**MultiThreading in java**

* **Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.
* Every thread defines a separate path of execution in java.
* A thread is explained in different ways.
* A thread is a lightweight process.
* A thread is a subpart of a process that can run individually.
* In java, multiple threads can run at a time, which enables the java to write multitasking programs. The multithreading is a specialized form of multitasking. All modern operating systems support multitasking.
* There are two types of multitasking, and they are as follows.
* Process-based multitasking
* Thread-based multitasking
* It is important to know the difference between process-based and thread-based multitasking. Let's distinguishboth.

Table

Description automatically generated

**Why Threads are used?**

* Now, we can understand why threads are being used as they had the advantage of being lightweight and can provide communication between multiple threads at a Low Cost contributing to effective multi-tasking within a shared memory environment.

# **Concurrency**

* Two task can operate concurrently and both make a progress
* Before multi-core processor

OS rapidly switch between different task, doing small portion of each task before moving to next so that all can be progress.

## Multithread

* Java makes concurrency available to the programmer using java.lang.Thread API’s
* Java programs can have multiple threads of execution
* Each thread has its own method stack and PC, allow it to use concurrency with other threads.

If Exception occurs, then only that thread will get terminated.

## Example of concurrency

* Stream video on the internet
  + Producer thread – download the video
  + Consumer thread – play the video

These activities perform simultaneously

* Threads are synchronized – coordinated.
  + To avoid choppy playback, the player thread does not being until there’s a sufficient amount of video available in memory.

**Life Cycle of Thread**

* There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State

2. Active State

3. Waiting/Blocked State

4. Timed Waiting State

5. Terminated State

Diagram

Description automatically generated

Diagram, schematic

Description automatically generated

**1. New**:In this state, a new thread begins its life cycle. This is also called a ***born thread***. The thread is in the new state if you create an instance of Thread class but before the invocation of the ***start()*** method.

**2. Runnable**:A thread becomes runnable after a newly born thread is started. In this state, a thread would be executing its task.

**3. Running**:When the thread scheduler selects the thread then, that thread would be in a running state.

**4. Non-Runnable (Blocked)**:The thread is still alive in this state, but currently, it is not eligible to run.

**5. Terminated**:A thread is terminated due to the following reasons:

* Either its ***run()*** method exists normally, i.e., the thread’s code has executed the program.
* Or due to some unusual errors like segmentation fault or an unhandled exception.

A thread that is in a terminated state does not consume ant cycle of the CPU.

**What is Main Thread?**

As we are familiar that, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, Similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution.

**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 void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. public void join(): waits for a thread to die.
5. public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
6. public int getPriority(): returns the priority of the thread.
7. public int setPriority(int priority): changes the priority of the thread.
8. public String getName(): returns the name of the thread.
9. public void setName(String name): changes the name of the thread.
10. public Thread currentThread(): returns the reference of currently executing thread.
11. public int getId(): returns the id of the thread.
12. public Thread.State getState(): returns the state of the thread.
13. public boolean isAlive(): tests if the thread is alive.
14. public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. public void suspend(): is used to suspend the thread(depricated).
16. public void resume(): is used to resume the suspended thread(depricated).
17. public void stop(): is used to stop the thread(depricated).
18. public boolean isDaemon(): tests if the thread is a daemon thread.
19. public void setDaemon(boolean b): marks the thread as daemon or user thread.
20. public void interrupt(): interrupts the thread.
21. public boolean isInterrupted(): tests if the thread has been interrupted.
22. public static boolean interrupted(): tests if the current thread has been interrupted.

Three ways to create thread  
There are three ways you can specify your code to be executed by a thread

1. By inheriting your class from the Thread class
2. By implementing the Runnable interface in your class
3. By using the [method reference](http://java-latte.blogspot.in/2014/03/method-references-in-java-8.html) to a method that takes no parameters and returns void

### **By inheriting your class from the Thread class**

When you inherit your class from the Thread class, you should override the run() method and provide the code to be executed by the thread.

