**Java Concepts**

1. **When to use Static Methods??**

**You should use static methods whenever**

1. **The code in the method is not dependent on instance creation and is not using any instance variable.**
2. **A particular piece of code is to be shared by all the instance methods.**
3. **The definition of the method should not be changed or overridden.**
4. **you are writing utility classes which should not be changed.**
5. **Instance variables can not be accessed inside a static method, to access it inside a static method we have to create an object to the class … and call the instance variable with that object.**
6. **static keyword??**
7. **When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.**
8. **When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.**

**Polymorphism:-**

**Topics:**

1. **Polymorphism**
2. **Static Polymorphism**
3. **Dynamic Polymorphism**
4. **Over Loading**
5. **Over riding**
6. **Up casting**
7. **Down casting**
8. **Type Promotion**
9. **Covariant Return Type**

**Polymorphism:- The ability of an object to take many forms.**

**POLY-MORPHISM**

**MANY FORMS**

**Eg : Frog = It can live on land (Terrestrial Animal) + It can live in water(Aquatic Animal)**

**Hence frog is a polymorphic object (real time example)**

* **Whenever an object passes inheritance test then that object is a polymorphic object.**

**As every objects extends object class, every object in java is automatically polymorphic in nature.**

**So To Implement polymorphism in java we have 2 ways :**

1. **Static Polymorphism (Method OverLoading)**
2. **Dynamic Polymorphism (Method OverRiding)**

**Static Polymorphism:- Static Polymorphism is achieved through method OverLoading.**

**Method OverLoading:- Methods having same name but different parameters (Methods should be in the same class).**

**Eg:- Add( int a, int b) ; Add( int a, long b); Add(long a, long b, long c)**

**They are of 2 types:**

1. **By changing number of arguments [Add(int a, int b) ; Add(int a, int b, int c)]**
2. **By Changing the data type of the arguments [Add( int a, int b) ; Add( int a,long b)]**

* **The internal functionality of all the methods is same.**
* **Overloading is not possible by changing the return type only, it gives compile time error due to ambiguity.**
* **This happens in the same class.**
* **COMPILE TIME POLYMORPHISM**

**Example:**

**public** **class** MethodOverLoading {

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

**return** (a+b);

}

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

**return** (a+b+c);

}

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

**return** (a+b);

}

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

MethodOverLoading obj = **new** MethodOverLoading();

System.***out***.println(obj.sum(5, 7));

System.***out***.println(obj.sum(3, 4, 6));

System.***out***.println(obj.sum(2.3, 4.5));

}

}

**Type Promotion:- Smaller data type is by default promoted to bigger one if no matching data type is found.**

**Eg: Add(int a , Long b) then we call Add(4,5) 🡪 5 is promoted to Long Data type**

**[Smaller things can be adjusted in the bigger box but bigger things can’t be adjusted in the smaller box]**

**In the above example 4,5 are the actual arguments and (int a, long b) are the formal arguments**

**3 cases:**

1. **Add(int a,int b)🡪 You call Add(4,5) {Actual arguments match the formal arguments}**
2. **Add(int a, long b) 🡪 You call Add(4,5) { Actual arguments do not match the formal arguments so 5 is promoted to long}**
3. **Add(int a, long b) ; Add(long a, int b) 🡪 You call Add(4,5) then whome to promote to long is ambiguous.**

**Dynamic Poymorphism:- Dynamic Polymorphism is achieved through method OverRiding.**

**Method OverRiding :- If a Child Class has the same method defined in the Parent Class. [ Inheritance involves ]**

1. **The child class takes the same methods of parent class but changes it’s definition.**
2. **It happens between child class and parent class.**
3. **The type of parameters and the number of parameters remains the same.**
4. **Which overriden method is going to be called is decided at runtime.**
5. **RUN TIME POLYMORPHISM**

**Example:**

**class** parent {

**public** **void** show() {

System.***out***.println("In parent class");

}

}

**class** subClass1 **extends** parent {

**public** **void** show() {

System.***out***.println("In subClass1");

}

}

**class** subClass2 **extends** parent {

**public** **void** show() {

System.***out***.println("In subClass2");

}

}

**public** **class** MethodOverRiding {

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

parent obj1 = **new** subClass1();

obj1.show();

parent obj2 = **new** subClass2();

obj2.show();

}

}

**class** Bank{

**float** getRateOfInterest(){**return** 0;}

}

**class** SBI **extends** Bank{

**float** getRateOfInterest(){**return** 8.4f;}

}

**class** ICICI **extends** Bank{

**float** getRateOfInterest(){**return** 7.3f;}

}

**class** AXIS **extends** Bank{

**float** getRateOfInterest(){**return** 9.7f;}

}

**class** TestPolymorphism{

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

Bank b;

b=**new** SBI();

System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());

b=**new** ICICI();

