**Java Interview Q&A**

**Q. Explain OOPs Concept? What is the different OOPs Concept?**

**Ans:-** There are 6 main OOPs concept we have in java. Those are:

1. **Classes**: Class is a user defined data type. It contains data member and member function. Class defines the properties of an object.

**Example:** Employee is an Object; It has different property we can say like – Employee Id, Employee name, Employee salary, Employee joining date etc.

1. **Objects**: Object is a real-time runtime entity in OOPs. Object contains data and code to manipulate that data. Each object is associated with class data type.
2. **Encapsulation**: Encapsulation is nothing but the wrapping up of data and methods into a single unit. That data is not directly accessible to outside world or classes, but methods can access those data and manipulate the operation. We have to specify access scope of data while creating a class.

**Example:**

**class** Employee {

**private** **int** EmpId; //only visible to its class methods

String EmpName="John";

**public** **void** employeeDetails(){

System.***out***.println("Employee Id: "+EmpId);

System.***out***.println("Employee Name: "+ EmpName);

}

}

**public** **class** TestPrograms {

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

Employee e1=**new** Employee();

e1.employeeDetails();

}

}

1. **Abstraction**: Abstraction refers to an act of representing an essential feature without including background details. Abstraction class contains data and abstract and non-abstract methods and the definition of abstract methods we must have to write into subclass.

**Example:**

**abstract** **class** Employee {

**int** EmpId; //Default access scope-subclass can see this fields

String EmpName="John";

**public** **abstract** **void** employeeDetails();

**public** **void** EnterEmpID() { //non-abstract method

EmpId=101;

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

}

}

**class** Organization **extends** Employee{

**public** **void** employeeDetails() {

System.***out***.println("Employee Id: "+EmpId);

System.***out***.println("Employee Name: "+ EmpName);

}

}

**public** **class** TestPrograms {

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

Employee e1=**new** Organization();

e1.EnterEmpID();

e1.employeeDetails();

}

}

1. **Inheritance**: Inheritance is process of deriving a new class from existing class. Parent class holds the common properties of deriving classes.
2. **Polymorphism**: Polymorphism means the ability to take more than one form. A single method name can be used to handle different set of parameters. We have compile time polymorphism and runtime polymorphism.

**Compile time polymorphism**: Method Overloading

**Runtime polymorphism**: Method Overriding

**Q. What is unchecked cast from object to type? How to resolved “Type safety” warning?**

**Ans:-** An unchecked cast warning in Java occurs when the compiler cannot verify that a cast is safe at compile time. This can happen when you are casting an object to a type that is not a super type or subtype of the object's actual type.

* To address an unchecked cast warning, you can either suppress the warning using the @SuppressWarnings("unchecked") annotation, or you can modify your code to ensure that the cast is safe.
* The ‘unchecked warnings’ is quite popular warning message in Java. However, if you insist this is an invalid warning, and there are no ways to solve it without compromising the existing program functionality. You may just use **@SuppressWarnings(“unchecked”)** to suppress unchecked warnings in Java.

1. **In Class:** If applied to class level, all the methods and members in this class will ignore the unchecked warnings message.

@SuppressWarnings("unchecked")

**public** **class** JSONFileHandling {

1. **In Method:** If applied to method level, only this method will ignore the unchecked warnings message.

@SuppressWarnings("unchecked")

**public** **void** writeToJSONFile() {

1. **In Property:** If applied to property level, only this property will ignore the unchecked warnings message.

@SuppressWarnings("unchecked")

Map<String, Object> map = (Map<String, Object>)address;

**Q. What are the different types of variable in Java?**

**Ans:-** Java variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. There are 3 types of variable in java:

1. **Local Variable:**
   * A Variable can be declared and used inside a method or a block or constructor is known as local variable.
   * Local variable can be accessible within specified block only. Outside method cannot access that variable.
   * When the variable declared and initialized a value within a block then that variable scope will be within that block only.
2. **Instance Variable:**
   * Instance variable are non-static and it can be declared inside a class and outside of any method.
   * Instance variables are declared in a class. And instance variables are created when an object of a class is created.
   * Instance variable can be accessed with object of a class.
3. **Static Variable/Class Variable:**
   * Static Variable is also known as Class variable.
   * The Variables are declared with static keyword within a class outside of any method.
   * Static Variable will create only one copy of variable it will be share among all the objects.

**Example:**

**class** Variable1{

String name; //Instance variable

**public** Variable1() {

name="John";

}

**static** String *lastName*="Smith"; //Static variable

}

**public** **class** TestPrograms {

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

**int** a=10; //Local variable

System.***out***.println("Local var: "+a);

Variable1 inVarObj = **new** Variable1();

System.***out***.println("Instance var: "+inVarObj.name);

System.***out***.println("Static var: "+ Variable1.*lastName*);

}

**Q. What are the different access modifiers in Java?**

**Ans:-** Access Modifier defines the scope of the variable, method and class within the package and outside the package. To define the scope of variable and method we have different access modifier. Those are:

1. **Default**: No keyword is required to specify default access modifier of a variable or method. The fields are visible or accessible within the current package only
2. **private:** private keyword is required to specify private access modifier of a fields. The fields are visible or accessible within the current class only. No other classes of same package and subclasses also cannot access these fields.
3. **public**: public keyword is required to specify public access modifier of a fields. The fields are visible or accessible everywhere. Anyone can access public data member and methods.
4. **protected**: protected keyword is required to specify protected access modifier of a fields. The fields are visible or accessible to all the classes and subclasses within same package as well as within the subclasses of other package.

**Q. What is Constructor? And Different types of constructor?**

**Ans:-** Constructor is called when an object is instantiated.

* Constructor is called when object of class is created.
* Constructor have same name as the class itself.
* It does not specify any return type not even void. Because it return the instance of a class.
* Default constructor provides default values of data types.

**Types:**

1. **Default Constructor:** A constructor does not have parameters is known as Default Constructor.
2. **Parameterized Constructor:** A constructor that has parameters is known as Parameterized constructor.
3. **Copy Constructor:** A Constructor copies the data from one object to other object.

