JAVA:

---

1. J2SE / JAVA SE

2. J2EE / JAVA EE

3. J2ME / JAVA ME

1. J2SE / JAVA SE:[Core Java]

--> Java 2 Standard Edition

--> It will cover only Fundemntals of JAVA.

--> Standalone Applications.

--> If we design and execute any application with out using Client-Server Arch then that application is called as Standalone Application.

--> If we design and execute any application with out distributing application logic over multiple machines then that application is called as "Standalone Application".

--> 5% of the applications are Standalone Applications.

--> It is a dependent PL for J2EE, Testing Tools, Salesforce, SAP, Hadoop,.....

--> Introduction, OOPs, Exception Handling, Multi Threadding, IOStreams, Collection Freamework,.....

2. J2EE / JAVA EE:

--> Java 2 Enterprise Edition

--> It will cover Server side Programming.

--> Distributed Applications / Enterprise Applications

--> If we design and execute any application on the basis of Client-Server Arch or by distributing application logic over Multiple machines then that application is called as Distributed Application or Enterprise Application.

--> 95% of the appl.

--> Servlets, Jsps, Jstl, EL, EJBs, Web Services,.....

3. J2ME / JAVA ME:

--> Java 2 Micro Edition

--> Micro programming

--> Mobile based Applications

--> If we design and execute any appl on the basis of Mobile H/W System then that application is called as Mobile Based Application.

--> less Demand.

Syllabus:

1.Multi Threadding

2.Networking

3.RMI

4.Arrays

5.Collection Framework

6.Generics

7.Internationalization[I18N]

8.JVM Arch

9.Garbage Collections

==========

Multi Threadding:

Q)What is the difference between Process, Procedure and Processor?

Ans:

Procedure is a set of isntructions representing a particular action.

Process is a flow of execution to perform a particular action by executing set of instructions[Procedure].

Processor is an H/W component to generate processes.

Initially, there are two models to execute applications.

1. Single Proess Mechanism / Single Tasking

2. Multi Process Mechanism / Multi Tasking

1. Single Proess Mechanism / Single Tasking:

--> This model allows only one process to execute the complete application.

--> It follows Sequerntial execution of the tasks.

--> It will increase application execution time.

--> It will reduce application performance.

2. Multi Process Mechanism / Multi Tasking:

--> This model allows more than one process at a time to execute the complete application.

--> It follows parallel execution of the tasks.

--> It will reduce application execution time.

--> It will increase application performance.

In Multi Process Mechanism, controlling is switched from one process context to another process context, so , this type of switching between process contexts is called as "Context Switching".

There are two types of Context Switchings.

1. Heavy Weight Context Switching

2. Light Weight COntext Switching

1. Heavy Weight Context Switching:

--> It is the context switching between two heavy weight components.

EX: Context Switching between two processes.

--> Heavy Weight Context switching will increase application execution time.

--> It will reduce application performance.

2. Light Weight COntext Switching:

--> It is the context switching between two light weight components.

EX: Context switching between two Threads.

--> Liht Weight Context switching will reduce application execution time.

--> It will increase application performance.

Q)What is the difference between Process and Thread?

Ans:

Process is flow of execution, it is heavy weight, it will take more execution time, it will reduce application performance.

Thread is flow of execution, it is light weight, it will take less execution time, it will increase application execution time.

Note: Now a days Majority of the technologies are designed on the basis of Threads.

There are two thread models to execute applications.

1. SIngle Thread Model / Single Tasking

2. Multi Thread Model / Multi Tasking

1. Single Thread Model / Single Tasking:

--> It allows only one thread to execute the comlete application.

--> It follows sequential execution of the applications.

--> It will increase application execution time.

--> It will reduce application performance.

2. Multi Thread Model / Multi Tasking

--> It allows more than one thread to execute the application application.

--> It follows parallel execution .

--> It will reduce application execution time.

--> It will increase application performance.

Note: Almost all the technologies using Multi Thread Model.

Java is Thread Based Programming Language, it is Multi Threadded Programming Language, it follows Multi Thread Model, it allows to create and execute more than one thread at a time to execute applications.

To prepare Threads, JAVA has provided predefined library in the form of java.lang package.

1. Thread

2. Runnable

Q)What is Thread and in how many ways we are able to create threads in Java applications?

Ans:

Thread is flow of execution to perform a particular task.

As per the predefined library provided by Java, there are two ways to create threads.

1. BY Extending java.lang.Thread class

2. By Implementing java.lang.Runnable interface

1. By Extending java.lang.Thread class:

```
class MyThread extends Thread{
        -----
}
```

2. By Implementing java.lang.Runnable interface:

```
class MyThread implements Runnable{
        ------
}
```

Threads Design:

---

1. Approach-1: By Extending Thread class
2. Approach-2: By Implementing Runnable Interface

1. Approach-1: By Extending Thread class:

---

a)Declare an User defined class.

b)Extend Uer defined class from java.lang.Thread class

c)Override Thread class run() method in user defined class.

Note: In run() method we must provide the application logic which we want to execute by creating new thread.

```
public void run()
```

Note: java.lang.Runnable interface contains only one method that is run().

d)In Main class, in main() method, Create thread and execute run() method provided implementation.

