Sample Questions :

1. **What is the relation between static methods, method overloading and method overriding ?**

Method Overloading means same method doing the same operations but with different number of parameters, different data type in parameters. Or we can say if Class has multiple methods having same name but different in parameters, its Method overloading.

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

A static method cant be overridden, because static method is bound with the class while instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area. As per Java coding convention, static methods should be accessed by class name rather than an object. Now, there is a catch, if we declare the same static method with same signature in the child class, it will work and take the Child class method but not the Parent class method. Also, if we mention @override keyword to the same method then it will not work.

So we can say, we can declare the same static method with same name in subclass, but we can’t override the parent class static method.

As per above explanation, In short, a static method can be overloaded, but can not be overridden in Java. If you declare,  another static method with same signature in derived class than the static method of superclass will be hidden, and any call to that static method in subclass will go to static method declared in that class itself. This is known as method hiding in Java.

Method overloading increases the readability of the program. There are two ways to overload the method in java -

1. By changing number of arguments
2. By changing the data type.

**Can we overload java main() method?**

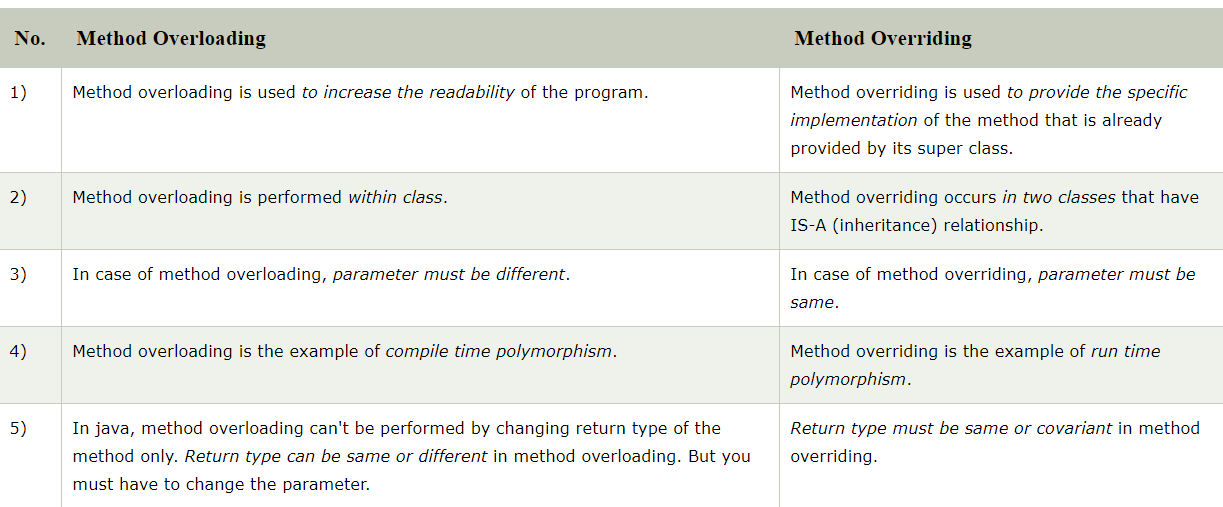
Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only.

**Method Overriding** **definition :** If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

Usage of Java Method Overriding :

1. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
2. Method overriding is used for runtime polymorphism

**Differences b/w Method Overloading and Overriding :**



1. **What is difference between Array List and Vector ?**

Answer : There are two differences between Vector and ArrayList. First, Vector is synchronized while ArrayList is not, and Second, Vector is a legacy class, and it support some legacy methods which are not part of Collections framework.

Those methods are : addElement(), elementAt(), firstElement() methods.

ArrayList uses the **Iterator** interface to traverse the elements. While A Vector can use both **Iterator** interface or **Enumeration** interface to traverse the elements. So here, Enumeration is the only legacy Interface.

ArrayList and Vector both implements List interface and maintains insertion order. Since they both implements List interface then Vector class will implement all the methods of List interface as well as Legacy methods too.

**Code example of Vector** :

Vector vec = **new** Vector<String>();

vec.add("First");

vec.add("second");

vec.addElement("Third");

Enumeration<String> e = vec.elements();

**while**(e.hasMoreElements()) {

System.***out***.println(e.nextElement());

}

When we create Vector Object like below, then it will create with initial by-default capacity of 10 and it will double the capacity after element next to its size going to add.

Case 1 : To create Vector –

// We are creating Vector object below with the inital default capacity of 10

Vector vec = **new** Vector();

System.***out***.println(vec.size()); // 0

System.***out***.println(vec.capacity()); // Original capacity or Default capacity is 10

vec.add("Test1");

vec.add("Test2");

System.***out***.println(vec.size()); // 2

System.***out***.println(vec.capacity()); // 10

Case 2 :

Vector vec3 = **new** Vector<String>(4);

System.***out***.println(vec3.size()); // 0

System.***out***.println(vec3.capacity()); // will be 4

vec3.add(1);

vec3.add("Feat1");

vec3.add("Feat2");

vec3.add("Feat3");

System.***out***.println(vec3.size()); // 4

System.***out***.println(vec3.capacity()); // 4

vec3.add("Feat4");

System.***out***.println(vec3.size()); // It becomes 5

System.***out***.println(vec3.capacity()); // It becomes 8 now on addition of 5th element, since actual capacity was 4. Double the capacity of 4 equals 8

vec3.add("Feat5");

vec3.add("Feat6");

vec3.add("Feat7");

vec3.add("Feat8");

System.***out***.println(vec3.size()); // It becomes 9

System.***out***.println(vec3.capacity()); // It becomes 16. Again double the capacity of 8

Case 3 :

Vector vec2 = **new** Vector<String>(4, 6); // initial capacity is 4 and its going to increment by 6 means, on insertion of 5th element

// automatically size will be 6+4 = 10 .. on 11th insertion it would be 16(10+6).

System.***out***.println(vec2.capacity()); // Here capacity will be 4 initially. But later it will be 10 and 16....

System.***out***.println(vec2.size());

1. **Imp Java Question – Can we make an Abstract class/Method final in Java ?**

**Answer** : No, we cannot make an abstract class or method final in Java because the abstract and final are the mutual exclusive concept. An abstract class is incomplete and can only be instantiated by extending a concrete class and implementing all abstract methods, while a final class is considered as complete and cannot be extended further. This means when we make an abstract class final, it cannot be extended hence it cannot be used and that's why Java compiler throws a compile-time error when you try to make an abstract class final in Java.  
In short, an abstract class cannot be final in Java, using both abstract and final modifier with a class is illegal in Java.

Same rule applies to abstract methods, we can’t make it final in Java. An abstract method must be override to get it used or implemented but if we make that abstract method final, it can’t be overridden.

Basically if we see, abstract means incomplete and final means complete.

Note : We can’t put final keyword before an abstract method, but we can put public or protected keyword easily.

**ABSTRACT CLASS :**

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body). It support Abstraction in Java. Now what is Abstraction, will say it’s an process of hiding the implementation details and showing only functionality to the user.

