Unit 1

Threads And MultiThreads

Any application can have multiple processes (instances). Each of this process can be assigned either as a single thread or multiple threads. We will see in this tutorial how to perform multiple tasks at the same time and also learn more about threads and synchronization between threads.

What is Single Thread?

A single thread is basically a lightweight and the smallest unit of processing. Java uses threads by using a "Thread Class".

There are two types of thread – **user thread and daemon thread** (daemon threads are used when we want to clean the application and are used in the background).

When an application first begins, user thread is created. Post that, we can create many user threads and daemon threads.

Single Thread Example:

```
1.  package demotest;
2.
3.  public class GuruThread
4.  {
5.          public static void main(String[] args) {
6.                  System.out.println("Single Thread");
7.          }
8.  }
```

Advantages of single thread:

- Reduces overhead in the application as single thread execute in the system
- Also, it reduces the maintenance cost of the application.

What is Multithreading in Java?

MULTITHREADING in Java is a process of executing two or more threads simultaneously to maximum utilization of CPU. Multithreaded applications execute two or more threads run concurrently. Hence, it is also known as Concurrency in Java. Each thread runs parallel to each other. Mulitple threads don't allocate separate memory area, hence they save memory. Also, context switching between threads takes less time.

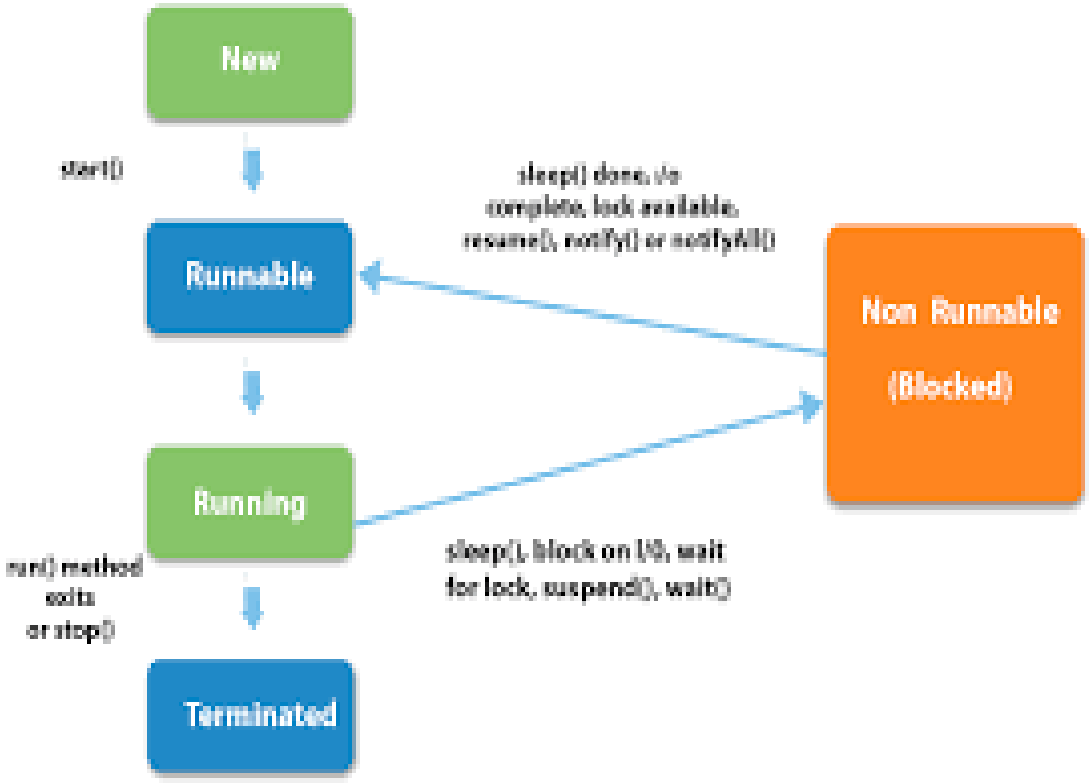
Example of Multi thread:

```
1.  package demotest;
2.
3.  public class GuruThread1 implements Runnable
4.  {
5.          public static void main(String[] args) {
6.           Thread guruThread1 = new Thread("Guru1");
7.           Thread guruThread2 = new Thread("Guru2");
8.           guruThread1.start();
9.           guruThread2.start();
10.          System.out.println("Thread names are following:");
11.          System.out.println(guruThread1.getName());
12.          System.out.println(guruThread2.getName());
13.     }
14.     @Override
15.     public void run() {
16.     }
17.
18. }
```

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times
- As such the threads are independent, the other threads won't get affected if one thread meets an exception.

Thread Life Cycle in Java



There are various stages of life cycle of thread as shown in above diagram:

1. New: In this phase, the thread is created using class "Thread class". It remains in this state till the program **starts** the thread. It is also known as born thread.
2. Runnable: In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3. Running: When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4. Non-runnable(waiting): This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5. Terminated(Dead): This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".

