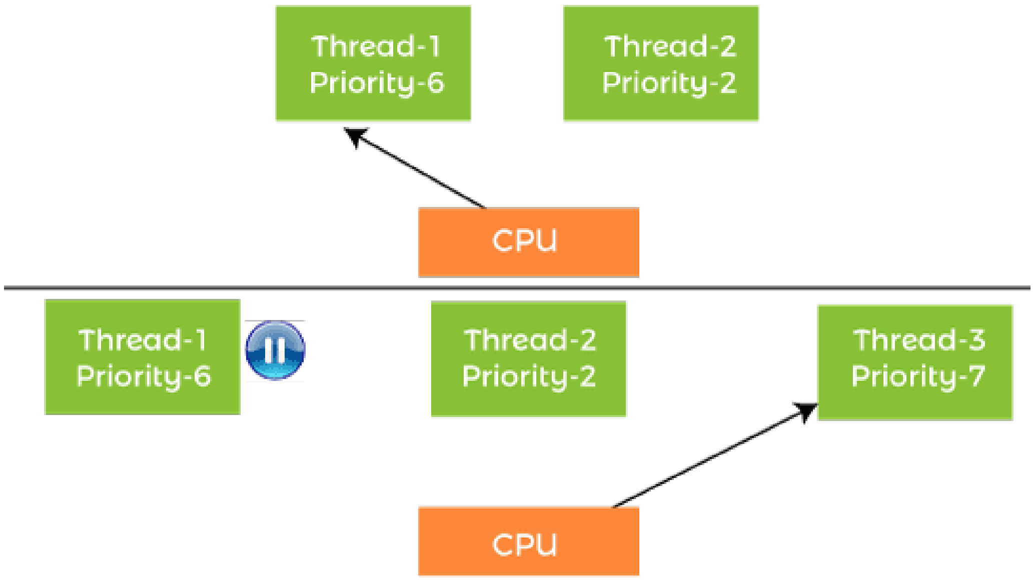


Time slicing scheduling

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

Preemptive-Priority Scheduling:

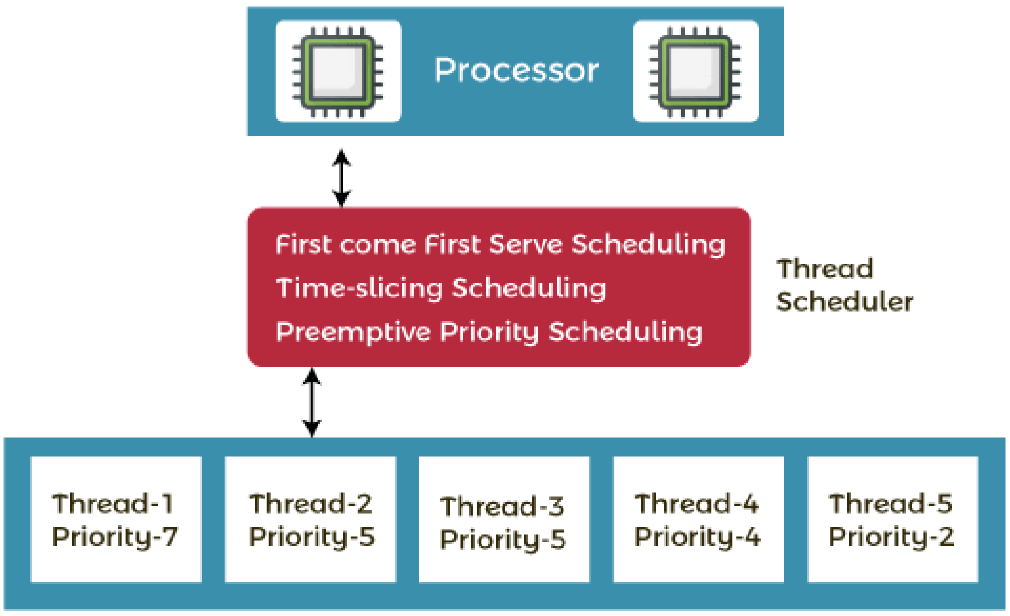
The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



Preemptive-Priority Scheduling

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

Working of the Java Thread Scheduler



Working of Thread Scheduler

Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

← Prev Next →

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

```
public static void sleep(long mls) throws InterruptedException
public static void sleep(long mls, int n) throws InterruptedException
```

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters:

The following are the parameters used in the sleep() method.

mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Example of the sleep() method in Java : on the custom thread

The following example shows how one can use the sleep() method on the custom thread.

FileName: TestSleepMethod1.java

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
  // the thread will sleep for the 500 milli seconds
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

Output:

```
1
1
2
2
3
3
```

```
4
4
```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

Example of the sleep() Method in Java : on the main thread

FileName: TestSleepMethod2.java

```
// important import statements
import java.lang.Thread;
import java.io.*;



public class TestSleepMethod2
{
  // main method
public static void main(String argvs[])
{

try {
for (int j = 0; j < 5; j++)
{

// The main thread sleeps for the 1000 milliseconds, which is 1 sec
// whenever the loop runs
Thread.sleep(1000);

// displaying the value of the variable
System.out.println(j);
}
}
catch (Exception expn)
{
// catching the exception
System.out.println(expn);
}
}
}
```

Output:

AD

```
0
1
2
3
4
```

Example of the sleep() Method in Java: When the sleeping time is -ive

The following example throws the exception IllegalArguementException when the time for sleeping is negative.

FileName: TestSleepMethod3.java

```
// important import statements
import java.lang.Thread;
import java.io.*;

public class TestSleepMethod3
{
// main method
public static void main(String argvs[])
{
// we can also use throws keyword followed by
// exception name for throwing the exception
try
{
for (int j = 0; j < 5; j++)
{

// it throws the exception IllegalArgumentException
// as the time is -ive which is -100
Thread.sleep(-100);

// displaying the variable's value
System.out.println(j);
}
}
catch (Exception expn)
{

// the exception iscaught here
System.out.println(expn);
}
}
}
```

Output:

```
java.lang.IllegalArgumentException: timeout value is negative
```

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Test it Now

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

← Prev Next →

What if we call Java run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

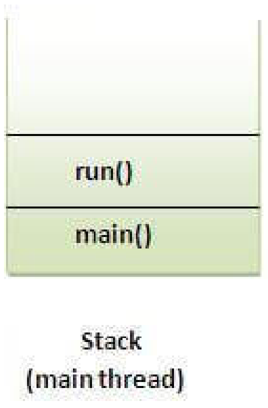
FileName: TestCallRun1.java

```
class TestCallRun1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

Test it Now

Output:

```
running...
```



Problem if you direct call run() method

FileName: TestCallRun2.java

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

Test it Now

Output:

