Day4: Object Collaboration

In Java applications, multiple classes collaborate with each other and provide the services. all the classes should be on the same path.

Example:

```
Employee.java:

package com.masai;
public class Employee{

 String id="E-1s11";
 String name="Ramesh";
 double salary=25000.00;
 String address="Hyderabad";

  public void displayDetails(){

    System.out.println("Employee Details");
    System.out.println("-------------------");
    System.out.println("Employee Id :"+id);
    System.out.println("Employee Name :"+name);
    System.out.println("Employee Saslary:"+salary);
    System.out.println("Employee Address:"+address);

  }
}

Main.java:

package com.masai;
public class Main{

  public static void main(String[] args){

      Employee emp=new Employee();
      emp.displayDetails();
  }
}
```

Note:- We can define a class as an instance member inside another class also, by this we establish

‘Has-A' relationship between 2 objects.

Example:

```
Address.java:-

package com.masai;
public class Address{

  String city;
  String state;
  String pincode;
}

Employee.java:-

package com.masai;
public class Employee{

  String empId;
  String empName;
  double salary;
  //Employee has Address
  Address address;

public static void main(String[] args){

  Employee emp1 = new Employee();

  System.out.println(emp1); //Employee object hashcode i.e Employee@323232
  System.out.println(emp1.empId); // null
  System.out.println(emp1.empName); // null
  System.out.println(emp1.salary); // 0.0
  System.out.println(emp1.address); // null

  Employee emp2 = new Employee();

  emp2.empId = "Emp-01";
  emp2.empName = "Ram";
  emp2.salary = 60000.00;

  emp2.address = new Address();

  emp2.address.city = "Coimbator";
  emp2.address.state = "Tamilnadu";
  emp2.address.pincode = "434322";
```

```
  System.out.println(emp2.empId); // Emp-01
  System.out.println(emp2.empName); // Ram
  System.out.println(emp2.salary); // 60000.00
  System.out.println(emp2.address); //Address object hashcode i.e Address@232423

  System.out.println(emp2.address.city); //Coimbator
  System.out.println(emp2.address.state); //Tamilnadu
  System.out.println(emp2.address.pincode); //434322
  }
}
```

Note: We can define the Address class inside the Employee class as a static member also.

```
Address.java:-
package com.masai;
public class Address{

  String city;
  String state;
  Srting pincode;
}

Employee.java:-

package com.masai;
public class Employee{

  String empId;
  String empName;
  double salary;
  //defining Address class as a static member
  static Address address;

public static void main(String[] args){

  Employee emp1 = new Employee();

  System.out.println(emp1); //Employee object hashcode i.e Employee@323232
  System.out.println(emp1.empId); // null
  System.out.println(emp1.empName); // null
  System.out.println(emp1.salary); // 0.0
  System.out.println(Employee.address); // null

  Employee emp2 = new Employee();

  emp2.empId = "Emp-01";
  emp2.empName = "Ram";
  emp2.salary = 60000.00;

  Employee.address = new Address();
//address = new Address(); //within the same class we can access the static members directly
//emp2.address=new Address(); we can access the static members with the help of object also
```

```
  Employee.address.city = "Coimbator";
  Employee.address.state = "Tamilnadu";
  Employee.address.pincode = "434322";



  System.out.println(emp2.empId); // Emp-01
  System.out.println(emp2.empName); // Ram
  System.out.println(emp2.salary); // 60000.00
  System.out.println(emp2.address); //Address object hashcode i.e Address@232423

  System.out.println(emp2.address.city); //Coimbator
  System.out.println(emp2.address.state); //Tamilnadu
  System.out.println(emp2.address.pincode); //434322
  }
}
```

Note: We can access the static members of a class from any static method directly whereas, to access the non-static members from the static method, we need to create an object.

All the non-static members are sharable, i.e. we can access the non-static members of a class from any non-static method directly.