**Example**:

**class** ConstructorTuto{

**int** a,b;

ConstructorTuto(){ //Default Constructor

**this**.a=10; **this**.b=20;

System.***out***.println("Default Constructor: "+**this**.a+" "+**this**.b);

}

ConstructorTuto(**int** a, **int** b){ //Parameterized Constructor

**this**.a=a; **this**.b=b;

System.***out***.println("Paramterized Constructor: "+**this**.a+" "+**this**.b);

}

ConstructorTuto(ConstructorTuto copyConstr){ //Copy Constructor

**this**.a=copyConstr.a; **this**.b=copyConstr.b;

System.***out***.println("Copy Constructor: "+**this**.a+" "+**this**.b);

}

**public** **void** print() {

System.***out***.println(a+" "+b);

}

}

**public** **class** TestProgram1 {

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

ConstructorTuto c1 = **new** ConstructorTuto(); //Called Default Constructor;

ConstructorTuto c2=**new** ConstructorTuto(30,40);//Called ParameterizedConstructor

ConstructorTuto c3 = **new** ConstructorTuto(c2); //Called Copy Constructor

c1.print();

c3.print();

}

}

**Q. What is Constructor Overloading?**

**Ans:-** When we create default constructor, parameterized constructor and copy constructor for a class then it is known as Constructor overloading. For a class we can have different parameterized constructor with different parameters.

**Q. What is Inheritance and what are the different types of inheritance?**

**Ans:-** Inheritance is the process of creating a new class from existing class. Parent class holds the common properties. By using inheritance we can achieve code reusability, method overriding, abstraction etc.

**Types:**

* 1. **Single Inheritance:** Here we can create one parent and one derived class.
  2. **Multilevel Inheritance:** A derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.
  3. **Hierarchical Inheritance:** One class serves as a superclass (base class) for more than one subclass.
  4. **Multiple** **Inheritance**: One class can have more than one superclass and inherit features from all parent classes. This can be achieving through Interface.
  5. **Hybrid Inheritance:** It is a combination of two or more inheritance type. This can be achieving through Interface.
* **‘extends’** keyword is used for inheriting one class into another.

**Example:**

**class** ConstructorTuto{

**int** a=10,b=20;

}

**class** Sum **extends** ConstructorTuto{

**int** sum = a+b;

**public** **void** print() {

System.***out***.println("Sum: "+sum);

}

}

**public** **class** TestProgram1 {

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

Sum c1 = **new** Sum();

c1.print();

}

}

**Q. What is the use of ‘*super*’ keyword?**

**Ans:-** The super keyword in Java is a reference variable that is used to refer to **parent** class objects

Super keyword used in below context:

1. When a derived class and base class has the same variable name
2. When parent and sub class having same named method, to avoid ambiguity we can use super keyword.
3. When we want to call parent class constructor then we can use super keyword.

**Example:**

**class** ConstructorTuto{

**int** a=10,b=20;

ConstructorTuto(**int** a){

**this**.a=a;

}

**public** **void** print() {

System.***out***.println("Sum Parent: "+(a+b));

}

}

**class** Sum **extends** ConstructorTuto{

**int** b=40, a=90;

Sum(**int** a){

**super**(a); //super class constructor

}

**public** **void** print() {

**super**.print(); //super class method method

**int** sum = a + **super**.b; //super class variable

System.***out***.println("Sum Child: "+sum);

}

}

**public** **class** TestProgram1 {

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

Sum c1 = **new** Sum(50);

c1.print();

}

}

Output: Sum Parent: 70

Sum Child: 110

**Q. What is the use of ‘*this*’ keyword?**

**Ans:-**

* ‘this’ is a reference variable that refers to the **current** object.
* It can be used to access instance variables and methods of the current object.
* this() used to invoke current class constructor.
* ‘this’ used to invoke current class methods and refer to current class variables.

**Example:**

**class** Sum{

**int** a,b;

Sum(){

**this**(10, 20); //constructor

}

Sum(**int** a, **int** b){

**this**.a=a;

**this**.b=b;

**int** sum = a + b; //constructor

System.***out***.println("Sum Child: "+sum);

}

}

**public** **class** TestProgram1 {

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

**new** Sum();

}

}

**Q. What is the use of ‘*final’* keyword? Can we override final method? Can we extend final class?**

**Ans:-** final keyword used in different context.

**Final Variable -> to create Constant Variable**

**Final Method -> to prevent Method Overriding**

**Final Class -> to prevent inheritance/cannot be extended to other class**

* Final keyword used to indicate that the variable, method, class cannot be modified or changed.
  1. **Final variables:** When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
* We must initialize a final variable with some value.
* The blank final variable can be initialized inside every constructor. If not initialized then compiler throws an error.
* Blank static final variable can be initialized in static block or static method.
  1. **Final methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class’s public API and should not be modified by subclasses.
  2. **Final classes:** When a class is declared as final, it cannot be extended to any other class. This is useful for classes that are intended to be used as is and should not be modified or extended.
* final variables will be participated into serialization directly by their values. Hence declaring a final variable as transient there is no use.

**Example:**

**final** **class** finalClass{ //Final Class

**public** **void** print() {

System.***out***.println("This is a final class");

}

}

**class** sum{

**final** **int** A=10; //Final Variable

**final** **void** print() { //Final Method

System.***out***.println("A: "+A);

}

}

**public** **class** TestProgram1 {

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

sum s1=**new** sum();

s1.print();

}

}

**Q. What is the use of ‘*static’* keyword?**

**Ans:-** static keyword is used to share the same variable or method among all objects of a given class.

**The *static* keyword is a non-access modifier in Java that is applicable for the following:**

1. **Static Blocks:** Static block get executed only once when the class gets loaded into memory first time. If you need to do the computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.
2. **Static Variables:** When a variable is declared as a static, that variable will be shared among all the objects of class. All instance of class will share the same static variable. Static block and variable will execute in order they are present in a program.
3. **Static Methods:** static method can access and process only static data. It allows only static variables. Static variable and methods can be accessed by using Class Name only. We cannot access it using object. Static methods won’t allow non static members (If we try to access non static member in static method compiler throws error - Cannot make a static reference to the non-static field).
4. **Nested Static Class:** A class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn’t need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

* A variable defined with static keyword is not serialized during serialization process. This variable will be loaded with current value defined in the class during deserialization.