```
1
2
3
4
1
2
3
4
```

As we can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

← Prev Next →

Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose *th* be the object the class Thread whose thread is doing its execution currently, then the *th.join();* statement ensures that *th* is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

Description of The Overloaded join() Method

join(): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

Syntax:

```
public final void join() throws InterruptedException
```

join(long mls): When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds) is over.

Syntax:

```
public final synchronized void join(long mls) throws InterruptedException, where mls is in millise
```

join(long mls, int nanos): When the join() method is invoked, the current thread stops its execution and go into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked called is dead or the wait for the specified time frame(in milliseconds + nanos) is over.

Syntax:

```
public final synchronized void join(long mls, int nanos) throws InterruptedException, where mls i
```

Example of join() Method in Java

The following program shows the usage of the join() method.

FileName: ThreadJoinExample.java

```
// A Java program for understanding
// the joining of threads

// import statement
import java.io.*;

// The ThreadJoin class is the child class of the class Thread
class ThreadJoin extends Thread
{
// overriding the run method
public void run()
{
for (int j = 0; j < 2; j++)
{
try
{
// sleeping the thread for 300 milli seconds
Thread.sleep(300);
System.out.println("The current thread name is: " + Thread.currentThread().getName());
}
// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught: " + e);
}
System.out.println( j );
}
}
}
```

```
public class ThreadJoinExample
{
// main method
public static void main (String argvs[])
{

// creating 3 threads
ThreadJoin th1 = new ThreadJoin();
ThreadJoin th2 = new ThreadJoin();
ThreadJoin th3 = new ThreadJoin();

// thread th1 starts
th1.start();

// starting the second thread after when
// the first thread th1 has ended or died.
try
{
System.out.println("The current thread name is: "+ Thread.currentThread().getName());

// invoking the join() method
th1.join();
}

// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught " + e);
}

// thread th2 starts
th2.start();

// starting the th3 thread after when the thread th2 has ended or died.
try
{
System.out.println("The current thread name is: " + Thread.currentThread().getName());
th2.join();
}

// catch block for catching the raised exception
catch(Exception e)
{
System.out.println("The exception has been caught " + e);
}

// thread th3 starts
th3.start();
}
}
```

Output:

```
The current thread name is: main
The current thread name is: Thread - 0
0
The current thread name is: Thread - 0
1
The current thread name is: main
The current thread name is: Thread - 1
0
The current thread name is: Thread - 1
1
The current thread name is: Thread - 2
0
The current thread name is: Thread - 2
1
```

Explanation: The above program shows that the second thread th2 begins after the first thread th1 has ended, and the thread th3 starts its work after the second thread th2 has ended or died.

The Join() Method: InterruptedException

We have learnt in the description of the join() method that whenever the interruption of the thread occurs, it leads to the throwing of InterruptedException. The following example shows the same.

FileName: ThreadJoinExample1.java

```
class ABC extends Thread
{
Thread threadToInterrupt;
// overriding the run() method
public void run()
{
// invoking the method interrupt
threadToInterrupt.interrupt();
}
}



public class ThreadJoinExample1
{
// main method
public static void main(String[] argvs)
{
try
{
// creating an object of the class ABC
ABC th1 = new ABC();

th1.threadToInterrupt = Thread.currentThread();
th1.start();

// invoking the join() method leads
// to the generation of InterruptedException
th1.join();
}
catch (InterruptedException ex)
{
System.out.println("The exception has been caught. " + ex);
}
}
}
```

Output:

```
The exception has been caught. java.lang.InterruptedException
```

AD

Some More Examples of the join() Method

Let' see some other examples.

Filename: TestJoinMethod1.java

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

Output:

AD

```
1
2
3
4
5
1
1
2
2
3
3
4
4
5
5
```

We can see in the above example, when t1 completes its task then t2 and t3 starts executing.

join(long miliseconds) Method Example

Filename: TestJoinMethod2.jav

```
class TestJoinMethod2 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod2 t1=new TestJoinMethod2();
 TestJoinMethod2 t2=new TestJoinMethod2();
 TestJoinMethod2 t3=new TestJoinMethod2();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

Output:

```
  1
 2
 3
 1
 4
 1
 2
 5
 2
 3
 3
 4
 4
 5
 5
```

In the above example, when t1 completes its task for 1500 milliseconds(3 times), then t2 and t3 start executing.

← Prev Next →

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

```
public String getName(): is used to return the name of a thread.
public void setName(String name): is used to change the name of a thread.
```

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

Example of naming a thread : Using setName() Method

FileName: TestMultiNaming1.java

```
class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  t1.start();
  t2.start();

  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

Test it Now

Output:

```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:Sonoo Jaiswal
running...
running...
```

Example of naming a thread : Without Using setName() Method

One can also set the name of a thread at the time of the creation of a thread, without using the setName() method. Observe the following code.

FileName: ThreadNamingExample.java

```
// A Java program that shows how one can
// set the name of a thread at the time
// of creation of the thread

// import statement
import java.io.*;

// The ThreadNameClass is the child class of the class Thread
class ThreadName extends Thread
{

// constructor of the class
ThreadName(String threadName)
{
// invoking the constructor of
// the superclass, which is Thread class.
super(threadName);
}

// overriding the method run()
public void run()
{
System.out.println(" The thread is executing....");
}
}

public class ThreadNamingExample
{
// main method
public static void main (String argvs[])
{
// creating two threads and settting their name
// using the contructor of the class
ThreadName th1 = new ThreadName("JavaTpoint1");
ThreadName th2 = new ThreadName("JavaTpoint2");

// invoking the getName() method to get the names
// of the thread created above
System.out.println("Thread - 1: " + th1.getName());
System.out.println("Thread - 2: " + th2.getName());



// invoking the start() method on both the threads
th1.start();
th2.start();
}
}
```

Output:

```
Thread - 1: JavaTpoint1
Thread - 2: JavaTpoint2
 The thread is executing....
 The thread is executing....
```

Current Thread

The currentThread() method returns a reference of the currently executing thread.