Note: To create Thread [Flow of execution] we have to use start() method from java.lang.Thread.

```
public void start()
```

Note: It will create new thread[Flow of execution] and it will send that thread to run() method to execute.

EX:

```
package com.durgasoft.app01.test;


class WelcomeThread extends Thread{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("WelcomeThread : "+i);
		}
	}
}


public class Test {
	public static void main(String[] args) {
		WelcomeThread welcomeThread = new WelcomeThread();
		welcomeThread.start();
		for(int i = 0; i < 10; i++) {
			System.out.println("Main Thread   : "+i);
		}
	}
}
```

Q) In Java applications, we have already first approach [Extending Thread class] to create threads then what is the requirement to go for Second approach [Implementing Runnable Interface] of creating Threads?

Ans:

In First Approach:

```
 class MyClass extends Thread {
 }
```

In GUI Appl.

```
class MyClass extends Frame{
}
```

In Java appl.

```
class MyClass extends Frame , Thread{
}
```

Single sub class is extended from more than one super class directly, ity represents Multiple Inhertitance, it is not supported in Java.

If we use first approach to create threads then we must extend java.lang.Thread class to user defined class, it is not possible to extend any other classes to user defined class.

To overcome the above problem, we have to use Second Approach to create threads.

IN Second approach of creating threads we will implement Runnable interface instead of extending Thread class.

```
class MyClass extends Frame implements Runnable{
        -----
}
```

2. Approach-2: By Implementing Runnable Interface

---

a)Declare an user defined class.

b)Implement java.lang.Runnable interface in User defined class.

c)Provide implementation for run() method in User defined class .

Note: Provide Application logic[Task] which we want to execute by creating new thread.

d)In Mian class and in main(), Create Thread and access run() method.

```
class MyThread implements Runnable{
	public void run(){
		-----Appl Logic----
	}
}

class Test{
	public static void main(String[] args){
		Case-1:
			MyThread mt = new MyThread();
			mt.start();
			Status: Compilation Error
		Case-2:
			MyThread mt = new MyThread();
			mt.run();
			Status: Here User Thread is not created, only Main Thread is executing MyThread class run() and main() method, here no Multi Threadding.
		Case-3:
			MyThread mt = new MyThread();
			Thread t = new Thread();
			t.start();
			Status: Here New Thread is created and it will be submitted to Thread class run() method, but not, MyThread class run() method.
		case-4:
			MyThread mt = new MyThread();
			Thread t = new Thread(mt);
			t.start();
			Status: Valid, Here new Thread is created and it will be submitted to MyThread class run() method.
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class WelcomeThread implements Runnable{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("Welcome Thread : "+i);
		}
	}
}
public class Test {
	public static void main(String[] args) {
		/*
		WelcomeThread welcomeThread = new WelcomeThread();
		welcomeThread.start();
		*/
		/*
		WelcomeThread welThread = new WelcomeThread();
		welThread.run();
		for(int i = 0; i < 10; i++) {
			System.out.println("Main Thread  : "+i);
		}
		*/
		/*
		WelcomeThread wt = new WelcomeThread();
		Thread t = new Thread();
		t.start();
		for(int i = 0; i < 10; i++) {
			System.out.println("Main Thread : "+i);
		}
		*/
		WelcomeThread wt = new WelcomeThread();
		Thread t = new Thread(wt);
```

```
                t.start();
                for(int i = 0; i < 10; i++) {
                        System.out.println("Main Thread  : "+i);
                }
        }
}
```

Thread Lifecycle:

It is the collective information of the threads right from its starting point to ending Point.

In Java applications, Threads are having the following states as part of its lifecycle.

1. New / Born State
2. Ready / Runnable State
3. Running State
4. Blocked State
5. Dead / Destroyed State

1. New / Born State:

--> In Java applications, when we create Thread class object automatically thread will come to "New or Born State".

2. Ready / Runnable State:

In Java applications, after creatying thread, if we want to execute thread , thread required memory and execution time , to get memory and execution time we have to access start() method.

In Java applications, after accessing start() and before allocating system resources to a thread , this state is called as "Ready / Runnable State".

3. Running State:

In Java applications, after accessing start() and after getting system resources like memory and execution time , this state is called as "Running State".

4. Blocked State:

To keep a running thread into Blocked state

a)Access sleep() method with a particular sleep time.

b)Access suspend() method over running thread.

c)Access wait() method over running thread.

d)When we perform IO Operations.

To getback thread from Blocked State to Ready or Runnable State

a)When sleep time is over.

b)If any other thread access resume() method.

c)If any other thread access notify() or notifyAll() methods.

d)When IO Operations are completed.

Note: If we want to keep a running thread to Ready / Runnable state directly then we have to use yield() method, it is not supported in Windows Operating system, because, it required priority based manipulations, Widows OS is not supporting Priority based Manipulations.

5. Dead / Destroyed State:

When we access stop() method over the running thread then that thread will come to Dead state.

Thread class Library:

Constructors:

---

1. public Thread()

--> It can be used to create Thread class object with the default values for thread name, thread priority value and thread group name.

