**Object Oriented Programming**

**(OOPs)**

**Object Oriented Programming and Design in Java**

Our clear objective in mind is to build a solid foundation in programming. Because solid and strong basics can lead to solve any programming problem.

And help us to write more elegant and machine and human friendly code.

Object-oriented programming is there for quite a while now but generally we seen many of the course cover only either **Concepts of OOPs with Programming Syntax** OR just basics of **Design Principles and Patterns.**

**So, it’s essential** that when we are learning **OOPs** we should also learn related things like

**Design Principles, Objected Oriented Design & Design Patterns** in detailsbecause without learning these our knowledge portfolio is not complete.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **OOPs Prerequisites** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Classes and Objects,

Variables and Methods,

Packages and Import Keywords

Access Modifiers and Non-Access Modifiers

**Classes and Objects** are the building block and foundation of programming.

**Class:** Class is a user-defined *blueprint* from which the individual objects are created OR *Design* from which objects are created.

It represents the set of **attributes** and **behavior** OR we can say **the variable and methods** simply.

Always remember in object-oriented programming we need to think everything in classes and objects.

1. **Example:-** Let’s take a use-case where **architect** is planning to build a residential colony. So, what a good architect will do first – He will start building a plan/blueprint/design and then the **Builder will start building a colony according to that architect design.**

In the same way, we can think of a class as a prototype/sketch or design of a residential colony. Which is design by that architect that contains all the details about the garden, departmental store, club house, and building and all the other things. So, based on these descriptions the builder builds the colony.

So, the builder can create any number of colonies in different cities with the same sketch / prototype / design. So, the colonies that the builder is building in different cities are called objects of the same design.

As the blueprint is divided further like in apartments another blueprint can be created for the department and multiple apartments can be created using that blueprint / design.

1. **Example 2:-** Let’s take a simple and basic example. Everything in this world is in form of objects or combination of object which has state and behavior.

Like **Dog** is an object who has its color and other physical attributes: as **State** and its walks, it barks: as **Behavior.**

**So,** we give this whole thing state and behavior as in class. In above example **Dog** is our class and class itself an object and it can be in different objects form like DOG class can be name: Jimmy Tommy, Ricky -🡪 as Objects which have **states and behaviors.**

**Like fan** also has its state like ON / OFF and behavior like spinning.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Class in Java** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Syntax: <**Modifier**> class <**Class Name**>

{

// variables; Class member

// methods;

}

e.g.

public class Fan {

private Boolean isOn;

private Boolean isOff;

public void turnOn() {

// statements.

}

Public void turnOff() {

// statements.

}

}

Location: D:\TechM\PT00640030\OOPS\My-Workspace

/\*

# How to define a class.

# How to create an object using a new operator

# How to access member function or method.

# What is a state of an object?

# We can have many classes in the java file but with only one public class.

# Always remember that the class name should be save with the main class name or the class which is "public" not any other class name like "private", "default"

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Variables, Methods & Constructor**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A class can have any of these types of variables.

* Instance variables
* Local variables
* Static variables (OR - Class variables)

Example:



**Instance Variables:** defined inside the class and outside the method. And the scope of these type of variables remains inside the class, in any method, block, or, constructor.

An instance variable initialized whenever we create an object and also, for each object a new copy of that variable is allocated.

**Local Variables:** defined inside the method, block, and constructor. The scope of such a variable is limited up to that method, block, and constructor where it’s defined. Such a variable initialized whenever the method called.

**Static Variables OR Class Variables:** These variables defined inside the class but outside the method but with **static** keyword.Such variables belong to the whole class not to the object. Therefore, the scope of static variable remains inside the class and they are initialized only once when execution start. Unlike instance variables which get initialized every time we create an object.Static variables initialized before initializing instance variable that is when the execution start. For every static object, only one copy of variable allocated to all the objects. Therefore, one copy of variable will be available for every object and due to this if any object operates on static variable the changes will be available to other objects too.

/\*

\* The role of adding **static** before any entity is to make that entity a class

\* entity. It means that adding static before **methods** and **variables** make them

\* class methods and **class variables** respectively, instead of instance methods

\* and instance variables.

\*

\* Hence, static methods and static variables can be directly access with the help

\* of class, which means that there is no need to create objects in order to

\* access static methods or static variables.

\*/

**Static methods:** When a method is declare with **static** keyword, it is known as static method. As discussed above, **any static member can be access before any objects of its class is create** and without reference to any object.

Example: // Making a static function

class MyClass

{

static void func()

{}

}

// Calling a static function

MyClass.func();

**Static main() method:** When the static keyword is added in the function definition of main() method, then it is known as static main() method.

class GfG

{

// Making a static main function

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

{}

}

**// public static void main(String[] args) - main method explanation**

**public** - access modifier

**static** - direct access without object

**void** - return type

**main** - name of function or method

**String**[] args - Java programming supports command line arguments and in command line argument we can give inputs, so in order to take input from command line we are using

"String[] args" which is string array of arguments.

our compiler will only search || **main(String[] args)**|| and this "**public static void**" we have to write by our self.

**Need of static in main() method:** Since main() method is the entry point of any Java application, hence making the main() method as static is mandatory due to following reasons:

Java **main()** method is always static, so that **compiler can call it without the creation of an object or before the creation of an object of the class**. In any Java program, the main() method is the starting point from where the compiler starts program execution. So, the compiler needs to call the main() method.

The static main() method makes it very clear for the JVM to call it for launching the Java Application. Otherwise, it would be required to specify the entry function for each Java application build, for the JVM to launch the application.

The method is static because otherwise there would be ambiguity which constructor should be called.

Example, if the class looks like this:

public class GfG{

protected GfG(int g){}

public void main(String[] args){

}

}

The JVM now enters an ambiguity state deciding whether it should call new GfG(int)? If yes, then what should it pass for g? If not, then should the JVM instantiate GfG without executing any constructor method?

There are too many edge cases and ambiguities like this for it to make sense for the JVM to have to instantiate a class before the entry point is called. That’s why main is static.

The main() method is static because it’s convenient for the JDK. Consider a scenario where it’s not mandatory to make main() method static. Then in this case, that just makes it harder on various IDEs to auto-detect the ‘launchable’ classes in a project. Hence making it a convention to make the entry method ‘main()’ as ‘public static void main(String[] args)’ is convenient.

**What if we don’t write “static” before the main method:** If we do not write “static” before the main method then, our program will be compiled without any compilation error(s). But at the time of execution, the JVM searches for the main method which is public, static, with a return type void and a String array as an argument. If such a method is not found, then an error is generated at the run time.

Example:

import java.io.\*;

class GFG {

public void main (String[] args) {

System.out.println("GFG!");

}

}

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

public static void main(String[] args)

or a JavaFX application class must extend javafx.application.Appl

Now let see these variables in coding example.

Always remember **static variable** and **static methods** always load first in memory.

We use **static** variable inside class similarly as **instance** variable.

**Static** variable is accessible within the class inside any method. To access **static** variable outside the class we use class name of static variable. Example

class Fan {

Public static int **staticVar**;

}

Public class MyClass {

System.out.println(Fan.**staticVar**); // Class name with (.) dot we access the static variable

}

We can also use the object reference.

Example: System.out.println(fan1.staticVar);

One point to remember is that no matter how many objects we create

System.out.println(Fan.staticVar);

System.out.println(myFan2.staticVar);

0

0

only one copy of static variable will be available to every object BUT

Unlike for instance variable a different copy of them created every time we create an object. For example, let’s call the method get();

myFan1.get();

In case of static variable, the change what we do on one static variable get reflected in all other variable which is operating with static variable BUT

in case of instance variable, the different copies will be created for every created object / variable.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Constructor\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Now to **initialize variables at runtime** we use constructors. Constructor is a special kind of method which does not have return type and is used to initialize objects and the name of constructors is same as its class.

Example: class MyClass {

// **Non-Parameterized Constructor below**

MyClass() {

//statements

}

// **Parameterized Constructor below**

MyClass(int a) {

// statements

}

}

**Types of constructors:** There are four types of constructors.

* **Default Constructor** – If you do not write constructor in your program / class then this constructor is create and called automatically by compiler.
* **Non-Parameterized Constructor –** A constructor with  **No-Argument** is non-parameterized.
* **Parameterized Constructor –** A constructor with an **argument** called the parameterized constructor. Parameterized constructor is use when we want to initialize a field of the class with our own values at the run time.
* **Copy Constructor -**- A copy constructor in a Java class is a constructor that creates an object using another object of the same Java class. That's helpful when we want to copy a complex object that has several fields, or when we want to make a deep copy of an existing object.