```
public static Thread currentThread()
```

Example of currentThread() method

FileName: TestMultiNaming2.java

```
class TestMultiNaming2 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[]){
  TestMultiNaming2 t1=new TestMultiNaming2();
  TestMultiNaming2 t2=new TestMultiNaming2();

  t1.start();
  t2.start();
 }
}
```

Test it Now

Output:

```
Thread-0
Thread-1
```

← Prev Next →

Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

FileName: ThreadPriorityExample.java

```
// Importing the required classes
import java.lang.*;

public class ThreadPriorityExample extends Thread
{

// Method 1
// Whenever the start() method is called by a thread
// the run() method is invoked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}

// the main method
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

// 1st Thread
// Displaying the priority of the thread
// using the getPriority() method
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 2nd Thread
// Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 3rd Thread
// // Display the priority of the thread
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);
// 6
System.out.println("Priority of the thread th1 is : " + th1.getPriority());

// 3
System.out.println("Priority of the thread th2 is : " + th2.getPriority());

// 9
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Main thread

// Displaying name of the currently executing thread
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// Priority of the main thread is 10 now
Thread.currentThread().setPriority(10);

System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
}
}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

We know that a thread with high priority will get preference over lower priority threads when it comes to the execution of threads. However, there can be other scenarios where two threads can have the same priority. All of the processing, in order to look after the threads, is done by the Java thread scheduler. Refer to the following example to comprehend what will happen if two threads have the same priority.

FileName: ThreadPriorityExample1.java

```
// importing the java.lang package
import java.lang.*;

public class ThreadPriorityExample1 extends Thread
{

// Method 1
// Whenever the start() method is called by a thread
// the run() method is invoked
public void run()
{
// the print statement
System.out.println("Inside the run() method");
}



// the main method
public static void main(String argvs[])
{

// Now, priority of the main thread is set to 7
Thread.currentThread().setPriority(7);

// the current thread is retrieved
// using the currentThread() method

// displaying the main thread priority
// using the getPriority() method of the Thread class
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

// creating a thread by creating an object of the class ThreadPriorityExample1
ThreadPriorityExample1 th1 = new ThreadPriorityExample1();
// th1 thread is the child of the main thread
// therefore, the th1 thread also gets the priority 7

// Displaying the priority of the current thread
System.out.println("Priority of the thread th1 is : " + th1.getPriority());
}
}
```

Output:

```
Priority of the main thread is : 7
Priority of the thread th1 is : 7
```

Explanation: If there are two threads that have the same priority, then one can not predict which thread will get the chance to execute first. The execution then is dependent on the thread scheduler's algorithm (First Come First Serve, Round-Robin, etc.)

Example of IllegalArgumentException

We know that if the value of the parameter *newPriority* of the method getPriority() goes out of the range (1 to 10), then we get the IllegalArgumentException. Let's observe the same with the help of an example.

FileName: IllegalArgumentException.java

```
// importing the java.lang package
import java.lang.*;

public class IllegalArgumentException extends Thread
{

// the main method
public static void main(String argvs[])
{

// Now, priority of the main thread is set to 17, which is greater than 10
Thread.currentThread().setPriority(17);
// The current thread is retrieved
// using the currentThread() method

// displaying the main thread priority
// using the getPriority() method of the Thread class
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());

}
}
```

When we execute the above program, we get the following exception:

```
Exception in thread "main" java.lang.IllegalArgumentException
        at java.base/java.lang.Thread.setPriority(Thread.java:1141)
        at IllegalArgumentException.main(IllegalArgumentException.java:12)
```

← Prev Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Daemon Thread in Java

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

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

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

Simple example of Daemon thread in java

File: MyThread.java

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();

  t1.setDaemon(true);//now t1 is daemon thread

  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

Test it Now

Output:

```
daemon thread work
user thread work
user thread work
```

Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

File: MyThread.java

```
class TestDaemonThread2 extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```

Test it Now

Output:

```
exception in thread main: java.lang.IllegalThreadStateException
```

← Prev Next →

AD

Youtube For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

Thread Pool Methods

newFixedThreadPool(int s): The method creates a thread pool of the fixed size s.

newCachedThreadPool(): The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

newSingleThreadExecutor(): The method creates a new thread.

Advantage of Java Thread Pool

Better performance It saves time because there is no need to create a new thread.

Real time usage

It is used in Servlet and JSP where the container creates a thread pool to process the request.

Example of Java Thread Pool

Let's see a simple example of the Java thread pool using ExecutorService and Executors.

File: WorkerThread.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }
     public void run() {
        System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
        processmessage();//call processmessage method that sleeps the thread for 2 seconds
        System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
    }
    private void processmessage() {
        try {  Thread.sleep(2000);  } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

File: TestThreadPool.java

```
public class TestThreadPool {
     public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);//calling execute method of ExecutorService
          }
        executor.shutdown();
        while (!executor.isTerminated()) {   }

        System.out.println("Finished all threads");
    }
 }
```

Output:

```
pool-1-thread-1 (Start) message = 0
pool-1-thread-2 (Start) message = 1
pool-1-thread-3 (Start) message = 2
pool-1-thread-5 (Start) message = 4
pool-1-thread-4 (Start) message = 3
pool-1-thread-2 (End)
pool-1-thread-2 (Start) message = 5
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 6
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 7
```

```
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 8
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 9
pool-1-thread-2 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-5 (End)
Finished all threads
```

download this example

Thread Pool Example: 2

Let's see another example of the thread pool.

FileName: ThreadPoolExample.java

```
// important import statements
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.text.SimpleDateFormat;



class Tasks implements Runnable
{
private String taskName;

// constructor of the class Tasks
public Tasks(String str)
{
// initializing the field taskName
taskName = str;
}

// Printing the task name and then sleeps for 1 sec
// The complete process is getting repeated five times
```

```java
public void run()
{
try
{
for (int j = 0; j <= 5; j++)
{
if (j == 0)
{
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");

//prints the initialization time for every task
System.out.println("Initialization time for the task name: "+ taskName + " = " + sdf.format(dt));

}
else
{
Date dt = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("hh : mm : ss");

// prints the execution time for every task
System.out.println("Time of execution for the task name: " + taskName + " = " +sdf.format(dt));

}

// 1000ms = 1 sec
Thread.sleep(1000);
}

System.out.println(taskName + " is complete.");
}

catch(InterruptedException ie)
{
ie.printStackTrace();
}
}
}
```

```
public class ThreadPoolExample
{
// Maximum number of threads in the thread pool
static final int MAX_TH = 3;

// main method
public static void main(String argvs[])
{
// Creating five new tasks
Runnable rb1 = new Tasks("task 1");
Runnable rb2 = new Tasks("task 2");
Runnable rb3 = new Tasks("task 3");
Runnable rb4 = new Tasks("task 4");
Runnable rb5 = new Tasks("task 5");

// creating a thread pool with MAX_TH number of
// threads size the pool size is fixed
ExecutorService pl = Executors.newFixedThreadPool(MAX_TH);

// passes the Task objects to the pool to execute (Step 3)
pl.execute(rb1);
pl.execute(rb2);
pl.execute(rb3);
pl.execute(rb4);
pl.execute(rb5);