Default Values for

1. Thread Name : Thread-0

2. Thread Priority Value : 5

3. Thread Group Name : main

EX:

```
package com.durgasoft.app01.test;

public class Test {
	public static void main(String[] args) {
		Thread t = new Thread();
		System.out.println(t);//Thread[Thread-0,5,main]
	}
}
```

2. public Thread(String name):

--> It can be used to create Thread class object with the specified name.

Note: With the above constructor we are able to provide a particular name to the thyread, but, the priority value of the thread will be 5[Default Priority Value] and Thread Group Name will be 'main'[Default Value].

EX:

```
package com.durgasoft.app01.test;

public class Test {
	public static void main(String[] args) {
		Thread t = new Thread("Core Java");
		System.out.println(t);//Thread[Core Java,5,main]
	}
}
```

3. public Thread (Runnable r):

--> It can be used to create Thread class object with the provided Runnable Reference.

Note: In this case no changes in default values of the Thread Properties.

Thread Name : Thread-0

Thread Priority Value : 5

Thread Group Name : main

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		Runnable r = new Thread();// Thread-0
		System.out.println(r);//Thread[Thread-0,5,main]
		Thread t = new Thread(r);// Thread-1
		System.out.println(t);//Thread[Thread-1,5,main]
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class HelloThread implements Runnable{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("HelloThread : "+i);
		}
	}
}
public class Test {
	public static void main(String[] args) {
		Runnable r = new HelloThread();
```

```
                Thread t = new Thread(r);
                t.start();
                for(int i = 0; i < 10; i++) {
                        System.out.println("MainThread  : "+i);
                }
        }
}
```

4. public Thread(Runnable r, String name)

--> It can be used to create thread class object with the provided Runnable reference and with the specified Thread Name.

Note: In this case, only Thread is changed and the remaining thread properties are having default values like Thread Priority Value is '5' and Thread Group Name is 'main'.

EX:

```
package com.durgasoft.app01.test;
public class Test {
        public static void main(String[] args) {
                Runnable r = new Thread();
                Thread t = new Thread(r,"Core Java");
                System.out.println(t);//Thread[Core Java,5,main]
        }
}
```

EX:

```
package com.durgasoft.app01.test;
class HelloThread implements Runnable{
        public void run() {
                for(int i = 0; i < 10; i++) {
                        System.out.println("HelloThread : "+i);
                }
```

```
	}
}
public class Test {
	public static void main(String[] args) {
		Runnable r = new HelloThread();
		Thread t = new Thread(r);
		t.start();
		for(int i = 0; i < 10; i++) {
			System.out.println("MainThread  : "+i);
		}
	}
}
```

5. public Thread(ThreadGroup tg, String name)

--> It can be used to create a Thread class object with the specified ThreadGroup Name and with the specified Thread Name.

Note: In Java applications, to provide a particular Thread Group Name we have to use java.lang.ThreadGroup class object.

```
	ThreadGroup threadGroup = new ThreadGroup("Group Name");
```

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		ThreadGroup threadGroup = new ThreadGroup("Java");
		Thread t = new Thread(threadGroup,"Core Java");
		System.out.println(t);//Thread[Core Java,5,Java]
	}
}
```

6. public Thread(ThreadGroup tg, Runnable r)

--> It can be used to create Thread class object withthe specified Thread Group Name and with the specified Runnable reference.

Note: In this case, only ThreadGroup Name will be changed and the remaining thread properties are having default values like Thread Priority Value is '5' and thread name is 'Thread-0'.

EX:

```
package com.durgasoft.app01.test;

public class Test {
	public static void main(String[] args) {
		Runnable r = new Thread();
		ThreadGroup threadGroup = new ThreadGroup("Java");
		Thread t = new Thread(threadGroup, r);
		System.out.println(t);//Thread[Thread-1,5,Java]
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class WelcomeThread implements Runnable{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("WelcomeThread : "+i);
		}
	}
}
public class Test {
	public static void main(String[] args) {
		Runnable r = new WelcomeThread();
		ThreadGroup threadGroup = new ThreadGroup("Java");
		Thread t = new Thread(threadGroup, r);
		t.start();
```

```
		for(int i = 0; i < 10; i++) {
			System.out.println("MainThread    : "+i);
		}
	}
}
```

7. public Thread(ThreadGroup tg, Runnable r, String name)

--> It can be used to create Thread class object with the specified ThreadGroup , Runnable reference and the provided Thread Name.

Note: In this case, Thread Group Name and Thread Name will be changed but Thread priority value will be the default value that is 5.

EX:

```
public class Test {
	public static void main(String[] args) {
		Runnable r = new Thread();
		ThreadGroup threadGroup = new ThreadGroup("Java");
		Thread t = new Thread(threadGroup,r,"Core Java");
		System.out.println(t);//Thread[Core Java,5,Java]
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class WelcomeThread implements Runnable{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("WelcomeThread : "+i);
		}
	}
}
```

```
public class Test {
	public static void main(String[] args) {
		Runnable r = new WelcomeThread();
		ThreadGroup threadGroup = new ThreadGroup("Java");
		Thread t = new Thread(threadGroup,r,"Core Java");
		t.start();
		for(int i = 0; i < 10; i++) {
			System.out.println("MianThread    : "+i);
		}
	}
}
```

Thread Class Metods:

---

1. public void setName(String name)

--> It can be used to provide a particular name to the Thread.

