Multithreading

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. 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.

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.

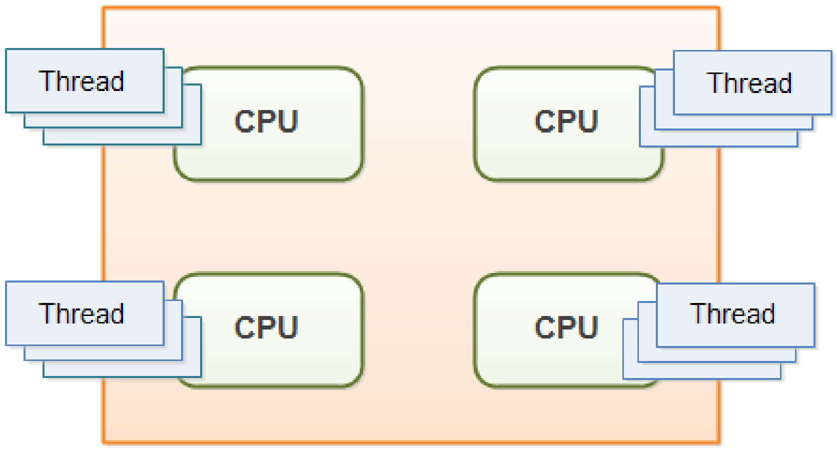

Figure 1: java-cpu-thread

Advantages of Multithreading : - It **doesn't block the user** because threads are independent and you can perform multiple operations at same time. - You **can perform many operations simultaneously** so it saves time. - Threads are **independent** so it doesn't affect other threads if exception occur 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 by two ways: - Process-based Multitasking(Multiprocessing) - Thread-based Multitasking(Multithreading)

Multiprocessing	Multithreading
Each process have its own address in memory i.e. each process allocates separate memory area.	Threads share the same address space.
Process is heavyweight.	Thread is lightweight.
Cost of communication between the process is high.	Cost of communication between the thread is low.
Context-switching require some time for saving & loading registers, memory maps, updating lists etc.	Context-switching between the threads takes less time than process.

NOTE : Context switching (aka process/task switching) is the switching of the CPU (central processing unit) from one process or thread to another.

Life Cycle of a Thread

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows: 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.

NOTE : 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.

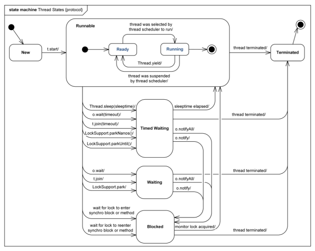

Figure 2: Java Thread Life Cycle

Creating a Thread

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:

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

public void start(): starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join(): waits for a thread to die.

public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive.

public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

public void suspend(): is used to suspend the thread(deprecated).

public void resume(): is used to resume the suspended thread(deprecated).

public void stop(): is used to stop the thread(deprecated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

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

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.

Creating Thread Codes :

1. Thread Example by extending Thread class

```
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();
    }
}
```

2. Thread Example by implementing Runnable interface

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

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 : We cannot start a thread twice. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown the moment it is started for the second time.

sleep() method in Java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

The Thread class provides two methods for sleep : 1. public static void sleep(long miliseconds)throws InterruptedException 2. public static void sleep(long miliseconds, int nanos)throws InterruptedException

```
Thread.sleep(500)
```

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.

run() method in Java

In Java, 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.

Problem if you use run() directly instead of start() "

There is no context-switching in the below program because here t1 and t2 will be treated as normal object not thread object. Output will be : 1 2 3 4 5 1 2 3 4 5 (One object will finish before starting next)

```
class TestCallRun 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();
    }
}
```

join() method in Java

Java Thread **join()** method can be used to pause the current thread execution, until the specified thread is dead.

