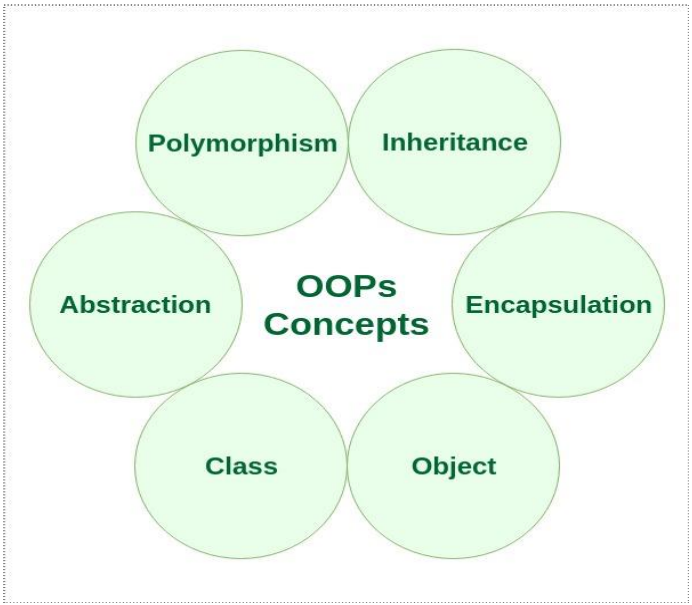
Chapter - 12

Object-Oriented Programming

- **Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data and code.
- The data is in the form of fields (often known as attributes or *properties*).
- The code is in the form of procedures (often known as *methods*).
- OOPS concepts are as follows:
    - Class
    - Object
    - Method
    - Message passing
    - Pillars of OOPs
        - Abstraction
        - Encapsulation
        - Inheritance
        - Polymorphism
            - Compile-time polymorphism
            - Runtime polymorphism



Class

- A class is a blueprint or template of an object.
- It is a user-defined data type.

- Inside a class, we can define variables, constants, member functions, and other functionality.
- A Class binds data and functions together in a single unit.
- A Class does not consume memory at run time.
- Note that classes are not considered as a data structure. It is a logical entity.
- Note that a class can exist without an object but vice-versa is not possible.
- In general, class declarations can include these components in order:
    - **Access Modifiers**: Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members.
        - **private** (accessible within the class where defined)
        - **default** or package-private (when no access modifier is specified)
        - **protected** (accessible only to classes that subclass your class directly within the current or different package)
        - **public** (accessible from any class)
        - **Note:** The order of the access modifiers from the least restrictive to the most restrictive:

public > protected > default > private

    - **Class name:** The class name should begin with the initial letter capitalized by convention.
    - **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
    - **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
    - **Body:** The class body is surrounded by braces, { }.

Object

- An Object is a real-world entity that has attributes, behavior, and properties.
- An Object is referred to as an instance of the class.
- An Object contains member functions, variables that we have defined in the class.
- An Object occupies space in the memory.
- The different objects have different states or attributes, and behaviors.
- Using class, a typical Java program allows to create the multiple objects, with the same behavior instead of writing their code multiple times.
- An object mainly consists of:
    - **State**: It is represented by the attributes of an object. It also reflects the properties of an object.
    - **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.

- **Identity**: It is a unique name given to an object that enables it to interact with other objects.

Method

- A method is a collection of statements that perform some specific task and return the result to the caller.
- A method can perform some specific task without returning anything.
- Methods allow us to **reuse** the code without retyping it, which is why they are considered **time savers**.
- In Java, every method must be part of some class.

Message Passing

- Objects communicate with one another by sending and receiving information to each other.
- A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results.
- Message passing involves specifying the name of the object, the name of the function and the information to be sent.