I Problem:

Collaborating classes and objects

Example:

```
Address.java:-

package com.masai;
public class Address{

  String city;
  String state;
  String pincode;

public void printAddress(){

  System.out.println("City :"+city);
  System.out.println("State :"+state);
  System.out.println("Pincode :"+pincode);


}
```

```
}

Employee.java:-

package com.masai;
public class Employee{

  String empId;
  String empName;
  double salary;

  Address address = new Address();

  public void showDetails(){

    System.out.println("Employee Id :"+empId);
    System.out.println("Employee Name :"+empName);
    System.out.println("Employee Salary :"+salary);

    System.out.println("Employee Address :");
    address.printAddress();




  }



public static void main(String[] args){

  Employee emp1 = new Employee();
  emp1.showDetails();

  Employee emp2 = new Employee();
  emp2.empId = "Emp-01";
  emp2.empName = "Ramesh";
  emp2.salary = 50000;

  emp2.address.city = "Coimbator";
  emp2.address.state = "Tamilnadu";
  emp2.address.pincode = "434322";

  emp2.showDetails();

  }
}
```

Example2:

```
  package com.masai;
  public class A {

    int x = 10;

    void funA(){
      System.out.println("inside funA of A class");
    }
  }

package com.masai;
public class Main {

  static int j=200;
  static A a1 = new A();

  public static void main(String[] args)
  {

    Main obj=new Main();
    //System.out.println(obj.j);// 200
    //System.out.println(obj.a1); //A@23232

    //obj.a1.funA();

    System.out.println(Main.a1);

    Main.a1.funA();
    //System.out.println();

  }

}
```

Note: println() method belongs to the “java.io.PrintStream” class. “out” is a variable of this PrintStream class, which is statically defined inside the System class.

Example

```
public class System{

  static PrintStream out = new PrintStream(...);

}

that's why we can use System.out.println();
```

Methods in Java:

A Method in java is a set of instructions, it will represent a particular action in java applications.

A **method** is a block of code that only runs when it is called.

We can pass data, known as parameters, into a method.

We can return some value from a method also.

Methods are used to perform certain actions, and they are also known as **functions**.

We define the method to reuse code: define the code once, and use it many times.

Syntax:
[Access_Modifiers] return_Type method_Name([param_List])[throws Exception_List]
{
----- instructions to represent a particular action-----

[return value]

}

Note: in Java, while defining a method, we must specify the return type, at least *void*.

Method name generally we take as verb, whereas variable names we take as noun.

There are 2 types of Methods in Java:

1. Concrete Method
2. Abstract Method

Difference between the Concrete method and Abstract method:

1. The concrete method is a method, it will have both method declaration and method
   implementation.

   example:

```
public void sayHello() //method declaration
{ // method implementation
    System.out.println("Welcome to Java");
}
```

A concrete method must have a body, at least zero body.

Example:

```
public void method1(){
  //Zero body with empty implementation
}
```

An abstract method is a method, it will have only a method declaration.

example:

```
public abstract void sayHello();
```

2. To declare concrete methods, no need to use any special keyword.
   To declare an abstract method, we must use the abstract keyword.
3. Concrete methods are allowed in concreate classes and in abstract classes also.
   whereas abstract methods are allowed in abstract classes and interfaces only.

Method with Parameter:

We can define a method with parameters also, to supply some input data to the method in order to perform an action.

In java applications, we are able to provide all primitive data types and all user defined data types(like Class, Interface, Enum, etc.) as parameter types.

Example1: method with parameter as primitive

```
public class Demo
{

  //method defination
  void fun1(int i){ //int type parameter, here i variable will act as local variable
    System.out.println("inside fun1 of Demo");
    System.out.println("the value of i is : "+i);
  }


  public static void main(String[] args)
  {
    Demo d1=new Demo();

    byte x=20;
    //method call
    d1.fun1(x); // any compatible to int we can pass as an argument.(implicit typecasting)
    //d1.fun1(10);

    double n1=100.00;
    //d1.fun1(n1); //Error
    //d1.fun1((int)n1); //valid

  }
}

Note: In this example main() method is calling the fun1() method on the Demo class object.
```

As we can take any primitive type as a parameter to a function, we can also take an Class reference also as a parameter.