To create a copy constructor, we can first declare a constructor that takes an object of the same type as a parameter:

public class Employee {

private int id;

private String name;

public Employee(Employee employee) {

}

}

Copy

Then, we copy each field of the input object into the new instance:

public class Employee {

private int id;

private String name;

public Employee(Employee employee) {

this.id = employee.id;

this.name = employee.name;

}

}

Constructor cannot be **abstract, final, static and synchronized.**

Constructor does not have **return type.**

**Why Not Final:** Child class inherit all the members *(variables and methods*) of the superclass **except constructor**. In other words **constructor** **cannot be inherited in Java therefore, you cannot override constructor**. So writing **final** before constructor makes no sense. Therefore Java does not allow **final** keyword before constructor.

Constructors are called at the time we create object. So, let’s create an object of class MyConstructor.

Example: MyConstructor myCons = new **MyConstructor();**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Methods**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Syntax for method declaration:**

<modifier><return type> <method Name> (<Parameters>)

{

<method body>

}

**To call a method:** <object> **.**<method name>(parameter name);

Method is use to show the behavior.

Example:

public int findSum(int a, int b) {

int sum = 0;

sum = a + b;

return sum;

}

**Note**: methodName (**findSum**) and data type of parameter (**int, int)** it has above, is called as

**method signature 🡪 i.e. findSum(int, int)**

Now to call a method with object followed by dot (.)

**Example**:

package com.oops.example;

class Addition {

public int findSum(int a, int b) {

int sum = 0;

sum = a + b;

return sum;

/\*

\* As we want to return the sum then we have to change the return type of

\* method from void to int.

\*/

}

}

public class MyMethods {

public static void main(String[] args) {

Addition obj = new **Addition();**

int result = obj.findSum(10, 20);

System.out.println("The sum is: " +result);

}

}

**Static Methods:** There are static methods that are used to access the static members.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Import & Packages** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Packages:** In java package is a mechanism to **keep group of classes**, and **interfaces** and **sub-packages** which has similar functionalities at the same place.

Packages provide a folder structure to organize classes, interfaces so that one can easily search and use them.

To create a package: Once you have created a project then right click on **src** folder 🡪 New 🡪 provide package-name.

Example: **Source Folder :** Object-Oriented-Programming/src

**Name:** com.basicstrong.business

Then we create classes, interfaces and sub-packages inside / under this created package which has similar behavior.

Let’s create class name **Company** and see below that **package** statement will be a very first statement in your code / java file.

Example: Company

package com.basicstrong.business;

public class Company {

public static void main(String[] args) {

// TODO Auto-generated method stub

}

}

So, packages are basically used to avoid the naming conflicts. Like if I want to create the class with same name “Company” with different purpose then I can do so by creating another package and then creating the same class inside it.

In java there are many classes such as Date.

Note: while importing the class for Date class make sure to import the correct class. i.e. there will be two class 1) import java.util.Date; 2) import java.sql.Date

Date myDate = new Date();

System.out.println(“The Date is: ” +myDate);

If the class which we have created “Company” is not a public class then we cannot use this class in any other package except the package in which it is declared.

We should always create a class within the package because it helps us to organize our classes into folder structure and make it easy to locate.

So, package helps to improve the re-usability.

Now the question arises

**Question:** **How can we use the classes which are present in other packages?**

**Answer:** For such requirement, java provide “**import”** keyword.

**Example:** If I want to use Calendar class functionality then we should import Calendar class which is present in **java.util** package.

Example:

import java.util.Calendar;

Calendar cal = Calendar.getInstance();

System.out.println(cal.getTime()); // This will print the current date and time.

So, the import is used to import built-in and user -defined packages into our java source file so that our class can refer to class that is in another package by directly using its name.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Different Access Modifiers** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are two type of modifiers in java.

* **Access Modifiers**
* **Non-Access Modifiers**

**Modifiers** are the keywords in object-oriented languages which **sets the accessibility of classes OR methods OR any members.**

There are four access modifiers in Java.

1. Public
2. Private
3. Protected
4. Default (when there is no modifier defined)

**We can specify the behavior of our class with the modifier. Like** using **public** modifier we specify that the class can be access everywhere.

If we declare any data member OR member function with **public** modifier then we can access them in any class with its object.

**Example:**

package com.oops.accessmodifier;

public class AccessModifier {

public static void main(String[] args) {

public void enrollEmployee(String name) {

System.out.println(name+"is enrolled. Welcome!");

}

}

}

So, above public class AccessModifier and public method enrollEmployee can be access everywhere in java universe. That is, we can use them outside this package too.

Example: The above created public class AccessModifier and method enrollEmployee we will try to access in another package by creating a class called “Person”

package com.basicstrong.business;

public class Person {

public static void main(String[] args) {

}

}

Since the **AccessModifier** class is public it can be access here in Person class. So, let’s create an object of **AccessModifier** class inside above **main** method of Person class.

AccessModifier accessModi = new AccessModifier();

Now with **accessModi** object we can call the public method **enrollEmployee()** method of **AccessModifier** class.

accessModi.enrollEmployee(“Pradeep”);

enrollEmployee() method is accessible as it’s declared public in AccessModifier class.

**Note:** If the method is public but class is not public then we cannot access the method.

Classes with no modifiers said to be **Default** classes. And scope for default class is up-to package level.

**Private** members cannot be access anywhere except in the classes itself where they are declared.

So, if you want to restrict the access of any data member then you can declare them private.

Mostly the data members are always declared as **private.**

**Example:** If we declare private String **empEmailId**; in class **Company** then this won’t be access outside of its class.

And, also you cannot make **local** variable as private, public and protected since the scope of local variable is already fixed up-to method level only.

Now the **Protected** members can be access everywhere in the same package and its sub-classes. But it cannot be access in other packages except its sub-classes.

Example: Create a class called **Person** in **accessmodifiers** package and access the **protected** method **enrollEmployee**() and **extends Person class** from **AccessModifier** class.

So, now the Person class is the child class of **AccessModifier** class and present in the same package **accessmodifiers.**

So, we can use **parent reference** and **child reference** to call the method.

We can use the child class reference **Person p = new Person();** and can call the method **p.enrollEmployee(name);** and I can also use parent class reference for child class as well like for example:

AccessModifier accessModi = new Person(); 🡪 Parent class(**AccessModifier**) reference(**accessModi)** have child class(**Person)** object.

AccessModifier accessModi = new AccessModifier();

accessModi.enrollEmployee("John");

Person p = new Person();

p.enrollEmployee(name); // Child Reference

AccessModifier accessModi = new Person();

accessModi.enrollEmployee(name); // Parent Reference

Person person = new AccessModifier();

person.enrollEmployee("koti"); // Child Can’t Hold Parent Reference

I can also call a method enrollEmployee() with child reference and parent reference within same package as well given above.

So, this is how protected works in package.

Note:

**The parent class can hold reference to both the parent and child objects**. If a parent class variable holds reference of the child class, and the value is present in both the classes, in general, the reference belongs to the parent class variable.

**Person** class (Child) of **business** package throw error in the case if we try to access the protected method of **AccessModifier** class(Parent) which is insider **accessmodifiers** package even when we extend the **AccessModifier** class because we cannot access the protected members outside the package in its child class with **parent reference**.

**So, to access the protected members** ( just comment the line: // AccessModifier accessModi = new AccessModifier(); in above code ) **we can access the protected methods with child class reference only.**

So, we need to take child class (**Person)** reference (**p)**  as given above in green color and should not use Parent class reference to access the method which is outside of the package given in red color.

Important:

/\*

* So, if you want to access the **protected** method in its sub-class that is

\* present outside the package then we can access them with **only child class**

**\* reference.**

* If you want to access **protected** method in different class but within the same

\* package then you may use any(Parent and Child) reference to call those method. It can be **child OR** **parent class reference.**

\*/

**OR another way to say above is:** So protected members can be access within the package and its sub-class. These sub classes can be in same package OR outside the package. The difference is that outside the package we can access them with child reference only and inside the package any reference.

**Important: Well** we cannot make top level class as **private** and **protected.**

**Why?**

Because there is no sense. We know defining a member as private limit its scope up-to that class only.

**Example**: Let’s suppose if we allow to define a class as private then the class can only be accessible within the entity in which it’s defined which is its package.

But for that we already have “default” in java. And

**Protected classes** – If the classes can be made protected then their scope should be limited to its package and its sub-package or we can say child package.

But there is no concept of package inheritance in java so need of declaring class as private and protected. Java does not require it.

However, the **inner** class can be **private** and **protected** as these are the class entities.

And the scope will be the same as the members of the class.

So, to sum up we can say that we have four access modifiers which are used to set the visibility scope of members.