Abstract class need to be extended and methods implemented. It can’t be instantiated. It can have Constructors and static methods also. It can have final methods which will force the subclass not to change the body of the method.

There are two ways to achieve abstraction in java:

1. Abstract class
2. Interface

We have 1 Abstract class below :

**abstract** **class** And {

**public** **static** **void** test1() { // can have static & public methods too

System.***out***.println(123);

}

**abstract** **void** test2(); // abstract method

And(){ // Abstract class can have Constructor as well

System.***out***.println("System created");

} }

Now, we have to call these methods, will make 1 class that will extend this abstract class :

**public** **class** abstractClass **extends** And {

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

And.*test1*(); // Since its an static method, it will be called using Class

And a = **new** abstractClass();

// And a = new And(); We can not instantiate an Abstract Class

a.test2();

}

//@Override // Optional to use

**void** test2() {

System.***out***.println("test");

} }

So, abstractClass has implemented the abstract method (test2) as well from And class.

Output will be:

123

System created

Test

Now other scenario , if we extend an abstract class that has an abstract method, we have to either implement that method or make the class abstract. It means that An abstract class can also extend another abstract class as well. We’ll see how.

**abstract** **class** Fire **extends** And {

}

Now, main class will look like :

**public** **class** abstractClass **extends** Fire {

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

And.*test1*();

And a = **new** abstractClass();

// And a = new And(); We can not instantiate an Abstract Class

a.test2();

Fire f = **new** abstractClass();

}

//@Override // Optional to use

**void** test2() {

System.***out***.println("test");

} }

Output will be :

123

System created

test

System created

We can write 1 Interface as well, and also a normal class or abstract class can implement that Interface. And after that normal rule will apply. Abstract class while don’t need to implement all Interface methods.

1. **Can abstract class implements interface in Java? do they require to implement all methods?**

**Ans**: Yes, an abstract class can implement an interface by using the implements keyword. Since they are abstract, they don’t need to implement all methods.

1. **Is it necessary for an abstract class to have an abstract method?**

**Ans :** No, It’s not mandatory for an abstract class to have any abstract method. You can make a class abstract in Java, by just using abstract keyword in class declaration

1. **Interface Complete Explanation –**

**Ans :** The interface in Java is *a mechanism to achieve*[abstraction](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java). It has static constants and abstract methods.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

**Why we use Interface :**

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance which is not possible with Java Classes.
3. Java Interface also represents Is-A Relationship
4. It cannot be instantiated just like the abstract class.
5. Since Java 8, we can have **default and static methods** in an interface.
6. Since Java 9, we can have **private methods** in an interface.

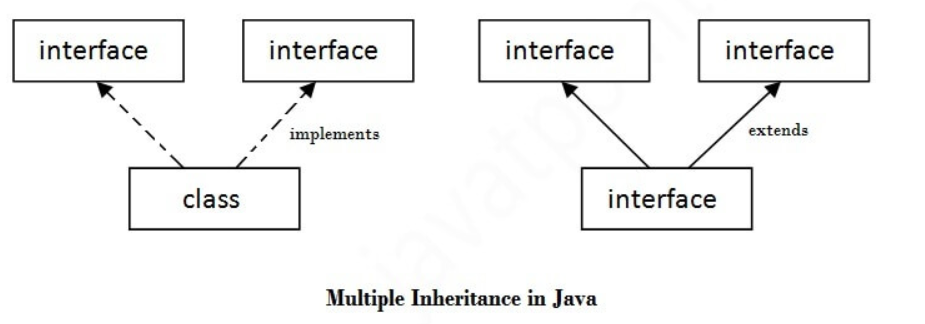
**Note** : An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Imp Point –

1. In other words, Interface fields are public, static and final by default, and the methods are by-default public and abstract.
2. The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.
3. A class extends another class, an interface extends another interface, but a **class implements an interface**

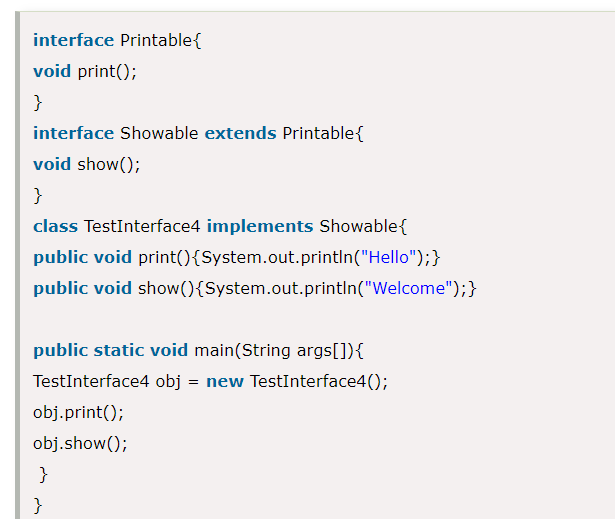
**Multiple Inheritance in Java by Interface :**

1. If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.
2. **Note :** Multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:



**Interface Inheritance :**

A class implements an interface, but one interface extends another interface.



Output :

Hello

Welcome

**Java 8 Default and Static Method in Interface :**

Since Java 8, we can have method body in interface. But we need to make it default method. Since Java 8, we can have static method in interface

**public** **class** InterfacePrac {

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

Drawable d=**new** Rectangle();

d.draw();

d.msg();

System.***out***.println(Drawable.cube(3));

}

}

**interface** Drawable{

**void** draw();

**static** **int** cube(**int** x){**return** x\*x\*x;} // this method has method body

**default** **void** msg(){System.***out***.println("default method");} // this method has method body

}

**class** Rectangle **implements** Drawable{

**public** **void** draw(){System.***out***.println("drawing rectangle");}

}

**Nested Interface :**

An interface i.e. declared within another interface or class is known as nested interface. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