**Example:**

**public** **class** TestProgram1 {

**static** **int** *a*=10; //Static Variable

**int** b=20; //Non-static Variable

**static** {

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

}

**public** **static** **void** print() { //static method

System.***out***.println("Static Method: A= "+*a*);

}

**static** **class** MyNestedClass {

**public** **void** disp(){

System.***out***.println("Nested Class: A = "+*a*);

}

}

}

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

System.***out***.println("Main Method");

TestProgram1.*print*();

TestProgram1.MyNestedClass n1=**new** TestProgram1.MyNestedClass();

n1.disp();

}

**Q. Can we print something on the console without creating main() method?**

**Ans:- Yes we can, by using static block.**

**Q. When to use static variables and methods?**

**Ans:-** Use the static variable for the property that is common to all objects. For example, in class Student, all students share the same college name. Use static methods for changing static variables.

**Q. Can we override static methods?**

**Ans:-** Yes. We can override static methods.

**Q. I have a parent class which contains private variable, public variable, static method and final method and if I extend parent class to child class what are the fields will be accessible and which method will be overridden?**

**Ans:-**

* + 1. First thing is that we cannot access any private fields outside parent class only that class methods can access private fields.
    2. Second thing is public variable can be accessed everywhere in class and other classes.
    3. Final method cannot be overridden
    4. Static method can be overridden
    5. And if we want to access static variable in other classes we have to make it public and we can access it using class name.

**Example:**

**class** test{

**private** **int** a=10;

**public** **static** **int** *c*=30;

**public** **int** b=20;

**public** **static** **void** print() {

System.***out***.println("Static method of parent class");

}

**public** **final** **void** print1() {

System.***out***.println("A: "+a);

}

}

**class** Test2 **extends** test{

**public** **void** print3() {

test.*c*=80;

System.***out***.println("Print3 emthod: "+test.*c*);

}

**public** **static** **void** print() {

System.***out***.println("Static method of child class");

}

}

**public** **class** TestProgram1 {

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

test.*c*=50;

System.***out***.println("Main method: "+test.*c*);

Test2 t2=**new** Test2();

t2.print1();

t2.print3();

test.*print*();

}

}

**Q. What is Abstraction? Give me a real time example? What is abstract class, abstract method in java?**

**Ans:-** Abstraction refers to the act of representing an essential feature without including background details. **Example**: Mobile Phone. We are using a mobile phone but we are not aware about the implementation of mobile functionality. We are also not aware of how they are working.

* Abstract keyword is used to define abstract classes and methods.
* Abstract class cannot be instantiated. Because an abstract methods does not have its implementation in abstract class.
* Abstract class can have both abstract and non-abstract methods.
* Abstract class can have constructor which are used to initialize a variable.
* Abstract class can contain instance variable which can be used by abstract class and subclass.
* Abstract class can implements interface.
* We can define static methods in an abstract class that can be called independently without an object.
* Abstract classes can also have final methods (methods that cannot be overridden)
* If a class contains at least one abstract method then compulsory should declare a class as abstract.
* If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level of Child class should provide implementation to the remaining abstract method.

**Example:**

**abstract** **class** AbstractClass{

**final** **int** a=10;

**int** b= 20;

**int** c;

**public** AbstractClass() {

c=40;

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

}

**final** **void** print1() {

System.***out***.println("Final non abstact method");

}

**static** **void** print2() {

System.***out***.println("Static non abstract method");

}

**abstract** **void** print3(); //If we try write a body here, compiler give an error 'Abstract methods do not specify a body'

**abstract** **void** print4();

}

**abstract** **class** subClass1 **extends** AbstractClass{

**void** print3() {

System.***out***.println("Abstract method body in subclass1");

}

}

**class** subclass2 **extends** subClass1{

**void** print4() {

System.***out***.println("Abstract method body in subclass2");

}

**void** print5() {

System.***out***.println("A: "+a);

System.***out***.println("B: "+b);

System.***out***.println("C: "+c);

}

}

**public** **class** TestProgram1 {

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

subclass2 s1 = **new** subclass2();

s1.print1();

AbstractClass.*print2*(); //static method called

s1.print3();

s1.print4();

s1.print5();

}

}

**Q. Difference between Abstraction and Encapsulation?**

**Ans:-**

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information. | Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world. |
| It solves an issue at the **design** level. | Encapsulation solves an issue at **implementation** level. |
| It focuses on the **external** lookout. | It focuses on **internal** working. |
| It can be implemented using **abstract classes** and **interfaces**. | It can be implemented by using the [**access modifiers**](https://www.javatpoint.com/access-modifiers) (private, public, protected). |
| It is the process of **gaining** information. | It is the process of **containing** the information. |
| In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities. | We use the **getters** and **setters** methods to hide the data. |
| The objects are **encapsulated** that helps to perform abstraction. | The object need not to **abstract** that result in encapsulation. |

**Q. Difference between Abstraction and Encapsulation?**

**Ans:-**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

**Q. What is Interface?**

**Ans:-** Interface is a mechanism to achieve abstraction and multiple inheritance.

* Interface contains all fields are public static final and methods are public abstract by default.
* All the methods are public and abstract only. And all the fields are public, static, and final by default.
* Interface contains abstract method that means it does not have method body.
* Class must implements an interface and must implements all the methods of an interface.
* To implement an interface to a class we have to use ***implements*** keyword.
* We cannot create an object of interface. But we can make the reference of it, that refers to implementation of class.
* Interface use only one access specifier i.e public.

**Example:-**

**interface** interfaceDemo{

**int** ***A***=10;

**public** **static** **final** **int** ***B***=20;

**void** print1(); //If we try write a body here, compiler give an error 'Abstract methods do not specify a body'

}

**class** class1 **implements** interfaceDemo{

**public** **void** print1() { //here we have to add public

System.***out***.println("Abstract method implementation");

}

**void** print2() {

System.***out***.println("A: "+***A***);

System.***out***.println("B: "+***B***);

}

}

**public** **class** TestProgram1 {

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

class1 c1 = **new** class1();

c1.print1();

c1.print2();

}

}

**Q. What is Polymorphism? And its type?**

**Ans:-** Polymorphism is an ability to take more than one form. It means single method name having different form or different parameters. There are two types of Polymorphism.

