MULTITHREADING

OBJECT ORIENTED PROGRAMMING LAB

SESSION - 10

Thread:

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

Multithreading:

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs. Java Multithreading is mostly used in games, animation etc.

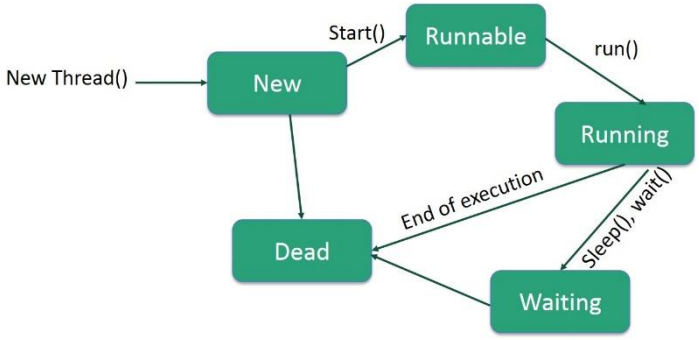
Multithreading and multitasking:

Multi-threading is a more "light weight" form of concurrency: there is less context per thread than per process. As a result thread lifetime, context switching and synchronisation costs are lower. The shared address space (noted above) means data sharing requires no extra work.

Multi-processing has the opposite benefits. Since processes are insulated from each other by the OS, an error in one process cannot bring down another process. Contrast this with multi-threading, in which an error in one thread can bring down all the threads in the process. Further, individual processes may run as different users and have different permissions.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle –

New – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Creating thread in Java:

1) By extending thread class

- The class should extend Java Thread class.
- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.
void run(): The new thread begins its life inside this method.

Example 10.1:

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

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example 10.2:

```
public class MyThread implements Runnable {
   public void run(){
     System.out.println("thread is running..");
   }
   public static void main(String[] args) {
     Thread t = new Thread(new MyThread());
     t.start();
}
```

Difference between Extends Thread class vs Implements Runnable Interface?

Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.

If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

Example 10.3:

```
import package lesson1;
class MyClass extends Thread {
        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println(Thread.currentThread().getId() + "
Value - " + i);
                }
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
public class Demo {
        public static void main(String[] args) {
                MyClass myClass = new MyClass();
                myClass.start();
                MyClass myClass2 = new MyClass();
                myClass2.start();
        }
}
```

Exercise 10.4:

```
import package lesson2;
class MyClass implements Runnable {
        public void run() {
                for(int i = 0; i < 10; i++){
                        System.out.println(Thread.currentThread().getId() + "
Value - " + i);
                }

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}

public class Demo {
        public static void main(String[] args){
                Thread t1 = new Thread(new MyClass());
                Thread t2 = new Thread(new MyClass());
                t1.start();
                t2.start();
        }
}
}
```

The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Example 10.5.

```
import package lesson3;


public class Demo {


	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Thread t1 = new Thread(new Runnable(){
			@Override
			public void run() {
				// TODO Auto-generated method stub
				for(int i = 0; i < 10; i++){

	System.out.println(Thread.currentThread().getId() + " Value - " + i);
				}
				try {
					Thread.sleep(1000);
				} catch (InterruptedException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}
			}
		});


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

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.

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example 10.6.

```
import package lesson4;


public class Demo {
	public static int count = 0;
	public static int count2 = 0;
	public static synchronized void incount() {
		count++;
	}
	public static void main(String[] args) {
		Thread t1 = new Thread(new Runnable(){
			@Override
			public void run(){
				for(int i = 0; i < 1000; i++){
					incount();
					count2++;
				}
			}
		});
```

```
Thread t2 = new Thread(new Runnable(){
                        @Override
                        public void run(){
                                for(int i = 0; i < 1000; i++){
                                        try {
                                                Thread.sleep(1);
                                        } catch (InterruptedException e) {

                                        e.printStackTrace();
                                        }
                                        incount();
                                }


                        }
                });

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

                try {
                        t1.join();
                        t2.join();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                System.out.println("Value " + count);


        }

}
```