* **Non-Access Modifiers**

There are other non-access modifiers / keyword, which achieve functionality like **abstract, final,** and **static.**

**Abstract Modifier/Keyword:** Abstract modifier is a modifier which is applicable to **Classes and Methods.**

If the class is **abstract** it means the class **is not fully implemented**. And

If the method is **abstract** which means it **does not have any implementation**.

Example: Abstract class:

**abstract** <class-Name> {

// It’s not fully implemented

}

Abstract method:

abstract void <method-Name>(); // Does not have implementation

**Note:** Always remember **abstract** **method** can only be define in **abstract class.**

**OR** we can say that if we declare any method as abstract then that class must be abstract too because the implementation is not complete.

Example: Below is an example of an abstract class.

package com.basicstrong.business;

public abstract class AbstractDemo {

}

**Abstract Method:** A method without any implementation OR definition is an abstract method.

Abstract method is declare with keyword **abstract.**

**Example:** public **abstract** void myMethod(); 🡪 There is no implementation, and this is how we declare the **abstract method**.

Even if we try to provide any implementation to this method the compiler will throw the error.

Abstract method even does not specify the body { }

**Always Remember:** We cannot **declare the abstract method** in **Non-abstract class**

However, it’s not mandatory to have at least one abstract method inside abstract class.

So, abstract class is a class which has incomplete implementation.

**Important:** Abstract class cannot be instantiated and that is why we cannot create an object of abstract class.

**Important:** So how we can provide the implementation of this abstract method? 🡪 We provide implementation to this method in its **sub-class/Implementation class/ child class.**

So, the child class is responsible to provide the implementation to this abstract method.

.

Example: To provide the implementation of abstract method using child/sub-class/implementation class.

1. Create an **abstract** class Department.
2. Create two abstract method inside abstract class – Department.
3. Create the Child class/sub-class and extends it from Department class.
4. Now you can give the implementation to these unimplemented methods which are in parent abstract class (Department).
5. After this we will create the driver class where we will create the object of Child-class object and get invoke to its method.( getEmployee(): IN case workspace example).
6. Code Snippet:

package com.basicstrong.business;

public abstract class Department {

public abstract int getEmployee();

public abstract void method();

}

class ChildDepartment extends Department {

@Override

public int getEmployee() {

// TODO Auto-generated method stub

return 100;

}

@Override

public void method() {

// TODO Auto-generated method stub

System.out.println("This is implementation of parent abstract method");

}

/\*

\* After this we need to create one more driver class where we will create the object of

\* ChildDepartment class and then access the getEmployee() method.

\*/

}

package com.basicstrong.business;

public class DepartmentDriver {

public static void main(String[] args) {

ChildDepartment cdObj = new ChildDepartment();

int employee = cdObj.getEmployee();

System.out.println("Department has " +employee+" employees");

}

}

1. Note that we can also create the **concrete-method** inside abstract class as well and these methods will be available to all their child class as well.

Example: public void **attendance**() {

System.out.println(“I am present”); 🡪 Now we can directly access this **concrete** method **attendance().** However, if you want to override this implementation then you can do so.

1. So, this method attendance() will be available to every implementation class of Department without giving an implementation as it already has an implementation.
2. Therefore, abstract classes also provide the feature of code re-usability so, you can just put your method in abstract class and multiple classes can extends and use that method.
3. Like every software application we have some common functions like to **create new file** or **open a file** or **Edit a file** etc. so these **common-functions** are **concrete methods** in any abstract class and **any uncommon function** are uncommon methods which can be implemented differently by support of the classes.
4. In our case function **attendance()** is also common to its sub-class like ChildDepartment and getEmployee() method is uncommon method.
5. **So abstract class is used to achieve an abstraction in java.**
6. **So, when you want a base class which should not be INSTANTIATED and have some common functionalities for each of its derived class with different implementation then we use abstract class and abstract method.**

**Final / Static Non-access Modifier**

**Final** keyword is non-access modifier which is applicable to **classes, methods, and variables.** Which is used to finalizing the implementation of them.

**Final** variables can be initialized only once, and we can never re-assign any different object or value to it.

**Static** keyword is non-access modifier which is applicable for **variables** and **methods** and used to define the class members independent from any instances.

**Static** members are not applicable for classes.

Example: public class NonAccessModifer {

final int a = 5;

public void myMethod() {

int b = 10;

System.out.println("This is Public method");

//a = 10; 🡪 Here we will get the error as we have already declared a as final variable which has 5 and now again, we cannot assign 10 value to it.

}

}

Now if we make method final then we cannot change its implementation that is we cannot override the final method.

If one parent class has a final method and if we try to override/ change its implementation this parent class method in child class, then we are not allowed to do so.

Example:

public class MyNonAccessModifer {

Public final void myMethod1() {

System.out.println(“This is my parent class final method”);

}

}

Class MyChildClass extends MyNonAccessModifier {

Public void myMethod1() { 🡪 In this line the compiler will through the error as we cannot override since its final in parent class.

System.out.println(“This is my Child class method which is overriding parent class method”);

}

}

**Similarly –** if we make the class as **final** then that class cannot be inherited by any class.

**Example:** public final class NonAccessModifer {

System.out.println(“This is parent final class”);

}

Class ChildNonAccessModifier extends NonAccessModifer { 🡪 Throw an error since parent is final

System.out.println(“This is child class which cannot inherit the parent class since its declared **final**”)

}

**Similarly:** Talking about **static** variables – When we declare any variable as **static** there is only one copy of static variable will be access, regardlessthe number of instances of the class.

These variables **are class variables which get memory only once in the class area at the time of class loading.**

**Example:**

package com.oops.accessmodifiers;

public class MyStaticClass {

static int a = 10; /\*These static variables are class variables which get memory only once in the class area at the time of class loading.\*/

static void display() {

System.out.println("The value for static variable is: " +a);

}

}

**Note:** Here is one thing 🡪 that we can **only use static variable inside the static method.**

If I try **to access any other variable inside static method, then compiler don’t let us to do that**.

**Example**:

Static int a = 5;

int c = 0;

Static void display() { 🡪 static method

System.out.println(“The Value of c is: ” +c); 🡪 Compiler will throw the error at c because we cannot access any non-static variable inside static method. But we can access only static variable a within it like: System.out.println(“The value of a is : ” +a);🡪 This will run fine as a is static variable

}

**Example**: Below static method 🡪 **display()** in **MyStaticClass** is accessed in another class 🡪 **MyNewDriverClass** using class object reference and directly by calling the class name respectively.

package com.oops.accessmodifiers;

public class MyStaticClass {

static int a = 10; /\* These static variables are class variables which get memory only once in the

class area at the time of class loading.\*/

static void display() {

System.out.println("The value for static variable is: " +a);

}

}

package com.oops.accessmodifiers;

public class MyNewDriverClass {

public static void main(String[] args) {

MyStaticClass msObj = new MyStaticClass();

System.out.println("Below using objectName which is not recommended \t");

msObj.display(); /\* As this is the static method we are doing / calling using object

reference. That is why its showing warning that display() method is a static method and it should be access in in static way. however, both using className and objectName will work fine.

So, we should do it / call it by using class name which is MyStaticClass. So, we should not create

object this above display() method \*/

// Call the static method display() of MyStaticClass using className.

Example below

System.out.println("\n");

System.out.println("Below using className which is recommended");

MyStaticClass.display(); // --> This is fine to access the static method using class name directly instead creating object.

}

}

Similarly, along with these we have another modifier too. Like **Synchronized, transient and volatile.**

**Synchronized** keyword is use to indicate that the method can be accessed by only one thread at a time.

**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.

**Volatile** is yet another way (like synchronized, atomic wrapper) of making class **thread safe**.

**Thread safe** means that a method or class instance (variable/object) can be used by multiple threads at the same time without any problem.

**A THREAD SAFE CLASS** is a class that guarantees the internal state of the class as well as returned values from methods are correct while invoked concurrently from multiple threads. The collections classes that are thread safe in java are stack, vector, properties, Hashtable etc.

**Volatile** keyword is use to mark a Java variable as "being stored in main memory".

Every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

**Volatile** keyword is used to mark a Java variable as "being stored in main memory".

The volatile modifier tells the JVM that writes to the field should always be synchronously flushed to memory, and that reads of the field should always read from memory. This means that fields marked as volatile can be safely accessed and updated in a multi-thread application without using native or standard library-based synchronization. Similarly, reads and writes to volatile fields are atomic.

**Conclusion:**

1. Defining classes and creating objects.
2. Everything around this world revolve around classes and objects.
3. Variables, methods and constructor and their types.
4. How to create a packages and importance of packages.
5. How to define user defined and predefined classes present in other classes.
6. How to define a scope of classes and class numbers with access modifiers.
7. Non-Access modifiers which use to define the other functionality like to define a method or class incomplete we use abstract method and final to finalize the implementation.