1. Compile Time Polymorphism (Static Binding) – Method Overloading
2. Run Time Polymorphism (Dynamic Binding) – Method Overriding

**Q. What is Method Overloading?**

**Ans:-** When a class having multiple method with the same name but different parameters is known as Method Overloading.

**Example:-**

**public** **class** TestProgram1 {

**public** **void** sum() {

System.***out***.println("Sum1 = "+(40+90));

}

**public** **void** sum(**int** a, **int** b) {

System.***out***.println("Sum2 = "+(a+b));

}

**public** **void** sum(**double** a, **double** b) {

System.***out***.println("Sum3 = "+(a+b));

}

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

TestProgram1 p1 = **new** TestProgram1();

p1.sum();

p1.sum(10, 20);

p1.sum(12.0, 15.0);

}

}

**Q. What is Method Overriding?**

**Ans:-** When parent class and subclass is having a same method name and same definition is known as Method Overriding. Runtime polymorphism is a process in which a function call to the overridden method is resolved at runtime.

When overridden method is called using subclass object then methods defined in the subclass is invoked and executed instead of parent class method. ‘super’ keyword used to invoke a parent class overridden method.

**Example:-**

**class** Parent{

**public** **void** print() {

System.***out***.println("Parent class method - print()");

}

}

**class** child **extends** Parent{

**public** **void** print() {

**super**.print(); //calling parent class method

System.***out***.println("Child class method - print()");

}

}

**public** **class** TestProgram1 {

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

child c1 = **new** child();

c1.print();

}

}

**Q. What is called method and calling method?**

**Ans:-** Called method means initialization of a method. Calling method means where we are using that initialized method.

**Q. What are the ways to create a String in Java?**

**Ans:-** String is a sequence of characters. We can create a string in 2 ways:

1. **String Literal:** To make java more memory efficient because no object will create if the string is present in the string constant pool.

Example:- String str=”John”;

1. **Using new Keyword:** String is created using new operator. JVM will create a new object in heap memory.

Example:- String str=new String(”John”);

**Q. What is String Class?**

**Ans:-** The String class represents character strings. String objects are **immutable**. It means String cannot be modified. String objects are immutable they can be shared. Strings are constant; their values cannot be changed after they are created.

Example:- String str1=”John”;

String str=new String(”John”);

**Q. How many String Objects will create for below code?**

**Ans:-** Whenever a String Object is created as a literal, the object will be created in the String constant pool. The string can also be declared using a **new** operator i.e. dynamically allocated. In case of String are dynamically allocated they are assigned a new memory location in the heap.

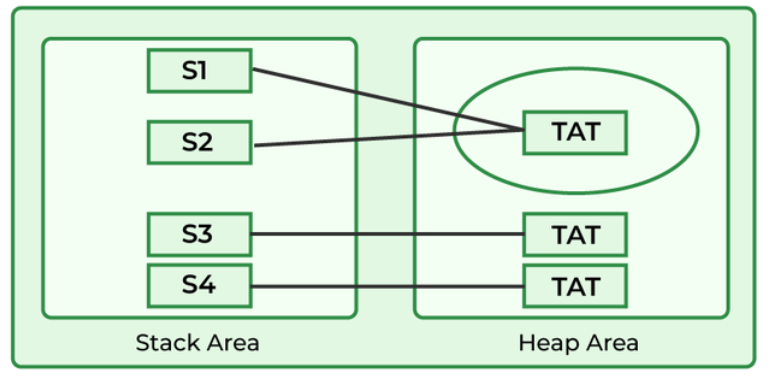
**Example:-**

String s1 = "TAT";

         String s2 = "TAT";

         String s3 = new String("TAT");

        String s4 = new String("TAT");



**Q. Why String objects are Immutable?**

**Ans:-** Java uses concept of string literal. If one of reference variable changes value of object, it will be affected by all reference variables.

**Q. What are different String Methods?**

**Ans:-** String str = “Java Question And Answer”;

* str.length():int
* str.charAt(int idx):char
* str.substr(int idx):String
* str.substr(int startIdx, int endIdx):String
* str.concat(String):String
* str.indexOf(String):int
* str.indexOf(String, int):int
* str.lastIndexOf(Stringt):int
* str.equals(Object):boolean
* str.equalsIgnoreCase(String):boolean
* str.compareTo(String):int
* str.compareToIgnoreCase(String):int
* str.toLowerCase():String
* str.toUpperCase():String
* str.trim():String
* str.replace(char old, char new):String
* str.split(String expr):String[]
* str.split(String expr, int limit):String[]
* str.isEmpty():boolean
* str.contains(charSequence):boolean

**Q. What are the different ways to compare a string?**

Ans:- There are many ways to compare two Strings in Java:

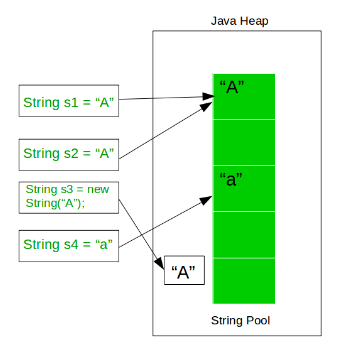
**Example:-**

String s1 = "A";

String s2 = "A";

String s3 = **new** String("A");

String s4 = "a";



1. **Using == operator**

== Operator is used to compare two or more than two **objects**. If they are referring to the same object then return true, otherwise return false.

Output:- s1==s2 -> true; s1==s3 -> false; s2==s4 -> false

1. **Using equals() operator**

equals() method is used to compare two given strings **content**. If all the contents of both the strings are same then it returns true, otherwise it returns false.

Output:- s1.equals(s2) -> true; s1.equals(s3) -> true; s2.equals(s4) -> false

1. **Using compareTo() operator**

compareTo() method is used to compare two string values based on Unicode value of character and return an int value may be less than, greater than or 0.