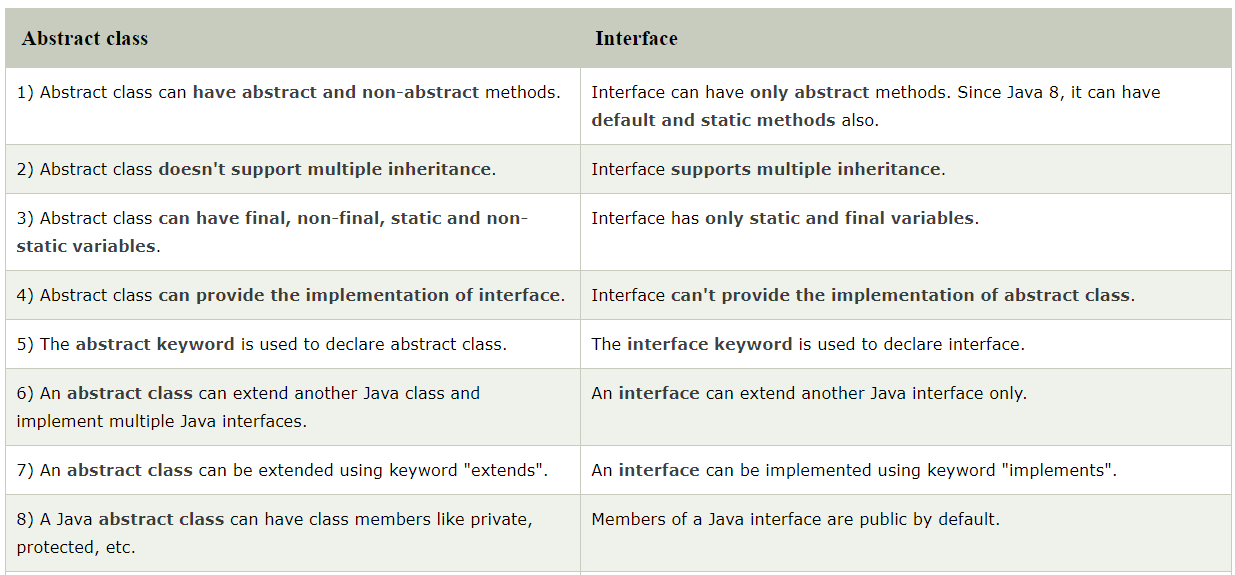
Ques Can we define a class inside the interface?

Ans : Yes, If we define a class inside the interface, java compiler creates a static nested class

1. **Difference between abstract class and interface in Java?**

**Ans :** Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.



1. **What is Final in Java –**

**Ans :** Final in Java is very important keyword and can be applied to class, method, and variables in Java. Final is a keyword or reserved word in Java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise a compilation error if you try to re-initialized final variables in Java.

Final Variable - Any variable either a member variable or local variable (declared inside method or block) modified by final keyword is called final variable. Final variables are often declared with static keyword in java and treated as constant.

Ex: public static final String abc = “Tushar”;

Abc = new String(“Mittal”); /// compilation error

Final Method - The final keyword in Java can also be applied to methods. A Java method with the final keyword is called a final method and it can not be overridden in the subclass.

**public** **class** Final {

**public** **static** **final** String ***loan*** = "loan";

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

System.***out***.println("test");

}

**public** **static** **final** **void** transfer() {

System.***out***.println("nine");

}

}

**class** vehicle **extends** Final{

**public** **void** transfer() { // Compilation error

}

Final Class - The final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer, and other wrapper classes

Create a final class and try to extend it. Will get compilation error.

Few more points :

1. Final variables are safe to share in multi threading env
2. Final and Immutable in Java - Final keyword helps to write an immutable class. Immutable classes are the one which can not be modified once it gets created and String is a primary example of an immutable and final class
3. The final member variable must be initialized at the time of declaration or inside the constructor, failure to do so will result in compilation error.
4. The local final variable must be initialized during declaration.
5. Final class can’t be inherited
6. Final and abstract are two opposite keyword and a final class can not be abstract in Java.
7. **Javascript code to enter the text in text field through Javascript Executor**. It required when in React.js code, text was getting input partially. In that case, this code will be helpful.

String givenInput2 = "Test data #45 &67 @!future SELENIUM";

String script2 = "var input=document.getElementById('%f');var nativeInputValueSetter = Object.getOwnPropertyDescriptor(window.HTMLInputElement.prototype, 'value').set;nativeInputValueSetter.call(input, '%g'); var ev2 = new Event('input', { bubbles: true}); input.dispatchEvent(ev2) ";

script2 = script2.replace("%f", textboxID);

script2 = script2.replace("%g", givenInput2);

jse.executeScript(script2, "");

**if** (textbox.getAttribute("value").length() <= 50) {

System.***out***.println("Textbox accepting alpha numeric & special characters");

} **else** {

System.***out***.println("Length issue entering alpha numeric & special characters");

softassertion.assertTrue(**false**);

}

String givenInput2 = "Test data #45 &67 @!future SELENIUM";

String script2 = "var input=document.getElementById('%f');var nativeInputValueSetter = Object.getOwnPropertyDescriptor(window.HTMLInputElement.prototype, 'value').set;nativeInputValueSetter.call(input, '%g'); var ev2 = new Event('input', { bubbles: true}); input.dispatchEvent(ev2) ";

script2 = script2.replace("%f", textboxID);

script2 = script2.replace("%g", givenInput2);

jse.executeScript(script2, "");

**if** (textbox.getAttribute("value").length() <= 50) {

System.***out***.println("Textbox accepting alpha numeric & special characters");

} **else** {

System.***out***.println("Length issue entering alpha numeric & special characters");

softassertion.assertTrue(**false**);

}

1. **Method to Sort Hashmap based on Values:**

**Code:**

HashMap hm = **new** HashMap();

hm.put("Zara", **new** Double(3434.34));

hm.put("Mahnaz", **new** Double(123.22));

hm.put("Ayan", **new** Double(1378.00));

hm.put("Daisy", **new** Double(99.22));

hm.put("Qadir", **new** Double(19.08));

Set<Map.Entry<String, Double>> set = hm.entrySet();

List<Double> ls = **new** ArrayList<Double>();

**for**(Map.Entry<String, Double> map : set)

{

Double a = map.getValue();

ls.add(a);

}

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

Collections.*sort*(ls);

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

LinkedHashMap<String, Double> newMap = **new** LinkedHashMap<String, Double>();

**for**(**int** i=0;i<ls.size();i++)

{

**for**(Map.Entry<String, Double> map : set)

{

**if** (ls.get(i).equals(map.getValue()))

{

newMap.put(map.getKey(), map.getValue());

}

}

}

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

1. **Method to Sort Hashmap based on Key:**

**Code:**

HashMap hm = **new** HashMap();

hm.put("Zara", **new** Double(3434.34));

hm.put("Mahnaz", **new** Double(123.22));

hm.put("Ayan", **new** Double(1378.00));

hm.put("Daisy", **new** Double(99.22));

hm.put("Qadir", **new** Double(19.08));

Set<Map.Entry<String, Double>> set = hm.entrySet();

List<String> list1 = **new** ArrayList<String>();

**for**(Map.Entry<String, Double> map : set)

{

String a = map.getKey();

list1.add(a);

}

System.***out***.println(list1);

Collections.*sort*(list1);

System.***out***.println(list1);

LinkedHashMap<String, Double> newMap = **new** LinkedHashMap<String, Double>();

**for**(**int** i=0;i<list1.size();i++)

{

**for**(Map.Entry<String, Double> map : set)

{

**if** (list1.get(i).equals(map.getKey()))

{

newMap.put(map.getKey(), map.getValue());

}

}

}

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

1. In above method, we saw Collections.sort() method to be used to sort the Collections items in ascending order. Now if we want it to be in reverse order/ descending. Do the below Code:

Collections.*sort*(list1,Collections.*reverseOrder*());

System.***out***.println(list1);

1. In the above method, we saw how to get the exact matching key for the respective value:

So its code :

If(“<Expected Value>”.equals(Map.Entry<String, Double> map.getValue)

{

Return map.getKey();

}

By this Code, we’ll get the expected key for the particular value

1. **Methods to Convert Set into List:**

There are multiple methods exist to do the same:

1. **Brute Force or Naive method :** Suppose there is a Hashmap (hm). Now

Set<String> set2 = hm.keySet(); // It will give all the Keys

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

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

**for**(String s : set2)

{

ls.add(s);

}

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

1. Simplest Way:

List<String> list2 = new ArrayList<String>(set2);

System.***out***.println(list2);

1. **Using Guava Library List.newArrayList(set)**: Lists.newArrayList(set) creates a mutable ArrayList instance containing the elements of the specified Set.