**class** MyThread **extends** Thread{

@Override

**public** **void** run() {

System.***out***.println("Running thread..");

}

}

**public** **class** InheritsThreadClassDemo {

**public** **static** **void** main(String args[]) {

MyThread t=**new** MyThread();

t.start();

}

}

### **By implementing the Runnable interface in your class**

You can create a class that implements the java.lang.Runnable interface. Runnable is a [functional interface](http://java-latte.blogspot.in/2014/02/functional-interface-and-lambda-in-java.html) and it is declared as follows  
@[FunctionalInterface](http://java-latte.blogspot.in/2014/02/functional-interface-and-lambda-in-java.html)  
**public interface Runnable**

**public** **class** RunnableInterfaceDemo {

**public** **static** **void** main(String[] args) {

Runnable runDemo = () -> System.***out***.println("Thread Running"); Thread t=**new** Thread(runDemo);

t.start();

Runnable runnable = () -> {

**try** {

String name = Thread.*currentThread*().getName();

System.***out***.println("Foo " + name);

TimeUnit.***SECONDS***.sleep(1);

System.***out***.println("Bar " + name);

}

**catch** (InterruptedException e) {

e.printStackTrace();

}

};

Thread thread = **new** Thread(runnable);

thread.start();

}

}

Output :

Thread Running

Foo Thread-1

Bar Thread-1

When you run the above code you'll notice the one second delay between the first and the second print statement. TimeUnit is a useful enum for working with units of time. Alternatively you can achieve the same by calling Thread.sleep(1000).

### **By using the**[**method reference**](http://java-latte.blogspot.in/2014/03/method-references-in-java-8.html)**to a method that takes no parameters and returns void**

**class** ThreadTest{

**public** **static** **void** execute() {

System.***out***.println("Thread using method reference");

}

}

**public** **class** ThreadUsingMethodReferenceDemo {

**public** **static** **void** main(String args[]) {

Thread t=**new** Thread(ThreadTest :: *execute*);

t.start();

}

}

**If you don’t understand method reference example then Please see below example.**

Java 8 lambda expressions avoids creating objects for functional interface. If in the application if there is already some method in some class, which we feel, is a perfect implementation of functional method of functional interface, wouldn’t be nice if we could refer this existing methods instead of using a lambda expression? This is exactly what we can do using method references.



1. **Java Thread Example by extending Thread class**

**public** **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();

}

}

1. **Java Thread Example by implementing Runnable interface**

**public** **class** Multi1 **implements** Runnable {

**public** **void** run(){

System.***out***.println("thread is running...");

}

**public** **static** **void** main(String args[]){

Multi1 m1=**new** Multi1();

Thread t1 =**new** Thread(m1); // Using the constructor Thread(Runnable r)

t1.start();

}

}

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

1. **Using the Thread Class: Thread(String Name)**

We can directly use the Thread class to spawn new threads using the constructors defined above.

**public** **class** MyThread1 {

// Main method

**public** **static** **void** main(String argvs[])

{

// creating an object of the Thread class using the constructor Thread(String name)

Thread t= **new** Thread("My first thread");

// the start() method moves the thread to the active state

t.start();

// getting the thread name by invoking the getName() method

String str = t.getName();

System.***out***.println(str);

}

}

1. **Using the Thread Class: Thread(Runnable r, String name)**

**public** **class** MyThread2 **implements** Runnable {

**public** **void** run()

{

System.***out***.println("Now the thread is running ..."); }

// main method

**public** **static** **void** main(String argvs[])

{

// creating an object of the class MyThread2

Runnable r1 = **new** MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)

Thread th1 = **new** Thread(r1, "My new thread");

// the start() method moves the thread to the active state

th1.start();

// getting the thread name by invoking the getName() method

String str = th1.getName();

System.***out***.println(str);

}

}

**Java Thread Priority**

* In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence.
* In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority.
* The thread with more priority allocates the processor first.
* The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.
* The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

1. MAX\_PRIORITY - It has the value 10 and indicates highest priority.
2. NORM\_PRIORITY - It has the value 5 and indicates normal priority.
3. MIN\_PRIORITY - It has the value 1 and indicates lowest priority.

The default priority of any thread is 5 (i.e. NORM\_PRIORITY).

