**Multithreading in java**

A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate path of execution that’s the reason if an exception occurs in one thread, it doesn’t affect the execution of other threads. All threads of a process share the common memory. The process of executing multiple threads simultaneously is known as multithreading.

* The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
* Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

**Thread vs Process**

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

**Multitasking vs Multithreading vs Multiprocessing vs parallel processing**

**Multitasking:**Ability to execute more than one task at the same time is known as multitasking.

**Multithreading:**We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

**Multiprocessing:** It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

Life cycle of a Thread (Thread States):

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



NEW – A thread that has not yet started is in this state.

RUNNABLE – A thread executing in the Java virtual machine is in this state.

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

NON- RUNNABLE(BLOCKED) – A thread that is blocked waiting for a monitor lock is in this state.

TERMINATED – A thread that has exited is in this state.

**Creating a thread in Java:**

There are two ways to create a thread in Java:

* By extending Thread class.
* By implementing Runnable interface.

**METHODS USED IN THREAD:**

getName(): It is used for Obtaining a thread’s name

getPriority(): Obtain a thread’s priority

isAlive(): Determine if a thread is still running

join(): Wait for a thread to terminate

run(): Entry point for the thread

sleep(): suspend a thread for a period of time

start(): start a thread by calling its run() method

**Thread creation by extending Thread class**

**Example:**

class MultithreadingDemo extends Thread{

public void run(){

System.out.println("My thread is in running state.");

}

public static void main(String args[]){

MultithreadingDemo obj=new MultithreadingDemo();

obj.start();

}

}

**Thread creation by implementing Runnable Interface**

**Example:**

class MultithreadingDemo implements Runnable{

public void run(){

System.out.println("My thread is in running state.");

}

public static void main(String args[]){

MultithreadingDemo obj=new MultithreadingDemo();

Thread tobj =new Thread(obj);

tobj.start();

}

}

**Output:**

My thread is in running state.

**Thread priorities**

* Thread priorities are the integers which decide how one thread should be treated with respect to the others.
* Thread priority decides when to switch from one running thread to another, process is called context switching
* A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
* A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
* To set the priority of the thread setPriority() method is used which is a method of the class Thread Class.
* In place of defining the priority in integers, we can use MIN\_PRIORITY, NORM\_PRIORITY or MAX\_PRIORITY.

**Thread join() method in Java**

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution).

**Why we use join() method?**

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.

## Without using join():

**Without using join()**

public class JoinExample2 {

public static void main(String[] args) {

Thread th1 = new Thread(new MyClass2(), "th1");

Thread th2 = new Thread(new MyClass2(), "th2");

Thread th3 = new Thread(new MyClass2(), "th3");

th1.start();

th2.start();

th3.start();

}

}

class MyClass2 implements Runnable{

@Override

public void run() {

Thread t = Thread.currentThread();

System.out.println("Thread started: "+t.getName());

try {

Thread.sleep(4000);

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("Thread ended: "+t.getName());

}

}

**Output:**

Thread started: th1

Thread started: th3

Thread started: th2

Thread ended: th1

Thread ended: th3

Thread ended: th2

**The same example with join()**

public class JoinExample {

public static void main(String[] args) {

Thread th1 = new Thread(new MyClass(), "th1");

Thread th2 = new Thread(new MyClass(), "th2");

Thread th3 = new Thread(new MyClass(), "th3");

// Start first thread immediately

th1.start();

/\* Start second thread(th2) once first thread(th1)

\* is dead

\*/

try {

th1.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

th2.start();

/\* Start third thread(th3) once second thread(th2)

\* is dead

\*/

try {

th2.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

th3.start();

// Displaying a message once third thread is dead

try {

th3.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("All three threads have finished execution");

}

}

class MyClass implements Runnable{

@Override

public void run() {

Thread t = Thread.currentThread();

System.out.println("Thread started: "+t.getName());

try {

Thread.sleep(4000);

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("Thread ended: "+t.getName());

}

}

Output:

Thread started: th1

Thread ended: th1

Thread started: th2

Thread ended: th2

Thread started: th3

Thread ended: th3

All three threads have finished execution

**Thread Scheduling**

* Execution of multiple threads on a single CPU, in some order, is called scheduling.

**Types of scheduling**

* Waiting and Notifying
  + Waiting [wait()] and notifying [notify(), notifyAll()] provides means of communication between threads that synchronize on the same object.
* wait(): when wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.
* notify(): This wakes up threads that called wait() on the same object and moves the thread to ready state.
* notifyAll(): This wakes up all the threads that called wait() on the same object.

**Why don’t we call run() method directly, why call start() method?**

We can call run() method if we want but then it would behave just like a normal method and we would not be able to take the advantage of multithreading.

When the run method gets called though start() method then a new separate thread is being allocated to the execution of run method, so if more than one thread calls start() method that means their run method is being executed by separate threads (these threads run simultaneously).

On the other hand if the run() method of these threads are being called directly then the execution of all of them is being handled by the same current thread and no multithreading will take place, hence the output would reflect the sequential execution of threads in the specified order.

**Calling run() method**

public class RunMethodExample implements Runnable{

public void run(){

for(int i=1;i<=3;i++){

try{

Thread.sleep(1000);

}catch(InterruptedException ie){

ie.printStackTrace();

}

System.out.println(i);

}

}

public static void main(String args[]){

Thread th1 = new Thread(new RunMethodExample(), "th1");

Thread th2 = new Thread(new RunMethodExample(), "th2");

th1.run();

th2.run();

}

}

Output:

1

2

3

1

2

3

As you can observe in the output that multithreading didn’t place here, it because both the run methods are being handled by the current thread. that treated them like normal methods and had them executed in the specified order rather then having them executed simultaneously. Thread scheduler didn’t play any role here.

**Calling start() method:**

Multithreading takes place and the output reflects simultaneous execution of threads.

public class RunMethodExample2 {

public void run(){

for(int i=1;i<=3;i++){

try{

Thread.sleep(1000);

}

catch(InterruptedException ie){

ie.printStackTrace();

}

System.out.println(i);

}

}

public static void main(String args[]){

Thread th1 = new Thread(new RunMethodExample(), "th1");

Thread th2 = new Thread(new RunMethodExample(), "th2");

th1.start();

th2.start();

}

}

Output:

1

1

2

2

3

3

**Synchronization**

* Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
* When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.

Synchronized(object)

{

// statement to be synchronized

}