System.***out***.println(Lists.*newArrayList*(set2));

1. **Super keyword in Java –**

**Ans :** The **super** keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

**Usage of Java super Keyword -**

* super can be used to refer immediate parent class instance variable.
* super can be used to invoke immediate parent class method.
* super() can be used to invoke immediate parent class constructor.

Detailed explanation – (1) When Super is used to refer immediate parent class instance variable :

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

**public** **class** InterfacePrac {

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

MultiThread m = **new** MultiThread();

m.paint();

}}

**class** MultiThread **extends** Nine {

String color = "Black";

**public** **void** paint() {

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

System.***out***.println(**super**.color);

}

}

**class** Nine {

String color = "White";

}

(2) – When Super is used to invoke parent class method and Constructor –

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

The super keyword can also be used to invoke the parent class constructor

**class** Nine {

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

System.***out***.println("Test Data");

}

Nine(){

System.***out***.println("Nine created");

}

**protected** **void** data() {

System.***out***.println("Not out");

}

}

**class** MultiThread **extends** Nine {

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

System.***out***.println(1234);

//super.run(); // If suppose method name is same as of Super class "Nine", then we have to call the super method like this

}

**void** bark() {

//run();

**super**.run(); // Calling run() method from Super class, explicitly calling it

}

MultiThread(){

**super**(); // We need to notice 1 thing here, either we can explitictly define super() to call Super class constructor or either we don't mention and JVM automatically will take it

}

}

**public** **class** InterfacePrac {

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

MultiThread m = **new** MultiThread(); // After this, MultiThread class constructor will execute automatically. Also, if its extending 1 super class, then Super class constructor will also run.

m.run(); // It will run run method of child class.

m.bark();

m.data();

}}

1. **Why Multiple Inheritance is not supported in Java –**

**Ans :** Multiple inheritance is not supported with java classes but it will support with Java interfaces. The main reason behind this is ambiguity around Diamond problem. Suppose there is a common method defined in 2 Java classes – foo().

Now, we take another class and going to extend both these classes, and we call that method, then Java compiler will not be able to decide which foo() method it should take.

1. **Can a non-static method access a static variable or call a static method –**

**Ans**: Yes, a non-static method can access a static variable or call a static method in Java. There is no problem with that because of static members i.e. both static variable and static methods belong to a class and can be called from anywhere, depending upon their access modifier. For example, if a static variable is private then it can only be accessed from the class itself, but you can access a public static variable from anywhere.  
  
Similarly, a private static method can be called from a non-static method of the same class but a public static method like main() can be called from anywhere.

But its vice-versa if we see, we can’t access non static variable or method from a static method directly. We can do this by using a reference object of the class and call it. Its because a static method forms a static context where only static members can be accessed.

1. **V IMP – Exception Handling. Difference between throw and throws.**

**Ans :**

//////////// ---------- ////////////// ---------- /////////////

1. **Multi threading in Java :**

**Ans : Multithreading in Java** is a process of executing multiple threads simultaneously. A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Why we prefer to use Multi-threading –

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2. You **can perform many operations together, so it saves time**.
3. IMP Point - Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread. It used a shared memory area.
4. Thread based multi-tasking (Multithreading) – Threads share the same address space. A thread is light weight.

**What is Thread in Java –**

Thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution.

Note : At a time, one thread is executed only.

Java provides Thread class to achieve thread programming. Thread class extends object class and implements Runnable interface.

**Start()** – This method is used to start the execution of the thread.

Run() – It is used to do an action for a thread.

currentThread() - It returns a reference to the currently executing thread object. – Mainly we use it like

Thread. currentThread().getName()

**Life Cycle of a Thread –**

1. New – The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. Runnable - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. Running - The thread is in running state if the thread scheduler has selected it.
4. Non Runnable - This is the state when the thread is still alive, but is currently not eligible to run.
5. Terminated - A thread is in terminated or dead state when its run() method exits.

2 Ways to create Thread -

1. We can create a Thread by extending Thread class – start() method of Thread class is used to start a newly created Thread. After this, a thread moves from new state to Runnable state.

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac i = **new** InterfacePrac();

i.start();

// Minute observation - If we extends Thread class to the current Class, then after creating an instance of the current class, we can call start() method to initiate the thread. No need to separately create an instance of Thread class. But if we didn't extend Thread class. Current class object instance won't be able to call start() method itself.

@Override

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

System.***out***.println("Thread started");

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

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

}

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

}

}

Note : Run() method will automatically triggers once start() method kicked out

Here, we have used keyword @override.. Its not mandatory to use this keyword. Its just a reference that says run() method belongs to Thread class.

1. We can create Thread by implementing Runnable interface as well. If we don’t define Thread class, we just implement Runnable interface, then we have to explicitly define instance object of Thread class to initiate a Thread. Also, we have to pass the current class instance as parameter of Thread class object creation.

**public** **class** InterfacePrac **implements** Runnable {

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

InterfacePrac i = **new** InterfacePrac();

Thread t = **new** Thread(i);

t.start();

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

System.***out***.println("Thread started");

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

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

}

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

}

}

Note : Run() method will automatically triggers once start() method kicked out

We have seen sleep method of Thread class. And we know that at a time, only 1 thread will be executed. So, if we sleep a thread for the specified time, the thread schedular picks up another thread and so on.

Note : We can’t start a thread twice. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

In above example, just mention 1 more sentence,

t.start()

We’ll get IllegalThreadStateException

**Imp Concept : What happen if we call run() method directly instead of start() method –**

Ans : Each thread starts in a separate call stack. Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

We’ll see this concept with 2 code examples:

Ex1 : We’ll just use 1 Thread, and will not initiate start() method, we just call out run() method. So logic is that, it will invoke run() method of Thread class, but it will run like a normal method, not a thread method, point is we have to call run() method explicitly,

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac i = **new** InterfacePrac();

**i.run();**

**}**

**public void run() {**

System.***out***.println("Thread started");

}

Output : Thread started.

But this will not run as a thread object.

Ex2: Now, we’ll use 2 Threads. And call 2 run() methods.

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac i = **new** InterfacePrac();

InterfacePrac i1 = **new** InterfacePrac();

//i.start();

//i1.start();

i.run();

i1.run();

}

@Override

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

System.***out***.println("Thread started");

**for**(**int** i=0;i<4;i++) {

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

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

}

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

}

}}

Output :

Thread started

0

1

2

3

Thread started

0

1

2

3

Logic is : that there is no context-switching because here i and i1 will be treated as normal object not thread object. So, it will run one after another.