**setPriority( ) method**

* The setPriority( ) method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).
* The regular use of the setPriority( ) method is as follows.
* Example

threadObject.setPriority(4); OR

threadObject.setPriority(MAX\_PRIORITY);

**getPriority( ) method**

* The getPriority( ) method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.
* The regular use of the getPriority( ) method is as follows.
* Example

String threadName = threadObject.getPriority();

**Look at the following example program on priority**

**public** **class** SampleThread **extends** Thread{

**public** **void** run() {

System.***out***.println("Inside SampleThread");

System.***out***.println("Current Thread: " + Thread.*currentThread*().getName());

}

}

**public** **class** SampleThreadTest {

**public** **static** **void** main(String[] args) {

SampleThread threadObject1 = **new** SampleThread();

SampleThread threadObject2 = **new** SampleThread();

threadObject1.setName("first");

threadObject2.setName("second");

threadObject1.setPriority(4);

threadObject2.setPriority(Thread.***MAX\_PRIORITY***);

threadObject1.start();

threadObject2.start();

}

}

**Java Thread Synchronization**

* The java programming language supports multithreading. The problem of shared resources occurs when two or more threads get execute at the same time. In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.
* The synchronization is the process of allowing only one thread to access a shared resource at a time.
* In java, the synchronization is achieved using the following concepts.

1. Mutual Exclusion
2. Inter thread communication

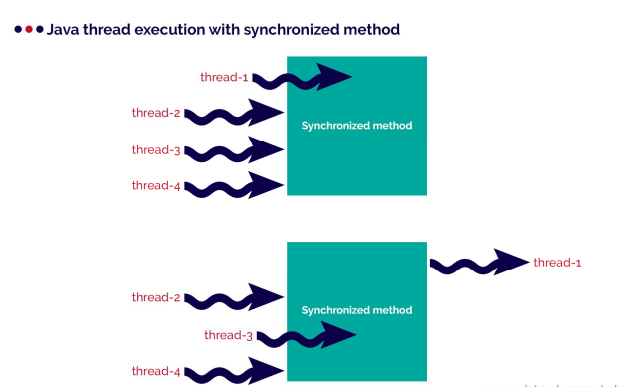
**Mutual Exclusion**

* Using the mutual exclusion process, we keep threads from interfering with one another while they accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

1. Synchronized method
2. Synchronized block

**Synchronized method**

When a method created using a synchronized keyword, it allows only one object to access it at a time. When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released. Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.



**Example of synchronized method:**

**public** **class** Table {

**synchronized** **void** printTable(**int** n) {

**for**(**int** i = 1; i <= 10; i++)

System.***out***.println(n + " \* " + i + " = " + i\*n);

}

}

**public** **class** Table1 **extends** Thread {

Table table = **new** Table();

**int** number;

Table1(Table table, **int** number){

**this**.table = table;

**this**.number = number;

}

**public** **void** run() {

table.printTable(number);

}

}

**public** **class** Table2 **extends** Thread {

Table table = **new** Table();

**int** number;

Table2(Table table, **int** number){

**this**.table = table;

**this**.number = number;

}

**public** **void** run() {

table.printTable(number);

}

}

**public** **class** Table\_ThreadSynchronizationExample {

**public** **static** **void** main(String[] args) {

Table table = **new** Table();

Table1 thread\_1 = **new** Table1(table, 5);

Table2 thread\_2 = **new** Table2(table, 10);

thread\_1.start();

thread\_2.start();

}

}

**Synchronized block**

* The synchronized block is used when we want to synchronize only a specific sequence of lines in a method. For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.
* The folllowing syntax is used to define a synchronized block.

**Syntax:**

synchronized(object){

... block code ...

}

**Example of synchronized block:**

**public** **class** SynchronizedBlock {

String name = "";

**public** **int** count = 0;

**public** **void** addName(String name, List<String> namesList){

**synchronized**(**this**){

**this**.name = name;

count++;

}

namesList.add(name);

}

**public** **int** getCount(){

**return** count;

}

}

**public** **class** SynchronizedBlockExample {

**public** **static** **void** main (String[] args)

{

SynchronizedBlock namesList\_1 = **new** SynchronizedBlock();

SynchronizedBlock namesList\_2 = **new** SynchronizedBlock();

List<String> list = **new** ArrayList<String>();

namesList\_1.addName("Rama", list);

namesList\_2.addName("Seetha", list);

System.***out***.println("Thread1: " + namesList\_1.name + ", " +

namesList\_1.getCount() + "\n");

System.***out***.println("Thread2: " + namesList\_2.name + ", " +

namesList\_2.getCount() + "\n");

}

}

**Java InterThread Communication**

* Inter thread communication is the concept where two or more threads communicate to solve the problem of polling. In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true. That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
* The inter-thread communication allows the synchronized threads to communicate with each other.
* Java provides the following methods to achieve inter thread communication.

1. wait( )
2. notify( )
3. notifyAll( )

* The following table gives detailed description about the above methods.