EXAMPLE -1

```
import java.io.*;
public class Student {

        int sage;

        public Student(String name) {
                // This constructor has one parameter, name.
                System.out.println("Name of student is :" + name );
        }

        public void setAge( int age ) {
                sage = age;
        }

        public int getAge( ) {
                System.out.println("Student's age is :" + sage );
                return sage;
        }


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

```
                /* Object creation */
                Student S = new Student( "XYZ" );

                /* Call class method to set age */
                S.setAge( 2 );

                /* Call another class method to get age */
                S.getAge( );

                /* You can access instance variable as follows as well */
                System.out.println("Variable Value :" + S.sage );
        }
}
```

EXAMPLE -2

```
import java.io.*;
public class Employee{

        String name;
        int age;
        String designation;
        double salary;

        // This is the constructor of the class Employee
        public Employee(String name){
                this.name = name;
        }

        // Assign the age of the Employee  to the variable age.
        public void empAge(int empAge){
                age = empAge;
        }

        /* Assign the designation to the variable designation.*/
        public void empDesignation(String empDesig){
                designation = empDesig;
        }

        /* Assign the salary to the variable  salary.*/
        public void empSalary(double empSalary){
                salary = empSalary;
        }
```

```
        /* Print the Employee details */
        Public void printEmployee(){
                System.out.println("Name:"+ name );
                System.out.println("Age:"+ age );
                System.out.println("Designation:"+ designation );
                System.out.println("Salary:"+ salary);
        }
}


import java.io.*;
public class EmployeeTest{

        public static void main(String args[]){

                /* Create two objects using constructor */
                Employee empOne =newEmployee("James Smith");
                Employee empTwo =newEmployee("Mary Anne");

                // Invoking methods for each object created
                empOne.empAge(26);
                empOne.empDesignation("Senior Software Engineer");
                empOne.empSalary(1000);
                empOne.printEmployee();

                empTwo.empAge(21);
                empTwo.empDesignation("Software Engineer");
                empTwo.empSalary(500);
                empTwo.printEmployee();
        }
}
```

Access Modifiers

- Access modifiers are used to control the accessibility to class, constructor, variable, method or data member.

- In other words, we can use access modifiers to protect data and behaviors from the outside world.

- There are four types of access modifiers available in java:
    - Public
    - Protected
    - Default – No keyword required

- Private

- The order of the access modifiers from the least restrictive to the most restrictive:
  - **public > protected > default > private**

Java public access modifier:

- When applied to a class, the class is accessible from any classes regardless of packages.
- This is the least restrictive access modifier which means the widest range of accessibility, or visibility.
- When applied to a member, the member is accessible from any classes.

EXAMPLE -3

```
package p1;

public class A {
	public int abc=10;
	public void display() {
		System.out.println("Hello World");
	}
}

package p2;
import p1.*;
public class B {

	public static void main(String[] args) {
		A obj = new A();
		obj.display();
		System.out.println(obj.abc);
	}
}
```

Output:

```
Hello World
10
```

Java protected access modifier:

- The protected access modifier is specified using the keyword **protected**.

- This is more restrictive than the public modifier. It is applied for members only.
  - There is no ‘protected’ class.

- When a member of a class is declared as protected,
  - It is accessible by only classes in the same package or by a subclass in different package.

EXAMPLE - 4

```
package p1;
public class A {
        protected int abc=10;
        protected void display() {
                System.out.println("Hello World");
        }
}

package p2;
import p1.*;
public class B {
        public static void main(String[] args) {
                A obj = new A();
                obj.display();
                System.out.println(obj.abc);
        }
}
```

Output: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method display() from the type A is not visible
at p2.B.main(B.java:9)

EXAMPLE - 5

```
package p1;

public class A {
        protected int abc=10;
        protected void display() {
                System.out.println("Hello World");
        }
}

package p2;
import p1.*;
public class B {
        public static void main(String[] args) {
```

```
            B obj = new B();
            obj.display();
            System.out.println(obj.abc);

        }
}
```

Output: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method display() is undefined for the type B
at p2.B.main(B.java:9)

EXAMPLE - 6

```
package p1;

public class A {
        protected int abc=10;
        protected void display() {
                System.out.println("Hello World");
   }
}

package p2;
import p1.*;
public class B extends A {     //Class B is subclass of A
        public static void main(String[] args) {
                B obj = new B();
                obj.display();
                System.out.println(obj.abc);
        }
}
```

Output:

Hello World
10

Java default access modifier:

- When no access modifier is specified for a class, method or data member
  - It is said to be having the **default** access modifier by default.
- It is more restrictive than the **protected** modifier.
- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

EXAMPLE - 7

```
package p1;

public class A {
	int abc=10;
	void display() {
		System.out.println("Hello World");
	}
}

package p2;
import p1.*;
public class B {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		//accessing class A from package p1
		A obj = new A();
		obj.display();
		System.out.println(obj.abc);
	}
}
```

Output: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method display() from the type A is not visible
at p2.B.main(B.java:9)

EXAMPLE - 8