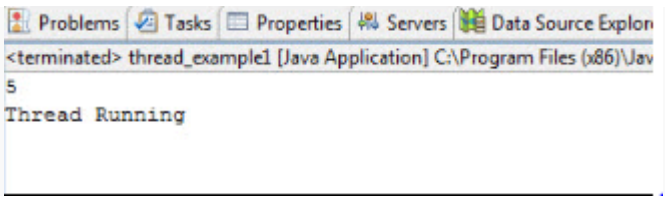
Some of the commonly used methods for threads are:

- start():This method starts the execution of the thread and JVM calls the run() method on the thread.
- Sleep(int milliseconds): This method makes the thread sleep hence the thread's execution will pause for milliseconds provided and after that, again the thread starts executing. This help in synchronization of the threads.
- getName():It returns the name of the thread.
- setPriority(int newpriority): It changes the priority of the thread.

Example code:

```
1.  package demotest;
2.  public class thread_example1 implements Runnable {
3.      @Override
4.      public void run() {
5.      }
6.      public static void main(String[] args) {
7.          Thread guruthread1 = new Thread();
8.          guruthread1.start();
9.          try {
10.             guruthread1.sleep(1000);
11.         } catch (InterruptedException e) {
12.             // TODO Auto-generated catch block
13.             e.printStackTrace();
14.         }
15.         guruthread1.setPriority(1);
16.         int gurupriority = guruthread1.getPriority();
17.         System.out.println(gurupriority);
18.         System.out.println("Thread Running");
19.   }
20. }
21.
```

Output :



Explanation of the code:

- **Code Line 2:** We are creating a class "thread_Example1" which is implementing the Runnable interface (it should be implemented by any class whose instances are intended to be executed by the thread.)
- **Code Line 4:** It overrides run method of the runnable interface as it is mandatory to override that method
- **Code Line 6:** Here we have defined the main method in which we will start the execution of the thread.
- **Code Line 7:** Here we are creating a new thread name as "guruthread1" by instantiating a new class of thread.
- **Code Line 8:** we will use "start" method of the thread using "guruthread1" instance. Here the thread will start executing.
- **Code Line 10:** Here we are using the "sleep" method of the thread using "guruthread1" instance. Hence, the thread will sleep for 1000 milliseconds.
- **Code 9-14:** Here we have put sleep method in try catch block as there is checked exception which occurs i.e. Interrupted exception.
- **Code Line 15:** Here we are setting the priority of the thread to 1 from whichever priority it was
- **Code Line 16:** Here we are getting the priority of the thread using getPriority()
- **Code Line 17:** Here we are printing the value fetched from getPriority
- **Code Line 18:** Here we are writing a text that thread is running.

Creating a Thread

There are two ways to create a thread:

1. **By extending Thread class**

1. By Extending Thread class

```
class Multi extends Thread                          // Extending thread class
{
public void run()                                   // run() method declared
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi t1=new Multi();                               //object initiated
t1.start();                                         // run() method called through start()
 }
}
```

Output: thread is running...

2. By implementing Runnable interface.

2. By implementing Runnable interface

```
class Multi3 implements Runnable                    // Implementing Runnable interface
{
public void run()
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();                             // object initiated for class
Thread t1 =new Thread(m1);                          // object initiated for thread
t1.start();
 } }
```

Output: thread is running...

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 int getPriority():** returns the priority of the thread.
4. **public String getName():** returns the name of the thread.
5. **public int getId():** returns the id of the thread.
6. **public void setName(String name):** changes the name of the thread.

Service Thread

- **Service thread are typically contain never ending loop for receiving and handling request.**
- **Such service threads can be convert to daemon threads.**
- **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.

Methods for Java Daemon thread by Thread class

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

```
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();
 }
}
```

Output:

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

Schedule a task for execution in Java

One of the methods in the Timer class is the void schedule(Timertask task, Date time) method. This method schedules the specified task for execution at the specified time. If the time is in the past, it schedules the task for immediate execution.

```
Method: public void schedule(Timertask task, Date time)
```

```
import java.util.*;
class MyTask extends TimerTask {
   public void run() {
      System.out.println("Task is running");
   }
}
public class Example {
   public static void main(String[] args) {
      Timer timer = new Timer(); // creating timer
      TimerTask task = new MyTask(); // creating timer task
      timer.schedule(task, new Date()); // scheduling the task
   }
   public void run() {
      System.out.println("Performing the given task");
   }
}
```

Output

```
Task is running
```

Class constructors

Timer()

This constructor creates a new timer.

Timer(String name)

This constructor creates a new timer whose associated thread has the specified name.

Timer(boolean isDaemon)

This constructor creates a new timer whose associated thread may be specified to run as a daemon.