**void wait( )** :It makes the current thread to pause its execution until other thread in the same monitor calls notify( )

**void notify( )**: It wakes up the thread that called wait( ) on the same object.

**void notifyAll()**:It wakes up all the threads that called wait( ) on the same object

**Note: %n and \n difference**

There is also one specifier that doesn't correspond to an argument. It is "%n" which outputs a line break. A "\n" can also be used in some cases, but since "%n" always outputs the correct platform-specific line separator, it is portable across platforms whereas"\n" is not

## Producer consumer problem

**Producer-consumer without synchronization**

* In this example producer will write int values to shared buffer and consumer consumes it. At the end the sum of values produce and consume should be same.
* The sample implementation of producer and consumer problem is as follows.

**public** **class** ItemQueue {

**int** item;

**boolean** valueSet = **false**;

**synchronized** **int** getItem()

{

**while** (!valueSet)

**try**

{

wait();

} **catch** (InterruptedException e)

{

System.***out***.println("InterruptedException caught");

}

System.***out***.println("Consummed:" + item);

valueSet = **false**;

**try**

{

Thread.*sleep*(1000);

} **catch** (InterruptedException e)

{

System.***out***.println("InterruptedException caught");

}

notify();

**return** item;

}

**synchronized** **void** putItem(**int** item) {

**while** (valueSet)

**try**

{

wait();

} **catch** (InterruptedException e)

{

System.***out***.println("InterruptedException caught");

}

**this**.item = item;

valueSet = **true**;

System.***out***.println("Produced: " + item);

**try**

{

Thread.*sleep*(1000);

} **catch** (InterruptedException e)

{

System.***out***.println("InterruptedException caught");

}

notify();

}

}

**public** **class** Producer2 **implements** Runnable {

ItemQueue itemQueue;

Producer2(ItemQueue itemQueue)

{

**this**.itemQueue = itemQueue;

**new** Thread(**this**, "Producer").start();

}

**public** **void** run() {

**int** i = 0;

**while**(**true**)

{

itemQueue.putItem(i++);

}

}

}

**public** **class** Consumer2 **implements** Runnable{

ItemQueue itemQueue;

Consumer2(ItemQueue itemQueue)

{

**this**.itemQueue = itemQueue;

**new** Thread(**this**, "Consumer").start();

}

**public** **void** run() {

**while**(**true**)

{

itemQueue.getItem();

}

}

}

**public** **class** ProducerConsumer {

**public** **static** **void** main(String args[]) {

ItemQueue itemQueue = **new** ItemQueue();

**new** Producer2(itemQueue);

**new** Consumer2(itemQueue);

}}

**How Deadlock occurs?**

* Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.
* Example:

**public** **class** TestThread {

**public** **static** Object *Lock1* = **new** Object();

**public** **static** Object *Lock2* = **new** Object();

**public** **static** **void** main(String args[]) {

ThreadDemo1 T1 = **new** ThreadDemo1();

ThreadDemo2 T2 = **new** ThreadDemo2();

T1.start();

T2.start();

}

**private** **static** **class** ThreadDemo1 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock1*) {

System.***out***.println("Thread 1: Holding lock 1...");

**try** { Thread.*sleep*(10); }

**catch** (InterruptedException e) {}

System.***out***.println("Thread 1: Waiting for lock 2...");

**synchronized** (*Lock2*) {

System.***out***.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

**private** **static** **class** ThreadDemo2 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock2*) {

System.***out***.println("Thread 2: Holding lock 2...");

**try** { Thread.*sleep*(10); }

**catch** (InterruptedException e) {}

System.***out***.println("Thread 2: Waiting for lock 1...");

**synchronized** (*Lock1*) {

System.***out***.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program –

**Output**- Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

**Deadlock Solution Example**

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

**package** com.Multi.Threading;

**public** **class** TestThread\_Deadlock\_Solution {

**public** **static** Object *Lock1* = **new** Object();

**public** **static** Object *Lock2* = **new** Object();

**public** **static** **void** main(String args[]) {

ThreadDemo1 T1 = **new** ThreadDemo1();

ThreadDemo2 T2 = **new** ThreadDemo2();

T1.start();

T2.start();

}

**private** **static** **class** ThreadDemo1 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock1*) {

System.***out***.println("Thread 1: Holding lock 1...");

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {}

System.***out***.println("Thread 1: Waiting for lock 2...");

**synchronized** (*Lock2*) {

System.***out***.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

**private** **static** **class** ThreadDemo2 **extends** Thread {

**public** **void** run() {

**synchronized** (*Lock1*) {

System.***out***.println("Thread 2: Holding lock 1...");

**try** {

Thread.*sleep*(10);

} **catch** (InterruptedException e) {}

System.***out***.println("Thread 2: Waiting for lock 2...");

**synchronized** (*Lock2*) {

System.***out***.println("Thread 2: Holding lock 1 & 2...");

}}

}

}

}

So just changing the order of the locks prevent the program in going into a deadlock situation and completes with the following result –

**Output**- Thread 1: Holding lock 1...

Thread 1: Waiting for lock 2...