```
package p1;

public class A {
	int abc=10;
	 void display() {
		System.out.println("Hello World");
	}
}

package p1;

public class B {
	public static void main(String[] args) {
		A obj = new A();
		obj.display();
		System.out.println(obj.abc);
	}
```

```
}
```

Output:

```
Hello World
10
```

Java private access modifier:

- The private access modifier is specified using the keyword **private**.
- This is the most restrictive access modifier in Java.
- The methods or data members declared as private are accessible only within the class in which they are declared.
- Any other **class of same package will not be able to access** these members.

EXAMPLE - 9

```
package p1;

public class A {
        int abc=10;
         private void display() {
                System.out.println("Hello World");
        }
}
public class B {
        public static void main(String[] args) {

                //trying to access private method of another class
                A obj = new A();
                obj.display();
                System.out.println(obj.abc);
        }
}
```

Output: Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method display() from the type A is not visible
at p1.B.main(A.java:16)

EXAMPLE – 10

```
package p1;

public class A {
         int abc=10;
         private void display() {
```

```
            System.out.println("Hello World");
        }

        public static void main(String[] args) {
            A obj = new A();
            obj.display();
            System.out.println(obj.abc);
        }
}
```

Output:

Hello World
10

Java static keyword

- The **static keyword** in Java is used for memory management mainly.
- java static keyword can be applied to :
  - Variable (also known as a class variable)
  - Method (also known as a class method)
  - Block
  - Nested class

1) Java static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- It makes your program **memory efficient** (i.e., it saves memory).
- The static variable gets memory only once in the class area at the time of class loading.

EXAMPLE – 11: Java Program to demonstrate the use of static variable

```
class Student{

    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
            rollno = r;
            name = n;
```

```
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}

//Test class to show the values of objects
public class TestStaticVariable1{
    public static void main(String args[]){
            Student s1 = new Student(111,"Karan");
            Student s2 = new Student(222,"Aryan");
            //we can change the college of all objects by the single line of code
            //Student.college="BBDIT";
            s1.display();
            s2.display();
    }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

2) Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

EXAMPLE – 12

```
//Java Program to demonstrate the use of a static method.

class Student{

        int rollno;
        String name;
        static String college = "ITS";
        //static method to change the value of static variable
        static void change(){
                college = "BBDIT";
        }

        //constructor to initialize the variable
        Student(int r, String n){
```

```
            rollno = r;
            name = n;
        }

        //method to display values
        void display(){System.out.println(rollno+" "+name+" "+college);}
}

//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
        Student.change();//calling change method

        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        Student s3 = new Student(333,"Sonoo");

        //calling display method
        s1.display();
        s2.display();
        s3.display();
    }
}
```

Output:

```
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT
```

NOTE: Why is the Java main method static?

- It is because the object is not required to call a static method.
- If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of class loading.

EXAMPLE – 13

```
class A2{
    static{System.out.println("static block is invoked");}
```

```
    public static void main(String args[]){
        System.out.println("Hello main");
    }
}
```

Output:

```
static block is invoked
Hello main
```

4) Java static nested class

- A static class i.e. created inside a class is called static nested class in java.
- It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

EXAMPLE – 14

```
class TestOuter1{
    static int data=30;
    static class Inner{
        void msg(){System.out.println("data is "+data);}
    }

    public static void main(String args[]){
        TestOuter1.Inner obj = new TestOuter1.Inner();
        obj.msg();
    }
}
```

Output:

```
data is 30
```

Encapsulation

- Encapsulation in Java refers to integrating data (variables) and code (methods) into a single unit.
- In encapsulation, a class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found.
- The aim of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this:
    - Declare class variables/attributes as private.
    - Private variables can only be accessed within the same class (an outside class has no access to it).