2. public String getName()

--> It can be used to get the present name of the Thread.

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		Thread t = new Thread();
		System.out.println(t.getName());//Thread-0
		t.setName("Core Java");
		System.out.println(t.getName());//Core Java
	}
}
```

3. public void setPriority(int priorityValue)

--> It can be used to set a particular Priority Value to the Thread.

Note: In Java applications, every thread will have a particular priority value and that priority values must be from 1 to 10. In the case of setPriority() method we must provide the priorityValue inbetween 1 and 10, if we provide priorityValu in out side of 1 to 10 range then JVM will raise an Exception like "java.lang.IllegalArgumentException".

In Java , to represent Thread Priority values , java.lang.Thread class has provided the following constants.

```
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIOITY = 5;
public static final int MAX_PRIORITY = 10;
```

4. public int getPriority()

--> It can be used to get the current priority value of the Thread.

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		Thread t = new Thread();
		System.out.println(t.getPriority());//5
		t.setPriority(3);
		System.out.println(t.getPriority());//3
		t.setPriority(Thread.MAX_PRIORITY-3);
		System.out.println(t.getPriority());//7
		t.setPriority(Thread.MIN_PRIORITY+Thread.NORM_PRIORITY);
		System.out.println(t.getPriority());//6

	}
}
```

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		Thread t = new Thread();
		System.out.println(t.getPriority());
		//t.setPriority(15);-->IllegalArgumentException
		//t.setPriority(-10);->IllegalArgumentException
		//t.setPriority(0);-->IllegalArgumentException
		System.out.println(t.getPriority());
	}
}
```

5. public static int activeCount()

--> It can be used to get the no of threads which are in active at present in our Java application.

Note: In Java applications, if we create a thread class object then the created thread is not active, when we access sdtart() method then only Thread will come active state.

EX:

```
package com.durgasoft.app01.test;
public class Test {
	public static void main(String[] args) {
		System.out.println(Thread.activeCount());//1
		Thread t1 = new Thread();
		t1.start();
		System.out.println(Thread.activeCount());//<=2
		Thread t2 = new Thread();
		t2.start();
		System.out.println(Thread.activeCount());//<=3
```

```
                Thread t3 = new Thread();
                t3.start();
                System.out.println(Thread.activeCount());//<=4
        }
}
```

6. public boolean isAlive()

--> It can be used to check whether a thared is in live[Active] or not. If the thread is in active state then isAlive() method will return 'true' value, if the thread is not in active then isAlive() method will return 'false' value.

Note: In Java applications, when we create Thread class object then the generated Thread is not in active, when we access start() method over the thread reference then only the generated thread will be in active.

EX:

```
package com.durgasoft.app01.test;
public class Test {
        public static void main(String[] args) {
                Thread t = new Thread();
                System.out.println(t.isAlive());
                t.start();
                for(int i = 0; i < 100; i++) {
                        System.out.println(t.isAlive()+" ");
                }
        }
}
```

7. public static Thread currentThread()

--> It can be used to get a Thread object which is executing the present instruction in Java applications.

EX:

```
package com.durgasoft.app01.test;
```

```
class A{
	void m1() {
		for(int i = 0; i < 10; i++) {
			System.out.println(Thread.currentThread().getName());
		}
	}
}
class Thread1 extends Thread{
	A a;
	Thread1(A a){
		this.a = a;
	}
	public void run() {
		a.m1();
	}
}
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		a.m1();
	}
}
class Thread3 extends Thread{
	A a;
	Thread3(A a){
		this.a = a;
	}
	public void run() {
		a.m1();
```

```
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();

		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);
		Thread3 t3 = new Thread3(a);

		t1.setName("AAA");
		t2.setName("BBB");
		t3.setName("CCC");

		t1.start();
		t2.start();
		t3.start();
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class MyThread extends Thread{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println(Thread.currentThread().getName());
		}
	}
}
public class Test {
	public static void main(String[] args) {
```

```
                MyThread mt1 = new MyThread();
                MyThread mt2 = new MyThread();
                MyThread mt3 = new MyThread();

                mt1.setName("AAA");
                mt2.setName("BBB");
                mt3.setName("CCC");

                mt1.start();
                mt2.start();
                mt3.start();
        }
}
```

8. public static void sleep(ling time)throws InterruptedException

--> It can be used to keep a running thread into sleep state untill the specified sleep time, if the sleep time is completed then the sleeping thread will come to active state automatically and continous its execution time.

EX:

```
package com.durgasoft.app01.test;

class WelcomeThread extends Thread{
        public void run() {
                for(int i = 0;i < 10; i++) {
                        try {
                                Thread.sleep(1000);
                                System.out.println("Welcome To Durgasoft");
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

```
public class Test {
	public static void main(String[] args) {
		WelcomeThread wt = new WelcomeThread();
		wt.start();
	}
}
```

9. public void join()throws InterruptedException

--> This method can be used to pass present [Main Thread] to Complete another thread, after completion of other thread automatically thepased thread will continue its execution part.