**Join() method in Thread –**

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

Ie. If we use join() method to any thread, and there are some other threads also persists in the program. Then the thread to which join() appends, that method will end its execution first. After that, other threads will starts its execution.

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac t1 = **new** InterfacePrac();

InterfacePrac t2 = **new** InterfacePrac();

InterfacePrac t3 = **new** InterfacePrac();

t1.start();

**try** {

t1.join();

} **catch** (Exception e) {

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

}

t2.start();

t3.start();

}

@Override

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

System.***out***.println("Thread started");

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

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

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

}

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

}

}

}

// Since we have attached join() with Thread t1. So Execution will stop for other threads before thread t1 completes its execution. After t1 completed its execution, then thread t2 and thread t3 act as a Thread object.

Few more methods of Thread class –

1. getName() – It returns the name of the thread.
2. setName() – It changes the name of the thread.
3. getId() – It returns the ID of the thread.
4. currentThread() – It returns a reference to the currently executing Thread object.

Note : By-default each thread has a name i.e. thread-0, thread-1 and so on

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac t1 = **new** InterfacePrac();

InterfacePrac t2 = **new** InterfacePrac();

InterfacePrac t3 = **new** InterfacePrac();

System.***out***.println(t1.getName());

System.***out***.println(t2.getName());

System.***out***.println(t1.getId());

t1.start();

t2.start();

System.***out***.println(Thread.*currentThread*());

t1.setName("Tushar"); // changes the name of this thread to be equal to argument name

System.***out***.println(t1.getName());

System.***out***.println(Thread.*currentThread*().getName());

t3.start();

t2.setName(“Tushar”);

System.***out***.println(Thread.*currentThread*());

System.***out***.println(Thread.*currentThread*().getName());

}

@Override

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

System.***out***.println("Thread started");

}

Priority of Thread :

Each thread has a priority. We have 3 constants defined in Thread class related to setting priority:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac t1 = **new** InterfacePrac();

InterfacePrac t3 = **new** InterfacePrac();

t3.setPriority(Thread.***MAX\_PRIORITY***);

t1.setPriority(Thread.***MIN\_PRIORITY***);

t1.start();

t3.start();

// Since we have set the priority Max priority to Thread t3, so it will run first before Thread t1. Also, Thread t3 will be assigned 1st thread name like thread-0

}

@Override

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

System.***out***.println("Thread started");

System.***out***.println(Thread.*currentThread*().getName());

System.***out***.println(Thread.*currentThread*().getId());

System.***out***.println(Thread.*currentThread*().getPriority());

}}

Output :

Thread started

Thread-0

Thread started

Thread-1

12

10

11

1

Imp Note : When we see like, we instantiated 3 or 4 Thread instances or Current class Objects in case of Runnable interface. All these instances will hit single method run(). Its like we are performing a single task by Multiple threads. Means all the threads will execute run() method. In backend but, each thread runs in a separate callstack. This is the example of Single Task by multiple Threads.

Now, we’ll see Example of Multiple Tasks by Multiple Threads :

Example 1 –

**public** **class** threadprac{

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

Simple1 s1 = **new** Simple1();

Simple2 s2 = **new** Simple2();

s1.start();

s2.start();

}}

**class** Simple1 **extends** Thread{

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

System.***out***.println("task one");

}

}

**class** Simple2 **extends** Thread{

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

System.***out***.println("task two");

}

}

In above example : We created 2 Classes that extend Thread class. And its objects in other class.

Example 2 : Two tasks by 2 Threads – This is a different and unique way to create Thread instances, this is called as Anonymous Inner class.

**public** **class** threadprac{

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

Thread t1 = **new** Thread() {

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

System.***out***.println("task three");

};

};

Thread t2=**new** Thread(){

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

System.***out***.println("task four");

}

};

t1.start();

t2.start();

}

}

VV Imp topic of Thread – We’ll see now the multiple ways to create Threads –

**public** **class** InterfacePrac **extends** Thread **implements** Runnable {

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

InterfacePrac t1 = **new** InterfacePrac();

Thread t4 = **new** Thread(); // We can create Thread from this type, but its of no use. If we call start() method as well from this Thread instance, it will not call run() method of Thread class

t4.start(); // No use it will not call run() method

Thread t5 = **new** Thread(t1); // This is the perfect way to create a Thread instance, and it will call run() method as well.

t5.start(); // It will call run() method of Thread class

Thread t6 = **new** Thread(**new** InterfacePrac()); // Parameter is just current class obj reference. Its an correct way to create Thread instance, it will call run() method

Thread t7 = **new** Thread(**new** InterfacePrac(),"Aditi"); // Thread will get an default name, but here we are giving the Thread name. Parameter are current class obj reference with new Thread Name. Its an correct way to create Thread instance, it will call run() method

System.***out***.println(t6.getName()); // Output - Thread-3

System.***out***.println(t7.getName()); // Output – Aditi

Thread t8 = **new** Thread("Finance"); // One more way of creating a Thread, to give name while creating itself

t8.start(); // If we create a Thread without any current class parameter, it will not hit run() method

System.***out***.println(t8.getName()); // Output – Finance

Thread t9 = **new** Thread(tg1,t3); // Parameters are first is Thread group instance member, 2nd is instance of current class

t9.start(); // It will hit run() method of Thread class

// Below are 2 ways we can create Thread object. But we need to see 2 things here, 1st way just gave Thread Group instance and Thread Name. I didn’t give Current class object reference here, then i try to initiate the Thread t10 by kicking start() method, It will not hit run() method, since it needs current class obj reference

Thread t10 = **new** Thread(**new** ThreadGroup("Mittal"),"t10ThreadName");

System.***out***.println(t10.getName());

t10.start();

// To overcome above prob, i create 1 more thread, this time i gave current class obj reference and try to give start() method. Now for this Thread object t11, it worked.

Thread t11 = **new** Thread(**new** ThreadGroup("Mittal"),t1,"t10ThreadName");

t11.start();

**Thread Group –**

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call. A ThreadGroup represents a set of threads. A thread group can also include the other thread group.

ThreadGroup tg1 = **new** ThreadGroup("Tushar Thread");

// Creating below 3 Threads inside Thread Group - Tushar Thread

Thread t12 = **new** Thread(tg1,**new** InterfacePrac(),"Thread One");

Thread t13 = **new** Thread(tg1,**new** InterfacePrac(),"Thread Two");

Thread t14 = **new** Thread(tg1,**new** InterfacePrac(),"Thread Three");

t12.start();

t13.start();

t14.start();

System.***out***.println(tg1.activeCount()); // It will give the estimate of all active threads running in this threadgroup and its subgroups.Recursively iterates over all subgroups inthis thread group.The value returned is only an estimate because the number ofthreads may change dynamically while this method traverses internaldata structures, and might be affected by the presence of certain system threads

System.***out***.println(tg1.activeGroupCount()); // Returns an estimate of the number of active groups in thisthread group and its subgroups. Recursively iterates overall subgroups in this thread group.