- However, public **get** and **set** methods are used to access and update the value of a private variable.
- The **set** method sets the value of variable and **get** method returns the variable value.
- Syntax for both is that they start with either get or set, followed by the name of the variable, with the first letter in upper case:

EXAMPLE -14:

```
class DemoEncap {

        private int ssnValue;
        private int employeeAge;
        private String employeeName;

        // We will employ get and set methods to use the class objects
        public void setEmployeeAge(int newValue) {
                employeeAge = newValue;
        }

        public void setEmployeeName(String newValue) {
                employeeName = newValue;
        }

        public void setEmployeeSSN(int newValue) {
                ssnValue = newValue;
        }

        public int getEmployeeSSN() {
                return ssnValue;
        }

        public String getEmployeeName() {
                return employeeName;
        }

        public int getEmployeeAge() {
                return employeeAge;
        }
}

public class TestEncapsulation {

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

```
        DemoEncap obj = new DemoEncap();
        obj.setEmployeeName("Mark");
        obj.setEmployeeAge(30);
        obj.setEmployeeSSN(12345);

        System.out.println("Employee SSN Code is: " + obj.getEmployeeSSN());
        System.out.println("Employee Name is: " + obj.getEmployeeName());
        System.out.println("Employee SSN Code is: " + obj.getEmployeeAge());
    }
}
```

Output:

```
Employee SSN Code is: 12345
Employee Name is: Mark
Employee SSN Code is: 30
```

Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Inheritance

- Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.
- The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).
    - Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

- The new class inherits the methods and fields from an existing class. Moreover, one can add new methods and fields in the current class also.
    - **The object of new class** acquires all the properties and behaviors of a parent object.
- The keyword **extends** is used to perform inheritance in Java.

EXAMPLE -15:

```
class Calculation {
        int z;

        public void addition(int x, int y) {
                z = x + y;
                System.out.println("The sum of the given numbers:"+ z);
        }

        public void Subtraction(int x, int y) {
                z = x - y;
                System.out.println("The difference between the given numbers:"+ z);
        }
}

public class My_Calculation extends Calculation {

        public void multiplication(int x, int y) {
                z = x * y;
                System.out.println("The product of the given numbers:"+ z);
        }

        public static void main(String args[]) {
                int a = 20, b = 10;
                System.out.println("The given numbers: a = 20 and b = 10 ");

                My_Calculation demo = new My_Calculation();
                demo.addition(a, b);
                demo.Subtraction(a, b);
                demo.multiplication(a, b);
        }
}
```

Output:

```
The given numbers: a = 20 and b = 10
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

Advantages of Inheritance

- It allows us to derive further classes. This creates a reusable model where we never change the existing classes, which improves the software development time required for program executions.
- The derived classes generate an extended set of properties, which ensures that programmers create dominant objects that are fully capable of performing the required independent tasks with loose connections to each other.
- Base classes can give rise to multiple derived classes, creating a dynamic hierarchy capable of providing excellent functionality under different conditions.
- It is possible to create complex inheritance, which offers the benefits of having all the properties present in the base classes. (Not possible in Java.)
- All common code belongs to the superclass and only needs to be executed and compiled once during a program operation.
- It is possible to override methods, which is excellent for defining empty method definitions in base classes and then overriding them, according to the specific use cases.
- It is possible to optimize the code and arrange the required functionality in an enhanced manner. This ensures that the program code is effective and provides better throughput results.

Abstraction

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
    - Ex: A car is viewed as a car rather than its individual components.
- Real-World example
    - Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.
- In java, abstraction can be achieved by abstract classes and interfaces.
    - Note: The 100% abstraction can be achieved using interfaces.
- Abstraction in Java can be best defined by using abstract keyword as access modifier with classes and methods:
    - **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
    - **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

- Properties of Abstract classes and Abstract methods:
  - An abstract class is a class that is declared with an abstract keyword.
  - An abstract class may or may not have all abstract methods. Some of them can be regular methods.
  - A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
  - Any class that contains one or more abstract methods must also be declared with an abstract keyword.
  - There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
  - An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.
- Notes on Interfaces:
  - Like **abstract classes**, interfaces **cannot** be used to create objects.
  - Interface methods do not have a body - the body is provided by the "implement" class
  - On implementation of an interface, you must override all of its methods
  - Interface methods are by default abstract and public
  - Interface attributes are by default public, static and final
  - An interface cannot contain a constructor (as it cannot be used to create objects)

EXAMPLE -16: Abstraction can also be achieved with class.

```
//Abstract class
abstract class Animalbehaviour {

        //Abstract method (does not have a body)
        public abstract void animalSound();

        //Regular method
        public void sleep() {
                System.out.println("Zzz");
        }
}

//Subclass (inherit from Animal)
class Pig extends Animalbehaviour {

        public void animalSound() {
                // The body of animalSound() is provided here
                System.out.println("The pig says: wee wee");
        }
}

public class Animal {
```

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

                Pig myPig = new Pig(); // Create a Pig object

                myPig.animalSound();

                myPig.sleep();
        }
}
```

EXAMPLE - 17: Abstraction can also be achieved with interfaces.