Output:- s1.compareTo(s2) -> 0 s1.compareTo(s3) -> 0

s2.compareTo(s4) -> -32 s4.compareTo(s1) -> 32

**Q. What is StringBuffer class in Java?**

**Ans:-** StringBuffer is a class it supports mutable strings. It allows modifying the contents of a string without creating a new object every time. StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. StringBuffer is less efficient than StringBuilder.

**Exmaple:-**

**public** **class** TestProgram1 {

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

StringBuffer s1 = **new** StringBuffer(); //way1 – initial capacity 16 length

StringBuffer s2 = **new** StringBuffer("Java"); //way2

StringBuffer s3 = **new** StringBuffer(20); //way3 – set capacity to 20 length

s1.append("Java Question");

System.***out***.println("String: "+s1);

String str = s1.toString();

System.***out***.println("String: "+str);

}

}

**Q. What is StringBuilder class in Java?**

**Ans:-** It is same as StringBuffer but only difference is that StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. StringBuilder is more efficient than StringBuffer.

**Q. What is difference between length and length()?**

**Ans:-** The length variable is applicable to an array but not for string objects whereas the length() method is applicable for string objects but not for arrays.

Example: int size = arr.length; //size of array object

int size = str.length(); //length of string object

**Q. What is an array? How to declare array in Java?**

**Ans:-** Array is a group of similar datatype of variable referred to by common name.

To find the size of an array. We have to use **arr.length** – return an integer value i.e. size of array.

**Declaration:** int arr[]; OR int[] arr;

**Instantiation:** int[] arr = new int[20];

**Initialization:** int[] arr = {10,20,40,1};

**Multi Dimentional Array:** It is arrays of arrays with each element of the array holding the reference of other array. Syntax: int[][] arr2D = new int[5][5]; //size=5\*5

Int[][][] arr3D = new int[][][];

**Q. How to copy one array to another array? What are the techniques to copy array?**

**Ans:-** Below are few techniques to copy one array to another array:

1. Copying Arrays Using Assignment Operator
2. Using Looping Construct to Copy Arrays
3. Copying Arrays Using arraycopy() method
4. Copying Arrays Using copyOfRange() method

**Example:**

**public** **class** TestProgram1 {

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

**int**[] sourceArr = {1, 2, 3, 4, 5, 6};

//1. Copying Arrays Using Assignment Operator

**int**[] destinationArr = sourceArr;

System.***out***.print("Method1: ");

**for**(**int** num : destinationArr) { //Array content display way1

System.***out***.print(num+" ");

}

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

//2. Using Looping Construct to Copy Arrays

**int**[] destiArr2= **new** **int**[sourceArr.length];

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

destiArr2[i]=sourceArr[i];

}

System.***out***.println("Method2: "+Arrays.*toString*(destiArr2)); //Array content display way2

//3. Copying Arrays Using arraycopy() method

**int**[] destiArr3= **new** **int**[sourceArr.length];

//copy whole data from one array to another

//System.arraycopy(src, srcPos, dest, destPos, length);

System.*arraycopy*(sourceArr, 0, destiArr3, 0, sourceArr.length);

System.***out***.println("Method3: "+Arrays.*toString*(destiArr3));

**int**[] destiArr4= **new** **int**[4];

//Copy partial data of source to destination array

System.*arraycopy*(sourceArr, 2, destiArr4, 0, 4);

System.***out***.println("Method3 partial: "+Arrays.*toString*(destiArr4));

//4. Copying Arrays Using copyOfRange() method

**int**[] destiArr5 = Arrays.*copyOfRange*(sourceArr, 0, sourceArr.length);

System.***out***.println("Method4: "+Arrays.*toString*(destiArr5));

**int**[] destiArr6 = Arrays.*copyOfRange*(sourceArr, 2, 5);

System.***out***.println("Method4 partial: "+Arrays.*toString*(destiArr6));

}

}

**Q. What is Jagged Array?**

**Ans:-** It is an array of array such that member arrays can be of different sizes i.e we can create 2D array but with a **variable no of columns** in each row. This is known as Jagged Array.

Syntax: int [][] arr= new int[rowsize][];

arr[0] = new int[row0\_colSize];

arr[1] = new int[row1\_colSize]; … till arr[rowsize] = new int[rowN\_colSize];

**Q. What is an Error?**

**Ans:-** Error indicates serious problem that a reasonable application should not try to catch.

**Q. What is Exception? What are different types of Exception?**

**Ans:-** An exception is an event that interrupts the normal flow of the program’s instructions. Exception occurs during the execution of a program and terminates the program. Exception can be catch and handled by the program.

Throwable is a base class of all type of Exception.

**Types of Exception:**

1. **Checked Exception/Compile Time Exception:** These Exception are checked at compile time by compiler.

Examples: FileNotFound, IOException, ClassNotFoundException etc.

1. **Unchecked Exception/Runtime Exception:** These Exception are caught at runtime. These Exception are not required to be caught or declared in throws clause. These Exception are usually caused by programming errors.

Examples: ArrayIndexOutOfBoundException, NULLPointerExeception, ArithmeticException, etc

**Q. What is try-catch block? What is the finally block? What is throw and throws?**

**Ans:-**

1. *try*: The try block contains a set of statements where an exception can occur. It means we can't use try block alone. The try block must be followed by either catch or finally.
2. *catch*: catch block is used to handle uncertain condition or used to handle exception thrown by try block.

Catch block must be written exact below the try block.

1. *throw*: throw keyword is used to explicitly throw an exception from block of code to the catch block. It will transfer control from try block to catch block.
2. *throws*: throws keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.
3. *finally*: finally block used to create common block of code which can be executed whether an Exception occur or not. It is executed after catch block if the catch block is there.

**Example:-**

**public** **class** TestProgram1 {

**public** **void** throwKeyword() {

**try** {

String s=**null**;

System.***out***.println(s.length());

**throw** **new** NullPointerException();

}

**finally** {

System.***out***.println("finally block");

}

}

**public** **void** tryCatchBlock() **throws** ArithmeticException{

**try** {

**int** div=1/0;

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

}

**catch**(Exception e) {

e.printStackTrace();

}

**finally** {

System.***out***.println("finally block");

}

}

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

TestProgram1 p1 = **new** TestProgram1();

//p1.throwKeyword();

p1.tryCatchBlock();

}

}

**Q. When finally block will not execute?**

Ans: When JVM exit while the try and catch code is being executed, then finally block may not execute. There is only one situation where finally block won’t be executed when we are using System.exit(0) method. When we are using System.exit(0) then JVM itself shutdown, hence in this case finally block won’t be executed.