Note: if a method takes a concreate class as a parameter, then in order to call that method, we can pass following 3 things:

a. same class object
b. its child class object //we will talk about the child classes in upcoming session
c. null

Example1: method with parameter as concrete class type.

```
package com.masai;
public class A{
```

```
  int i=10;

  void funA(){
    System.out.println("inside funA of A");
  }

}

package com.masai;
public class Demo
{

  //method with class as parameter
  void fun1(A a1){   // here to call this method we can pass 3 possible values
             // 1.same class obj, 2 .child class obj, 3. null

    System.out.println("inside fun1 of Demo");
    System.out.println("the value of a1 is : "+a1);
    a1.funA();
  }


  public static void main(String[] args)
  {
    Demo d1=new Demo();

    //A a5=new A();
    //d1.fun1(a5);

    d1.fun1(new A());

  }
}
```

To the above method call , if we supply the null value then fun1() method will be called but inside the method , when control reaches to the **a1.funA(); ,** it will throw a runtime exception called “**NullPointerException**”.

So, it is always recommended to check the null value inside the method definition, before accessing any member from the supplied object.

Example:

We Problem:

```
package com.masai;
public class A{

  int i=10;
```

```
  void funA(){
    System.out.println("inside funA of A");
  }

}

package com.masai;
public class Demo
{


  void fun1(A a1){  //never use such identifiers

    if(a1 != null){

        System.out.println("inside fun1 of Demo");
        System.out.println("the value of a1 is : "+a1);
        a1.funA();
      }else
          System.out.println("supplied value is null");


  }


  public static void main(String[] args)
  {
    Demo d1=new Demo();


    d1.fun1(new A());
    d1.fun1(null);

  }
}
```

Method with return type:

A method must have a return type, at least **void.**

void return type indicates that, the method does not return any value, it will perform some task there only.

When a method has a return type other than void, then author of that method should not close the method body without returning appropriate value.

method with return type as primitive:

```
package com.masai;
public class Demo
{

  int fun1(){

        System.out.println("inside fun1 of Demo");
        //return 100;

        //int x=200;
        //return x;

        //we can return any value which is compatible with the int type(smaller than int)

        byte b=10;
        return b;

        //long x=20;
        //return x; //Error

        //return (int)x; //OK

  }


  public static void main(String[] args)
  {
    Demo d1=new Demo();


    d1.fun1(); //here method will be called but returned value will be unreferenced
              // hence, it will reaches to the garbage collector

    //to utilize the returned value, we need to hold that value inside a variable
    //the variable on which we hold that value should be either of the same type or bigger
    // than the specified type.

    int x=d1.fun1();
    System.out.println("Returned value is "+x);

    long y=d1.fun1(); //implicit typecasting
    System.out.println("Returned value is "+x);

    //byte b= d1.fun1(); // Error
    byte b = (byte)d1.fun1(); //explicit type casting



  }
}
```

method with return type as Class:

We can define a method with class as a return type also, if a method is defined to return a class type then that method should return one of the following :

1. Same class object
2. child class object of the specified type
3. null

Example:

```
package com.masai;
public class A{

  int i=10;

  void funA(){
    System.out.println("inside funA of A");
  }

}


package com.masai;
public class Demo
{


  public A fun1(){ //return type as class A type

      System.out.println("inside fun1 of Demo");

      //A a1=new A();
      //return a1;

      return new A();
  }


  public static void main(String[] args)
  {
    Demo d1=new Demo();

    d1.fun1(); //here returned A class object will reaches to the GC.

    //to hold the returned value we have 2 options:
    //1.to the same class variable
    //2.to the parent class variable

    A a1= d1.fun1(); //to the same class variable
    a1.funA();


    Object obj = d1.fun1(); //to the parent class variable
```

```
  }
}
```

Note: In Java, there is a class called Object class which act as a parent/super class of any java class directly or indirectly.

This Object class belongs from “java.lang” package.

To the variable of Object class, we can assign any class object.

Polymorphism:

Defining more than one functionality with the same name in the same class is known as polymorphism.

The main advantage of Polymorphism is "Flexibility".

We have 2 type of polymorphism:

1. **Static polymorphism**: If the Polymorphism is existed at compilation time then it is called as Static Polymorphism. example: method overloading (same method name, but the parameter will be different)

   It is also know as compile time polymorphism, i.e. which method will be called, decided at compile time only.

