***Java MultiThreading***

* **Multiprocessing and multithreading, both are used to achieve multitasking.**
* But we use **multithreading** than multiprocessing because **threads share a common 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.

### Multitasking

* + Multitasking is a process of **executing multiple tasks simultaneously**. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:
  + **Process-based** **Multitasking**(Multiprocessing)
  + **Thread-based Multitasking**(Multithreading)

### Process-based Multitasking (Multiprocessing)

* Each process have its **own address in memory** i.e. each process allocates separate memory area.
* **Process** is **heavyweight**.
* Cost of communication between the process is **high**.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

### Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.
  + **Note: At least one process is required for each thread.**
  + **Note: At a time one thread is executed only.**

|  |
| --- |
| Life cycle of a Thread (Thread States) A thread can be in one of the five states. According to sun, there is only 4 states in**thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.   1. [Life cycle of a thread](http://www.javatpoint.com/life-cycle-of-a-thread)    1. [New](http://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew)    2. [Runnable](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable)    3. [Running](http://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning)    4. [Non-Runnable (Blocked)](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked)    5. [Terminated](http://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)   But for better understanding the threads, we are explaining it in the 5 states.  The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:   1. New 2. Runnable 3. Running 4. Non-Runnable (Blocked) 5. Terminated   thread life cycle in java 1) New The thread is in new state if you **create an instance of Thread class but before the invocation of start()** method. |

### 2) Runnable

The thread is in **runnable state after invocation of start()** method, but the **thread scheduler has not selected it to be the running thread.**

### 3) Running

The thread is in **running state** if the **thread scheduler has selected it**.

### 4) Non-Runnable (Blocked)

This is the state when the thread is **still alive, but is currently not eligible to run**.

### 5) Terminated

A thread is in terminated or dead state when its **run() method exits**.

### 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) //will be used in ThreadGroup concept * Thread(Runnable r,String name) //will be used in ThreadGroup concept |

### 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. |

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread **starts(with new callstack)**. * The thread moves from **New state to the Runnable state**. * When the thread gets a **chance to execute**, its target **run() method will run**. |

* NOTE : - If you **are not extending the Thread class**,your class object would **not be treated as a thread** object.**So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute**.
* ***Thread Scheduler in Java***
* Thread scheduler in java is the **part of the JVM** that decides **which thread should run**.
* There is **no guarantee that which runnable thread will be chosen to run by the thread scheduler**.
* Only one thread at a time can run in a single process.
* The **thread scheduler** mainly uses **preemptive** or **time slicing scheduling** to schedule the threads.
* **Difference between preemptive scheduling and time slicing**
  + Under **preemptive** scheduling, the **highest priority task executes** until it enters the **waiting** or **dead states** or a **higher priority task** comes into existence. Under **time slicing**, a task **executes for a predefined slice of time** and then reenters the pool of ready tasks. **The scheduler then determines which task should execute next**, based on priority and other factors.
* ***NOTE : - 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.
* ***What if we call run() method directly instead start() method?***
  + Each thread **starts** in a **separate call stack**.
* Invoking the **run() method from main thread**, the run() method goes onto the current call stack rather than at the beginning of a new call stack.
* ***Join method in Thread Class***
  + ***The join() method waits for a thread to die.***
  + It will **holds the Starting of** all the **other threads** till it **executes completely**.

**public** **class** OverLo **extends** Thread{

**public** **void** run() {

**int** i=0;

**while**(i<10){

// logic to print 1 to 5 on hold of 0.5 sec

}

**public** **static** **void** main(String[] args) {

OverLo o1 = **new** OverLo();

OverLo o2 = **new** OverLo();

OverLo o3 = **new** OverLo();

o1.start();

**try** {

***==> ==> o1.join();***

} **catch** (InterruptedException e) {}

o2.start();

o3.start();

}}

output :-

**1 2 3 4 5** 1 1 2 2 3 3 4 4 5 5

* *Thread Priority*

**3 constants defined in Thread class:**

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

* To set the priority of Thread

TestMultiPriority1 m1=new TestMultiPriority1();

TestMultiPriority1 m2=new TestMultiPriority1();

**m1.setPriority(Thread.MIN\_PRIORITY);**

**m2.setPriority(Thread.MAX\_PRIORITY);**

m1.start();

m2.start();

Daemon Thread in Java

* Daemon thread in java is a **service provider thread** that **provides services to the user thread**. Its life **depends 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.

**IMP\_Notes\_DeamonThreads**

* 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.
* JVM **terminates the daemon thread** if there is **no user thread**.
* If you want to **make a user thread as Daemon**, it **must not be started** otherwise it will **throw IllegalThreadStateException**.