**Q. When to use the finally block? Usage/Applications of finally block?**

**Ans:** **Application of finally block:** The use of finally block is resource deallocation. This means all the resources such as Network Connections, and Database Connections, which we opened in the try block, is needed to be closed so that we won’t lose our resources as opened.

**Q. What are Autoboxing, Unboxing, Type Casting, Widening, and Narrowing?**

**Ans:-**

* **Autoboxing**: It refers to the converting a primitive value into object of corresponding wrapper class.
* **Unboxing**: It refers to the converting object of wrapper class into corresponding primitive values.
* **Type casting**: To store a value of one datatype into a variable of another datatype
* **Widening / Promotion**: The process of assigning a smaller type to larger datatype
* **Narrowing**: The process of assigning a larger type to smaller datatype

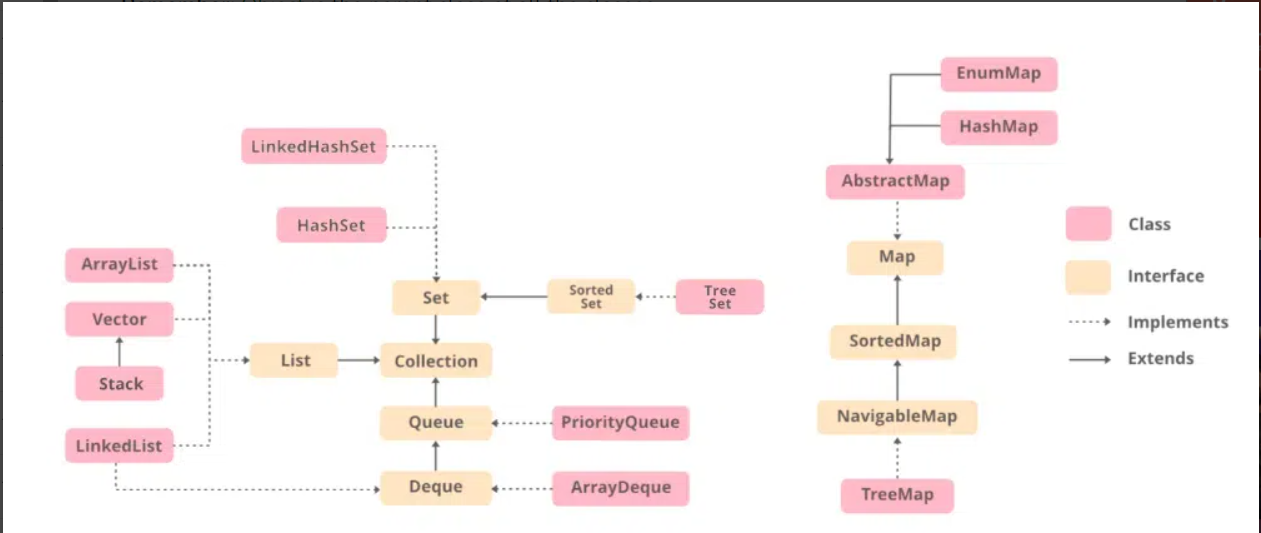
|  |  |
| --- | --- |
| Datatype | Wrapper Class |
| int | Integer |
| long | long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

**Q. What is Java Collection and its structure?**

**Ans:-** **Collections** class in Java is one of the utility classes in Java Collections Framework. The java.util package contains the Collections class in Java. ***Object*** is the parent class of all the classes.

**Interface**: Collection, Iterable, List, Set, SortedSet Queue, Deque, Map, SortedMap, Iterator

**Classes**: ArrayList, LinkedList, HashSet, LinkedHashSet, TreeSet, Vector, Stack, PriorityQueue, ArrayDeque, HashMap, TreeMap



**Q. Explain all collection interface and classes?**

**Ans:-**

1. **Iterator Interface:** Iterator interface provides the facility of iterating the elements in a forward direction only. There are only three methods in the Iterator interface. They are:

a) hasNext(): boolean-> It returns true if the iterator has more elements otherwise it returns false.

b) next(): Object-> It returns the element and moves the cursor pointer to the next element.

c) remove(): void-> It removes the last elements returned by the iterator. It is less used.

Example:-

1. **Iterable Interface:** The Iterable interface is the root interface for all the collection classes. The main functionality of this interface is to provide an iterator for collection.

We can iterate elements of java iterable by obtaining the iterator from it using iterator() method.

It contains only one abstract method. It returns the iterator over the elements of type T. To iterate collection using iterator we can use methods: hasNext(), next().

Example:-

1. **List Interface:** List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the **ordered collection** of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use:

List <data-type> list1= new ArrayList();

List <data-type> list2 = new LinkedList();

List <data-type> list3 = new Vector();

List <data-type> list4 = new Stack();

* 1. **ArrayList Class:** The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.
  2. **LinkedList Class:** LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.
  3. **Vector Class:** Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, it is synchronized and contains many methods that are not the part of Collection framework.
  4. **Stack Class:** The stack is the subclass of Vector. It implements the **last-in-first-out** data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

1. **Set Interface:** Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the **unordered** set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

Set<data-type> s1 = new HashSet<data-type>();

Set<data-type> s2 = new LinkedHashSet<data-type>();

Set<data-type> s3 = new TreeSet<data-type>();

* 1. **HashSet Class:** HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.
  2. **LinkedHashSet Class:** LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

1. **SortedSet:** SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

SortedSet<data-type> set = new TreeSet();

* 1. **TreeSet Class:** Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

1. **Queue Interface:** Queue interface maintains the **first-in-first-out** order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

Queue<String> q1 = new PriorityQueue();

* 1. **PriorityQueue Class:** The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

1. **Deque:** Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends. Deque can be instantiated as:

Deque d = new ArrayDeque();

* 1. **ArrayDeque Class:** ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

1. **Map Interface:** A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys. This interface doesn’t support duplicate keys because same key cannot have multiple mapping. However, it allows duplicate values in different key. A map is useful if there is data and we want to perform operation based on key.
   1. **HashMap Class:** It provides implementation of Map. It stores data in key-value pair. To access value we must know key. It uses a technique called Hashing
   2. **TreeMap Class:** Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.
   3. **LinkedHashMap Class:** It is similar to HashMap but only difference is that it maintains the order of insertion.
2. **HashTable:** Hashtable class implements Hashtable, which maps key to values. It store data in hashtable. In hashtable we specify an object that is used as a key and value we want to associate to that key. The key is then hashed and resulting hash code is used as the index at which value is stored within has table.