2. **Dynamic Polymorphism:** If the Polymorphism is existed at runtime then that Polymorphism is called as Dynamic Polymorphism. example: method overriding:- (same method name ,and same parameter)

   Method overriding we achieve through inheritance.

   It is also know as run time polymorphism, i.e. which method will be called , decided at runtime.

   Static Polymorphism Example:

```
package com.masai;
public class Demo
{
```

```
  void fun1(byte b){

    System.out.println("inside fun1(byte) of Demo");
    //500 line of code

   }

  void fun1(int i){

    System.out.println("inside fun1(int) of Demo");
    //10000 line of code

   }

  public static void main(String[] args)
  {
    Demo d1=new Demo();
    byte x=20;
    d1.fun1(x);  // it will give the priority to the nearest one.
  }
}
```

Constructor:

It is a special type of non-static method, which will be executed automatically only at the time of creating an object.

Example:

```
Demo d1=new Demo();
```

The meaning of the above statement is, creating the object of Demo class by executing zero argument constructor of Demo class.

Syntax:

[Access_Modifier] Class_Name([Param_List])[throws Exception_List]

{

...body of the constructor...

}

Note:- we can have a dot java file of a class, without a constructor, but we can't have a dot class file of a class without a constructor.

At least default constructor must be there inside the dot class file of a class.

Inside our class, if we keep any constructor, then that constructor will be executed at time of creating object of our class, but if we don't keep any constructor explicitly inside our class, then **java compiler** will provide a default constructor to the dot class file.

Default constructor given by the compiler will always be public and zero argument.

Example: In Demo class

```
public Demo(){

}
```

This default constructor given by the java compiler seems like an empty constructor, but strictly speaking there is a hidden statement is there as a first line of this default constructor. (it is a “super();” it is a call to the parent class constructor, we will discuss about this statement inside the Inheritance concept).

if we want to execute certain logics at the time of creating an object of a class then we should place those logics inside the constructor of that class.

For object creation, constructor execution must be there.

Until and unless our constructor get executed completely, our object will not be created completely, it will be in the creation process.

The main utilization of constructors is to provide initializations for the instance variable.

Difference between normal non-static method and a constructor:

1. Method name can be anything, but the name of the constructor must be the class name.

   Note:- we can take a method name as a class name also, but it is a bad practice.

2. A method must have a return type , at least void, where as a constructor should not have any return type.
3. A method we can call on an object, whenever we want to call. but a constructor will be called on an object automatically only one time at the time of creating that object.
4. A method can be static, where static keyword is not applicable with a constructor.
5. A method can be abstract where as abstract keyword is not applicable with the constructor.
6. final keyword is applicable with the method, whereas it is not applicable with the constructor.

Similarities between normal non-static method and a constructor:

We can overload a constructor also, and rules of constructor overloading is similar to the rules of method overloading.

Note: if we keep any constructor in our dot java file, then java compiler won't provide default constructor to our dot class file.

Example:

```
package com.masai;
public class Demo{

  Demo(){
    System.out.println("Inside constructor of Demo");
  }

  public static void main(String[] args){
      Demo d1=new Demo();
  }


}
```

We Problem:

Constructor Overloading example:

```
package com.masai;
public class Employee {

  String empId;
  String name;
  double salary;

  //zero argument constructor
  public Employee() {

    empId="Emp-01";
    name= "Ramesh";
    salary = 50000.00;
  }


  //overloaded parameterized constructor
  public Employee(String empId, String name, double salary) {
    this.empId = empId;
    this.name = name;
    this.salary = salary;
  }


  public void showDetails() {
```

```
    System.out.println("Employee Id :"+empId);
    System.out.println("Employee Name :"+name);
    System.out.println("Salary is :"+salary);

  }

  public static void main(String[] args) {

    Employee emp1=new Employee();
    emp1.showDetails();

    Employee emp2=new Employee("Emp-02", "Dinesh", 40000.00);
    emp2.showDetails();


  }
}
```

‘this’ keyword:

'this' keyword points to the current object.

Whenever it is required to point an object from a method which is under execution because of that object then we use 'this' keyword.