**Object-Oriented Programming Concepts**

Object oriented programming paradigm that provides many concepts such as **abstraction**, **encapsulation,** **inheritance,** and **polymorphism**.

Here we are going to see each of these concepts in details with example.

The main concept of using OOP is to implement the real-world entity. It’s paradigm that uses objects and class as its core entities by programmer to think in terms of real-life objects.

This makes the code **cleaner, Reusable, maintainable and scalable.**

**Apart** this, we are also going to discuss the other important terms **Association, Aggregation & Composition.**

**Data Hiding**

Data hiding is a technique to hide internal data object details i.e. data members / data components that means any outsiders OR unauthenticated users should not access internal data directly.

Data which is specific to Application OR program or to any authenticated users should not be access by anyone without any validation. Only after completing all the validation process one can access the data.

For example: To access the google account we need to provide the proper **username** and **password** and only after completing this validation we can access our google account.

Even though if we are authenticated user then also, we must go through the validation process.

Let’s see how we can achieve data hiding programmatically

Let’s say we have class **BankAccount** and in this class we have data members. And **We achieve data hiding by declaring data members / attributes as private**

So, lets achieve by declaring data members “**balance”, userName, and password** as **private .**

**private double balance = 150000;**

**private String userName = "Pradeep Tiwari";**

**private String password = "password123";**

Now let’s create a function / method “**getWalletBalance()**” to access the information of wallet and this will be the public method.In this method we will pass the **userName** and **password** as parameter to get the wallet balance.

So, before returning a details(**balance**) which user asked for, we can put a validation.

Example: We can ask for **userName** and **Password** and validate them by using below condition and only then can return the balance.

Example:

public double **getWalletBanlance(String userName, String password) {**

If(userName.equals(**userName**) && password.equals(**password**)) {

return **balance**;

}

else {

return 0.0;

}

**}**

Now let’s create its object inside the main method.

BankAccount acc = new BankAccount();

System.out.println(acc.**getWalletBalance**(“Pradeep Tiwari”, “password123”));

}

So, this is how we achieve the **data hiding.**

**What we have done here is : We declared variable balance is private so, these cannot be accessed outside the class. And, we implemented the validation steps in getWalletBanlance() method before returning the data(balance) as result**.

By doing this what we are achieving is nothing but the **security of data.** By using private and some validation we are protecting our data from unauthenticated users.

So, security is the biggest advantages of data hiding.

And it’s highly recommended to declare data members as private by using private access modifiers.

Above example attached below:



**Abstraction**

Data Hiding concept above which talks about **hiding of data** and

Now comes **abstraction, which is about hiding the internal implementation.**

Abstraction in java is achieve by using **abstract classes and Interfaces.**

Before understanding abstraction, Let’s get understand the **abstract classes 🡪 abstract classes** are classes with incomplete/partial implementation.

Abstract classes **cannot be instantiated (means we cannot create & assign a value to variable i.e. private int a=10; OR AbstractClass abstractClassObj c= new AbstractClass(); )** and **we** **can have abstract and concrete method in abstract classes.**

Now comes **Interfaces 🡪 Interface** is a blueprint of the class which specify what a class must do (**Not How**). **Interface** is **Service Requirement Specification.**

**Interfaces** only have method declaration 🡪 That is interface does not have implementation of methods.

All the methods are **abstract** and **public** by default whether we are declaring them with **public** and **abstract** keyword or not.

See how we can create an interface.

Example:

<modifier> **interface** InterfaceDemo {

void method1();

void method2();

}

Given above we have **modifier** first, then **interface** keyword, with named **InterfaceDemo** and inside this interface we have two method namely **method1()** and **method2()** and

These method **method1()** and **method2()** are **abstract** and **public** by default no matter whether we are declaring these methods as public and abstract or not.

**Notice** here: **if** **every method is abstract inside interface** then the **interface** can be called as **pure abstract class.** But only until java 1.7 version because from java 1.8 version we are allowed to add **default** and **static methods** in an interface.

And from java 1.9 version also we can define a method **abstract, default, static and private** as per our requirement. So, these are the enhancement in an interface.

Example: Let’s create an interface **InterfaceDemo**

So, data members (variables) inside interface always will be **static and final.**

**i.e.** static final String a = "static constant"; 🡪

In above variable declaration we are not using any access modifiers because inside interface the declared data member would be **public** by default as per **interface** **property**. The declared members without **public access modifier** will be **public** no matters whether we are declaring it public or not.

Now comes method 🡪 we can define abstract method like.

<modifier> <return type> <method name> <parenthesis> <semicolon>

Example: void m1(); 🡪 **We do not provide any implementation as this is an abstract method.**

void m2(); 🡪 This is also an abstract method.

Until java version 1.7 we were able to declare method as **public** and **abstract** which is by **default** but **java version 1.8 onwards,** we can declare method as **static** and **default** as well.

Example: **static** void m3() { System.out.println(“static method inside interface from java 1.8 onwards”);

} 🡪 static method can have body and implementation as well from java version 1.8 onwards

**default** int m4() {

return 0;

} 🡪 default method also has body and implementation.

**So, to default and static method we provide the implementation. Static method as they are static, we can access them using interface name directly because we know that for static method, we do not need to create the object, we can access them directly by class name or interface name by those method**.

For default method, we provide the default implementation so, if we do not want to give implementation to this method in our class in which we are implementing this interface we are not required to do so, because it is a default method. So, by default the implementation of this method will be available directly to all the classes that are implementing this interface.

For these method **method1()** and **method2()** the classes that are implementing are required to give implementation of **method1()** and **method2()** because they are not having any implementation. But in case of **default** method we can provide the implementation of default so that it’s not necessary for the class implementing this interface to provide this implementation or override this interface.

With **static** method we can also use **private** 🡪 We can use **private** access modifiers with **static** **method**();

But with **default** we cannot use **private** access modifiers. And this version of **private** came from java version 1.9 onward also we can have **private** method inside the **interface.**

**Private void method5() {**

**}**

**Private** method uses inside interface only. No other class implementing this interface can use this private method or access that private method.

Why this **private** method came into concept because this **private** method as helper methods to other methods. So, if you want to provide any help to this default method or static method then we can do that using **private** method.

Let’s take an example where a class is going to implement this interface.

Class **Test** implements **InterfaceDemo {**

**@Override**

public void m1() {

// TODO Auto-generated method stub

}

**@Override**

public void m2() {

// TODO Auto-generated method stub

}

// By default, this class Test will throw an error (until we don’t override/implement the existing method in an interface) because the since it’s implementing interface – **InterfaceDemo** and it has some **abstract methods** (m1() and m2()) which needs to be implemented here in **Test** class .

Example: So, the kind of error it’s going to throw is : “The type Test must implement the inherited abstract method InterfaceDemo.m2()”

**}**

**So,** we need to override / implement those methods (m1() and m2()) inside implementing class **Test** as well.

And if you **do not want to implement all the abstract methods** of the interface what you can do is to make this class – **Test as abstract** i.e. **abstract class Test implements InterfaceDemo.** Because if the class **Test** is abstract as well then it’s totally depend on us that which selective methods we want to implement. Because abstract class can have abstract and non-abstract method as well.

**Note:** Ifwe want to implement interface using abstract class OR **As** abstract class cannot be instantiated and if we have requirement where we want to override only 2 method out of 3 of interface then 2 out of 3 methods we can override in abstract class and remaining 1 method in by creating another class which further extends abstract class and then in main method class calling by creating 3rd class object and calling all three methods as 3rd class would have all 3 method implementation since multilevel inheritance is taking place.

Example: <https://www.geeksforgeeks.org/implement-interface-using-abstract-class-in-java/>

Again, to sum up all these: Interface is a blueprint of a class which has abstract method, along with **default**, **static (Java 1.8 onwards)** and **private(Java 1.9 onwards)** methods. Abstract class and interface are quite similar to each other but these two are having some key differences between abstract class and interface are given below:

Abstract classes can have **private** methods. Interfaces cannot(till java 1.8).

Abstract classes can have **instance**, **final**, **non-final, static and non-static** variables. The interface has only **static** and **final** variables.

**Key Differences between abstract class and interface.**

**Abstract class:**

* Abstract class can extend only one class. (Since **multiple inheritance** is not allowed)
* Abstract class can have abstract methods and concrete methods.
* “**abstract”** keyword is required in declaration of abstract methods.
* Abstract class can have instance, **static final, non-final** variables with any access specifiers/modifiers.
* We can use abstract classes to provide the base for sub-classes to extends it and implement the abstract methods and use the non-abstract implemented method which are defined in abstract class.