// pool is shutdown
pl.shutdown();
}
}
```

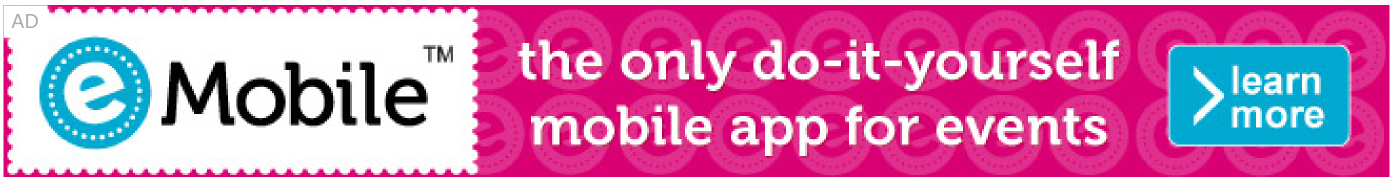
Output:

```
Initialization time for the task name: task 1 = 06 : 13 : 02
Initialization time for the task name: task 2 = 06 : 13 : 02
Initialization time for the task name: task 3 = 06 : 13 : 02
```

```
Time of execution for the task name: task 1 = 06 : 13 : 04
Time of execution for the task name: task 2 = 06 : 13 : 04
Time of execution for the task name: task 3 = 06 : 13 : 04
Time of execution for the task name: task 1 = 06 : 13 : 05
Time of execution for the task name: task 2 = 06 : 13 : 05
Time of execution for the task name: task 3 = 06 : 13 : 05
Time of execution for the task name: task 1 = 06 : 13 : 06
Time of execution for the task name: task 2 = 06 : 13 : 06
Time of execution for the task name: task 3 = 06 : 13 : 06
Time of execution for the task name: task 1 = 06 : 13 : 07
Time of execution for the task name: task 2 = 06 : 13 : 07
Time of execution for the task name: task 3 = 06 : 13 : 07
Time of execution for the task name: task 1 = 06 : 13 : 08
Time of execution for the task name: task 2 = 06 : 13 : 08
Time of execution for the task name: task 3 = 06 : 13 : 08
task 2 is complete.
Initialization time for the task name: task 4 = 06 : 13 : 09
task 1 is complete.
Initialization time for the task name: task 5 = 06 : 13 : 09
task 3 is complete.
Time of execution for the task name: task 4 = 06 : 13 : 10
Time of execution for the task name: task 5 = 06 : 13 : 10
Time of execution for the task name: task 4 = 06 : 13 : 11
Time of execution for the task name: task 5 = 06 : 13 : 11
Time of execution for the task name: task 4 = 06 : 13 : 12
Time of execution for the task name: task 5 = 06 : 13 : 12
Time of execution for the task name: task 4 = 06 : 13 : 13
Time of execution for the task name: task 5 = 06 : 13 : 13
Time of execution for the task name: task 4 = 06 : 13 : 14
Time of execution for the task name: task 5 = 06 : 13 : 14
task 4 is complete.
task 5 is complete.
```

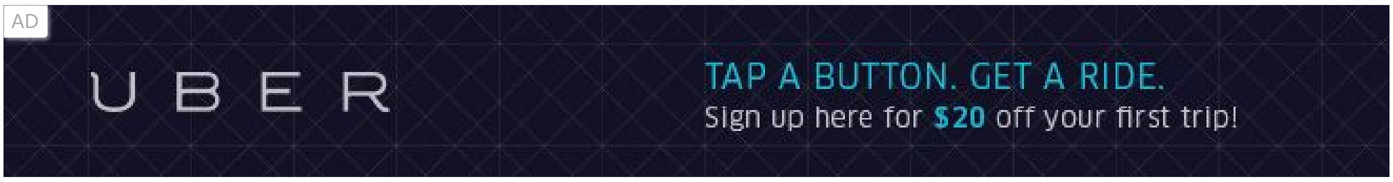
Explanation: It is evident by looking at the output of the program that tasks 4 and 5 are executed only when the thread has an idle thread. Until then, the extra tasks are put in the queue.

The takeaway from the above example is when one wants to execute 50 tasks but is not willing to create 50 threads. In such a case, one can create a pool of 10 threads. Thus, 10 out of 50 tasks are assigned, and the rest are put in the queue. Whenever any thread out of 10 threads becomes idle, it picks up the 11th task. The other pending tasks are treated the same way.

AD


Risks involved in Thread Pools

The following are the risk involved in the thread pools.

AD

UBER
TAP A BUTTON. GET A RIDE.
Sign up here for $20 off your first trip!

Deadlock: It is a known fact that deadlock can come in any program that involves multithreading, and a thread pool introduces another scenario of deadlock. Consider a scenario where all the threads that are executing are waiting for the results from the threads that are blocked and waiting in the queue because of the non-availability of threads for the execution.

Thread Leakage: Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task. For example, when a thread throws the exception and the pool class is not able to catch this exception, then the thread exits and reduces the thread pool size by 1. If the same thing repeats a number of times, then there are fair chances that the pool will become empty, and hence, there are no threads available in the pool for executing other requests.

Resource Thrashing: A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Points to Remember

Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation, as explained above.

Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.

AD

In the end, the thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

One needs to understand the tasks to effectively tune the thread pool. If the given tasks are contrasting, then one should look for pools for executing different varieties of tasks so that one can properly tune them.

To reduce the probability of running JVM out of memory, one can control the maximum threads that can run in JVM. The thread pool cannot create new threads after it has reached the maximum limit.

A thread pool can use the same used thread if the thread has finished its execution. Thus, the time and resources used for the creation of a new thread are saved.

Tuning the Thread Pool

The accurate size of a thread pool is decided by the number of available processors and the type of tasks the threads have to execute. If a system has the P processors that have only got the computation type processes, then the maximum size of the thread pool of P or P + 1 achieves the maximum efficiency. However, the tasks may have to wait for I/O, and in such a scenario, one has to take into consideration the ratio of the waiting time (W) and the service time (S) for the request; resulting in the maximum size of the pool P * (1 + W / S) for the maximum efficiency.

Conclusion

A thread pool is a very handy tool for organizing applications, especially on the server-side. Concept-wise, a thread pool is very easy to comprehend. However, one may have to look at a lot of issues when dealing with a thread pool. It is because the thread pool comes with some risks involved it (risks are discussed above).

← Prev Next →

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

> Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with a given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.

3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.

17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1,new MyRunnable(),"one");
Thread t2 = new Thread(tg1,new MyRunnable(),"two");
Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

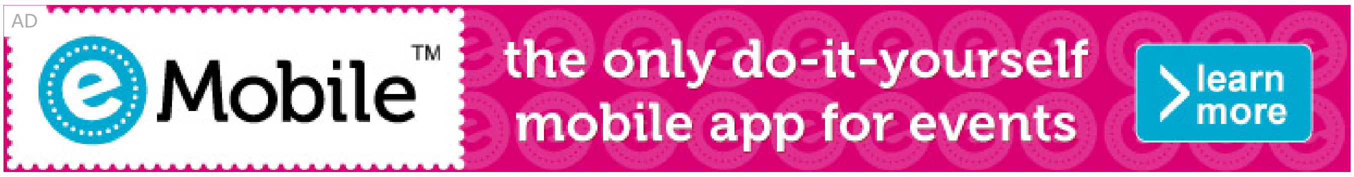
File: ThreadGroupDemo.java

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
          System.out.println(Thread.currentThread().getName());
    }
   public static void main(String[] args) {
      ThreadGroupDemo runnable = new ThreadGroupDemo();
          ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

          Thread t1 = new Thread(tg1, runnable,"one");
          t1.start();
          Thread t2 = new Thread(tg1, runnable,"two");
          t2.start();
```

```
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();

        System.out.println("Thread Group Name: "+tg1.getName());
       tg1.list();

  }
 }
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
```



