**Introduction To Java - Day 03/04**

**Object oriented Programming Vs Procedural Programming**

Already Covered in Day 01 pg.3

As discussed earlier Object Oriented Programming is a methodology where we program the real life systems by representing every entity that makes up that system, in the form of classes which become the building blocks of the automated system.

Classes act as a blueprint, design & idea representing any real life entity. For example when we are programming for an automobile manufacturer, Vehicle is a real time entity, now in the program we can represent that entity in the form of a class.

We are actually classifying all the characteristics & behavioral aspects of a vehicle in a class called Vehicle.

So the characteristics could be the engineCapacity, chasisNumber , vehicleNumber , color , size, weight,price etc.

The Behaviors could be move, transport things , carry people etc.

So classes hold the characteristics through variables and behaviors/functionalities are represented through methods .

**Objects of a class or Instance of a class:**

As discussed earlier , the classes are abstract or not physically available but an idea or design, which can be exhibited through the instances/objects of a class.

So to experience the features and the behaviors of a class we need to instantiate the class,

When we have a class Vehicle, we can have Maruti Esteem with the following numbers

KA05 w 3234

KA06 w 3235

KA07 w 3237 etc as instances of a class Vehicle

We need to understand that all these vehicles will share the common behavior and characteristics, but the values might be different, for example each of these vehicles will have its own unique VehicleNumber, its own engineCapacity etc.

Similarly if we have class called “SoftwareEngineer” where we have encapsulated the characteristics & behaviors of a SoftwareEngineer

We can have the following instances of the SoftwareEngineers.

Prashanth

Sushanth

Rajesh

KiranKumar are all SoftwareEngineers , they are all programmatically termed as instances / objects of the class SoftwareEngineer

Each of them will share common characteristics like empCode,department, salary, designation, but the values for these characteristics will be different for each SoftwareEngineer.

Therefore the building blocks of Object Oriented Programming are classes.

The main pillar features of Object Oriented Programming are:

Inheritance

Polymorphism

Encapsulation

Abstraction

Data Security

Syntax to create a class would be

class SoftwareEngineer

{  
 int empCode;

String engineerName;

String designation;

String department;

float salary;

public void createSoftware()

{

}

public void solveProblem()

{

}

}

Syntax to create an instance would be:

SoftwareEngineer sumanGupta = new SoftwareEngineer();

SoftwareEngineer kiranKumar = new SoftwareEngineer();

SoftwareEngineer s1 = new SoftwareEngineer();

Classes have methods and attributes.

Methods or Functions represent the behavior and attributes represent the characteristics

Attributes are represented through variables.

**Access Modifiers (Specifiers):**  
When we create classes which encapsulate the characteristics (attributes) & behavior (methods) in it, we can restrict the access

These were called “Access Specifiers” earlier.

The allowed access modifiers in java are

1. private
2. public
3. protected
4. default (which is not a keyword , when no access modifier is specified it is considered default)

**private:** private identifiers can be accessible only within the class where it has been declared.

For eg.

class Employee

{  
 String employeeId; // default access modifiers

private String employeeName;

public int employeeSalary;

protected float incomeTax;

}

Identifiers be it a variable, method or an object which are declared **private** can be accessible only within that class where it has been declared.

Identifiers that are declared **public** can be accessible within the package , outside the class where it has been declared, as well as outside the package .

Identifiers that are declared protected can be accessible only by the derived classes, outside the package, though within the package , they are accessible by other classes.

Identifiers that are declared without any access modifiers i.e which are considered **default** are accessible anywhere within the package of that class , where it has been declared.

**Methods :** Method or a function is a group of instructions that are defined to do a particular task, which has a name, return type and parameters along with access modifiers. In some languages they are called **“functions”.**

Methods provide us the feature of reusability, once written they can be called anywhere else in the program(By importing libraries even outside the program).

For example :

public boolean checkIfDOJIsValid(Date dateOfJoining)

{

//Instructions

return true; //or false

}

Methods can have different signatures

**For example : Method with parameters and void return type.**

class PayRoll

{

public void calculateSalary(int basic, int hra,int cca)

{

Int grossSalary = basic + hra + cca;

System.out.println(“The Gross Salary is “+grossSalary);

}

}

Here in the above snippet, the method calculateSalary() takes 3 arguments of integer type and returns nothing since its return type is void but prints the message about gross salary, to call the above method, we need an object of the class PayRoll as shown below

PayRoll payRoll = new PayRoll();

payRoll.calculateSalary(1000,250,275); // Calling the method using instance of the class

**Example of Method with a parameter and float as return type**

class Geometry

{

public float calculateAreaOfCircle(int radius)

{

float pi = 3.14f;

float area= pi \* radius \* radius ;

return area;

}

}

In the above snippet, this method calculateAreaOfCircle takes radius as an argument and finds the area and returns it.