System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());

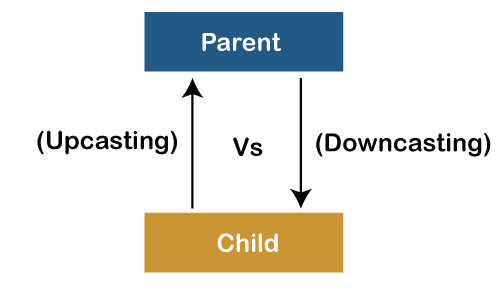
b=**new** AXIS();

System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());

}

}

**UpCasting & DownCasting :-**

****

1. **A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting.**
2. **Parent and Child objects are two types of objects. So, there are two types of typecasting possible for an object, i.e**., **Parent to Child and Child to Parent or can say Upcasting and Downcasting.**
3. **Upcasting = we typecast a child object to a parent object.**
4. **Downcasting = we typecast  a parent object to a child object.**
5. **We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.**
6. **In Upcasting, we assign a parent class reference object to the child class.**
7. **PRV = Parent Reference Variable ( reference variable holds the memory address of the object in the heap memory)**
8. **When we say = new ChildObject then the memory is created for the child class. Hence the someMethod of child class is called.**

**class** A{}

**class** B **extends** A{}

A a=**new** B();//upcasting

1. **We can’t achieve polymorphism with data members or variables ,it is only possible with methods.**
2. **With parent reference variable if you call the instance variable of child then also the instance variable of parent will be called if it is present in the parent class.**
3. **If child class does not override the method of the parent class then the method of the parent class will be called even though the object of the child is created with parent reference variable .**

**Upcasting Example:**

**class** A{

**void** PrintData() {

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

}

}

**class** B **extends** A {

**void** PrintData() {

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

}

}

**public** **class** UpCasting {

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

A obj1 = **new** B(); //  Implicitly

A obj2 = (A)**new** B(); // Explicitly

obj1.PrintData();

obj2.PrintData();

}

}

**Downcasting Example:**

//Parent class

**class** A1 {

String name;

**void** showMessage()

{

System.***out***.println("Parent method is called");

}

}

//Child class

**class** B1 **extends** A1 {

**int** age;

@Override

**void** showMessage()

{

System.***out***.println("Child method is called");

}

}

**public** **class** Downcasting{

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

{

A1 p = **new** B1();

p.name = "Shubham";

// Performing Downcasting Implicitly

//B c = new A(); // it gives compile-time error

// Performing Downcasting Explicitly

B1 c = (B1)p;

c.age = 18;

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

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

c.showMessage();

}

}

**Covariant Return Type:- It means the return type may vary during overriding(only after java5).**

**[Changing the return type of the overridden method]**

**Before java5 it was not allowed to override any method if return type is changed in the child class.**

**Before JDK 5.0, it was not possible to**[**override**](https://www.geeksforgeeks.org/overriding-in-java/)**a method by changing the return type. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be exactly same as that of parent class method. Overriding method was said to be invariant with respect to return type.**

**Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. Overriding method becomes variant with respect to return type.**

**Example:**

**class** parent{

parent Show() {

System.***out***.println("parent class");

**return** **this**; // or return new parent();

}

}

**class** child **extends** parent{

@Override

child Show() {

**super**.Show();

System.***out***.println("child class");

**return** **this**; // or return new child();

}

}

**public** **class** CoVariantReturnType {

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

child co = **new** child();

co.Show();

}

}

**Output:**

parent class

child clas

**Class:**-

1. **A class in a blue print/user defined datatype in java that describes the behavior/state that the object of its type support.**
2. **Class is a collection of objects and it doesn’t take any space in the memory**
3. **Class is a collection of variables and methods.**
4. **It represents the set of properties or methods that are common to all objects of one type.**
5. **Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.**

* **In general, class declarations can include these components, in order:**

1. **Modifiers: A class can be public or has default access (Refer**[**this**](https://www.geeksforgeeks.org/access-specifiers-for-classes-or-interfaces-in-java/)**for details).**
2. **class keyword: class keyword is used to create a class.**
3. **Class name: The name should begin with an initial letter (capitalized by convention).**
4. **Superclass(if any): The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.**
5. **Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.**
6. **Body: The class body surrounded by braces, { }.**

**There are 2 types of classes**

**1) Pre-defined Class 🡪 java developers have already defined the functionalities of these classes. We can just import them in our program and use them. Eg- Scanner, String etc.**

**2) User-defined Class 🡪 Classes which are defined by the user in there programs. Eg- class A**

**Object:- An object is an instance of a class that executes the class. Once the object is created it takes up space like other variables in the memory. It is created by using the new keyword. Once you create an object of a class, using it you can access the members of the class.**