Thread Pool Methods Example: int activeCount()

Let's see how one can use the method activeCount().

FileName: ActiveCountExample.java

```
// code that illustrates the activeCount() method

// import statement
import java.lang.*;



class ThreadNew extends Thread
{
// constructor of the  class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run method
public void run()
{

for (int j = 0; j < 1000; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}
}
}
}

public class ActiveCountExample
{
// main method
public static void main(String argvs[])
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("The parent group of threads");

ThreadNew th1 = new ThreadNew("first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("second", tg);
System.out.println("Starting the second");

// checking the number of active thread by invoking the activeCount() method
System.out.println("The total number of active threads are: " + tg.activeCount());
}
}
```

Output:

```
Starting the first
Starting the second
The total number of active threads are: 2
```

Thread Pool Methods Example: int activeGroupCount()

Now, we will learn how one can use the activeGroupCount() method in the code.

FileName: ActiveGroupCountExample.java

```
// Java code illustrating the activeGroupCount() method

// import statement
import java.lang.*;



class ThreadNew extends Thread
{
// constructor of the  class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}

}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}

public class ActiveGroupCountExample
{
// main method
public static void main(String argvs[])
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("The parent group of threads");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// checking the number of active thread by invoking the activeGroupCount() method
System.out.println("The total number of active thread groups are: " + tg.activeGroupCount());
}
}
```

Output:

AD

```
Starting the first
Starting the second
```

```
The total number of active thread groups are: 1
the second thread has finished executing
the first thread has finished executing
```

Thread Pool Methods Example: void destroy()

Now, we will learn how one can use the destroy() method in the code.

FileName: DestroyExample.java

```
// Code illustrating the destroy() method

// import statement
import java.lang.*;



class ThreadNew extends Thread
{
// constructor of the  class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}
}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}

public class DestroyExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// waiting until the other threads has been finished
th1.join();
th2.join();

// destroying the child thread group
tg1.destroy();
System.out.println(tg1.getName() + " is destroyed.");

// destroying the parent thread group
tg.destroy();
System.out.println(tg.getName() + " is destroyed.");
}
}
```

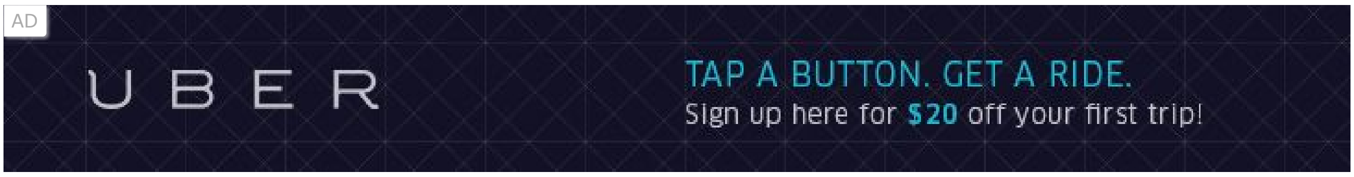
Output:

```
Starting the first
Starting the second
the first thread has finished executing
the second thread has finished executing
the child group is destroyed.
the parent group is destroyed.
```

Thread Pool Methods Example: int enumerate()

Now, we will learn how one can use the enumerate() method in the code.

FileName: EnumerateExample.java

AD

UBER
TAP A BUTTON. GET A RIDE.
Sign up here for $20 off your first trip!

```
// Code illustrating the enumerate() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}


}


System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}


public class EnumerateExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// returning the number of threads kept in this array
Thread[] grp = new Thread[tg.activeCount()];
int cnt = tg.enumerate(grp);
for (int j = 0; j < cnt; j++)
{
System.out.println("Thread " + grp[j].getName() + " is found.");
}
}
}
```

Output:

```
Starting the first
Starting the second
Thread the first is found.
Thread the second is found.
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: int getMaxPriority()

The following code shows the working of the getMaxPriority() method.

FileName: GetMaxPriorityExample.java

```
// Code illustrating the getMaxPriority() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
```

```
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}


}


System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}


public class GetMaxPriorityExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

int priority = tg.getMaxPriority();

System.out.println("The maximum priority of the parent ThreadGroup: " + priority);



}
}
```

Output:

```
Starting the first
Starting the second
The maximum priority of the parent ThreadGroup: 10
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: ThreadGroup getParent()

Now, we will learn how one can use the getParent() method in the code.

FileName: GetParentExample.java

```
// Code illustrating the getParent() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered" + e);
}


}


System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}

public class GetMaxPriorityExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// printing the parent ThreadGroup
// of both child and parent threads
System.out.println("The ParentThreadGroup for " + tg.getName() + " is " + tg.getParent().getName());
System.out.println("The ParentThreadGroup for " + tg1.getName() + " is " + tg1.getParent().getName()



}
}
```

Output:

```
Starting the first
Starting the second
The ParentThreadGroup for the parent group is main
The ParentThreadGroup for the child group is the parent group
the first thread has finished executing
the second thread has finished executing
```

Thread Pool Methods Example: void interrupt()

The following program illustrates how one can use the interrupt() method.

FileName: InterruptExample.java

```
// Code illustrating the interrupt() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered " + e);
}


}


System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}


public class InterruptExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

// invoking the interrupt method
tg.interrupt();

}
}
```

Output:

```
Starting the first
Starting the second
The exception has been encountered java.lang.InterruptedException: sleep interrupted
The exception has been encountered java.lang.InterruptedException: sleep interrupted
the second thread has finished executing
the first thread has finished executing
```

Thread Pool Methods Example: boolean isDaemon()

The following program illustrates how one can use the isDaemon() method.