To call the above method

Geometry geometry = new Geometry(); // Instantiating the Geometry class

float areaOfCircle = geometry.calculateRadius(5) ;

// invoking the calculateRadius method and passing an argument and holding the result //in the float variable areaOfCircle

**Method Overloading :** Method Overloading is a feature, by which a method can do multiple tasks through different function/method signatures.

Method signature denotes the number of parameters, types of parameters & sequence of parameters.

public void calculateSalary(double basicSalary,double hra,double cca) {}

public void calculateSalary(double basicSalary,double hra,double cca,double deductions){}

public void calculateSalary(double basicSalary,double hra,double cca,double deductions,String employeeName){ }

In the above example, calculateSalary is overloaded.

Method Overloading leads to Polymorphism, since the same method is behaving in multiple ways with different function signatures.

Another example:

1. public void add(int a,int b,int c)
2. public void add(String str1,String str2,String str3)
3. public void add(int a, int b,float c)
4. public void add(float a,int b,int c)

In the above signatures in case 1 and 2 Number of parameters are same, but the types are different, in the case 2 and 3 the number of parameters are different, In the case 3 and 4 number of parameters is same (2 int and 1 float) but the sequence is different, hence in all these cases we can say the method is overloaded.

Also we should note that keeping all the parameters the same and changing the return type alone will not lead to Method Overloading, it gives compile time error.

**Constructors & Constructor Overloading**

Constructors are special types of functions which will have the same name as that of the class name and will be invoked automatically when the classes are instantiated.

Constructors should be always public and should not contain any return type (we can make private constructors only for Singleton classes which you would learn little later)

Constructors can be overloaded

class Employee

{

String employeeId;

String employeeName;

String employeePhone;

int employeeSalary;

//Default Constructor

public Employee()

{

super();

employeeId = “E001”;

employeeName=”Harsha”;

employeePhone=”9839939933”;

employeeSalary = 1000;

}

// Overloaded Constructor or Parameterized Constructor

public Employee(String employeeId, String employeeName, String employeePhone, int employeeSalary)

{

this.employeeId = employeeId;

this.employeeName = employeeName;

this.employeePhone = employeePhone;

this.employeeSalary = employeeSalary;

}

public void displayDetails()

{

System.out.println(“Employee Id “+employeeId);

System.out.println(“Employee Name “+employeeName);

System.out.println(“Employee Phone “+employeePhone);

System.out.println(“Employee Salary “+employeeSalary);

}

public static void main(String[] args)

{

//Following statement would invoke the default constructor

Employee employee1 = new Employee();

employee1.displayDetails();

//Following statement would invoke the Overloaded constructors

Employee employee2 = new Employee(“E002”,”Kiran”,”9289299922”,2000);

employee2.displayDetails();

}

}

The above code would print the following output

Employee Id E001

Employee Name Harsha

Employee Phone 9839939933

Employee Salary 1000

Employee Id E002

Employee Name Kiran

Employee Phone 9289299922

Employee Salary 2000

“super” keyword represents the instance of the base class and super() means that we are invoking the parent class constructor with out parameters , if we are not deriving our class from any class, the by default every class is inherited from Object class, which means we are invoking its constructor, always super() should be the first statement within the constructor, after which other initializations should take place

Constructors thus help us in initializing the data members/variables , also when we create an Object of any class the constructor is automatically invoked and the memory allocation for the objects takes place with the help of a new operator , which we use to create an instance.

“This” keyword represents an instance of the current class, when we are initializing the class data members , to differentiate between the passed arguments and the class level data members, this keyword is used to refer to the variables of the class and initialize it to the values of the passed argument values through the overloaded constructor.

this.employeeId = employeeId

(data member of class) (passed argument)

**Getters & Setters (Accessor / Mutator Methods)**

Since the data members in a class are supposed to be private , if we need to access these data members , for changing its value or to retrieve the values from the outside world, Getters/Setters act as a channel.