EX:

```
package com.durgasoft.app01.test;
class WelcomeThread extends Thread{
	public void run() {
		for(int i = 0; i < 10; i++) {
			System.out.println("WelcomeThread : "+i);
		}
	}
}
public class Test {
	public static void main(String[] args)throws Exception {
		WelcomeThread wt = new WelcomeThread();
		wt.start();
		wt.join();
		for(int i = 0; i < 10; i++) {
			System.out.println("MainThread    : "+i);
		}
	}
}
```

Daemon Threads:

Daemon Thread is a thread running internally and providing services to some other thread, Damon threads will be terminated automatically when the threads which are taking services from daemon threads are terminated.

EX: Garbage Collector inside JVM is Daemon Thread.

When we start JVM to execute a particular Java Application, internally, Garbage Collector will be executed and it is providing Garbage Collection service to JVM, where Garbage Collector thread will be terminated automatically when JVM thread is terminated.

In Java applications, to make a thread as daemon thread we must use the following method.

public void setDaemon(boolean b)

--> if b value is true then the thread will be

daemon thread.

--> If b value is false then the thread will not be

daemon thread.

Note: If we want to make a thread as daemon thread then we have to access setDamon(true) method before accessing start() method, because, once if we start a thread then it will not allow to change its state, if we access setDamon(true) method after acessing start() method then JVM will raise an exception like "java.lang.IllegalThreadStateException".

To check whether a thread is daemon thread or not we have to use the following method .

public boolean isDaemon()

EX:

```
package com.durgasoft.app01.test;
```

```
class GarbageCollector extends Thread{
	public void run() {
		while(true) {
			System.out.println("Garbage Collector Thread");
		}
	}
}
public class Test {
	public static void main(String[] args){
		GarbageCollector gc = new GarbageCollector();

		gc.start();
		gc.setDaemon(true);
		for(int i = 0; i < 10; i++) {
			System.out.println("JVM Thread");
		}
		System.out.println(gc.isDaemon());
	}
}
```

Concurrent Threads:

In Multi Threadding, we may create more than one thread and we may execute more than one thread at a time.

If we execute more than one thread on single data item or on single program then that threads are called as "Concurrent Threads" and that process is called as "Threads Concurrency".

In the case of Threads concurrncy there may be chances are available to get Data Inconsistency, it will provide wrong results in Java applications.

To overcome the above problems we need "Threadsafe" resources.

If any resource allows more than one thread at a time with out providing data inconsistency then that resource is called as "Threadsafe Resource".

If we want to make a resource as threadsafe resource then we have to use the following approaches.

1. Use Local Variables Over Instance Variable.

2. Use Immutable Objects Over Mutable Objects.

3. Use Synchronization.

---

1. Use Local Variables Over Instance Variable:

In Java applications, if we declare any variable inside a method then that variable is called as Local Variable.

In Java applications, Local variables data will be stored in Stack memory.

If we create multiple threads in java applications then a seperate stack will be created in stack memory for each and every thread.

If any thread access the local variable of a method then the respective local variable value will be stored in the threads respective stack only.

If multiple threads access the same local variable then multiple copies of the respective local variable will be stored in threads respective stacks.

If any thread performs modification on local variable then that modification is available upto the threads respective stack only, one thread modifification is not available to other threads.

If we declare any non static variable at class level then that variable is called as an instance variable.

In java applications, instance variables data will be stored inside the objects in heap memory.

In general, Heap memory objects are shared objects for multiple threads, all the threads can access data, it may provide data inconsistency.

EX:

```
package com.durgasoft.app01.test;
class A{
	int i = 10;
	void increment() {
		i = i + 10;
		System.out.println(Thread.currentThread()+" : "+i);
	}
}
class Thread1 extends Thread{
	A a;
	Thread1(A a){
		this.a = a;
	}
	public void run() {
		a.increment();
	}
}
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
```

```
                a.increment();
        }
}
public class Test {
        public static void main(String[] args){
                A a = new A();
                Thread1 t1 = new Thread1(a);
                Thread2 t2 = new Thread2(a);
                t1.start();
                t2.start();
        }
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
        void increment() {
                int i = 10;
                i = i + 10;
                System.out.println(Thread.currentThread().getName()+" : "+i);
        }
}
class Thread1 extends Thread{
        A a;
        Thread1(A a){
                this.a = a;
        }
        public void run() {
                a.increment();
        }
}
```

```
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		a.increment();
	}
}
public class Test {
	public static void main(String[] args){
		A a = new A();
		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);

		t1.setName("Thread-1");
		t2.setName("Thread-2");
		t1.start();
		t2.start();
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
	int i = 10;
	void increment() {
		int j = 10;

		i = i + 10;
		j = j + 10;
```

```
                System.out.println(Thread.currentThread().getName()+" : Instance
Variable : "+i);
                System.out.println(Thread.currentThread().getName()+" : Local
Variable    : "+j);


        }
}
class Thread1 extends Thread{
        A a;
        Thread1(A a){
                this.a = a;
        }
        public void run() {
                a.increment();
        }
}
class Thread2 extends Thread{
        A a;
        Thread2(A a){
                this.a = a;
        }
        public void run() {
                a.increment();
        }
}
public class Test {
        public static void main(String[] args){
                A a = new A();


                Thread1 t1 = new Thread1(a);
                Thread2 t2 = new Thread2(a);
```

```
		t1.setName("Thread-1");
		t2.setName("Thread-2");

		t1.start();
		t2.start();
	}
}
```

2. Use Immutable Objects Over Mutable Objects:

Mutable Object is a Java object, it allows modifications on its content directly.