### 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. |

Java Thread Pool (It is Concept of **Java Concurrent** API)

* **Java Thread pool** represents a **group of worker threads** that **are waiting for the job and reuse many times**.
* In case of thread pool, **a group of fixed size threads are created**. A thread from the thread pool is pulled out and **assigned a job by the service provider**. After **completion of the job, thread is contained in the thread pool again**.
* **Advantage of Java Thread Pool**
  + Better performance It **saves time because there is no need to create new thread**.
* **Real time usage**
  + It is **used in Servlet and JSP where container creates a thread pool to process the request**.
* There is a following class and Interfaces in java.concurrent package
  + **ExecutorService (I)**
  + **Executors (C)**
* With use of **Runnable Interface** also,
* With the help of Executors class’s newFixedThreadPool(int) method, we are creating new thread pool and storing its instance in ExecutorServices Interface type.

ExecutorServices executor = Executors.newFixedThreadPool(5);

* And by creating main Thread’s Class object we are storing in Runnable interface type

Runnable mainThreadObj = new MainThread();

* By calling method of executor (the object which we have created ) we are starting the threads.

**executor.execute(*mainThreadObj*);** //Starting the Thread by **passing mainThread object**

executor.shutdown(); // Stoping the Thread

while(!executor.isTerminated()){}

* **Example Full Program:-**

import java.util.concurrent.Executors;

import java.util .concurrent.ExecutorService;

class SimpleThread extends Thread{

String msg;

SimpleThread(String s){

this.msg =s;

}

public void run(){

System.out.println(Thread.currentThread().getName()+"Start"+msg);

process();

System.out.println(Thread.currentThread().getName()+"Stop");

}

static void process(){

try{

Thread.sleep(500);

}catch(Exception e){}}}

public class ThreadPool{

public static void main(String[] a){

ExecutorService executor = Executors.newFixedThreadPool(5);

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

Runnable simplethread = new SimpleThread(" "+i);

executor.execute(simplethread);

}

executor.shutdown();

while(!executor.isTerminated()){}

System.out.println("Finished All Threads...");

}}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](http://www.javatpoint.com/ShutdownHook-thread)[**← prev**](http://www.javatpoint.com/java-thread-pool)  ThreadGroup in Java   * Java provides a **convenient way to group multiple threads in a single**   **object**. In such way, **we can suspend, resume or interrupt 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.  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 given parent group and name. |   Important methods of ThreadGroup class  There are many methods in ThreadGroup class. A list of important methods are given below.   |  |  |  | | --- | --- | --- | | **No.** | **Method** | **Description** | | 1) | int activeCount() | returns no. of threads running in current group. | | 2) | int activeGroupCount() | returns a no. of active group in this thread group. | | 3) | void destroy() | destroys this thread group and all its sub groups. | | 4) | String getName() | returns the name of this group. | | 5) | ThreadGroupget Parent() | returns the parent of this group. | | 6) | void interrupt() | interrupts all threads of this group. | | 7) | void list() | prints information of this group to standard console. | |

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

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

**Example :-**

**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();

**tg1.intrupt();**

**tg1.list();**

**tg1.koeBhiMethodofThreadPoolClass();**

Java Shutdown Hook

* If we want to **execute some code before JVM shuts down**, **use shutdown hook**.
* The shutdown hook can be used to **perform cleanup resource** or **save the state when JVM shuts down normally or abruptly**. Performing **clean resource means closing log file**, **sending some alerts or something else.**

**The addShutdownHook(Thread hook) method**

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

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

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

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

**Factory method**

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

Simple example of Shutdown Hook

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) {}

} }

**Note: The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.**

* **Performing same task by MultiThreading with use of Anonymous class.**

|  |
| --- |
| Same example as above by annonymous class that extends Thread class:  *Program of performing two tasks by two threads*  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();   }  }  Same example as above by annonymous class that implements *Runnable interface:*  Program of performing two tasks by two threads  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();   }} |

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 annonymous object etc.

1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

3) By annonymous object:

1. **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:

1. **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.

1. **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

TestGarbage1 s1=**new** TestGarbage1();

  TestGarbage1 s2=**new** TestGarbage1();

  s1=**null**;

  s2=**null**;

**System.gc();**

 }}

**Note: Neither finalization nor garbage collection is guaranteed.**

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 windows 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("c:\\Windows\\System32\\shutdown -s -t 0");**

 }}

* To know about free and total memory in JVM

Runtime **r**=Runtime.getRuntime();

System.out.println("Total Memory: "+**r.totalMemory()**);

System.out.println("Free Memory: "+**r.freeMemory()**);

Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.