System.***out***.println(tg1.getName()); // Print the name of Thread group

System.out.println(Thread.currentThread().getName()); // gives name of the running thread

tg1.list(); // It will list our all the active Threads in the given Thread group

// Thread.currentThread().getThreadGroup().interrupt(); // used to interrupt all the threads in the given Thread group

**Daemon Thread in Java –**

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically. It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads. It is a low priority thread.

Ques : Why JVM terminates the daemon thread if there is no User thread /

Ans : The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

**public** **class** InterfacePrac **extends** Thread {

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

InterfacePrac t1 = **new** InterfacePrac();

InterfacePrac t2 = **new** InterfacePrac();

t2.start();

t1.setDaemon(**true**); // We can set any thread as a daemon thread. Its a correct way to do.But if we define this before start() method for that particular thread, then we'll get IllegalThreadSTateException

t1.start(); }

@Override

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

System.***out***.println("New Thread started");

**if**(Thread.*currentThread*().isDaemon()){//checking for daemon thread

System.***out***.println("daemon thread work");

}

**else**{

System.***out***.println("user thread work");

}

}

Garbage Collection –

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

protected void finalize(){}

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

public static void gc(){}

#### Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

**public** **class** InterfacePrac {

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

InterfacePrac t1 = **new** InterfacePrac();

InterfacePrac t2 = **new** InterfacePrac();

t1 = **null**;

t2 = **null**;

System.*gc*();

}

@Override

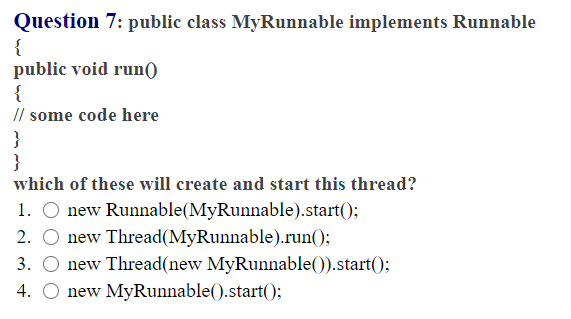
**protected** **void** finalize() **throws** Throwable {

System.***out***.println("garbage");

**super**.finalize();

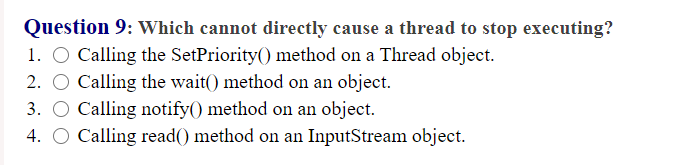
}

Ques -



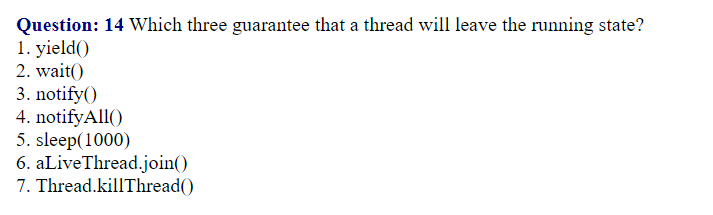
Ans – Option no 3

Ques:



Ans – Option 3

Ques:



Ans : 2,5,6

//////////// ---------- ////////////// ---------- /////////////

1. **Java Synchronization? Difference b/w Synchronized and not-Synchronized in Java ?**

**Ans** : Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time. Synchronization in java is the capability *to control the access of multiple threads to any shared resource*. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

There are 2 types of thread synchronization –

Mutual Exclusive and Cooperation (Inter-thread communication)

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

***By synchronized method, by synchronized block or by static synchronization***

Concept of lock in Java – This synchronization is implemented in Java with a concept called monitors or lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

**Detailed explanation of synchronized method -**

Example 1 : In below program, we will create 2 Java classes that extends Thread class. We’ll create 1 more class and define an synchronized method there.

**class** Table{

**synchronized** **void** printTable(**int** n){ //method not synchronized

**for**(**int** i=1;i<=3;i++){

System.***out***.println(n\*i);

**try**{

Thread.*sleep*(400);

}

**catch**(Exception e){

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

}

}

}

}

**class** MyThread1 **extends** Thread{

Table t;

MyThread1(Table t){

**this**.t=t;

}

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

t.printTable(5);

}

}

**class** MyThread2 **extends** Thread{

Table t;

MyThread2(Table t){

**this**.t=t;

}

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

t.printTable(100);

}

}

**public** **class** InterfacePrac {

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

Table obj = **new** Table();//only one object

MyThread1 t1=**new** MyThread1(obj);

MyThread2 t2=**new** MyThread2(obj);

// Creating 2 different class objects using another class object

t1.start();

t2.start();

}

}

Output : // In synchronized manner --

5

10

15

100

200

300

Explanation - We didn’t create the run() method inside MyThread classes as synchronized, if we create that synchronization will start from there itself. We define that method as synchronized which is going to hit by both MyThread classes object. It’s like a shared resource method, now due to synchronized keyword, it will allow only 1 class obj reference at a time. Then the other one.

Example 2 : From above example, we will remove MyThread2 class from the code, we just created 1 more object from MyThread1 class. Then we invoke the same program. Again it will call the print Table() method, but one by one.

**class** Table{

**synchronized** **void** printTable(**int** n){//method not synchronized

**for**(**int** i=1;i<=3;i++){

System.***out***.println(n\*i); // i = 1,2,3,4,5 if suppose n=5

**try**{

Thread.*sleep*(400);

}

**catch**(Exception e){

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

}

}

}

}

**class** MyThread1 **extends** Thread{

Table t;

MyThread1(Table t){

**this**.t=t;

}

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

t.printTable(5);

}

}

**public** **class** InterfacePrac {

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

Table obj = **new** Table();//only one object

MyThread1 t1=**new** MyThread1(obj);

MyThread1 t2=**new** MyThread1(obj);

// Creating 2 different class objects using another class object

t1.start();

t2.start();

obj.printTable(10);

}

}

Output :

10

20

30

5

10

15

5

10

15

Example 3 : If we just make run() method as synchronized, it won’t help much. It will give values like un synchronized manner only.

**Detailed explanation of synchronized block –**

Synchronized block can be used to perform synchronization on any specific resource of the method. If we put all the codes of the method in the synchronized block, it will work same as the synchronized method. Java provides a way of creating threads and synchronizing their task by using synchronized blocks. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.