EX: StringBuffer

If we submit multiple threads on single SingBuffer object to perform modifications then all the threads modifications are allowed in the same object, it will override one thread modifications with another thread modifications, it will provide data inconsistency, it will not make the respective resource as Threadsafe.

EX:

```
package com.durgasoft.app01.test;

class A{
	StringBuffer sb = new StringBuffer("Durga");
	void add() {
		sb = sb.append("soft");
		System.out.println(Thread.currentThread()+" : "+sb);
	}
}

class Thread1 extends Thread{
	A a;
	Thread1(A a){
```

```
		this.a = a;
	}
	public void run() {
		a.add();
	}
}

class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		a.add();
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();
		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);
		t1.start();
		t2.start();
	}
}
```

Immutable Objects are normal Java objects, they will not allow modifications on their content, if we are trying to perform modifications on its content then data is allowed for modifications but the resultent modified data will not be stored back in original object , where the resultent modified data will be stored by creating new Object.

EX: String class objects are Immutable objects.

All Wrapper Classes Objects are Immutable.

In Java applications, if we submit more than one thread on single String object [Immutable Object] and if more than one thread is performing modifications on single immutable object content, then, at each and every modification a seperate new String object will be created , where new modified data will be stored, old Object data is remains same , this approach will not allow overriding one thread modifications with another thread modifications, this approach will provide data consistency, it will make the resource as Threadsafe resource.

EX:

```
package com.durgasoft.app01.test;


class A{
	String str1 = new String("Durga");


	void add() {
		String str2 = str1.concat("soft");
		System.out.println(Thread.currentThread().getName()+" : "+str2);
	}
}
class Thread1 extends Thread{
	A a;
	Thread1(A a){
		this.a = a;
	}
	public void run() {
		a.add();
	}
}
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
```

```
	}
	public void run() {
		a.add();
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();
		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);

		t1.setName("Thread1");
		t2.setName("Thread2");

		t1.start();
		t2.start();
	}
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
	String str1 = new String("Durga");
	StringBuffer sb1 = new StringBuffer("Durga");

	void modify() {
		String str2 = str1.concat("soft");
		StringBuffer sb2 = sb1.append("soft");
		System.out.println("str2 : "+Thread.currentThread().getName()+" :
"+str2);
```

```
                System.out.println("sb   : "+Thread.currentThread().getName()+" :
"+sb2);
        }
}
class Thread1 extends Thread{
        A a;
        Thread1(A a){
                this.a = a;
        }
        public void run() {
                a.modify();
        }
}
class Thread2 extends Thread{
        A a;
        Thread2(A a){
                this.a = a;
        }
        public void run() {
                a.modify();
        }
}
public class Test {
        public static void main(String[] args) {
                A a = new A();
                Thread1 t1 = new Thread1(a);
                Thread2 t2 = new Thread2(a);
                t1.setName("Thread-1");
                t2.setName("Thread-2");
                t1.start();
                t2.start();
        }
```

}

3. Use Synchronization:

Synchronization is a mechanism, it able to allow only one thread at a time, it unable to allow more than one thread at a time, it will allow other threads after completing the present thread execution.

In general, in Java applications, when we execute more than one thread on single data item , when we perform modifications on single data item more than one thread and when one thread modifications are overridden with another thread modifications there we will get data inconsistency, but, synchronization is allowing only one thread, there is no scope to override one thread modification with another thread modifications , there is no scope to get data inconsistency, so synchronization is always providing data consistency, bydefault, all synchronized resources are Threadsafe.

Note: Bydefault, all synchronized resources are Threadsafe, but, all Threadsafe resources need not be synchronized resources, because, we are able to make resource as Threadsafe resource in multiple ways like by declaring local variables over instance variables and by using immutable objects over mutable Objects,.....

In Java applications, we are able to achieve synchronization by using an access modifier in the form of "synchronized".

In Java applications, synchronization is goingon on the basis of Locking Mechanisms.

When we submit more than one thread to the synchronized area, where Lock Manager will assign lock to a particuler thread, here which thread aquire lock from Lock Manager that thread is eligible to execute synchronized area, when a thread execution is completed in synchronized area then that thread has to submit lock back to Lock manager, when Lock Manager gets Lock from a thread from synchronized area then Lock Manager will assign that lock to another thread which is in waiting state. If any thread is executing synchronized area then Lock Manager will assign Lock to other threads.

In Java applications, we are able to achieve synchronization in the following two ways.