**Q. What is Comparable and Comparator in java?**

**Ans:-** identifi

**Q. What are java identifiers?**

**Ans:-** identifiers are used for identification purposes. Java Identifiers can be a class name, method name, variable name, or label.

**Q. What are the different Operators in Java?**

**Ans:-** Operators in Java are the symbols used for performing specific operations in Java.

Types of operators in Java:

1. **Arithmetic Operators**: Addition (+), Subtraction (-), Multiplication (\*), Division (/), Remainder (%)
2. **Unary Operators**: Negative (-), positive (+)
3. **Assignment Operator**:
4. Simple Assignment: = operator
5. Compound Assignment: +=, -=, \*=, /=, %=
6. **Relational Operators**: Less Than (<), Greater Than (>), Less Than equal (<=), Greater Than equal (>=), Not equal (!=), Equal to (==)
7. **Logical Operators:** AND (&&), OR (||), NOT (!)
8. **Ternary Operator:** (? and : ) Syntax-> Expression **?** True Statement **:** False Statement
9. **Bitwise Operators:** Bitwise OR (|), Bitwise AND (&), Bitwise XOR (^), Bitwise Complement (~)
10. **Shift Operators**: Signed Right Shift(>>), Unsigned Right Shift(>>>), Signed Left Shift(<<),Unsigned Left Shift (<<<)
11. **instanceof** **operator**: The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

Syntax: **public** **class** TestProgram1 {

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

TestProgram1 t1 = **new** TestProgram1();

**if**(t1 **instanceof** TestProgram1) {

System.***out***.println("t1 object is instance of TestProgram1 class");

}

**else**{

System.***out***.println("t1 object is not instance of TestProgram1 class");

}

}

}

**Q. How to use bitwise operator and shift operator?**

**Ans:-**

1. Bitwise OR (|)

This operator is a binary operator, denoted by ‘|’. It returns bit by bit OR of input values, i.e., if either of the bits is 1, it gives 1, else it shows 0.

1. Bitwise AND (&)

This operator is a binary operator, denoted by ‘&’. It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it shows 0.

1. Bitwise XOR (^)

This operator is a binary operator, denoted by ‘^’. It returns bit by bit XOR of input values, i.e., if corresponding bits are different, it gives 1, else it shows 0.

1. Bitwise Complement (~)

This operator is a unary operator, denoted by ‘~’. It returns the two’s complement representation of the input value, i.e., with all bits inverted, which mean it, makes every 0 to 1, and every 1 to 0.

1. Signed Left Shift (<<)

The left shift operator moves all bits by a given number of bits to the left.

1. Signed Right Shift (>>)

The right shift operator moves all bits by a given number of bits to the right.

1. Unsigned Right Shift (>>>)

It is the same as the signed right shift, but the vacant leftmost position is filled with 0 instead of the sign bit.

**Example:**

**public** **class** TestProgram1 {

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

**int** a=4, b=8; //a=0100 b=1000

System.***out***.println("Bitwsie OR: "+(a|b));

System.***out***.println("Bitwsie AND: "+(a&b));

System.***out***.println("Bitwsie XOR: "+(a^b));

System.***out***.println("Bitwsie NOT: "+(~a));

System.***out***.println("Bitwsie Left Shift: "+(a<<2));

System.***out***.println("Bitwsie Right Shift: "+(a>>2));

System.***out***.println("Bitwsie Unsigned Right: "+(a>>>2));

}

}

**Output:**

Bitwsie OR: 12

Bitwsie AND: 0

Bitwsie XOR: 12

Bitwsie NOT: -5

Bitwsie Left Shift: 16

Bitwsie Right Shift: 1

Bitwsie Unsigned Right: 1

**Q. What is Marker Interface in Java?**

**Ans:-** An interface that does not contain methods, fields, and constants is known as marker interface. In other words, an empty interface is known as marker interface or tag interface.

* It delivers the run-time type information about an object. It is the reason that the JVM and compiler have additional information about an object.
* Java marker interface are useful if we have information about the class and that information never changes
* It is used to logically divide the code and a good way to categorize code. It is more useful for developing API and in frameworks like Spring.
* **Built-in Marker Interface:**
  1. **Cloneable Interface:** It belongs to java.lang package. It generates replica (copy) of an object with different name. We can implement the interface in the class of which class object to be cloned. It indicates the clone() method of the Object class.
  2. **Serializable Interface:** It belongs to java.io package. If we want to make the class serializable, we must implement the Serializable interface. If a class implements the Serializable interface, we can serialize or deserialize the state of an object of that class.
  3. **Remote Interface:** It belongs to java.rmi package. It marks an object as remote that can be accessed from another machine (host). We must implement the Remote interface if we want to make an object as remote.

**Example of implementation:**

|  |  |  |
| --- | --- | --- |
| **Cloneable** | **Serializable** | **Remote** |
| **public class** TestProgram1 **implements** Cloneable | **public** **class** TestProgram1 **implements** Serializable | **public** **class** TestProgram1 **implements** Remote |

**Q. What is Serialization and Deserialization in Java?**

**Ans:-** Serialization is the process of converting the state of object to byte stream. Deserialization is the process of converting byte stream to object.

* If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.
* Only non-static data members are saved via Serialization process.
* Static data members and transient data members are not saved via Serialization process. So, if you don’t want to save value of a non-static data member then make it transient.
* Constructor of object is never called when an object is deserialized.
* The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.
* To make a Java object serializable we implement the java.io.Serializable interface.
* The **ObjectOutputStream** class contains **writeObject()** method for serializing an Object.
* The **ObjectInputStream** class contains **readObject()** method for deserializing an object.

