**<https://javabeginnerstutorial.com/core-java-tutorial/transient-vs-static-variable-java/>**

## Use of serialVersionUID

***serialVersionUID*** is a version number associated to each serializable class by serialization runtime. This version number is used during deserialization process to verify that the sender and receiver of a serialized object have loaded class for that object which is compatible with respect to serialization.

* Defining a ***serialVersionUID***field in serializable class is **not mandatory**.
* If a serializable class have explicit ***serialVersionUID***then this field should be of type ***long****and must be static and final.*
* If there is no ***serialVersionUID***fielddefined explicitly then serialization runtime will calculate default value for that class. Which can vary based on compiler implementation. Hence it is advisable to define ***serialVersionUID.***
* It is advised to use private access modifier for ***serialVersionUID.***

**MultiThreading :**

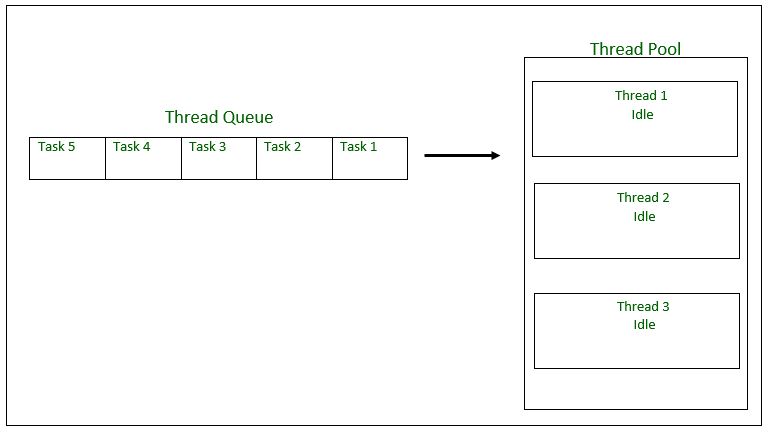
Thread Pools in Java

Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks.  A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

**What is ThreadPool in Java?**

**A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.** Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

* Interface–**ExecutorService** and the class-**ThreadPoolExecutor** is responsible for Thread pool mechanism.
* To use thread pools, we first create a object of **ExecutorService** and pass a set of tasks to it. **ThreadPoolExecutor** class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tpinit.jpg)

*Thread Pool Initialization with size = 3 threads. Task Queue = 5 Runnable Objects*

**Executor Thread Pool Methods**

**Method** **Description**

newFixedThreadPool(int) Creates a fixed size thread pool.

newCachedThreadPool() Creates a thread pool that creates new

threads as needed, but will reuse previously

constructed threads when they are available

newSingleThreadExecutor() Creates a single thread.

In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

**Steps to be followed**

1. Create a task(Runnable Object) to execute

2. Create Executor Pool using Executors

3. Pass tasks to Executor Pool

4. Shutdown the Executor Pool

**Task.java**

// Java program to illustrate

// ThreadPool

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

// Task class to be executed (Step 1)

class **Task** implements Runnable

{

    private String name;

    public Task(String s)

    {

        name = s;

    }

    // Prints task name and sleeps for 1s

    // This Whole process is repeated 5 times

    public void run()

    {

        try

        {

            for (int i = 0; i<=5; i++)

            {

                if (i==0)

                {

                    Date d = new Date();

                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");

                    System.out.println("Initialization Time for"

                            + " task name - "+ name +" = " +ft.format(d));

                    //prints the initialization time for every task

                }

                else

                {

                    Date d = new Date();

                    SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");

                    System.out.println("Executing Time for task name - "+

                            name +" = " +ft.format(d));

                    // prints the execution time for every task

                }

                Thread.sleep(1000);

            }

            System.out.println(name+" complete");

        }

        catch(InterruptedException e)

        {

            e.printStackTrace();

        }

    }

}

**Test.java**

public class Test

{

     // Maximum number of threads in thread pool

    static final int MAX\_T = 3;

    public static void main(String[] args)

    {

        // creates five tasks

        Runnable r1 = new Task("task 1");

        Runnable r2 = new Task("task 2");

        Runnable r3 = new Task("task 3");

        Runnable r4 = new Task("task 4");

        Runnable r5 = new Task("task 5");