1. Synchronized Methods

2. Synchronized Blocks

1. Synchronized Methods:

Synchronized Method is a normal java method with 'synchronized' keyword, it allows only one thread at a time to execute method body, it will not allow more than one thread at a time to execute method body, after completing the execution of the present thread only other threads are allowed.

```
synchronized void m1(){
        ------
        -------
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
        synchronized void display() {
                for(int i = 0; i < 10; i++) {
                        System.out.println(Thread.currentThread().getName());
                }
        }
}
class Thread1 extends Thread{
        A a;
        Thread1(A a){
                this.a = a;
        }
        public void run() {
                a.display();
        }
}
```

```
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		a.display();
	}
}
class Thread3 extends Thread{
	A a;
	Thread3(A a){
		this.a = a;
	}
	public void run() {
		a.display();
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();

		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);
		Thread3 t3 = new Thread3(a);

		t1.setName("AAA");
		t2.setName("BBB");
		t3.setName("CCC");

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

```
                t3.start();

        }
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
        synchronized void display() {
                for(int i = 0; i < 10; i++) {
                        System.out.println(Thread.currentThread().getName());
                }
        }
}
class DisplayThread extends Thread{
        A a;
        public DisplayThread(A a) {
                this.a = a;
        }
        public void run() {
                a.display();
        }
}
public class Test {
        public static void main(String[] args) {
                A a = new A();
                DisplayThread thread1 = new DisplayThread(a);
                DisplayThread thread2 = new DisplayThread(a);
                DisplayThread thread3 = new DisplayThread(a);

                thread1.setName("Thread-1");
```

```
                thread2.setName("Thread-2");
                thread3.setName("Thread-3");

                thread1.start();
                thread2.start();
                thread3.start();

        }
}
```

In Java applications, synchronization allows only one thread at a time, it follows sequential execution of the threads, it will increase application execution time, it will reduce application performance.

In Java applications, dont use synchronization unneccessarily, if the requirement is existed then onle use synchronization otherwise it is not suggestible to use synchronize because it will reduce application performance.

Q)In Java applications, to achieve synchronization we have already synchronized methods approach then what is the requirement to use Synchronized blocks?

Ans:

In Java applications, to achieve synchronization if we use synchronized method then synchronization will be provided through out the method irrespective of the actual requirment. If we have requirement of synchronization upto some of the instructions inside a method and if we use synchronized method then synchronization will applied through out the method unndeccerily, it will reduce application performance.

In the above context, if we want to improve application performance, we have to provide synchronization upto the required no of instructions not through out the method, if we want to provide synchrnozation upto the required no of instructions we have to use "Synchronized Blocks".

2. Synchronized Blocks:

It is a set of instructions, which are able to allow only one thread at a time, it will not allow more than one thread at a time, after completion of the present thread execution only other threads are allowed.

Syntax:

```
synchronized(ObjectToLock){
        -------
        -------
        -------
}
```

EX:

```
package com.durgasoft.app01.test;
class A{
        void display() {
                System.out.println("Before Synchronized Block :
"+Thread.currentThread().getName());
                synchronized (this) {
                        for(int i = 0; i < 10; i++) {
                                System.out.println("Inside Synchronized Block :
"+Thread.currentThread().getName());
                        }
                }

        }
}
class Thread1 extends Thread{
        A a;
        Thread1(A a){
                this.a = a;
```

```
	}
	public void run() {
		a.display();
	}
}
class Thread2 extends Thread{
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		a.display();
	}
}
class Thread3 extends Thread{
	A a;
	Thread3(A a){
		this.a = a;
	}
	public void run() {
		a.display();
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();
		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);
		Thread3 t3 = new Thread3(a);

		t1.setName("AAA");
		t2.setName("BBB");
```

```
                t3.setName("CCC");


                t1.start();
                t2.start();
                t3.start();
        }
}
```

Inter Thread Communication:

--> The process of providing communication between more than one thread inorder to complete a particular task or a particular job is called as Inter Thread Communication.

--> To provide Inter Thread communication we have to use the following methods from java.lang.Object class.

a)wait(): To keep a running thread in waiting state.

b)notify(): To give activation state to a particular waiting thread

c)notifyAll(): To give Notification to all the threads which are in waiting state.

--> To use the above methods in java applications we have to provide Synchronization, because, the above methods are able to perform their functionalities under synchronization only.

--> In general, Inter Thread COmmunication is Much usefull while providing solutions to the problems like "Producer-Consumer Problems".

--> In Producer-Consumer problem, both producer and consumer are two threads, where Producer has to produce an item then consumer has to consume that item, it has to be performed upto infinite no of times.

Logic:

boolean flag : If its value is true then Producer is Producing an Item

and Consumer must be in waiting state.

If its value is false then Consumer is consuming an item and Producer must be in waiting state.

int count : It will manage Item Count.

If flag == true:

Producer:

Produce an Item

giving tern to Consumer [flag = false].

Giving notification to Consumer

Going to waiting state.

Consumer :

waiting state

If flag == false:

Producer:

waiting state

Consumer :

Consume the item

Giving tern to Producer [flag = true]

Giving Notification to Producer

Going to waiting state