Timer(String name, boolean isDaemon)

This constructor creates a new timer whose associated thread has the specified name, and may be specified to run as a daemon.

Class methods

Void cancel():

This method terminates this timer, discarding any currently scheduled tasks.

Void run():

This method will run the task class.

Void schedule(TimerTask task, Date time):

This method schedules the specified task for execution at the specified time.

Class declaration:

```
public abstract class TimerTask
   extends Object
   implements Runnable
```

Thread safe variables

As we know Java has a feature, Multithreading, which is a process of running multiple threads simultaneously. When multiple threads are working on the same data, and the value of our data is changing, that scenario is not thread-safe and we will get inconsistent results. When a thread is already working on an object and preventing another thread on working on the same object, this process is called Thread-Safety.

Java ThreadLocal

ThreadLocal in Java is another way to achieve thread-safety apart from writing immutable classes. Thread local can be considered as a scope of access like session scope or request scope. In thread local, you can set any object and this object will be local and global to the specific thread which is accessing this object.

Java ThreadLocal class provides thread-local variables. It enables you to create variables that can only be read and write by the same thread. If two threads are executing the same code and that code has a reference to a ThreadLocal variable then the two threads can't see the local variable of each other.

Creating a ThreadLocal variable

The variable of ThreadLocal can be created using the following code:

```
private ThreadLocal threadLocal = new ThreadLocal();
```

In the above code, you instantiate a new ThreadLocal object. This object is only need to be done only once per thread. Two threads can't see each other's ThreadLocal variables even if two threads set different values on same ThreadLocal object.

Accessing a ThreadLocal

Once a ThreadLocal has been created you can set the value to be stored in it like this:

```
threadLocal.set(183);
```

This ThreadLocal value can be read like this:

```
Integer threadLocalValue = (Integer) threadLocal.get();
```

ThreadLocal Example

```
public class ThreadLocalExp
{
    public static class MyRunnable implements Runnable
    {
        private ThreadLocal<Integer> threadLocal =
                new ThreadLocal<Integer>();
        @Override
        public void run() {
            threadLocal.set( (int) (Math.random() * 50D) );
            try
            {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.println(threadLocal.get());
```

```
        }
    }
    public static void main(String[] args)
    {
        MyRunnable runnableInstance = new MyRunnable();

        Thread t1 = new Thread(runnableInstance);
        Thread t2 = new Thread(runnableInstance);
        // this will call run() method
        t1.start();
        t2.start();
    }
}
```

Output:

16

32

Example 2:

```
package com.journaldev.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
```

```
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default
Formatter = "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //formatter pattern is changed here by thread, but it won't reflect to other threads
        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
    }
```

}

Output:

```
Thread Name= 0 default Formatter = yyyyMMdd HHmm
Thread Name= 1 default Formatter = yyyyMMdd HHmm
Thread Name= 0 formatter = M/d/yy h:mm a
Thread Name= 2 default Formatter = yyyyMMdd HHmm
Thread Name= 1 formatter = M/d/yy h:mm a
Thread Name= 3 default Formatter = yyyyMMdd HHmm
Thread Name= 4 default Formatter = yyyyMMdd HHmm
Thread Name= 4 formatter = M/d/yy h:mm a
Thread Name= 5 default Formatter = yyyyMMdd HHmm
Thread Name= 2 formatter = M/d/yy h:mm a
Thread Name= 3 formatter = M/d/yy h:mm a
Thread Name= 6 default Formatter = yyyyMMdd HHmm
Thread Name= 5 formatter = M/d/yy h:mm a
Thread Name= 6 formatter = M/d/yy h:mm a
Thread Name= 7 default Formatter = yyyyMMdd HHmm
Thread Name= 8 default Formatter = yyyyMMdd HHmm
Thread Name= 8 formatter = M/d/yy h:mm a
Thread Name= 7 formatter = M/d/yy h:mm a
Thread Name= 9 default Formatter = yyyyMMdd HHmm
Thread Name= 9 formatter = M/d/yy h:mm a
```

Java ThreadLocal class

Java ThreadLocal class provides thread-local variables. It enables you to create variables that can only be read and write by the same thread. If two threads are executing the same code and that code has a reference to a ThreadLocal variable then the two threads can't see the local variable of each other.

Java ThreadLocal methods

void set():This method sets the current thread's copy of this thread-local variable to the specified value.

Void remove(): This method removes the current threads value for this thread local variable.

Synchronizing Threads

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement −

Syntax:
```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Multithreading Example with Synchronization

```
1. //example of java synchronized method
2. class Table{
3.  synchronized void printTable(int n){//synchronized method
4.    for(int i=1;i<=5;i++){
5.      System.out.println(n*i);
6.      try{
7.       Thread.sleep(400);
8.      }catch(Exception e){System.out.println(e);}
9.    }
10.
```

```
 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
```

```
400
500
```

Why use Synchronization?

The synchronization is mainly used to:

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