        // creates a thread pool with MAX\_T no. of

        // threads as the fixed pool size(Step 2)

**ExecutorService pool = Executors.newFixedThreadPool(MAX\_T);**

        // passes the Task objects to the pool to execute (Step 3)

        pool.execute(r1);

        pool.execute(r2);

        pool.execute(r3);

        pool.execute(r4);

        pool.execute(r5);

        // pool shutdown ( Step 4)

        pool.shutdown();

    }

}

Initialization Time for task name - task 2 = 02:32:56

Initialization Time for task name - task 1 = 02:32:56

Initialization Time for task name - task 3 = 02:32:56

Executing Time for task name - task 1 = 02:32:57

Executing Time for task name - task 2 = 02:32:57

Executing Time for task name - task 3 = 02:32:57

Executing Time for task name - task 1 = 02:32:58

Executing Time for task name - task 2 = 02:32:58

Executing Time for task name - task 3 = 02:32:58

Executing Time for task name - task 1 = 02:32:59

Executing Time for task name - task 2 = 02:32:59

Executing Time for task name - task 3 = 02:32:59

Executing Time for task name - task 1 = 02:33:00

Executing Time for task name - task 3 = 02:33:00

Executing Time for task name - task 2 = 02:33:00

Executing Time for task name - task 2 = 02:33:01

Executing Time for task name - task 1 = 02:33:01

Executing Time for task name - task 3 = 02:33:01

task 2 complete

task 1 complete

task 3 complete

Initialization Time for task name - task 5 = 02:33:02

Initialization Time for task name - task 4 = 02:33:02

Executing Time for task name - task 4 = 02:33:03

Executing Time for task name - task 5 = 02:33:03

Executing Time for task name - task 5 = 02:33:04

Executing Time for task name - task 4 = 02:33:04

Executing Time for task name - task 4 = 02:33:05

Executing Time for task name - task 5 = 02:33:05

Executing Time for task name - task 5 = 02:33:06

Executing Time for task name - task 4 = 02:33:06

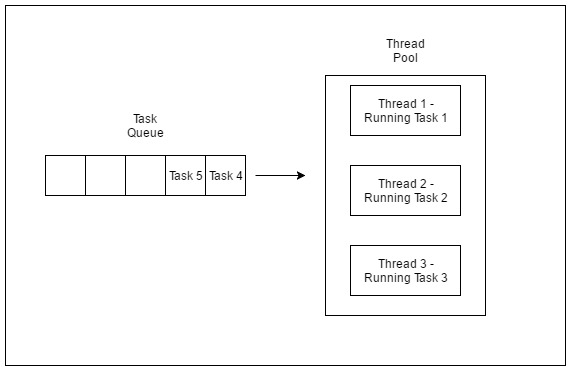
Executing Time for task name - task 5 = 02:33:07

Executing Time for task name - task 4 = 02:33:07

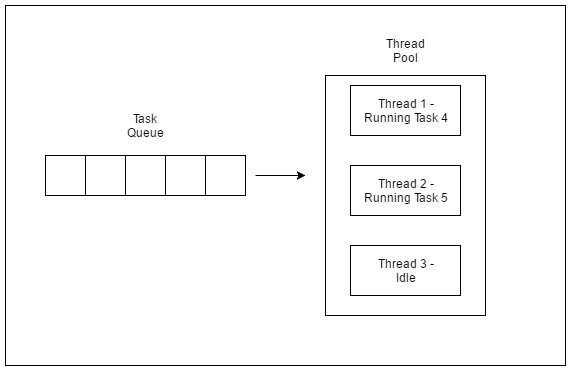
task 5 complete

task 4 complete

As seen in the execution of the program, the task 4 or task 5 are executed only when a thread in the pool becomes idle. Until then, the extra tasks are placed in a queue.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tprun1.jpg)

*Thread Pool executing first three tasks*

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tprun2.jpg)

**Volatile Keyword in java:**

**class** SharedObj

{

// Changes made to sharedVar in one thread

// may not immediately reflect in other thread

**static int** sharedVar = 6;

}

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVariable**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the [write policy](https://en.wikipedia.org/wiki/CPU_cache#Write_policies) of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

class SharedObj

{

// volatile keyword here makes sure that

// the changes made in one thread are

// immediately reflect in other thread

static **volatile** int sharedVar = 6;

}