The syntax for getters and setters are as follows:

class Employee

{

String employeeId;

public String getEmployeeId()

{

return employeeId;

}

public void setEmployeeId(String employeeId)

{

this.employeeId = employeeId;

}

public void displayEmployee()

{

System.out.println(“EmployeeId “+employeeId);

}

public static void main(String[] args)

{

Employee employee = new Employee();

employee.setEmployeeId(“E001”);

employee.displayEmployee();

System.out.println(“Employee Id is “+employee.getEmployeeId());

}

}

**“Getters”** are referred to as Accessors and **“Setters”** are referred to as Mutators, since we are able to retrieve the data using “Getters” and we are mutating/changing the values using Setters.

Depending upon the need of the business we can make these getters/setters as private or public, if we make both of them public , which means the variable is with read/write facility, if we make the getter public and setter private , then the variable becomes read Only and if we make setter public and getter private, it becomes write only,

**INHERITANCE :**

Inheritance is a feature by which one can derive a new class from the existing class, so that the new class would inherit all the characteristics & the behaviors from the parent class.

The new class is also called Derived class, Sub class or a Child class and the Parent class is also called Super Class or a Base class.

For example when a system is developed for an Automobile Mfg Company, assume we have already created a class called “Vehicle” with the following characteristics.

chasisNo, color, length, width, weight, engineNo

And with the following behavior:(methods)

getVehiclePrice(), getVehicleDetails(), transport() etc

Now later when somebody wants to deal with TwoWheelers and FourWheelers one need not create these classes from the scratch.

One can derive these classes from the existing Vehicle class

class Vehicle

{

String chasisNo;

String color;

int length;

int width;

int weight;

long engineNo;

public void getVehiclePrice()

{

}

public void getVehicleDetails()

{  
}

public void transport()

{

}

}

class TwoWheeler extends Vehicle class FourWheeler extends Vehicle { {

String handleBarType; String steeringType;

public void carry2People() public void carry4People()

{ {

} }

} }

Thus Inheritance is a feature by which one can derive a new class from an existing class, so that the new class would automatically inherit all the characteristics and behaviors from the parent class , along with which , one can specialize features in the derived class, which means the derived classes can contain additional features.

Inheritance mostly helps us in implementing reusability.

Different types of inheritance are:

Simple inheritance

Multiple inheritance [Not supported in Java]

Single Level Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Hybrid Inheritance [Not supported in Java]

**Simple Inheritance:** When a class is Inherited from only One class it is said to be Simple Inheritance.



**Multiple Inheritance:** When a class is Inherited from more than one class, it is said to be Multiple Inheritance(Not supported in Java).



**Single Level Inheritance:** When we have only one level of Inheritance

 

**Multi Level Inheritance:** When we have multiple levels of inheritances



**Hierarchical Inheritance:**



**Hybrid Inheritance:** A combination of all types of Inheritances(Not supported in Java):



**Polymorphism:**

Polymorphism is a feature by which an entity can exhibit itself in multiple forms.

For example the operator + can behave in many ways

* Can be used to add integers int x=100; int y=200; int result = x + y;
* Can also be used to concatenate Strings

String str1 = “Hello”;

String str2 = “World”;

String resultStr = str1+str2;

resultStr would store “HelloWorld”;

Here Operator + exhibits itself in multiple forms

Polymorphism in Java can be implemented with the help of following mechanisms

Method Overloading

Method Overriding

Interfaces

**Method Overloading:**

Method Overloading is a feature by which a single method can perform multiple functionalities through different function signatures.

**Method Overriding:**

Method Overriding or Function Overriding is a feature where a method/function of a base class is redefined/implemented in the derived class with a different functionality retaining its method signature intact.

class **Shapes**

{  
 public void draw()

{

System.out.println(“**Drawing Shapes**”);

}

}

class **Rectangle** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Rectangles**”);

}

}

class **Triangle** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Traingles**”);

}

}

class **Square** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Squares**”);

}

}

public class Main

{

Shapes shapes;

shapes = new Shapes();

shapes.draw(); //Prints Drawing Shapes

shapes = new Rectangle();

shapes.draw(); //Prints Drawing Rectangles

shapes = new Triangle();

shapes.draw(); //Prints Drawing Triangles

shapes = new Square();

shapes.draw(); //Prints Drawing Squares

}

@Override is an annotation to inform the compiler, saying that this method is already there in the Base Class which is being redefined in the derived class

Annotations are special instructions to the Compiler about the behavior of the annotated content.you will learn more on Annotations later.