FileName: IsDaemonExample.java

```
// Code illustrating the isDaemon() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered" + e);
}
}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}

public class IsDaemonExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

if (tg.isDaemon() == true)
{
System.out.println("The group is a daemon group.");
}
else
{
System.out.println("The group is not a daemon group.");
}

}
}
```

Output:

```
Starting the first
Starting the second
The group is not a daemon group.
the second thread has finished executing
the first thread has finished executing
```

Thread Pool Methods Example: boolean isDestroyed()

The following program illustrates how one can use the isDestroyed() method.

FileName: IsDestroyedExample.java

```java
// Code illustrating the isDestroyed() method

// import statement
import java.lang.*;


class ThreadNew extends Thread
{
// constructor of the class
ThreadNew(String tName, ThreadGroup tgrp)
{
super(tgrp, tName);
start();
}

// overriding the run() method
public void run()
{

for (int j = 0; j < 100; j++)
{
try
{
Thread.sleep(5);
}
catch (InterruptedException e)
{
System.out.println("The exception has been encountered" + e);
}

}

System.out.println(Thread.currentThread().getName() + " thread has finished executing");
}
}

public class IsDestroyedExample
{
// main method
public static void main(String argvs[]) throws SecurityException, InterruptedException
{
// creating the thread group
ThreadGroup tg = new ThreadGroup("the parent group");

ThreadGroup tg1 = new ThreadGroup(tg, "the child group");

ThreadNew th1 = new ThreadNew("the first", tg);
System.out.println("Starting the first");

ThreadNew th2 = new ThreadNew("the second", tg);
System.out.println("Starting the second");

if (tg.isDestroyed() == true)
{
System.out.println("The group has been destroyed.");
}
else
{
System.out.println("The group has not been destroyed.");
}

}
}
```

Output:

```
Starting the first
Starting the second
The group has not been destroyed.
the first thread has finished executing
the second thread has finished executing
```

← Prev Next →

Java Shutdown Hook

A special construct that facilitates the developers to add some code that has to be run when the Java Virtual Machine (JVM) is shutting down is known as the **Java shutdown hook**. The Java shutdown hook comes in very handy in the cases where one needs to perform some special cleanup work when the JVM is shutting down. Note that handling an operation such as invoking a special method before the JVM terminates does not work using a general construct when the JVM is shutting down due to some external factors. For example, whenever a kill request is generated by the operating system or due to resource is not allocated because of the lack of free memory, then in such a case, it is not possible to invoke the procedure. The shutdown hook solves this problem comfortably by providing an arbitrary block of code.

Taking at a surface level, learning about the shutdown hook is straightforward. All one has to do is to derive a class using the java.lang.Thread class, and then provide the code for the task one wants to do in the run() method when the JVM is going to shut down. For registering the instance of the derived class as the shutdown hook, one has to invoke the method Runtime.getRuntime().addShutdownHook(Thread), whereas for removing the already registered shutdown hook, one has to invoke the removeShutdownHook(Thread) method.

In nutshell, the shutdown hook can be used to perform cleanup resources or save the state when JVM shuts down normally or abruptly. Performing clean resources means closing log files, sending some alerts, or something else. So if you want to execute some code before JVM shuts down, use the shutdown hook.

When does the JVM shut down?

The JVM shuts down when:

- user presses ctrl+c on the command prompt
- System.exit(int) method is invoked
- user logoff
- user shutdown etc.

The addShutdownHook(Thread hook) method

The addShutdownHook() method of the Runtime class is used to register the thread with the Virtual Machine.

Syntax:

```
public void addShutdownHook(Thread hook){}
```

The object of the Runtime class can be obtained by calling the static factory method getRuntime(). For example:

```
Runtime r = Runtime.getRuntime();
```

The removeShutdownHook(Thread hook) method

The removeShutdownHook() method of the Runtime class is invoked to remove the registration of the already registered shutdown hooks.

Syntax:

```
public boolean removeShutdownHook(Thread hook){ }
```

True value is returned by the method, when the method successfully de-register the registered hooks; otherwise returns false.

Factory method

The method that returns the instance of a class is known as factory method.

Simple example of Shutdown Hook

FileName: MyThread.java

```
class MyThread extends Thread{
    public void run(){
        System.out.println("shut down hook task completed..");
    }
}

public class TestShutdown1{
public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();
r.addShutdownHook(new MyThread());
System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e) {}
}
}
```

Output:

```
Now main sleeping... press ctrl+c to exit
shut down hook task completed.
```

Same example of Shutdown Hook by anonymous class:

FileName: TestShutdown2.java

```
public class TestShutdown2{
public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new Thread(){
public void run(){
    System.out.println("shut down hook task completed..");
    }
}
);

System.out.println("Now main sleeping... press ctrl+c to exit");
try{Thread.sleep(3000);}catch (Exception e) {}
}
}
```

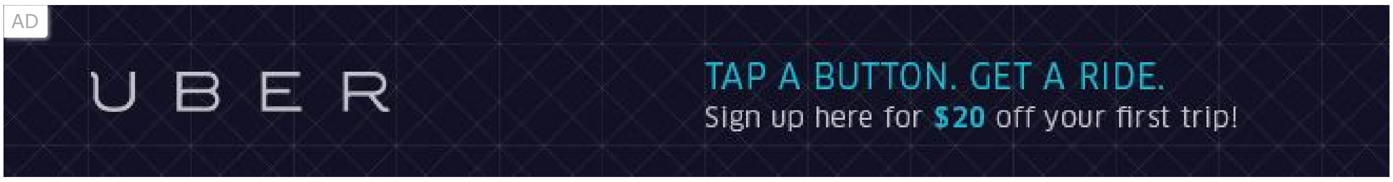
Output:

```
Now main sleeping... press ctrl+c to exit
shut down hook task completed.
```

Removing the registered shutdown hook example

The following example shows how one can use the removeShutdownHook() method to remove the registered shutdown hook.

FileName: RemoveHookExample.java


AD
UBER
TAP A BUTTON. GET A RIDE.
Sign up here for $20 off your first trip!

```
public class RemoveHookExample
{

// the Msg class is derived from the Thread class
static class Msg extends Thread
{

public void run()
{
System.out.println("Bye ...");
}
}

// main method
public static void main(String[] argvs)
{
try
{
// creating an object of the class Msg
Msg ms = new Msg();

// registering the Msg object as the shutdown hook
Runtime.getRuntime().addShutdownHook(ms);

// printing the current state of program
System.out.println("The program is beginning ...");

// causing the thread to sleep for 2 seconds
System.out.println("Waiting for 2 seconds ...");
Thread.sleep(2000);

// removing the hook
Runtime.getRuntime().removeShutdownHook(ms);

// printing the message program is terminating
System.out.println("The program is terminating ...");
}
catch (Exception ex)
{
ex.printStackTrace();
}
}
}
```

Output:

```
The program is beginning ...
Waiting for 2 seconds ...
The program is terminating ...
```

Points to Remember

There are some important points to keep in mind while working with the shutdown hook.