**Interface:**

* Interface can implement any number of interfaces
* Interface can have **public** & **abstract** methods (By default) , static, default(1.8 onwards) and **private** methods(1.9 onwards).
* “**abstract**” keyword is not required in method declaration as the declared method will be abstract by default. By default, all methods are abstract.
* Interface can have only **static final** variable by default.
* Interface cannot have non-static variable.
* We can use **interface** when we want to create service requirement specification for any class that is blueprint of any class to specify that **what a class must do.**

**OOP’s Concept in Java**

**1st Pillar | Abstraction:** abstraction is an **act of representing the essentials features** **without including the background details** or explanations.

It simply means that hiding the internal implementation and just show casing the offered services.

Example: We use several mobile applications in daily life, and we know what functionality they offer. Like using WhatsApp, we can send a file to any of our friend but in this scenario of sending message or file to friend we don’t know the its technical internal implementation / internal processing will be hidden from the users. If you are a technical person, then only you will be knowing the technical internal details.

**“Hiding the internal details and showing the essentials things to the users is ABSTRACTION “**

**Important:** Don’t mess-up **Abstraction** with the **Data Hiding!**

**Data Hiding Vs Abstraction**

Data Hiding is the **hiding of data** where abstraction is **hiding the internal details** **of implementation.**

**Important:** In java we achieve **data hiding** by declaring data members as **private** and **abstraction** by using **abstract class and Interface**.

An interface in java is a **collection of abstract methods and static constants** whereas **abstract class** can contain **abstract methods and concrete methods.** And **you cannot create the object of abstract class.**

Now in terms of functionality: You can achieve the abstraction using **interface and abstract class as well.**

**Refer:** Code example **in eclipse.**

**First example** of **abstraction is using interface** and

**second example** of **abstraction is using abstract class.**

**Abstraction using interface:**

**Example 1:** Lets create a main class “**Abstraction**” with public static void main(String[] args) method and then create an **interface** “**Mobile**” with in it and one class “**Apple**” which will implement the “**Mobile”** interface .

Example:

interface Mobile {

void calling(String number);

void sendMessage(String message); /\* public & abstract method by default, which does not have any implementation \*/

}

class Apple implements Mobile /\* Now let’s implement a method defined in interface Mobile \*/

{

private ArrayList<String> contacts = new ArrayList<String>(); /\*create a contacts list to add \*/

/\*Declare a method given below which will add the numbers to the contacts list and now the implementation is complete. \*/

public void addContacts(String number)

{

contacts.add(number);

}

@Override

public void calling(String number)

{

System.out.println(“Calling…” +number);

}

@Override

public void sendMessage(String message)

{

System.out.println(“Sending…” +message);

}

Public class Abstraction {

Public static void main(String[] args) {

Apple myPhone = new Apple();

myPhone.calling(“9878665878”);

myPhone.sendMessage(“Hello, how are you?”);

myPhone.addContacts(“9835463546”);

}

If you want to send a message to number of people, then you can send by declaring a list of contacts. Declare the **contacts** list above inside the class Apple.

**So, this is how you can achieve the abstraction in java using interface.**

As you can see at interface level both the methods are abstract so, the implementation is hidden at the interface level. For any class that is implementing “**Mobile** **interface** is basically a **blueprint” 🡪 OR base design that each class is going to implement**.

At interface level everything is abstract. In this above example we have achieved nearly the **100 %** abstraction using interface since we cannot write any concrete method to interface.

**Now let’s see how we can achieve the ABSTRACTION using abstract class**

**Second Example:**

**Abstraction using abstract class:**  Let’s create one abstract class “**Mobiles”**

abstract class **Mobiles** {

ArrayList<String> contacts = new ArrayList<String>();

abstract void calling(String number);

abstract void sendMessage(String message);

public void addContact(String contact) { /\* Let’s create a concrete method addContact() method\*/

contacts.add(contact);

}

}

**Note:** if a method definition is given in the same class it’s declared / called concrete class. (Given above)

In case of interface we are taking this method “**addContact()**” inside the **implementing class** because this method was a concrete method but this time we can take this method in the **abstract class** **itself** because we can provide the implementation to this method in abstract class that is we can have concrete methods inside abstract class.

So, the class (**Samsung**) which extends this abstract class “**Mobiles**”

class **Samsung** extends **Mobiles** {

@Override

void calling(String number) {

System.out.println("Calling..." + number); // Implementation

}

@Override

void sendMessage(String message) {

System.out.println("Sending Message..." + message); // Implementation

}

Now in the main method(**public static void main(String args[])**) as well we need to create the Samsung class object as well and call the implemented methods like below:

Samsung myNewPhone = new Samsung();

myNewPhone.calling("8798094040");

myNewPhone.sendMessage("Hi there?");

We won’t call the concrete method addContact() 🡪 (**addContact(String contact))** because in abstract class it’s already implemented so no need to call.

However, if you want to override and change and modify the implementation I can do so inside the Samsung class but if I don’t do so I can access this method addContact() in Samsung class too because this Samsung class inheriting / extending Mobiles class so by default this addContact() methods also available for the Samsung class.

Abstraction helps us to reduce the complexity and used to provide the solution to real world problem.

So, the basic advantages of abstraction are **security, enhancement, maintainability, and modularity.**

**2nd Pillar | Encapsulation:** Encapsulation is a **process of grouping data members and members function** into single unit is known as **encapsulation.**

If you are creating a **class** it means you are doing an **encapsulation** because class is kind of **container** or **cell** or **capsule** which **encapsulate a set of methods, attributes, and properties** to provide its internal functionality to other classes.

So, every class in java in an example of encapsulation. Encapsulation also allow a class to change its internal implementation without hurting the overall functioning of the system. “**That idea of encapsulation is to hide how a class does it but to allow requesting what to do”**.

In that sense if you observe above **bold** statement in sense that if any component follows **data hiding and abstraction** [**Data Hiding + Abstraction = Encapsulation**] then it’s encapsulation.

For example, we will create a class **AccountInfo** which will have **balance as data member and setBalance() and getBalance() methods as member function.**

**So, if you want to set the balance then setBalance() method and if you want to show / display the balance then getBalance() method. Both will be the public method.**

**Imp: balance as data member will be private while both member function / method getBalance() and setBalance() will be public**

**There is a combination of data hiding + abstraction . So, let’s see this in example:**

1. Let’s create main method class named - **Encapsulation**
2. Create an **AccountInfo** class within **Encapsulation** outside its body.
3. Declare private double type “**balance” as** data member in **AccountInfo**.
4. And to **set** and **displaying** the **balance** data member we have to create **setters**() and **getters**() method.
5. Now to access/set/display the data member “**balance”** create “**AccountInfo”** class object in main method class **“Encapsulation”**
6. Now set the balance first and then display the balance using setters() and getters() method respectively.

Example:

package OOPS;

class AccountInfo {

private double balance; // private data member = data hiding

public double getBalance() { /\*

\* Member function to get balance or to access private data member balance

\*/

return this.balance;

}

public void setBalance(double balance) { // Member function to set the balance

this.balance = balance;

}

}

public class Encapsulation {

public static void main(String[] args) {

AccountInfo acc = new AccountInfo();

acc.setBalance(500000);

System.out.println("The balance is " + acc.getBalance());

}

}

So, as we said encapsulation is a combination of data hiding and abstraction so**, data hiding** we are achieving through by declaring **private** data member 🡪 “**private double balance**” and **abstraction** through this “**setBalance()” and “getBalance()”** are invoking but with not know what the implementation is, we know only what it gonna set/update by setBalance() and return by getBalance(). But there may be some validation/implementation before returning the balance that are necessary to be performed and also there may be some more implementation / validation before setting the value.

For example: In amazon retail shopping site we know if we click on “Buy Now” the item will get purchased but we do not know along with that internally what is going on behind the scene. What data it’s taking and returning.

So, this is the way we are achieving the **data hiding** + **abstraction** and thus it’s known as **encapsulation**.

So, talking about the advantages of using encapsulation: “**maintainability, flexibility, reusability and user would not be knowing what is going on behind the scene.”**

Now there is a term called **Tightly Encapsulated Class** as above **AccountInfo** class is having the single data member and that data member is **private** so, the class in which each and every data member is **private** that class is called **Tightly Encapsulated Class.** So, in our case **AccountInfo** is tightly encapsulated class.

**3rd Pillar | Inheritance:**  Inheritance 🡪 is an **IS-A Relationship, [ HAS-A Relationship(Association) –** in which we will discuss about **composition**, **aggregation** which are part of **Has-A Relationship**] and then the difference between the **IS-A Relationship** and **HAS-A Relationship**.