Following 3 main job of this keyword:

1. Points to the current class obj.
2. Differentiate between local and instance variable.
3. Calling a constructor of a class from another constructor of the same class.

Pointing to the current class object

```
package com.masai;
public class Demo
{
  //instance variable
  int x=100;

  void fun1(){

    //local variable
```

```
    int x=500;

    System.out.println("inside fun1() of Demo");
    System.out.println(this); //Demo@232323 current class obj
    System.out.println(this.x); // 100
    System.out.println(x); //500


   }

  public static void main(String[] args)
  {

    Demo d1=new Demo();
    System.out.println(d1); // Demo@232323
    d1.fun1();

    //System.out.println(this); //Compilation Error

  }
}
```

Note: ‘this’ keyword we can not use inside the static area.

Calling a constructor explicitly:

A constructor will be called automatically whenever we create object of a class.

We can call a constructor of a class explicitly also, but that call must be :

1. From the another constructor of the same class (using **'this'** keyword)
2. From the constructor of its child class. (using **'super'** keyword)

Note:- if we call a constructor explicitly from the another constructor then that call must be the first line.

Example:

```
package com.masai;
public class Demo
{


  public Demo(){
    this(10);
    System.out.println("inside zero argument constructor Demo()");
  }
```

```
  public Demo(int x){
    this(100, 200);
    System.out.println("inside one argument constructor Demo(int x)");
  }

  public Demo(int x,int y){
    this("Hello");
    System.out.println("inside two argument constructor Demo(int x,int y)");
  }

  public Demo(String s){
    //this(); //it will become recursive call
    System.out.println("inside one(String) argument constructor Demo(String s)");
  }


  public static void main(String[] args){


      Demo d1=new Demo();

  }
}
```

Pure encapsulation:

This concept says that, we need to mark our data as private and expose those data through the public methods (like getter and setters methods).

Java Bean class aka POJO (plain old java object):

It is a reusable, purely encapsulated java class which should have following properties:

1. The class must be public
2. All the fields should be private
3. For each field there should be corresponding public getter and setter method.
4. It should have zero argument constructor.
5. It may have parameterized constructor (it is not the minimum requirement)

Java Bean class is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Note: There are two ways to provide values to the object. One way is by constructor and second is by setter method.

Advantages of JavaBean

The following are the advantages of JavaBean

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Example:

Student.java:

```
package com.masai;
public class Student{

private int roll;
private String name;
private int marks;

//zero argument constructor
public Student(){

}

//parameterized constructor
public Student(int roll,String name,int marks){
  this.roll=roll;
  this.name=name;
  this.marks=marks;

}

public void setRoll(int roll){
```

```
  this.roll=roll;
}

public int getRoll(){
  return roll;
}

public void setName(String name){
  this.name=name;
}

public String getName(){
  return name;
}

public void setMarks(int marks){
  this.marks= marks;
}

public int getMarks(){
  return marks;
}

public void showDetails(){
  System.out.println("Roll is :"+roll);
  System.out.println("Name is :"+name);
  System.out.println("Marks is :"+marks);
}
}
```

Demo.java

```
package com.masai;
public class Demo
{

  public static void main(String[] args){

    //using zero argument constructor
    Student student1 = new Student();
    student1.setRoll(100);
    student1.setName("Ram");
    student1.setMarks(800);


    //using parameterized constructor
    Student student2 = new Student(101, "Ramesh", 850);


    System.out.println("Student1 details: ");
    student1.showDetails();
```

```
    System.out.println("Student2 details using getter methods");
    System.out.println("Roll is :"+student2.getRoll());
    System.out.println("Name is :"+student2.getName());
    System.out.println("Marks is :"+student2.getMarks());
  }
}
```

You Problem:

Write a java class Main, and place following 4 overloaded constructor inside this class :

public Main(){

    System.out.println(”inside Main()”);

}

public Main(int x){

    System.out.println(”inside Main(int x)”);

}

public Main(double d){

    System.out.println(”inside Main(double d)”);

}

public Main(String s){

    System.out.println(”inside Main(String s)”);

}

inside the main method create only one object of the Main class and call above all constructor.

Reference :

https://www.javatpoint.com/java-bean