Thread 1: Holding lock 1 & 2...

Thread 2: Holding lock 1...

Thread 2: Waiting for lock 2...

Thread 2: Holding lock 1 & 2...

## Fork-Join

* The fork-join framework allows to break a certain task on several workers and then wait for the result to combine them. It leverages multi-processor machine's capacity to great extent.

**Fork**

* Fork is a process in which a task splits itself into smaller and independent sub-tasks which can be executed concurrently
* Syntax

Sum left = new Sum(array, low, mid);

left.fork();

Here Sum is a subclass of RecursiveTask and left.fork() spilts the task into sub-tasks.

**Join**

* Join is a process in which a task join all the results of sub-tasks once the subtasks have finished executing, otherwise it keeps waiting.
* Syntax

left.join();

Here left is an object of Sum class

**ForkJoinPool**

* It is a special thread pool designed to work with fork-and-join task splitting.
* Syntax

ForkJoinPool forkJoinPool = new ForkJoinPool(4);

Here a new ForkJoinPool with a parallelism level of 4 CPUs.

**RecursiveTask**

RecursiveTask represents a task which returns a value.

Syntax

class Sum extends RecursiveTask<Long> {

@Override

protected Long compute() { return null; }

}

**Example ForkJoin**

**public** **class** TestForkJoin {

**public** **static** **void** main(**final** String[] arguments) **throws** InterruptedException,

ExecutionException {

**int** nThreads = Runtime.*getRuntime*().availableProcessors();

System.***out***.println(nThreads);

**int**[] numbers = **new** **int**[10000];

**for**(**int** i = 0; i < numbers.length; i++) {

numbers[i] = i;

}

/\* Compute the sum of 20 for test purpose.

\* int sum=0;

for(int i = 0; i < 20; ++i)

sum += numbers[i];

System.out.println("Sum="+sum);\*/

ForkJoinPool forkJoinPool = **new** ForkJoinPool(nThreads);

Long result = forkJoinPool.invoke(**new** Sum(numbers,0,numbers.length));

System.***out***.println(result);

}

**static** **class** Sum **extends** RecursiveTask<Long> {

**int** low;

**int** high;

**int**[] array;

Sum(**int**[] array, **int** low, **int** high) {

**this**.array = array;

**this**.low = low;

**this**.high = high;

}

**protected** Long compute() {

**if**(high - low <= 10) {

**long** sum = 0;

**for**(**int** i = low; i < high; ++i)

sum += array[i];

**return** sum;

} **else** {

**int** mid = low + (high - low) / 2;

Sum left = **new** Sum(array, low, mid);

Sum right = **new** Sum(array, mid, high);

left.fork();

**long** rightResult = right.compute();

**long** leftResult = left.join();

**return** leftResult + rightResult;

}

}

}

}

**Interview Questions**

**1) Which collection classes are thread-safe in Java?**

* A **thread-safe** class is a class that guarantees the internal state of the class as well as returned values from methods, are correct while invoked concurrently from multiple threads. The collection classes that are **thread-safe** in Java are **Stack**, **Vector**, **Properties**, **Hashtable**, etc.

Stack

* The **Stack**class in Java implements the stack data structure that is based on the principle of **LIFO**. So, the **Stack**class can support many operations such as **push, pop, peek, search, empty**, etc.

**Example**

import java.util.\*;

public class StackTest {

   public static void main (String[] args) {

      Stack<Integer> stack = new Stack<Integer>();

      stack.push(5);

      stack.push(7);

      stack.push(9);

      Integer num1 = (Integer)stack.pop();

      System.out.println("The element popped is: " + num1);

      Integer num2 = (Integer)stack.peek();

      System.out.println(" The element on stack top is: " + num2);

   }

}

**Output**

The element popped is: 9

The element on stack top is: 7

**2) String is thread safe in Java**

* **String** is immutable ( once created cannot be changed )object . The object created as a **String** is stored in the Constant **String** Pool. Every immutable object in **Java** is **thread safe** , that implies **String** is also **thread safe** . **String** cannot be used by two **threads** simultaneously.