There are three join methods : 1. public void join() 2. public void join(long millis) 3. public void join(long millis, int nanos)

Examples :

```
t1.join();   \\Current thread will be paused, until t1 is dead. (t1 will start executing)
```

```
t1.join(1500);  \\Current thread will be paused, t1 will execute for 1500 milliseconds.
```

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. But, we can change the name of the thread by using setName() method. - **public String getName():** is used to get the name of a thread.

```
t1.getName()
```

- **public void setName(String name):** is used to change the name of a thread.

```
t1.setName("My sweet thread");
```

Getting the Current Thread

public static Thread currentThread(): The currentThread() method returns a reference of currently executing thread.

```
public void run(){
 System.out.println(Thread.currentThread().getName());
}
```

Priority of a Thread

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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.

The 3 constants defined in Thread class: - public static int MIN_PRIORITY - public static int NORM_PRIORITY - 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.

```
System.out.println("thread priority is:"+t1.getPriority());
```

```
t1.setPriority(Thread.MIN_PRIORITY);
```

Daemon Thread

Daemon thread in java is a service provider thread that provides services to the user thread. It has no role in life than to serve user threads. Its life depends on user threads i.e. when all the user threads dies, JVM terminates this thread automatically. It is a low priority thread.

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

NOTE : 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.

Methods for Daemon Thread: - public void setDaemon(boolean status): is used to mark the current thread as daemon thread or user thread.

```
t1.setDaemon(true);   //Now, t1 is a daemon thread
t1.start()
```

- **public boolean isDaemon():** is used to check that current is daemon.

```
if(t1.isDaemon()) {
 ...
}
```

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

```
t1.start();
t1.setDaemon(true);     //will Throw Exception Here
```

Java Thread Pool

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 is 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.

Example Syntax:

```
ExecutorService executor = Executors.newFixedThreadPool(5);  //creating a pool of 5 threads
```

```
Runnable worker = new ...
executor.execute(worker);   //calling execute method of ExecutorService
```

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

NOTE : Now suspend(), resume() and stop() methods are **deprecated**.

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 :

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)

ThreadGroup getParent()

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.

Creating group of 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();
```

Java 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. So if you want to execute some code before JVM shuts down, use shutdown hook.

The JVM shuts down when: user presses ctrl+c on the command prompt, System.exit(int) method is invoked, user logoff, shutdown, etc.

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

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

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

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

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

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

Multitask

Each thread run in a separate **callstack**.

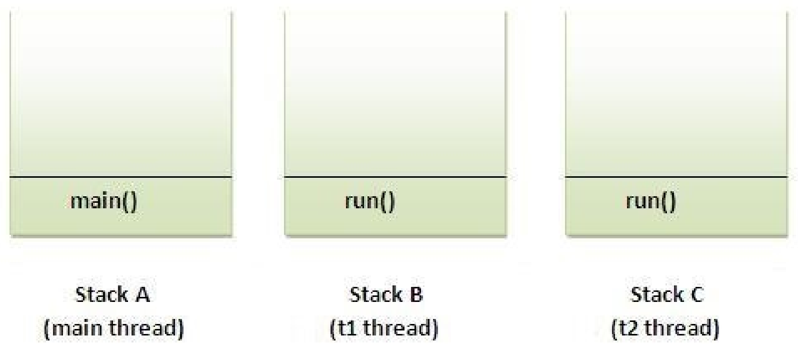

Figure 3: callstack

```
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();
        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 an object is unreferenced: - By nulling the reference

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

- 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
```

- By anonymous object

```
new Employee();
```

- etc.

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. This method is defined in System class as:

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

```
System.gc();    // explicitely invoking the garbage collector
```

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

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
 }
}
```

Shutdown system in Java :

```
Runtime.getRuntime().exec("shutdown -s -t 0");  // Shutdown
```

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.

```
Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");  // Windows Shutdown
```

```
Runtime.getRuntime().exec("shutdown -r -t 0");  // Restart
```

Runtime availableProcessors() method :

```
System.out.println(Runtime.getRuntime().availableProcessors());
```

Runtime freeMemory() and totalMemory() method :

```
System.out.println("Total Memory: "+Runtime.totalMemory());
System.out.println("Free Memory: "+Runtime.freeMemory());
```