No guarantee for the execution of the shutdown hooks: The first and the most important thing to keep in mind is that there is no certainty about the execution of the shutdown hook. In some scenarios, the shutdown hooks will not execute at all. For example, if the JVM gets crashed due to some internal error, then there is no scope for the shutdown hooks. When the operating system gives the SYSKILL signal, then also it is not possible for the shutdown hooks to come into picture.

Note that when the application is terminated normally the shutdown hooks are called (all threads of the application is finished or terminated). Also, when the operating system is shut down or the user presses the ctrl + c the shutdown hooks are invoked.

Before completion, the shutdown hooks can be stopped forcefully: It is a special case of the above discussed point. Whenever a shutdown hooks start to execute, one can forcefully terminate it by shutting down the system. In this case, the operating system for a specific amount of time. If the job is not done in that frame of time, then the system has no other choice than to forcefully terminate the running hooks.

There can be more than one shutdown hooks, but there execution order is not guaranteed: The JVM can execute the shutdown hooks in any arbitrary order. Even concurrent execution of the shutdown hooks are also possible.

Within shutdown hooks, it is not allowed to unregister or register the shutdown hooks: When the JVM initiates the shutdown sequence, one can not remove or add more any existing shutdown hooks. If one tries to do so, the IllegalStateException is thrown by the JVM.

The Runtime.halt() can stop the shutdown sequence that has been started: Only the Runtime.halt(), which terminates the JVM forcefully, can stop the started shutdown sequence, which also means that invoking the System.exit() method will not work within a shutdown hook.

Security permissions are required when using shutdown hooks: If one is using the Java Security Managers, then the Java code that is responsible for removing or adding the shutdown hooks need to get the shutdown hooks permission at the runtime. If one invokes the method without getting the permission in the secure environment, then it will raise the SecurityException.

← Prev Next →

How to perform single task by multiple threads in Java?

If you have to perform a single task by many threads, have only one run() method. For example:

Program of performing single task by multiple threads

FileName: TestMultitasking1.java

```
class TestMultitasking1 extends Thread{
 public void run(){
   System.out.println("task one");
 }
 public static void main(String args[]){
  TestMultitasking1 t1=new TestMultitasking1();
  TestMultitasking1 t2=new TestMultitasking1();
  TestMultitasking1 t3=new TestMultitasking1();

  t1.start();
  t2.start();
  t3.start();
 }
}
```

Test it Now

Output:

```
task one
task one
task one
```

Program of performing single task by multiple threads

FileName: TestMultitasking2.java

```
class TestMultitasking2 implements Runnable{
```

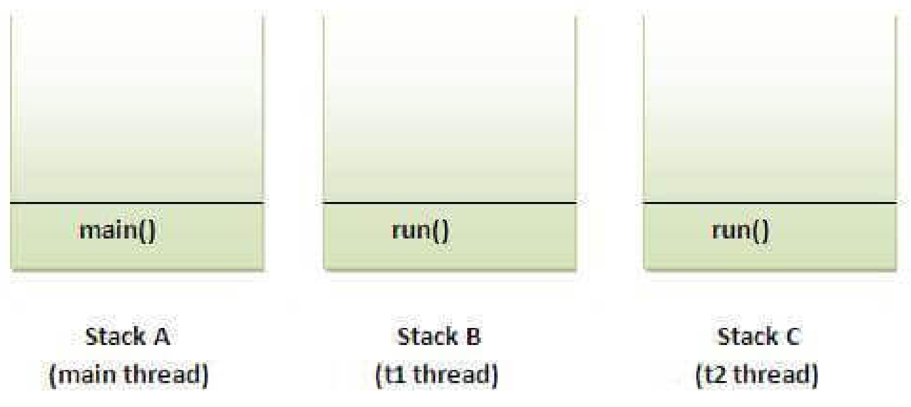
```
public void run(){
System.out.println("task one");
}

public static void main(String args[]){
Thread t1 =new Thread(new TestMultitasking2());//passing annonymous object of TestMultitasking
Thread t2 =new Thread(new TestMultitasking2());

t1.start();
t2.start();

 }
}
```

Test it Now

Output:

```
task one
task one
```

Note: Each thread run in a separate callstack.



How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example:

Program of performing two tasks by two threads

FileName: TestMultitasking3.java

```
class Simple1 extends Thread{
 public void run(){
   System.out.println("task one");
 }
}

class Simple2 extends Thread{
 public void run(){
   System.out.println("task two");
 }
}

 class TestMultitasking3{
 public static void main(String args[]){
  Simple1 t1=new Simple1();
  Simple2 t2=new Simple2();

  t1.start();
  t2.start();
 }
}
```

Test it Now

Output:

```
task one
task two
```

Same example as above by anonymous class that extends Thread class:

Program of performing two tasks by two threads

FileName: TestMultitasking4.java

```
class TestMultitasking4{
 public static void main(String args[]){
  Thread t1=new Thread(){
    public void run(){
      System.out.println("task one");
    }
  };
  Thread t2=new Thread(){
    public void run(){
      System.out.println("task two");
    }
  };


  t1.start();
  t2.start();
 }
}
```

Test it Now

Output:

```
task one
task two
```

Same example as above by anonymous class that implements Runnable interface:

Program of performing two tasks by two threads

FileName: TestMultitasking5.java

```
class TestMultitasking5{
 public static void main(String args[]){
  Runnable r1=new Runnable(){
    public void run(){
      System.out.println("task one");
    }
  };

  Runnable r2=new Runnable(){
    public void run(){
      System.out.println("task two");
    }
  };

  Thread t1=new Thread(r1);
  Thread t2=new Thread(r2);

  t1.start();
  t2.start();
 }
}
```

Test it Now

Output:

```
task one
task two
```

Printing even and odd numbers using two threads

To print the even and odd numbers using the two threads, we will use the synchronized block and the notify() method. Observe the following program.