```
// Interface
//An interface is a completely "abstract class" that is used
//to group related methods with empty bodies
interface Animalbehaviour {

        public void animalSound(); // interface method (does not have a body)

        public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
//To access the interface methods, the interface must be "implemented" (like
//inherited) by another class with the implements keyword (instead of extends)
class Pig implements Animalbehaviour {

        public void animalSound() {

                // The body of animalSound() is provided here
                System.out.println("The pig says: wee wee");
        }

        public void sleep() {

                // The body of sleep() is provided here
                System.out.println("Zzz");
        }
}

public class Animal {

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

```
        Pig myPig = new Pig(); // Create a Pig object

        myPig.animalSound();

        myPig.sleep();
    }
}
```

- Benefits of Abstraction
    - Abstraction offers several benefits and therefore, it forms the basis of all OOP languages. Here are some excellent advantages of using this specific principle:
        - It creates a barrier, which protects the implementation layer from the code users.
        - It also offers flexibility in terms of later changing the way implementation is carried out. This may be done to improve the coupling structure to loosen it up. A loose system is beneficial, as it allows all involved parties to make the best use of a working contract created through an application interface.
        - It makes it easier to perform debugging and find out which working layer is at fault for a particular problem.
        - It can be delivered through interfaces, allowing the easy correction of code use by an end user or the identification of wrong implementation scheme placed by the developer.
        - It also allows to set up for a divide and conquer policy for breaking a large program into smaller sections, which are easier to correct and implement by setting up interfaces that connect different parts of the complex program.

Polymorphism

- Polymorphism is derived from 2 Greek words: "poly" and "morphs".
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- **Real-life example of** Polymorphism:
    - A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
- Polymorphism is considered one of the important features of Object-Oriented Programming.
- Polymorphism allows us to perform a single action in different ways.
- Polymorphism is the ability of an object to take on many forms.
- Java makes software more reliable and maintainable with the use of polymorphism.

- Polymorphism is a great boon for software maintenance: if a new subclass is added, the code in the main program does not change.
- In Java polymorphism is mainly divided into two types:
    - *Compile-time Polymorphism:*
        - It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But Java doesn't support the Operator Overloading.
    - *Runtime Polymorphism:*
        - It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

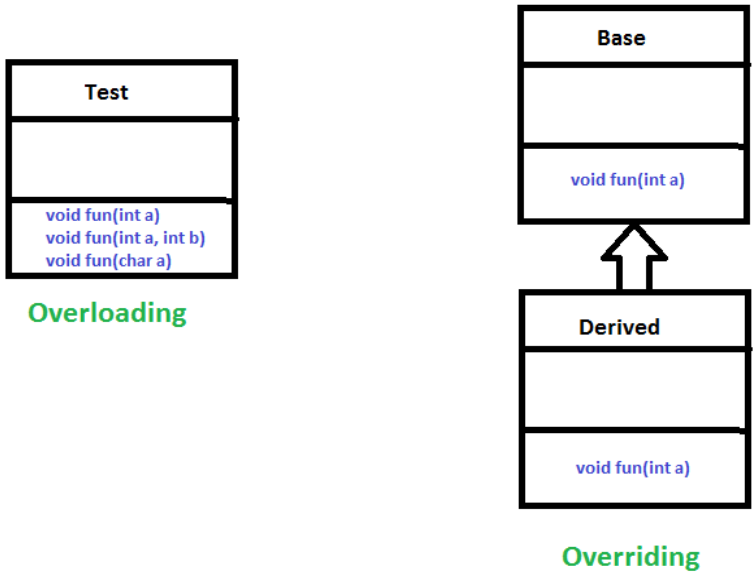


Figure: Overloading and Overriding

- We can achieve polymorphism in Java using the following ways:

    1. **Method Overriding**
    2. **Method Overloading**

Method Overriding

- **Method overriding** occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.
- This is also termed as *run-time polymorphism*, because the object takes different shapes during the program execution.