That mean when we should go for IS-A Relationship and when we should go for HAS-A Relationship.

**Relationship Between Classes:**

**Inheritance:** Inheritance is a mechanism to create a new class by deriving the old class.

Old class

extends

New Class

Ability of a new class to be created from an existing class by extending it is called inheritance.

For example : A class **IOException** is derived class of **Exception** So, here IOException is a sub-class/child class/ derived class of an Exception which inherit all the property of Exception class.

Exception class is called **Base class / Super class / parent class.**

And the keyword **extends** is used by the **sub-class** to inherit the features of **super class.**

Inheritance is related to the specialization, one of the most important relationships among object in real world is specialization which can be described as IS-A Relationship.

In inheritance the total functionality of a parent class is totally available to its child class object and this kind of relationship which is known as **IS-A Relationship**.

For example, if we say that an IT - developer is a specialize kind of **employee** of a company and similarly the project manager is also an **employee** of that company. Both of them (Developer and Project Manager) have different attribute and behavior like name, salary & working respectively but are employee of same company. Here **employee** is a **base class** and developer and Project Manager are the sub-classes.

Employee

**Developer Project Manager**

**IS-A Relationship**

So, we can say that **Developer** and **Project Manager** is in **relationship** with **Employee** because developer is an employee and Project Manager is also an employee.

**IS-A Relationship:**

**Syntax:**

Class **Developer** extends Employee {

}

class **ProjectManager** extends Employee {

}

All the java API’s are based on the concept of inheritance. Like many times we heard saying that **Object** class is **a parent class of every java class**. So**, common methods** written inside **object** class will be by default available to the each and every class with without re-writing it. So, inheritance is important since it leads to **re-usability** of code.

We have five types of inheritance:

Single Inheritance

Multilevel Inheritance

Multiple Inheritance

Hierarchical Inheritance and

Hybrid Inheritance

Let’s understand all these inheritance by code example:

1. **Single Inheritance:** In single inheritance there is only **one** **parent** and **one child** – So, this relationship is single inheritance.

A

extends

B

**Example:** Let’s create a class called “ **Inheritance”.**

Class **A** {

}

Class **B** extends **A** {

} // Level 1

public class **Inheritance** {

public static void main(String[] args) {

}

}

1. **Multilevel Inheritance:** In **Multilevel inheritance:** Ifthere is more than one levels. Class **A** is a base class for class **B** and there is another class **C** which is derived class of class **B**. This is multilevel inheritance.

A

extends

B

extends

C

**Example:**

Class **A** {

}

Class **B** extends **A** {

} // Level 1

Class **C** extends **B** {

} // Level 2

public class **Inheritance** {

public static void main(String[] args) {

}

}

1. **Multiple Inheritance:** In **Multiple Inheritance** when **a class has more than one parents** then inheritance is called multiple inheritance.

Class A

Class B

method(){} method(){}

extends extends

Class C

So, above is an example of multiple inheritance. You have class **A**, class **B** and class **C**. Here class **C** is extending the both class **A and B.** So, basically, it’s multiple parent and single one child relationship.

**BUT JAVA DOES NOT PROVIDE SUPPORT TO MULTIPLPE INHERITANCE. WHY?**  If a class **C** extends both class **A & B** then there is a chance of **Diamond Ambiguity problem:** That is if any method presents with the same name inside both the class so, this is ambiguity that which method is to call on **runtime** if it’s invoked!

This is also called the **Diamond Access Problem.**

**HOWEVER, IN INTERFACES MULTIPLE INHERITNACE IS POSSIBLE BECAUSE INTERFACES HAVE ONLY DECLARATION NOT THE IMPLEMENTATION. BECAUSE IF THERE IS SIMILAR METHOD WITH THE SAME NAME PRESENT WITH BOTH INTERFACES A & B THEN THERE IS ONLY MULTIPLE DECLARATION BUT WE HAVE NOT PROVIDED THE IMPLEMENTATION**

Interface A

Interface B

**method();** **method();**

implements implements

Class C

**Important!**

**Example**: If there is method with same name(**method();)** present in both interface **A & B** then in case if the class **C** is providing the implementation for interface **B method then at the runtime when we call the method the implementation it would take will be provided by class C of interface B method only so, there won’t be any ambiguity between choosing which interface method to implement.**

But one more thing that **after java 8** we may write **default** method in interfaces and classes can implement two or more interfaces. Let’s assume interface A and interface B both having default method with name **method();** So, both are having **same name**, both are **default** , that is why we are providing the implementation and **suppose the class C is not providing the implementation to these interface method and the class which will create the object of class C and then invoke the method then which method will be called ?**

Both have provided the method implementation to these default methods in interfaces so in that ways we have solution to resolve this in implementing class we can explicitly specify which default method is to be used.

For Example: In class C if you are overriding the method or in any class where you are creating the object you can call that method of the particular interface by using the **interface name** and then the **super keyword** and then the **method name**. Let’s see how.

Code Example:

Let first try checking in eclipse whether one class can extends more than one class or not?

Class A {

}

Class B {

}

Class C extends A,B {

} 🡪 This is not possible at all it will throw an error in eclipse .

**BUT** if we take interfaces then it’s possible for a class to implements more than one interfaces.

Interface A {

}

Interface B {

}

Class C implements A,B {

} 🡪 This is possible

**Now** we were talking about the scenario where we have two default method with same name in the two interfaces.

Code example:

package OOPS;

// Single Inheritance

class A { // So, class 'A' is a parent class of class B;

}

class B extends A { // Child class 'B' which extends parent class 'A';

} // Level 1 = From class A to Class B extends A

// Multilevel Inheritance

class C extends B {

} // Level 2 = From class B extends A \_To\_ class C extends B

**// Multiple Inheritance: More than one parent**

interface AA {

default void method() {

System.out.println("Default method of interface AA");

}

interface AAA {

default void method() {

System.out.println("Default method of interface AAA");

}

}

class BB implements AA, AAA {

@Override

public void method() {

// TODO Auto-generated method stub

AA.super.method();

}

}

/\* It's necessary to override the method

\* using interface name "**AA**" then using "**super**" keyword and then **method**

\* name. we tell the compiler that which implementation to took to avoid

\* ambiguity

\*/

/\*

\* Since we are implementing **AA** and **AAA** interface in class **BB** the method would start throwing error until we override these default methods which are with same name. So, we will override them in above if we want to pick another implementation then simply change the interface from AA to AAA. like **AAA.super.method;** \*/

}

public class Inheritance {

public static void main(String[] args) {

}

}

So, as we seen in above code example that we must override the default method in implementing class when we implement the interface AA and AAA and then **to avoid the ambiguity** we must use **interface name then super keyword and then method name together.**

**Example: @override**

**Public void Method() {**

**AA.super.method();**

**}**

**So, always remember there nothing like multiple inheritance in java for the classes.**

**We have just interfaces through which we can achieve multiple inheritance.**

1. **Hierarchical inheritance:** When a **single base class** or **parent class** has **multiple child class** then it’s called as Hierarchical inheritance.

Class A

extendsextends

Class C

Class B

**Important:**

S**o, one class can implement any number of interface and can extends only one class.**

**Example:**

**class BB extends A implements AA, AAA {**

**}**

1. **Hybrid Inheritance:** Hybrid inheritance is a combination of **multiple inheritance and multilevel inheritance**.

So, there are multiple parents and multiple children in this inheritance. This type of inheritance is also not provided by java. As we know multiple classes can extends one class, but single class cannot extend multiple classes that is why hybrid inheritance is not allowed in java.

Class A

extends extends

Class C

Class B

extends extends

Class D

Given above in Hybrid inheritance the class B and class C is extending the class A, until then everything is possible, **BUT class D is trying to extend class B and class C which is not possible.**

1. **Cyclic Inheritance: Now** comes ,cyclic inheritance when **one class inherit itself** or s**ub-class** that is called **cyclic inheritance.**

**Example:**

Class A

**extends extends**

Class B

**Given** above **class B is extending class A** and **class A is also extending class B**. So, this is creating a kind if cycle and this type of inheritance is not possible in java. Also, it’s not required because there is no use when both of class is having each other attributes and method then why to create two different classes! Just create one and put all the attributes and method in one class itself.

**IS-A Relationship** is nothing but an inheritance. So, this is one of the relationships between classes.

**Similarly,** we have another relationship between classes it’s called **Association –** and we can term that as a “**HAS-A Relationship”**

**Association | HAS-A Relation:- Association OR HAS-A relationship** is an **act of establishing a relationship between two different classes** **through their object.**

**For example:** We use object of two different classes in a class and make them interact with each other.

Class B

Class A

Association

There are two forms of **Association**:🡪 **Aggregation and Composition**

**Aggregation :** is a special case of Association and **Association is said to be Aggregation if both the entities that are associated can exit independently**.