```
class A{
    boolean flag = true;
    int count = 0;

    public synchronized void produce(){
        try{
```

```
                while(true){
                        if(flag == true){
                                count = count + 1;
                                sopln("Producer Produced Item - "+count);
                                flag = false;
                                notify();
                                wait();
                        }else{
                                wait();
                        }
                }
        }catch(Exception e){
                e.printStackTrace();
        }
    }
    public synchronized void consume(){
        try{
                while(true){
                        if(flag == true){
                                wait();
                        }else{
                                Sopln("Consumer Consumed Item - "+count);
                                flag = true;
                                notify();
                                wait();
                        }
                }
        }catch(Exception e){
                e.printStackTrace();
        }
    }
}
```

Deadlocks:

--> Deadlock is a situation where more than one thread is depending on each other in circular dependency.

--> In Java applications, when we get Deadlock , JVM will struck its execution over the program.

--> In Java applications, once if we get Deadloack then it is not possible to come back from Deadlock, in Java applications, we have to prevent deack lock situations, because, we unable to have any recovery mechanism from Deadlock, we are able to use prevention mechanisms.

EX: RegisterCourse: courseName, trainerName

CancelCourse : courseName, trainerName

Level1: RegisterCourse Thread holds courseName

CancelCourse Thread Holds trainerName

Level2: RegisterCourse Thread trying to get trainerName resource which was already locked by CancelCourse Thread.

CancelCourse Thread trying to get courseName resource which was already locked by RegisterCourse Thread.

```
package com.durgasoft.app01.test;
class RegisterCourse extends Thread{
        Object courseName;
        Object trainerName;
        RegisterCourse(Object courseName, Object trainerName){
                this.courseName = courseName;
                this.trainerName = trainerName;
```

```
	}
	public void run() {
		synchronized (courseName) {
			System.out.println("RegisterCourse Thread Holds courseName Rsource and waiting for trainerName");
			synchronized (trainerName) {
				System.out.println("RegisterCourse Thread Holds both courseName and trainerName , so that, Course Reistration is Success");
			}
		}
	}
}
class CancelCourse extends Thread{
	Object courseName;
	Object trainerName;
	CancelCourse(Object courseName, Object trainerName){
		this.courseName = courseName;
		this.trainerName = trainerName;
	}
	public void run() {
		synchronized (trainerName) {
			System.out.println("CancelCourse Thread Holds trainerName and waiting for courseName resource");
			synchronized (courseName) {
				System.out.println("CancelCourse Thread Holds both trainerName and courseName, so that, Course Cancellation is Success");
			}
		}
	}
}
public class Test {
	public static void main(String[] args) {
		Object courseName = new Object();
```

```
                Object trainerName = new Object();

                RegisterCourse rc = new RegisterCourse(courseName, trainerName);
                CancelCourse cc = new CancelCourse(courseName, trainerName);

                rc.start();
                cc.start();
        }
}
```

ThreadLocal:

In general, in Java applications, there are four types of Scopes for the data.

1. private
2. <default>
3. protected
4. public

Where 'private' scope is able to make available data upto the present class.

Where '<default>' scope is able to make available data upto the present package.

Where 'protected' scope is able to make available data upto the present package and the chaild class in other packages.

Where 'public' scope is able to make available data through out the application.

Similarily, i want make available data upto all the resources which are accessed by a particular thread, for this, we have to define a seperate scope for the data that is 'ThreadScope'.

In Java applications, if we want to define 'ThreaScope' for the Data Java has provided a predefined class in the form of 'java.lang.ThreadLocal' class.

Where 'ThreadLocal' class has provide the following methods to keep data in ThreadSCope, To get data from ThreadScope, to remove data from ThreadScope and to provide default data in ThreadScope.

1. public void set(Object data): To set data in ThreaScope.

2. public Object get(): To get data from ThreadScope.

3. public void remove(): To remove Data from ThreadScope.

4. public void initialValue(): To provide initial value in ThreadSCope , it will be executed to send value when we access data from ThreadScope with out setting data.

```
package com.durgasoft.app01.test;

class ThreadScope extends ThreadLocal<String>{

	@Override

	protected String initialValue() {

		return "Data is not defined in the Scope";

	}

}

class A{

	void m1() {

		System.out.println("m1() : Thread-1 Scope :
"+Thread1.threadScope.get());

		System.out.println("m1() : Thread-2 Scope :
"+Thread2.threadScope.get());

	}

	void m2() {

		System.out.println("m2() : Thread-2 Scope :
"+Thread2.threadScope.get());

		System.out.println("m2() : Thread-1 Scope :
"+Thread1.threadScope.get());

	}

}
```

```
class Thread1 extends Thread{
	static ThreadScope threadScope = new ThreadScope();
	A a;
	Thread1(A a){
		this.a = a;
	}
	public void run() {
		threadScope.set("AAA");
		a.m1();
	}
}
class Thread2 extends Thread{
	static ThreadScope threadScope = new ThreadScope();
	A a;
	Thread2(A a){
		this.a = a;
	}
	public void run() {
		threadScope.set("BBB");
		a.m2();
	}
}
public class Test {
	public static void main(String[] args) {
		A a = new A();

		Thread1 t1 = new Thread1(a);
		Thread2 t2 = new Thread2(a);

		t1.start();
		t2.start();
	}
```

}