Example 1 :

**class** Table{

**void** printTable(**int** n){ //method not synchronized

**synchronized** (**this**) { // here this keyword refers to current class object reference & block is synchronized

**for**(**int** i=1;i<=3;i++){

System.***out***.println(n\*i);

**try**{

Thread.*sleep*(200);

}

**catch**(Exception e){

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

}

}}

}

}

**class** MyThread1 **extends** Thread{

Table t;

MyThread1(Table t){

**this**.t=t;

}

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

t.printTable(5);

}

}

**class** MyThread2 **extends** Thread{

Table t;

MyThread2(Table t){

**this**.t=t;

}

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

t.printTable(100);

}

}

**public** **class** InterfacePrac {

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

Table obj = **new** Table();//only one object

MyThread1 t1=**new** MyThread1(obj);

MyThread2 t2=**new** MyThread2(obj);

// Creating 2 different class objects using another class object

t1.start();

t2.start();

}

}

**Output**

5

10

15

100

200

300

**Example 2 : We can use synchronized block inside run() method like below, just update MyThread classes like below and remove synchronized block from printable() method from above program. Output will be in synchronized way.**

**class** MyThread2 **extends** Thread{

Table t;

MyThread2(Table t){

**this**.t=t;

}

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

**synchronized** (t) {

t.printTable(100);

}

}

}

**Detailed explanation of static synchronized block –**

If you make any static method as synchronized, the lock will be on the class not on object.

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

T1 and t2 belongs to 1 object.

And t3 and t4 belongs to diff object.

Thread like t1 and t3 will run separately but values can intervene. Its not like a synchronized values. To overcome this prob, static synchronized block came into picture.

**class** Table{

**static** **synchronized** **void** printTable(**int** n){ //method not synchronized

**for**(**int** i=1;i<=3;i++){

System.***out***.println(n\*i);

**try**{

Thread.*sleep*(200);

}

**catch**(Exception e){

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

}

}}

}

**class** MyThread1 **extends** Thread{

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

Table.*printTable*(10);

}

}

**class** MyThread2 **extends** Thread{

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

Table.*printTable*(5);

}

}

**public** **class** InterfacePrac {

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

MyThread1 t1=**new** MyThread1();

MyThread2 t2=**new** MyThread2();

t1.start();

t2.start();

}

}

Inter Thread Communication in Java –

What is Polling in Java ?

Ans - The process of testing a condition repeatedly till it becomes true is known as polling. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.

To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().**

All these methods belong to object class as final so that all classes have them.

IMP Point to note : They must be used within a synchronized block only.

**wait**()-It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().

**notify**()-It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.

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

**public** **class** InterfacePrac {

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

**final** PC p = **new** PC();

// Anonymous inner class

Thread t1 = **new** Thread(**new** Runnable() {

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

**try** {

p.produce();

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

// Anonymous inner class

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

**try** {

p.consume();

} **catch** (Exception e) {

e.printStackTrace();

}

}

});

t1.start();

t2.start();

t1.join();

t2.join();

}

**public** **static** **class** PC{

**public** **void** produce()**throws** InterruptedException

{

// synchronized block ensures only one thread

// running at a time.

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

{

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

// releases the lock on shared resource. It releases the lock it holds on PC Object. After wait() method call, produce() thread goes on waiting until all other threads have been terminated.

wait();

System.out.println("Tushar");

// and waits till some other method invokes notify().

System.***out***.println("Resumed");

}

}

**public** **void** consume()**throws** InterruptedException

{

// this makes the produce thread to run first.

/// Thread.sleep(1000);

Scanner s = **new** Scanner(System.***in***);

// synchronized block ensures only one thread

// running at a time.

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

{

System.***out***.println("Waiting for return key.");

s.nextLine();

System.***out***.println("Return key pressed");

// notifies the produce thread that it

// can wake up.

notify();

System.out.println("Mittal");

// Sleep

Thread.*sleep*(2000);

}

}

}

}

Output :

producer thread running

Waiting for return key.

Return key pressed

Mittal

Tushar

Resumed

**Explanation** – Use of synchronized block ensures that only one thread at a time runs. Since the wait() is called on produce method, the controller will shift to consume() method. It prints “Waiting for return key”. The moment we press some key, it invokes notify() method, it will not release lock from consume method. Notify() method will just wake produce() thread from sleep that its time came to do some action.

In the consume method, we use 1 Thread.sleep method as well. So it means, that after notify() method, it will stop for 1 sec, then program terminates, then control will go to produce method.

If any print statement exists in consume() method, then it will print/execute that also before terminating the whole program.

Once the control goes to produce(), then the program terminates.

//////////// ---------- ////////////// ---------- /////////////

1. **Why characters array is better than String for storing password in Java ?**

**Ans** : The answer is mainly around Security reasons.

1. Since Strings are immutable in Java if we store password as plain text it will be available in memory until Garbage collector clears it and since String are used in String pool for reusability there is pretty high chance that it will be remain in memory for long duration, which pose a security threat. Now any one who has access to memory dump can find the password in clear text. Also we know String is immutable, we can’t change the contents of String while we can set all element of Character array to zero or blank if we use Char []. So Storing password in character array clearly mitigates security risk of stealing password.
2. Java itself recommends using getPassword() method of JPasswordField which returns a char[] and deprecated getText() method which returns password in clear text stating security reason
3. **IMP Topic - Nested Class**

**Ans**: If we define a class within another class, such classes are known as nested class. Few points :

1. The scope of nested class is bounded by scope of its enclosing class (parent class)
2. Nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.
3. As a member of its enclosing class, a nested class can be declared *private*, public, protected, or default.
4. Nested classes are divided into two categories:

* static nested class : Nested classes that are declared static are called **static nested classes.**
* inner class : An inner class is a **non-static nested class.**

Now, we’ll see in detail these both types of Nested class –

**Static Nested class –**

In the case of static nested class, Without an outer class object existing, there may be a static nested class object. i.e., an object of a static nested class is not strongly associated with the outer class object.

**public** **class** NestedClassPrac {

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

OuterClass.staticNestedClass object = **new** OuterClass.staticNestedClass();

// syntax to create object of static nested inner class

object.*display*();

}

}

**class** OuterClass {

**static** **int** *outerCount* = 10;

**int** nineDigit = 12;

**private** **int** outerPrivate = 30;

**private** **static** **int** *Decibal* = 40;

**public** String test = "Data";

**static** **void** running() {

System.***out***.println("Partition");

}

// static inner class

**static** **class** staticNestedClass {

**static** **void** display() {

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

//System.out.println(nineDigit); // static nested class can't directly access non static member

//System.out.println(outerPrivate); // static nested class can't directly access non static member

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

//System.out.println(test); // Can't access public member as well

*running*();

}

}

}

**Inner class** – This is also divided into 2 categories, - Local classes and Anonymous classes –

**public** **class** NestedClassPrac {

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

OuterClass outer = **new** OuterClass();

OuterClass.staticNestedClass innerobject = outer.**new** staticNestedClass();

// This was we have to create Inner clas object

innerobject.display();

}

}

**class** OuterClass {

**static** **int** *outerCount* = 10;

**int** nineDigit = 20;

**private** **int** outerPrivate = 30;

**private** **static** **int** *Decibal* = 40;

**public** String test = "Data";

**static** **void** running() {

System.***out***.println("Partition");

}

// local inner class

**class** staticNestedClass {

**void** display() {

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

System.***out***.println(nineDigit); // can access non static member

System.***out***.println(outerPrivate); // can access private member as well

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

System.***out***.println(test); // Can't access public member as well

*running*();

}

}

}

1. **Static block in Java and Exit method -**

**Ans** : If you put a System.exit(0) at the end of the static-block, it will run with no errors in Java 6 and below (without a valid main!). This is because the static block is executed *before* a valid main method is searched for, so if you exit the program at the end of the static block, you will receive no errors.