Teacher

Employee

College

Office

**HAS - A**

These two entities are associated in a way that one contains other in **HAS-A Relationship.**

**Example:-** In above example **“College has Teacher” OR Office has an Employee”**

So, here **College** is a separate entity and so as **Teacher** and **Office** is also a separate entity so as **Employee** and destroying one entity will not impact / affect other. Like without college teacher may exist in different environment and same for office employee – An employee can also work in some other office/environment(WFH) if the office entity gets ended.

Let’s see the code example: Below we have main class **Aggregation** and we will create another two class **College** and **Teacher.** This **College** class has **Teacher** class entities like **name and teacher**

Then we will create a public constructor of **College** class and pass the teacher class entities name and teacher as in parameter to it.

**Then** we will create a getter method / **getTeachers()** propertywhich will return teacher entities.

Now let’s override the **toString**() method of **object** class and return the **teacher** and **subject.**

Public String toString() {

Return “[Teacher : “+name+ “, Subject : ”+subject+” ]”;

}

Now we have this class called **Aggregation** and inside this class we have main method and here we will create the object of **Teacher** class and then will add that teacher object to the list and then we will pass that list along with the name of the college to the college object.

Let’s see how 🡪

So, first we have created the list of teachers type separately and then we have passed it into **College** class.

package OOPS;

import java.util.ArrayList;

import java.util.List;

class College {

String name;

private List<Teacher> teacher;

public College(String name, List<Teacher> teacher) {

this.name = name;

this.teacher = teacher;

}

public List<Teacher> getTeachers() {

return teacher;

}

}

class Teacher {

String name;

String subject;

public Teacher(String name, String subject) {

this.name=name;

this.subject=subject;

}

@Override

public String toString() {

return "[Teacher : "+name+", Subject : "+subject+"]";

}

}

public class Aggregation {

public static void main(String[] args) {

List<Teacher> teachers = new ArrayList<Teacher>();

teachers.add(new Teacher("Pradeep", "Java"));

teachers.add(new Teacher("Tiwari", "OOPS"));

System.out.println("The Teachers and subjects are: " +teachers);

College college = new College("GEU", teachers); /\*

\* Here we are passing the "Teacher" class object --> **teachers** to the College class. So hence it's called aggregation.

\*/

}

}

So, in this program we have passed object of teacher class to the College class. So, this type of association is **Aggregation.**

And **same is called HAS-A Relationship** andWe can say it’s like **College** has **Teacher.**

**So, IS-A Relationship is all about inheritance where we inherit all the functionality of parent class and make available to the child class. Where HAS-A Relationship** OR **Association**  we create an object of some other class like we are doing in this class and we are passing the object of **Teacher** class in **College** class. So, this is association and this above example is of **Aggregation🡪 where we are externally providing the object “teachers” to this College class. So, aggregation is where one entity contain other one, but both can survive independently. So, without this College class Teacher class can survive. independently.**

**Remember: Code re-use is best achieved by aggregation.**

**Now** another form of **Association that is 🡪 Composition**

**Composition:** Composition is **a stronger form of Association in which two entities are highly dependent on each other**.

Example: – Consider the case of car having engine OR human having a heart. – A car cannot exist without an engine and same the human cannot exist without the heart.

In **Composition** the bond between two entities are too stronger and one entity cannot exist without other.

**Code Example:** Below we created the main method class – **Composition. Then** we created another **House class and Kitchen class within same source file.** Then we created the private object **kitchen** of the **Kitchen class in House class.**

We created private object **food** in Kitchen class and **getter()** 🡪**getFood**() and **setter** 🡪 **setFood**() property of it.

Then we create one public constructor 🡪 public House() in **House** class and then initialize the **kitchen** object 🡪 **kitchen = new Kitchen()** and set the food **pizza within it 🡪 kitchen.setFood(“Pizza”).**  We have also declared the getFood() method and within it we initialize kitchen object and call getFood() method as in return statement.

**Example:**

package OOPS;

class House {

private Kitchen kitchen;

public House() {

kitchen = new Kitchen();

kitchen.setFood("Pizza");

}

public String getFood() {

kitchen = new Kitchen();

return kitchen.getFood();

}

}

class Kitchen {

private String food;

public String getFood() {

return food;

}

public void setFood(String food) {

this.food = food;

}

}

public class Composition {

public static void main(String[] args) {

House house = new House();

System.out.println(house.getFood());

}

}

So, basically, we are creating the object of Kitchen class in House class and through constructor we are initializing it and setting and getting the value within it.

So, above is composition as there is no existence of **kitchen** object without **House object.** When you create the house object it creates kitchen object and when you destroy the house object then kitchen object will destroy automatically.

So, we can say **House has Kitchen and Kitchen is a part of House** and this shows the strong composition between that since kitchen cannot exist individually.

So Now let’s create the object inside the main method.

House house = new House() 🡪 and we are not passing anything within it and even then, by not passing it will create the object of kitchen and when this house object gets destroyed then the kitchen object will also get destroyed automatically.

**So, what is the main difference between Aggregation and Composition?**

**Well to answer this: - In Aggregation the contained object can survive without container object and in Composition the contained object cannot survive without its container object.**

**4th Pillar | Polymorphism:**  **Poly + morphism** 🡪 **Many + Forms 🡪 Polymorphism** is an ability of something to take many forms.

**In Java we can say that polymorphism is an ability of an object to take many forms.**

So, first let’s see about the **method signature** = **<method\_name>(<argument\_type)**

**Method Signature:** Method signature is an actually the **method name** followed by the **argument types** that it has.

Example:

Class MyClass {

public void **display**(**String info**) { 🡪 Highlighted in bold is a method signature.

System.out.println(“info”);

}

MyClass obj = new MyClass();

obj.display(“info”);

}

**Remember** that **return type** is not included in method signature.

**Who uses method signature?** 🡪 Method signature is used by the **compiler** to resolve the method call. When we call any method with object it searches for that method by its method signature.

Like in above example class **MyClass** has method - > public void **display**(String info) and we are calling display method with its class object **obj**.

**Note:** We cannot write more than one method with same method signature in class.

Even though if we try to write method with different return type then also it will through the error saying that this “method String display is duplicate in type MyClass”

Example: public String display(String info) {

}

**Now** Let’s discuss the first way that java provide to achieve polymorphism it’s **Method Overloading.**

**Method Overloading:** is a feature that allows class to have more than one method with same name but having different argument list. That is, we can have different method in a class with same name but different method signature.

For example: In Below class we have two method with the same name but with two different arguments list OR method signature. And when we call these methods the compiler will resolve the method call with basis of argument we pass. If we pass the value integer, then display(int) will be called and if we pass the string value then display(String) will be called as well.

class MyOverloading {

Public void display(String info) {

System.out.println(“info”);

}

Public String display(int info) {

System.out.println(info);

}

MyOverloading obj = new MyOverloding();

Obj.display(200); 🡪 display(int) will be called by compiler; and if we pass

Obj.display(“Pradeep”); 🡪 display(String) will be called by the compiler.

So, since compiler will resolve the method call that is it check which method going to be call at compile time there this is also called **Compile Time Polymorphism OR early binding.**

**What do we achieve by method overloading / what are its advantages?**

* We don’t have to create and remember different names for functions doing the same thing.

Let’s take an example:

package OOPS;

class Addition {

public void sum(int a, int b) {

System.out.println("Sum is " + (a + b));

}

public void sum(double a, double b) {

System.out.println("Sum is " + (a + b));

}

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

System.out.println("Sum is " + (a + b + c));

}

}

public class Polymorphism {

public static void main(String[] args) {

Addition obj = new Addition();

obj.sum(10.50, 20.20); 🡪 compiler will call 🡪 **sum(double a, double b)** method.

obj.sum(10, 20); 🡪 compiler will call 🡪 **sum(int a, int b)** method.

obj.sum(10, 10, 30); 🡪 compiler will call 🡪 **sum(int a, int b, int c)** method.

/\*

\* Method selection will be chosen by compiler based on the argument we passed

\* depend on type and number of arguments.

\*/

}

}

**Remember the case:** If we don’t have any such method with float value in argument and if we call that method with object then compiler won’t through any error because float value passed / called in method get promoted to double value.

**Example**: byte 🡪 short 🡪 int 🡪 long 🡪 float 🡪 double: - get promoted in this hierarchy from Small 🡪 Big Data type.

Also, **char** can be promoted to **int. Kind** of implicit type conversion by compiler.