FileName: OddEvenExample.java

AD

```
// Java program that prints the odd and even numbers using two threads.
// the time complexity of the program is O(N), where N is the number up to which we
// are displaying the numbers
public class OddEvenExample
{
// Starting the counter
int contr = 1;
static int NUM;
// Method for printing the odd numbers
public void displayOddNumber()
{
// note that synchronized blocks are necessary for the code for getting the desired
// output. If we remove the synchronized blocks, we will get an exception.
synchronized (this)
{
// Printing the numbers till NUM
while (contr < NUM)
{
// If the contr is even then display
while (contr % 2 == 0)
{
// handling the exception handle
try
{
wait();
}
catch (InterruptedException ex)
{
ex.printStackTrace();
}
}
// Printing the number
System.out.print(contr + " ");
// Incrementing the contr
contr = contr + 1;
// notifying the thread which is waiting for this lock
notify();
}
}
}
// Method for printing the even numbers
public void displayEvenNumber()
{
synchronized (this)
{
// Printing the number till NUM
while (contr < NUM)
{
// If the count is odd then display
while (contr % 2 == 1)
{
// handling the exception
try
{
wait();
}
catch (InterruptedException ex)
{
ex.printStackTrace();
}
}
// Printing the number
System.out.print(contr + " ");
// Incrementing the contr
contr = contr +1;
// Notifying to the 2nd thread
notify();
}
}
}
// main method
public static void main(String[] argvs)
{
// The NUM is given
NUM = 20;
```

```
// creating an object of the class OddEvenExample
OddEvenExample oe = new OddEvenExample();
// creating a thread th1
Thread th1 = new Thread(new Runnable()
{
public void run()
{
// invoking the method displayEvenNumber() using the thread th1
oe.displayEvenNumber();
}
});
// creating a thread th2
Thread th2 = new Thread(new Runnable()
{
public void run()
{
// invoking the method displayOddNumber() using the thread th2
oe.displayOddNumber();
}
});
// starting both of the threads
th1.start();
th2.start();
}
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

← Prev   Next →

Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

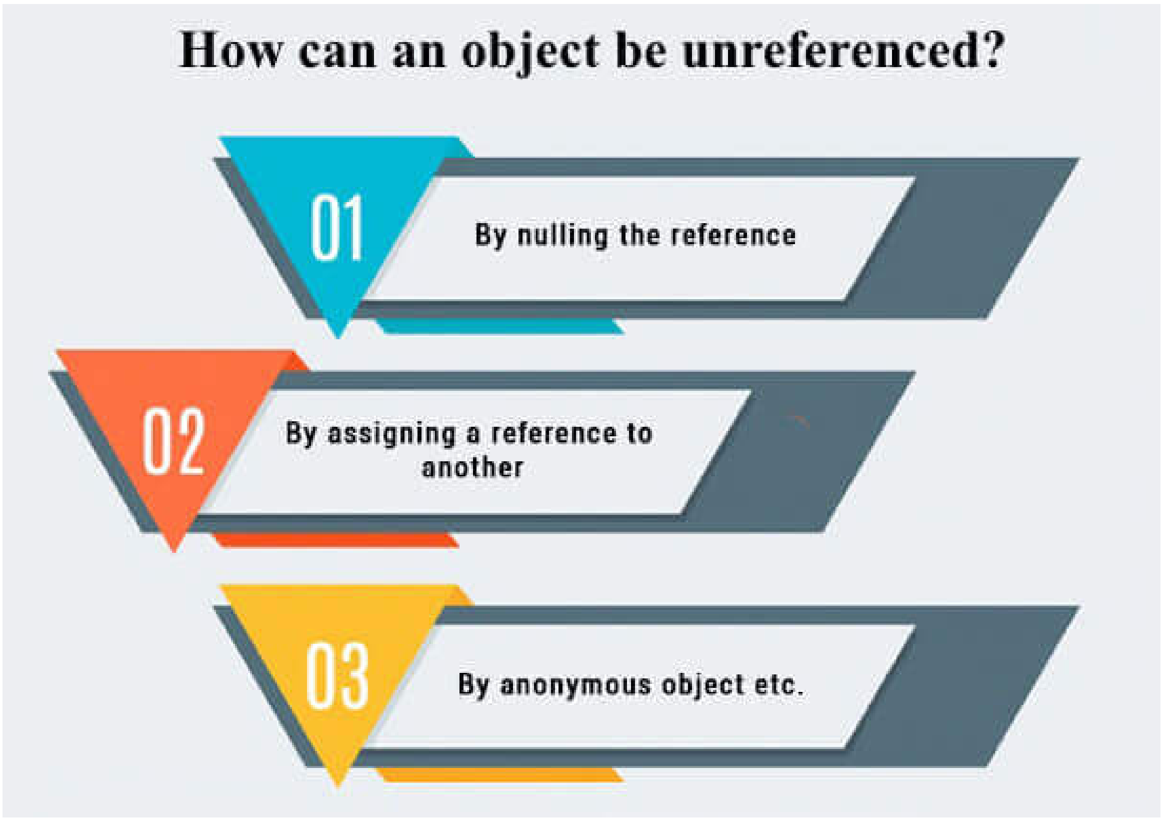
- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

AD


How can an object be unreferenced?
01
By nulling the reference
02
By assigning a reference to another
03
By anonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){}
```

> Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

> Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

Test it Now

```
object is garbage collected
object is garbage collected
```

> Note: Neither finalization nor garbage collection is guaranteed.

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Java Runtime class

Java Runtime class is used *to interact with java runtime environment*. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

Important methods of Java Runtime class

No.	Method	Description
1)	public static Runtime getRuntime()	returns the instance of Runtime class.
2)	public void exit(int status)	terminates the current virtual machine.
3)	public void addShutdownHook(Thread hook)	registers new hook thread.
4)	public Process exec(String command)throws IOException	executes given command in a separate process.
5)	public int availableProcessors()	returns no. of available processors.
6)	public long freeMemory()	returns amount of free memory in JVM.
7)	public long totalMemory()	returns amount of total memory in JVM.

Java Runtime exec() method

```
public class Runtime1{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("notepad");//will open a new notepad
 }
}
```

How to shutdown system in Java

You can use *shutdown -s* command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

```
public class Runtime2{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("shutdown -s -t 0");
 }
}
```

AD

How to shutdown windows system in Java

```
public class Runtime2{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
 }
}
```

How to restart system in Java

```
public class Runtime3{
 public static void main(String args[])throws Exception{
  Runtime.getRuntime().exec("shutdown -r -t 0");
 }
}
```

Java Runtime availableProcessors()

```
public class Runtime4{
 public static void main(String args[])throws Exception{
  System.out.println(Runtime.getRuntime().availableProcessors());
 }
}
```

Java Runtime freeMemory() and totalMemory() method

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

```
public class MemoryTest{
 public static void main(String args[])throws Exception{
  Runtime r=Runtime.getRuntime();
  System.out.println("Total Memory: "+r.totalMemory());
  System.out.println("Free Memory: "+r.freeMemory());

  for(int i=0;i<10000;i++){
   new MemoryTest();
  }
  System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());
  System.gc();
  System.out.println("After gc(), Free Memory: "+r.freeMemory());
 }
}
```

```
Total Memory: 100139008
Free Memory: 99474824
After creating 10000 instance, Free Memory: 99310552
After gc(), Free Memory: 100182832
```

← Prev Next →