**StringBuffer**

* StringBuffer is mutable means one can change the value of the object . The object created through StringBuffer is stored in the heap. StringBuffer has the same methods as the StringBuilder , but each method in StringBuffer is synchronized that is StringBuffer is thread safe .
* Due to this it does not allow two threads to simultaneously access the same method . Each method can be accessed by one thread at a time .
* But being thread safe has disadvantages too as the performance of the StringBuffer hits due to thread safe property .
* Thus StringBuilder is faster than the StringBuffer when calling the same methods of each class.
* String Buffer can be converted to the string by using toString() method.
* StringBuffer demo1 = new StringBuffer("Hello") ; // object stored in heap and its value can be changed .
* demo1=new StringBuffer("Bye"); // statement is right as it modifies the value which is allowed in the StringBuffer

**StringBuilder**

* StringBuilder is same as the StringBuffer , that is it stores the object in heap and it can also be modified . The main difference between the StringBuffer and StringBuilder is that StringBuilder is also not thread safe. StringBuilder is fast as it is not thread safe .
* StringBuilder demo2= new StringBuilder("Hello"); // object too is stored in the heap and its value can be modified
* demo2=new StringBuilder("Bye"); // statement is right as it modifies the value which is allowed in the StringBuilder.

**3) What is volatile variable.**

* The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.
* In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs.

Diagram

Description automatically generated

With non-volatile variables there are no guarantees about when the Java Virtual Machine (JVM) reads data from main memory into CPU caches, or writes data from CPU caches to main memory. This can cause several problems which I will explain in the following sections.

Imagine a situation in which two or more threads have access to a shared object which contains a counter variable declared like this:

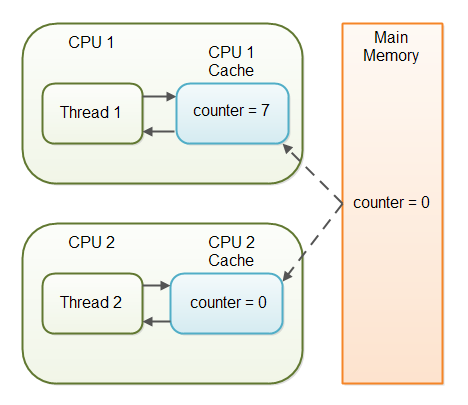
public class SharedObject {

public int counter = 0;

}

Imagine too, that only Thread 1 increments the counter variable, but both Thread 1 and Thread 2 may read the counter variable from time to time.

If the counter variable is not declared volatile there is no guarantee about when the value of the counter variable is written from the CPU cache back to main memory. This means, that the counter variable value in the CPU cache may not be the same as in main memory. This situation is illustrated here:



The problem with threads not seeing the latest value of a variable because it has not yet been written back to main memory by another thread, is called a "visibility" problem. The updates of one thread are not visible to other threads.

## The Java volatile Visibility Guarantee

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

Here is how the volatile declaration of the counter variable looks:

public class SharedObject {

public **volatile** int counter = 0;

}

Declaring a variable volatile thus *guarantees the visibility* for other threads of writes to that variable.

How Deadlock occurs?

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.

A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 2...");

try { Thread.sleep(10); }

catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 1...");

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}

**Output**

Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program by pressing CTRL+C.

## Deadlock Solution Example

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

public class TestThread {

public static Object Lock1 = new Object();

public static Object Lock2 = new Object();

public static void main(String args[]) {

ThreadDemo1 T1 = new ThreadDemo1();

ThreadDemo2 T2 = new ThreadDemo2();

T1.start();

T2.start();

}

private static class ThreadDemo1 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 1: Holding lock 1...");

try {

Thread.sleep(10);

} catch (InterruptedException e) {}

System.out.println("Thread 1: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 1: Holding lock 1 & 2...");

}

}

}

}

private static class ThreadDemo2 extends Thread {

public void run() {

synchronized (Lock1) {

System.out.println("Thread 2: Holding lock 1...");

try {

Thread.sleep(10);

} catch (InterruptedException e) {}

System.out.println("Thread 2: Waiting for lock 2...");

synchronized (Lock2) {

System.out.println("Thread 2: Holding lock 1 & 2...");

}

}

}

}

}