**Syntax – Class-Name Obj-Name = new Class-Name();**

**Class-Name = Name of the class**

**Obj-name = Object reference**

**New = allocates Dynamic memory**

**Class-Name() – Constructor**

// Class Declaration

public class Dog

{

    // Instance Variables

    String name;

    String breed;

    int age;

    String color;

    // Constructor Declaration of Class

    public Dog(String name, String breed,

                   int age, String color)

    {

        this.name = name;

        this.breed = breed;

        this.age = age;

        this.color = color;}

    // method 1

    public String getName()

    {

        return name;

    }

    // method 2

    public String getBreed()

    {

        return breed;

    }

    // method 3

    public int getAge()

    {

        return age;

    }

    // method 4

    public String getColor()

    {

        return color;

    }

    @Override

    public String toString()

    {

        return("Hi my name is "+ this.getName()+

               ".\nMy breed,age and color are " +

               this.getBreed()+"," + this.getAge()+

               ","+ this.getColor());

    }

    public static void main(String[] args)

    {

        Dog tuffy = new Dog("tuffy","papillon", 5, "white");

        System.out.println(tuffy.toString());

    }

}

**Static Binding and Dynamic Binding**

**Connecting a method call to the method body is known as binding.**

**There are two types of binding**

1. **Static Binding (also known as Early Binding).**
2. **Dynamic Binding (also known as Late Binding).**

**static binding**

**When type of the object is determined at compiled time(by the compiler), it is known as static binding.**

**If there is any private, final or static method in a class, there is static binding.**

**Example:**

**class** Dog{

**private** **void** eat(){System.out.println("dog is eating...");}

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

  Dog d1=**new** Dog();

  d1.eat();

 }

}

### Dynamic binding

**When type of the object is determined at run-time, it is known as dynamic binding.**

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

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

  Animal a=**new** Dog();

  a.eat();

 }

}

**In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.**

**Abstraction :- Hiding internal implementations and showcasing or highlighting only the services**

1. **Data abstraction is the process of hiding certain details and showing only essential information to the user.**
2. **Abstraction can be achieved with either abstract classes or interfaces.**

**(Using GUI Screens and Interfaces we can implement abstraction )**

**Real Time Examples : ATM , in ATM after inserting the card , it asks for the pin ,then it checks weather it a valid pin or not ,here we don’t know the internal implementation ,like how it is getting validated , what is the Algorithm used , where is the data stored , in which language the code is written etc.**

**Advantages :**

1. **Security**
2. **Without effecting end user happily we can perform any operation.(Enhancement)**
3. **Maintainability**
4. **Modularity**

#### Points to Remember

* **An abstract class must be declared with an abstract keyword.**
* **It can have abstract and non-abstract methods.**
* **It cannot be instantiated.**
* **It can have**[**constructors**](https://www.javatpoint.com/java-constructor)**and static methods also.**
* **It can have final methods which will force the subclass not to change the body of the method.**

### Example of Abstract class that has an abstract method

**1)**

**abstract class Bike{**

**abstract void run();**

**}**

**class Honda4 extends Bike{**

**void run(){System.out.println("running safely");}**

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

**Bike obj = new Honda4();**

**obj.run();**

**}**

**}**

**2)**

**abstract class Shape{**

**abstract void draw();**

**}**

**//In real scenario, implementation is provided by others i.e. unknown by end user**

**class Rectangle extends Shape{**

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

**}**

**class Circle1 extends Shape{**

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

**}**

**//In real scenario, method is called by programmer or user**

**class TestAbstraction1{**

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

**Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method**

**s.draw();**

**}**

**3)**

**//Example of an abstract class that has abstract and non-abstract methods**

**abstract class Bike{**

**Bike(){System.out.println("bike is created");}**

**abstract void run();**

**void changeGear(){System.out.println("gear changed");}**

**}**

**//Creating a Child class which inherits Abstract class**

**class Honda extends Bike{**

**void run(){System.out.println("running safely..");}**

**}**

**//Creating a Test class which calls abstract and non-abstract methods**

**class TestAbstraction2{**

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

**Bike obj = new Honda();**

**obj.run();**

**obj.changeGear();**

**}**

**}**

# Interface in Java

1. **An interface in Java is a blueprint of a class. It has static constants and abstract methods.**
2. **It is used to achieve abstraction and multiple**[**inheritance in Java**](https://www.javatpoint.com/inheritance-in-java)
3. **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members**





**interface printable{**

**void print();**

**}**

**class A6 implements printable{**

**public void print(){System.out.println("Hello");}**

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

**A6 obj = new A6();**

**obj.print();**

**}**

**}**

**/Interface declaration: by first user**

**interface Drawable{**

**void draw();**

**}**

**//Implementation: by second user**

**class Rectangle implements Drawable{**

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

**}**

**class Circle implements Drawable{**

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

**}**

**//Using interface: by third user**

**class TestInterface1{**

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

**Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()**

**d.draw();**

**}}**

**Encapsulation:-**

1. **Encapsulation in Java** **is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.**
2. **We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.**

**Advantage of Encapsulation in Java**

1. **By providing only a setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods.**

### Read-Only class

//A Java class which has only getter methods.