**Remember, you cannot perform reverse of the above hierarchy like double 🡪 to Float and so.**

**So,** before running the code compiler check the method call and resolve it early at compile time that is why it’s call **early binding.**

If in case java would not have supported the **method overloading** then we would have created the different method with different name like as above **sum(),** **sum1(), sum2() and sum3()** like so.

**Also, remember that by changing the return type of method we do not perform method overloading because the return type is not considered in method signature and compiler resolve the method call by its method signature and return type of method is not included.**

Suppose in big project and application we have written an API that brings the data to the user on the basis of input. Suppose we have method “**getDetails**(int **id**, String **name**)” and later suppose the requirement changes the newer version of the application or some new code and this method is not required, and some enhance version of this method is required and programmer need to change the logic for the existing application and suppose this method is used by other services so we won’t change / remove this particular as this is used by other services.

THEN 🡪 we should declare another method with the same name as:- **getDetails**(int **id**, String **name**, String **password**) so this is the method with newly added argument password in the parameter. So, this is how we can overload the method and problem will get resolved.

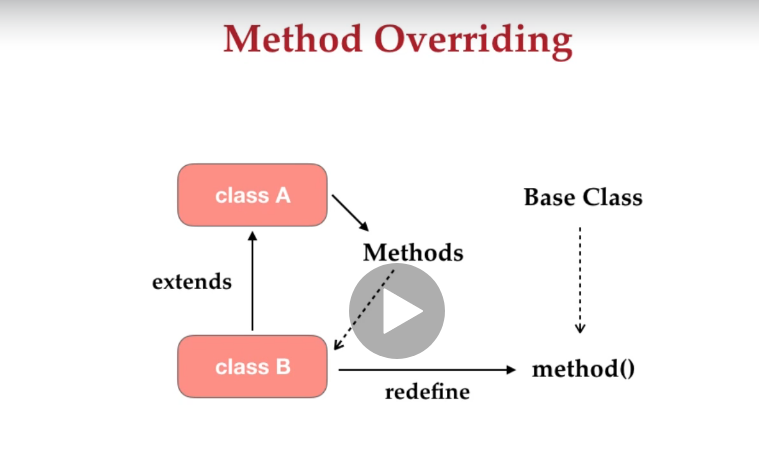
**So, here the use / benefit of overloading the method OR application of overloading the method.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Method Overriding: 🡪** We know when we extend any class then all of its **methods** get available to its sub-class and we can use them as per our requirement. If we want to implement different implementation to those methods, then we can do so by **method overriding.**

**Method Overriding:** allows us to re-define the implementation of methodswhich is provided by the base class.

**So, Method Overriding** is a feature that allow a sub-class / child class to provide the implementation to the method that is provided by its the base class.

****

**For example:** Let’s suppose we have class Base and one method show() given as below and one Derived class which is extending a Base class.

Public class Base {

Public void show() {

// Implementation

}

}

Class Derived extends Base {

Public void show() {

// New implementation.

}

Now if we want to provide the new implementation to method show () then we can do by overriding the method. Now the question arises – how to call these methods? And who will decide the which version is overridden method is called!

Unlike Method Overloading and Method Overriding JVM does the job of method resolution.

When we call the method then basis of object type it determine which method is going to be called.

Like: We are creating an object like this **Base base = new Base();**

**base.show(); Here 🡪** in this we are calling the object of Base with show method which is actually the parent class So, JVM will call the show() method of parent Base class.

AND

Next calling method would be of Derived class like:

**Derived derived = new Derived();**

**derived.show();**

}

Since derived class is inheriting the implementation from parent so, in this case we are calling method with derived class object. So, **JVM** will first check if this derived class has provided any implementation to this(it’s) method and if yes then this will call show method of derived class object.

Otherwise if derived class is not overriding the implementation provided by base class then it will call the method present in the base class.

Now the case comes of when we have parent type of reference but child type of object. Like

Base obj = new Derived(); 🡪 Here we are using PARENT-TYPE REFERENCE FOR CHILD OBJECT( Base obj) and then we are invoking a show method. In this scenario the reference is of Base type, BUT object is still of child 🡪 derived type.

And since the basis of runtime object the method resolution will take place hence this time the show() method of derived class will be called**.**

**Base obj = new Derived(); 🡪** Always remember we are creating the object of derived class with new keyword that is why the method(show() method of derived class) that is going to be invoked at runtime will be that which is present in the derived class.

**Now** there is one more case which is having **child reference and parent object**

**Derived obj = new Base(); 🡪 This is NOT valid at all.**

**Since, we cannot have parent object in child reference.**

Always remember that **it’s types of referred object** (🡪 new Derived();) **determine which version of an overridden method will be executed. Not the type of reference variable. And since the runtime object determine which method is going to be called hence method overriding is also called as Runtime Polymorphism OR late binding.**

**Reference Type Object Type**

**Base obj = new Base();**

**Parent type reference Child type object : Resultant it will call the child method()**

**Base obj = new Child();**

Now let’s take a code example:

package OOPS;

class Vehicle {

public void run() {

System.out.println("Vehicle is running...");

}

}

class Car extends Vehicle {

public void run() {

System.out.println("Car is running...");

}

}

public class Overriding {

public static void main(String[] args) {

Vehicle v = new Vehicle(); // Parent Case: Parent Type Reference and Parent Type Object.

v.run();

Car c = new Car(); // Child Case: Child Type Reference and Child Type Object.

c.run();

Vehicle vObj = new Car(); // Case of Parent Type Reference and Child Class Object.

vObj.run();

//Car cObj = new vehicle(); // Case of Child Type Reference and Parent Type Object - Which is not possible.

//cObj.run();

}

}

So, here **Car** class is extending / inheriting the parent class **Vehicle** and its functionality BUT it’s **overriding** the **run()** method that its parent is implementing. So, when I run this 🡪 Child c = new Child(); then it will print the “Car is running…”

If there is no **run()** method inside the child class **Car** and then if we are extending the parent class **Vehicle,** then it will print the parent class **Vehicle** implementation. Like: “Vehicle is running…”

Now we will take the scenario where **parent class reference** and **child class object.**

i.e. 🡪 Vehicle vObj = new Car(); and output would be 🡪 “ Car is running…” 🡪 Because the object this parent class reference **vObj** is pulling of child class **new Car();** So, it will print that implementation.

Car cObj = new Vehicle(); 🡪 This is not possible in java. Because **child type reference cannot hold the parent object** so, it will throw the error at this line.

So, this is method overriding, we can provide the multiple implementation to the same method and can call any of them as per our requirements.

The best example for this is **toString()** method of **object** class. **toString**() 🡪 It returns the string representation of object and we can override it and provide our own implementation to get the desired output.

**Some more important points about** **method overriding.**

* The first one is you cannot have more and provide less to any overridden method 🡪 This says we cannot decrease the visibility of overridden method.

For example: if overridden method is **public in parent class** then we **cannot set its visibility** to **protected or private**. If we do so, then we will get compile time error saying cannot produce the visibility of inherited method from the base class.

**Base class:** public void method() {

}

**Derived class:** Protected void method() { } AND Private void method() {}

However, you can provide more to it and that is if this method is protected in base class then we can increase the visibility to public in the derived class.

* We cannot override the **final** methods and **static** methods. If we want our implementation final and so, that no one can change it then **we can define own methods as** **final**.

Example: Public **final** void method() {

System.out.println(“This is the final method implementation.”);

}

Declare the method as **final** if you want your implementation to be final so, no one can change it.

And even if you try to override the **final** method then compiler won’t let us to do so.

Now comes the **static** methods: If you want to hide implementation from the derived class then we may define our method as static and if we try to override the static method then we won’t get any error because it will be considered as new static method in the derived class.

Example: public static void method() {

System.out.println(“This is the static method implementation ”);

}

**Declare your methods as static if you want to hide your implementation from the derived class.**

Derived class remain unaware of the static method present in the parent class. So, static method created with the same name, same method signature as of in the base class will be just considered as new method in the derived class. So, we cannot override the **static and final methods.**

Also remember that we cannot override the **private methods** as well. The reason is obvious that **private method scope outer than outside the class where they are defined.**

Example: **private** void method() {

}

* Now next is the concept of **invoking the base class implementation from sub-class:** We can call the base class implementations in the overriding method using **super** keyword.

Example: **Base class:** public void method()

{

System.out.println(“Base class Method”);

}

**Derived class:** public void method()

**{**

**Super**.method();

**}**

**Summary**

* Data Hiding
* Interfaces and abstract classes
* Abstraction
* Encapsulation
* Inheritance (IS-A Relationship)
* Association (HAS-A Relationship)-Aggregation, Composition
* Polymorphism