**However,** this behavior was changed in Java 7; now you must include an explicit main, even if it might never be reached. Means if we put System.exit() method as well, still we have to mention 1 main method explicitly. Also, class initialization is not mandatory to do here.

In Java 7 and above, the answer to the question is false, but in Java 6 and below the answer is indeed true. Question is can we run Static block without having a main method.

If we do so, we’ll get below error :

Error: Main method not found in class Test, please define the main method as:

public static void main(String[] args)

**Static Block** - In Java, Static Block is used to initialize the static data members. Important point to note is that static block is executed before the main method at the time of class loading.

class staticExample {

    // static block

    static

    {

        System.out.println("Inside Static Block.");

    }

    // main method

    public static void main(String args[])

    {

        System.out.println("Inside main method.");

    }

}

Output:

Inside Static block

Inside main method

Now, we have see **System.exit(0)** method here, what this method generally does. This method is for the successful termination of the Java program. This method exits current program by terminating running Java virtual machine.

**exit(0)** : Generally used to indicate successful termination.  
**exit(1) or exit(-1) or any other non-zero value** – Generally indicates unsuccessful termination.

**Note :**This method does not return any value.

1. **JAVA HashSet , HashMap and their difference :**

**HashSet** :

Java HashSet class is used to store unique elements. It uses hash table internally to store the elements. It implements Set interface and extends the AbstractSet class.

It creates a collection that uses hash table for storage. A hash table stores information by using a mechanism called hashing. HashSet does not maintain any order of elements.

HashSet contains only unique elements. It allows to store a null Value. It is non synchronized.

**Some methods of HashSet are :**

Add(E e) – Used to add any specific element to the set

Clear() – It removes all the elements of the set.

Contains(Object o) – It return true if Set contains that specific element.

isEmpty() – It returns true if set contains no elements.

Remove(Object o) – Used to remove any particular element from Set

Size() – To get the size of the Set

NOTE : A list can contain duplicate elements, but Set can’t.

Ex Code :

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

HashSet<String> hs = **new** HashSet<String>();

// Adding elements

hs.add("Mohan");

hs.add("Rohan");

hs.add("Sohan");

hs.add("Mohan"); // It will not allow the duplicate element

// Displaying HashSet

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

hs.remove("Rohan");

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

**for**(String element:hs) {

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

}

Iterator<String> itr = hs.iterator();

**while**(itr.hasNext()) {

System.***out***.println(itr.next()); }

**HashMap** : Java HashMap class is an implementation of Map interface based on hash table. It stores elements in key & value pairs which is denoted as HashMap<Key, Value> or HashMap<K, V>.

It uses  a hashtable to store the map. HashMap does not maintain order of its element. It allows only unique keys. It is non synchronized. It permits null values and the null key

**Code Example :**

HashMap<Integer, String> hashMap = **new** HashMap<Integer, String>();

hashMap.put(1, "One");

hashMap.put(2, "Two");

hashMap.put(3, "Three");

hashMap.put(4, "Four");

// Displaying HashMap

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

hashMap.remove(2);

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

**for**(Map.Entry<Integer, String> k: hashMap.entrySet()) {

System.***out***.println(k.getValue());

System.***out***.println(k.getKey());

}

hashMap.replace(3, "Nine");

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

**Difference b/w HashSet and HashMap :**

HashSet method automatically override equals() and hashcode() method so that we can check for equality and no duplicate value is stored in our set.

1. HashMap Stores data in form of  key-value pair while HashSet Store only objects.
2. In hash map hashcode value is calculated using key object.

There is no major diff b/w these 2 things.

1. **Primitive Data Types in Java –**

Ans :There are 8 Primitive Data Types in Java ;

Byte – Value ranges from -128 to 127, Short, Int, Long, float, Double, Char and Boolean.

1. **What are Access Modifier in Java –**

Ans : Access modifiers are keywords in Java that are used to set accessibility. An access modifier restricts the access of a class, constructor, data member and method in another class.

Java language has four access modifier to control access level for classes and its members.

1. Default: Default has scope only inside the same package
2. Public: Public has scope that is visible everywhere
3. Protected: Protected has scope within the class and all sub classes of same package or either diff package
4. Private: Private has scope only within the classes

**Default Access Modifier** - If we don’t specify any access modifier then it is treated as default modifier. It is used to set accessibility within the package. It means we can not access its method or class from outside the package.

We’ll see this concept with an example :

Create a class “staticPrac” inside the Package “com.practice.Test” and a method show() without any modifier. Which means a default modifier will get assign to it by self.

**package** com.practice.Test;

**public** **class** staticPrac {

**int** a = 10;

**void** show() {

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

}

}

Now, create 1 more class “Modifier” in different package “com.automation.Java”. And import the above class as well. Now just create a object for the above class and try to call show() method. It won’t allow. Since its out of its package.

**package** com.automation.Java;

**import** com.practice.Test.staticPrac;

**public** **class** Modifier {

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

System.***out***.println("Test Data");

staticPrac s = **new** staticPrac();

s.show() // this will give the compile error.

}

}

**Public Access Modifier** - public access modifier is used to set public accessibility to a variable, method or a class. Any variable or method which is declared as public can be accessible from anywhere in the application.

We can use just above example and in show() method we just add keyword “public” like below :

Public void show() {

System.out.println(a);

}

After this, we can now call this method in another class of other package.

**Protected Access Modifier -** Protected modifier protects the variable, method from accessible from outside the class. It is accessible within class, and in the child class (inheritance) whether child is located in the same package or some other package.

We’ll see below code , in below class, we can see method and 1 variable are declared as protected.

**package** com.practice.Test;

**public** **class** staticPrac {

**int** a = 10;

**public** String abc = "Data";

**protected** String xyz = "Tushar";

**protected** **void** show() {

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

System.***out***.println(456);

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

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

}

}

Now, if we want to access these methods and protected variable, it has to be done from child class. We are creating 1 child class in diff package, by below way we can access those protected methods. 1 thing we need to note here that, we have to create object of the child class only if we want to access those protected methods.

**package** com.automation.Java;

**import** com.practice.Test.staticPrac;

**public** **class** Modifier **extends** staticPrac {

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

System.***out***.println("Test Data");

staticPrac s = **new** staticPrac(); // from this reference, it won’t call

Modifier m = **new** Modifier();

m.show();

}

}

**Private Access Modifier** - Private modifier is most restricted modifier which allows accessibility within same class only. We can set this modifier to any variable, method or even constructor as well.

Just create any private method or private variable and try to access from any other class of same package or diff package. It won’t allow, will give a Compile time error.