**public** **class** Student{

//private data member

**private** String college="AKG";

//getter method for college

**public** String getCollege(){

**return** college;

}

}

**Now, you can't change the value of the college data member which is "AKG".**

### Write-Only class

**//A Java class which has only setter methods.**

**public class Student{**

**//private data member**

**private String college;**

**//getter method for college**

**public void setCollege(String college){**

**this.college=college;**

**}**

**}**

**Now, you can't get the value of the college, you can only change the value of college data member.**

1. It provides you the **control over the data**.
2. It is a way to achieve **data hiding** in Java
3. The encapsulate class is **easy to test**.

**Simple Example of Encapsulation in Java**

***File: Student.java***

**//A Java class which is a fully encapsulated class.**

**//It has a private data member and getter and setter methods.**

**package com.javatpoint;**

**public class Student{**

**//private data member**

**private String name;**

**//getter method for name**

**public String getName(){**

**return name;**

**}**

**//setter method for name**

**public void setName(String name){**

**this.name=name**

**}**

**}**

***File: Test.java***

**//A Java class to test the encapsulated class.**

**package com.javatpoint;**

**class Test{**

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

**//creating instance of the encapsulated class**

**Student s=new Student();**

**//setting value in the name member**

**s.setName("vijay");**

**//getting value of the name member**

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

**}**

**}**

***File: Account.java***

**//A Account class which is a fully encapsulated class.**

**//It has a private data member and getter and setter methods.**

**class Account {**

**//private data members**

**private long acc\_no;**

**private String name,email;**

**private float amount;**

**//public getter and setter methods**

**public long getAcc\_no() {**

**return acc\_no;**

**}**

**public void setAcc\_no(long acc\_no) {**

**this.acc\_no = acc\_no;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getEmail() {**

**return email;**

**}**

**public void setEmail(String email) {**

**this.email = email;**

**}**

**public float getAmount() {**

**return amount;**

**}**

**public void setAmount(float amount) {**

**this.amount = amount;**

**}**

**}**

***File: TestAccount.java***

**//A Java class to test the encapsulated class Account.**

**public class TestEncapsulation {**

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

**//creating instance of Account class**

**Account acc=new Account();**

**//setting values through setter methods**

**acc.setAcc\_no(7560504000L);**

**acc.setName("Sonoo Jaiswal");**

**acc.setEmail("sonoojaiswal@javatpoint.com");**

**acc.setAmount(500000f);**

**//getting values through getter methods**

**System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());**

**}**

**}**

**Inheritance :-** Inheritance is a mechanism that **allows one class to inherit properties** or behaviors from another class.

* **It is also called as IS-A relationship**
* **Using extends keyword we can implement inheritance**
* **Why inheritance: 1) Code Reusability 2) For**[**Method Overriding**](https://www.javatpoint.com/method-overriding-in-java)

**Example:**

**class Employee{**

**float salary=40000;**

**}**

**class Programmer extends Employee{**

**int bonus=10000;**

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

**Programmer p=new Programmer();**

**System.out.println("Programmer salary is:"+p.salary);**

**System.out.println("Bonus of Programmer is:"+p.bonus);**

**}**

## Types of inheritance in java

* **there can be three types of inheritance in java: single, multilevel and hierarchical.**



* **multiple and hybrid inheritance is supported through interface only**

## Single Inheritance Example

**class Animal{**

**void eat(){System.out.println("eating...");}**

**}**

**class Dog extends Animal{**

**void bark(){System.out.println("barking...");}**

**}**

**class TestInheritance{**

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

**Dog d=new Dog();**

**d.bark();**

**d.eat();**

**}}**

## Multilevel Inheritance Example

**class Animal{**

**void eat(){System.out.println("eating...");}**

**}**

**class Dog extends Animal{**

**void bark(){System.out.println("barking...");}**

**}**

**class BabyDog extends Dog{**

**void weep(){System.out.println("weeping...");}**

**}**

**class TestInheritance2{**

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

**BabyDog d=new BabyDog();**

**d.weep();**

**d.bark();**

**d.eat();**

**}}**

## Hierarchical Inheritance Example

**lass Animal{**

**void eat(){System.out.println("eating...");}**

**}**

**class Dog extends Animal{**

**void bark(){System.out.println("barking...");}**

**}**

**class Cat extends Animal{**

**void meow(){System.out.println("meowing...");}**

**}**

**class TestInheritance3{**

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

**Cat c=new Cat();**

**c.meow();**

**c.eat();**

**//c.bark();//C.T.Error**

**}}**