**Static Binding(Compile Time Binding) vs Dynamic Binding(Runtime Binding)**

Binding is a process by which an instance of a class is associated with the address of the method.If this Binding takes place at the time of compilation, it is termed as Static Binding and if this Binding takes place at the runtime, it is termed as RunTime binding.

Method overloading leads to static binding and Method overriding leads to Dynamic Binding.

This is also termed as function/method resolution.

Polymorphism achieved through “Static Binding” is called Static Polymorphism

&

Polymorphism achieved through “Dynamic Binding” is called Dynamic Polymorphism

For example when we invoke the overloaded functions using an Object

For example obj.calculateSalary() by passing the arguments as shown in the example for Method Overloading, at the time of compilation the resolution is clear, that the Object is to be associated with the respective method with so and so signature…..but in Overriding, this resolution is delayed and takes place during the runtime, hence overriding leads to dynamic binding , resulting in Dynamic Polymorphism.

For example in the above example of Overriding

When an object

Say

When we call as in the following steps:

Shapes shapes;

shapes = new Shapes();

shapes.draw(); //Prints Drawing Shapes

shapes = new Rectangle();

shapes.draw(); //Prints Drawing Rectangles

shapes = new Triangle();

shapes.draw(); //Prints Drawing Triangles

shapes = new Square();

shapes.draw(); //Prints Drawing Squares

At the time of compilation shapes.draw() is not resolved because shapes object can be associated with the draw() method of the base class as well as that of the Derived class, therefore we say resolution is not clear, but during the runtime the Object gets clear meaning that it is an instance of Rectangle Class or Triangle class, therefore its resolution gets clarity, hence we term this as Late Binding or Dynamic Binding or Runtime Binding leading to Dynamic Polymorphism.

Static Polymorphism vs Dynamic Polymorphism

Performance High Low

Extensibility Low High

Maintainability Low High

Flexibility Low High

**Encapsulation** ::

Encapsulation is a feature by which we hide all the non essential characteristics from the end users accessibility (End user need not be always the customer, when we create modules , a specific module could be utilized by another module, the other module becomes end user too with respect to the first module)

Encapsulation provides thus security to the content of a class.

Another way of understanding Encapsulation is that it helps us in binding the data members and methods inside a class with the apt access modifiers, so that any outside entity does not easily tamper with the internal contents, but instead can utilize the features alone.

Here above “non essential characteristics” we mean to say with respect to the user, whereas with respect to the System every characteristic is an essential one.

For example, for the usage of an electronic gadget, we have switches and controllers using which we control it,but all the circuitry is hidden so that nobody tampers with it.

So hiding all the non essential characteristics from the end user is Encapsulation.

**Abstraction :**

Is a feature by which we provide accessibility to the essential characteristics to the end user .

For example : In a GearBox there are multiple Gear Wheels with different teeth, when we change the gear , the shaft is shifted to a Wheel of different numbered teeth, this mechanism is a complex activity, hence if user has to manually do this it is going to be laborious, hence the user is provided with the lever , using which he can easily shift the gear and get the effect, he is accessible to the Lever, if the lever is also sealed the user might not be able to use the Gear System, so providing accessibility to the essential characteristics is Abstraction

Always whenever there is complexity, it has to be abstracted from the user and easier accessibility is to be provided to the user. Thus Abstraction comes into picture

Abstraction usually leads to Polymorphism.

In the given example:

abstract class **Shapes**

{  
 public abstract void draw();

}

class **Rectangle** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Rectangles**”);

}

}

class **Triangle** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Traingles**”);

}

}

class **Square** extends **Shapes**

{

@Override

public void draw()

{

System.out.println(“**Drawing Squares**”);

}

}

public class Main

{

Shapes shapes;

shapes = new Rectangle();

shapes.draw(); //Prints Drawing Rectangles

shapes = new Triangle();

shapes.draw(); //Prints Drawing Triangles

shapes = new Square();

shapes.draw(); //Prints Drawing Squares

}

Abstract class is a class which has at least one abstract method, which means, wherever we feel that, there is going to be change in future, we need not create a concrete class with all methods implemented, we can make abstract class, which has a scope for different implementations like in the above example drawing Rectangles, Squares etc and in future we can, if required , extend it for Polygons and Trapezoid etc.

And all these implementations are / can be very complex and we need not expose it to the end user, what user accesses is the “Shapes” class object referencing Rectangles, Triangles etc dynamically based on the need in the runtime.