**Advantages of Serialization:**

* To save/persist state of an object.
* To travel an object across a network.
* The primary purpose of serialization in Java is to enable the persistence and transfer of objects between different systems or across different network connections. It allows objects to be saved to a file system or sent over a network as a byte stream and then reconstructed back into objects when needed.

**Note:** Refer below question use of transient keyword and its example to get clarification on serialization and deserialization process in java.

**Q. What is the use of ‘*transient* keyword?**

**Ans:-** *transient* is a variables modifier used in serialization.

* At the time of serialization, if we don’t want to save value of a particular variable in a file, then we use transient keyword.
* When JVM comes across transient keyword, it ignores original value of the variable and save default value of that variable data type while deserialization.
* transient keyword plays an important role to meet security constraints. There are various real-life examples where we don’t want to save private data in file.
* It is good habit to use transient keyword with private confidential fields of a class during serialization.
* **transient variables:** A variable defined with transient keyword is not serialized during serialization process. This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).
* **static Variables:** A variable defined with static keyword is not serialized during serialization process. This variable will be loaded with current value defined in the class during deserialization.
* **transient and static:** Since static fields are not part of state of the object, there is no use/impact of using transient keyword with static variables. However there is no compilation error.
* **transient and final:** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as transient. There is no compile-time error though.

**Examples:**

**public** **class** TestProgram1 **implements** Serializable{

**int** a = 10, b = 20; //Normal Variables

**transient** **static** **int** *c* = 30; //transient-static variable

**static** **int** *d* = 40; //static variable

**transient** **final** **int** g = 50; //transient-final variable

**final** **int** h = 60; //final variable

**transient** **int** k = 70; //transient variable

**public** **void** printData() {

System.***out***.println("Normal Variable a = "+a);

System.***out***.println("Normal Variable b = "+b);

System.***out***.println("transient static Variable value = "+*c*);

System.***out***.println("static Variable value = "+*d*);

System.***out***.println("transient final Variable value = "+g);

System.***out***.println("final Variable value = "+h);

System.***out***.println("transient Variable value = "+k);

}

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

TestProgram1 p1 = **new** TestProgram1();

//Serialization

FileOutputStream out = **new** FileOutputStream("Seri\_Deserialization.txt");

ObjectOutputStream serializationObj = **new** ObjectOutputStream(out);

//serialized data using writeObject() method

serializationObj.writeObject(p1);

serializationObj.close();

out.close();

System.***out***.println("Before Serialization:");

p1.printData();

*d*=50;

**try** {

//De-serialization

FileInputStream inFile = **new** FileInputStream("Seri\_Deserialization.txt");

ObjectInputStream deserializationObj = **new** ObjectInputStream(inFile);

//de-serialized data using readObject() method

p1 = (TestProgram1)deserializationObj.readObject();

deserializationObj.close();

inFile.close();

System.***out***.println("\nAfter Serialization:");

p1.printData();

}

**catch**(ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**Output:**

Before Serialization:

Normal Variable a = 10

Normal Variable b = 20

transient static Variable value = 30

static Variable value = 40

transient final Variable value = 50

final Variable value = 60

transient Variable value = 70

After Serialization:

Normal Variable a = 10

Normal Variable b = 20

transient static Variable value = 30

static Variable value = **50**

transient final Variable value = 50

final Variable value = 60

transient Variable value = **0**

**Q. What is Multithreading in Java? Thread Life Cycle and its Methods?**

**Ans:-** Synchronizati

**Q. What is Deadlock? How to resolve deadlock situation?**

**Ans:-** Synchronizatn

**Q. What is the use of *‘volatile*’ keyword in java?**

**Ans:-** *volatile* keyword makes sure that the changes made in one thread are immediately reflect in other thread.

* *volatile* tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself.
* *volatile* variables have the visibility features of synchronized but not the atomicity features. The values of the volatile variable will never be cached and all writes and reads will be done to and from the main memory.

**Example:**

**Q. What is mean by thread-safe?**

**Ans:-**Thread-safe means that a method or class instance can be used by multiple threads at the same time without any problem.

**Q. What is Synchronization in Java?**

**Ans:-** 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. The synchronization is mainly used to prevent thread interference and used to prevent consistency problem.

**Q. What Singleton Class in Java?**

**Ans:-** A java singleton class is a class that can have only one object (an instance of the class) at a time. A class must ensure that only single instance should be created and single object can be used by all other classes.

* There are two forms of singleton design patterns, which are:

**Early Instantiation:** The object creation takes place at the load time.

**Lazy Instantiation:** The object creation is done according to the requirement.

* To instantiate a singleton class, we use the **getInstance()** method.
* To create the singleton class, we need to have static member of class, private constructor and static method.

**Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.

**Private constructor:** It will prevent to instantiate the Singleton class from outside the class.

**Static method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

* **Advantage:** Saves memory because object is not created at each request. Only single instance is reused again and again.
* **Usage:** Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

**Example:**

**class** singletonExample{

**private** **static** **int** *num*=10;

**public** String str;

**private** singletonExample() {

str="Hello Singleton Concept";

}

**private** **static** singletonExample *instance1*=**null**;

**public** **static** **synchronized** singletonExample getInstance() {

**if**(*instance1*==**null**)

*instance1* = **new** singletonExample();

**return** *instance1*;

}

}

**public** **class** TestProgram1{

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

singletonExample s1 = singletonExample.*getInstance*();

singletonExample s2 = singletonExample.*getInstance*();

singletonExample s3 = singletonExample.*getInstance*();

s1.str = (s1.str).toUpperCase();

System.***out***.println("String from s1: "+s1.str);

System.***out***.println("String from s2: "+s2.str);

System.***out***.println("String from s3: "+s3.str);

}

}

**Output:**

String from s1: HELLO SINGLETON CONCEPT

String from s2: HELLO SINGLETON CONCEPT

String from s3: HELLO SINGLETON CONCEPT

**Q. What are the different Design Patterns in Java?**

**Ans:-** An

**Q. What is Lambda Expression in Java?**

**Ans:-** An