Abstract class cannot be instantiated.

Hence Abstraction leads to polymorphism as well as providing access only to essential characteristics.

We need not get confused between normal inheritance vs Abstraction.

For example when we have a “TextBox” class which can accept inputs from the user, further if we need a TextBox which can take only numerics, we can extend a New NumericTextBox from TextBox class, which can be normal inheritance, i.e we may need implementation in base class plus different extended implementations.

But what we saw in Abstraction is that there is complexity as well as need for change in future, hence we can make it abstract and give flexibility to the user to implement it in different ways as well as we are providing accessibility to only essential characteristics of the system to the user.

**Relationships in OOP:**

Specialization

Association

Aggregation

Composition

**Specialization:** Specialization is denoting Inheritance where we derive a new class from the existing class with added features in the derived class , but without disturbing the originality of the base class, hence we specialize in the derived classes therefore it is termed as Specialization.

**Association :** Association is a kind of relationship that is established among the objects of classes, Association can be of the following types:

One-One

One-Many

Many-One

Many-Many

For example a Bank can have multiple Employees

Therefore the relationship between Bank and Employee is **One - Many**

Similarly a Student can opt multiple Courses and one Course might be opted by multiple students, the association between Student and Course is termed **Many - Many**

In a Sales application , there can be Multiple Products purchase by a Customer, therefore association between Product and Customer is **Many-One**

In a situation, where One Citizen should hold only one PanCard, AadharCard

If Citizen is a class and PanCard is a class it becomes **One - One** relationship.

**Aggregation & Composition :**

Aggregation & Composition are Containment kind of relationships.

Aggregation & Composition follows a Has-A kind of relationship.

Aggregation is a relationship where the Contained entity exists even if the the Containing entity ceases to exist,But these are again business and requirement specific,

for example take for instance Hospital class and a Lab Class

In a scenario If Multiple Hospitals reach out to a Common Lab for sample check, then even if the Hospital ceases to exist , Lab still exists, so in this scenario it is said to be Hospital Aggregates Lab



**Composition :** Composition exhibits Has-A relationship where the Contained entity also ceases to exist, if the containing entity ceases to exist.

Take for an instance ScienceLab in a College, If the College is demolished the Lab is also destroyed, here we can say College class composes Lab class



**static & final : Non Access Modifiers:**

static & final are termed as non-access modifiers since they denote the behavior of the entity, be it a variable,instance or a method, but do not denote the scope or accessibility of the entity, for accessibility we have access modifiers.

Object , variable or method when defined as static denotes that there is going to be only one copy of it for innumerable instances.

class StaticExample

{

int nonStaticVar;

static int staticVar;

public void nonStaticMethod()

{

nonStaticVar++;

staticVar++;

System.out.println(“Static Var in Non Static Method “+staticVar);

System.out.println(“Non Static Var in Non Static Method “+nonStaticVar);

}

public static void staticMethod()

{

staticVar++;

System.out.println(“Static Var in Static Method”+staticVar);

}

public static void main(String[] args)

{

StaticExample stat1 = new StaticExample();

StaticExample.staticMethod();

// static method can be invoked through class and instance is not needed to call it

stat1.nonStaticMethod();

StaticExample stat2 = new StaticExample();

StaticExample.staticMethod();

stat2.nonStaticMethod();

}

}

The output is as follows

Static Var in Static Method :1

Static Var in Non Static Method :2

Non Static Var in Non Static Method :1

Static Var in Static Method :3

Static Var in Non Static Method :4

Non Static Var in Non Static Method :1

Static methods can access only static variables , non static methods can access static as well as non static variables, irrespective of the instance since the static variable is common for all objects, it is incremented from 1 to 4, whereas non static variable is specific to instance , hence for the first instance it gets incremented to 1 and for another instance the non static variable is specific to the new object, so it is incremented freshly from 0 to 1.

**final :** Keyword is used to make a variable constant, which cannot be changed

For example:

final int varx = 10;

varx +=5 **will throw an error since final prohibits one from changing the value**

final method means that it cannot be inherited

class FinalClass

{

public final void finalMethod()

{

System.out.println(“this method is final”);

}

}

class InheritedFinalClass extends FinalClass

{

‘@Override

public void finalMethod() **// This will throw an error since final method cannot be overridden**

{

}

}

**A final class by default has all methods final and final class cannot be inherited**

—------------------------------------EOP-------------------------------------