Example - 18: Java Program for Method Overriding

```
class Vehicle{

        //defining a method
        void run(){
                System.out.println("Vehicle is moving");
        }
}

//Creating a child class
class Car extends Vehicle{

        //defining the same method as in the parent class
        void run(){
                System.out.println("car is running safely");
        }

        public static void main(String args[]){

                //creating object of base class
                Vehicle obj1 = new Vehicle();
                obj1.run(); //calling method

                //creating object of child class
                Car obj2 = new Car();
                Obj2.run();//calling method
        }
}
```

Output

```
Vehicle is moving
car is running safely
```

Method Overloading

- When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**.
- Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.
- This is defined not during the execution, but is fixed at the time of program compilation, which occurs prior to execution having fixed behavior.
- It is also termed as *compile-time polymorphism*.
- **Example:** Java program for Method Overloading by Using Different Numbers of Arguments

Example - 19: Java program to illustrate overloading using simple class.

```
public class Polymorphism {
	public static void main(String[] args) {
		double a=3.0, b=4, c=5;
		int width=5, height=7;
		int radius=5;

		Shape S = new Shape();
		System.out.println("The  Circle  Area : " + S.computeArea(radius));
		System.out.println("The  Rectangle  Area : " + S.computeArea(width, height));
		System.out.println("The  Triangle  Area : " + S.computeArea(a,b,c));
	}
}

class Shape {

	// compute the area of rectangle
	public int computeArea(int x, int y) {
		int width = x;
		int height=y;
		return width * height;
	}

	// compute the area of circle
	public double computeArea(int radius) {
		return Math.PI * radius * radius;
	}

	// compute the area of triangle
	public double computeArea(double a, double b, double c) {
		double s = (a + b + c) / 2;
		return Math.sqrt(s * (s - a) * (s - b) * (s - c));
	}
}
```

Output:

```
The  Circle  Area : 78.53981633974483
The  Rectangle  Area : 35
The  Triangle  Area : 6.0
```

Example - 20: Java program to illustrate overloading using abstract parent class, abstract method and public child classes.

```
//parent class
public abstract class Shape {
```

```
        protected int x, y;

        //abstract method
        public abstract double computeArea();
}

//Creating Child Class
public class Rectangle extends Shape {
        double width=5, height=7;
        public double computeArea() {
            return width * height;
        }
}

//Creating Child Class
public class Circle extends Shape {
        double radius=5;
        public double computeArea() {
            return Math.PI * radius * radius;
        }
}

//Creating Child Class
public class Triangle extends Shape {
        double a=3.0, b=4, c=5;
        public double computeArea() {
            double s = (a + b + c) / 2;
            return Math.sqrt(s * (s - a) * (s - b) * (s - c));
        }
}

public class Polymorphism {
         public static void main(String[] args) {

                Shape S;
                S= new Circle();
                System.out.println("The  Circle  Area : " + S);

                S= new Rectangle();
                System.out.println("The  Rectangle  Area : " + S);

                S= new Triangle();
                System.out.println("The  Triangle  Area : " + S);
        }
}
```

Output:

The Circle Area: 78.53981633974483
The Rectangle Area: 35.0
The Triangle Area: 6.0

Advantages of Polymorphism

- It ensures that programmers can first fully test and finalize their code before implementing it in a variety of ways. This way, it is possible to always use consistent elements in the final program.
- Developers do not need to take care of the naming schemes, as polymorphism ensures that the same name can represent different data instances, such as int, double, and other available options in the OOP language.
- It reduces the present coupling, ensuring that we can create flexible program objects.
- It improves the code efficiency, when a program becomes long with complex functions placed within it, as it ensures that several instances and situations can be adequately handled.
- Closely related operations become possible by using method overloading, where we can use the same name to designate different methods, each with their own set of parameters.
- It allows the use of different constructors that can initialize class objects. This ensures that we have program flexibility, with access to multiple initializations.
- It is carried out during inheritance principle application as well, as general definitions of a superclass can be employed by various objects that add their specific definitions, using the overriding of the available class methods that are not instantiated properly in the superclass.
- It is excellent for reducing recompilation needs, by ensuring that even sections of a class can have a reusable structure, while the altering requirements may be achieved with the use of polymorphism.

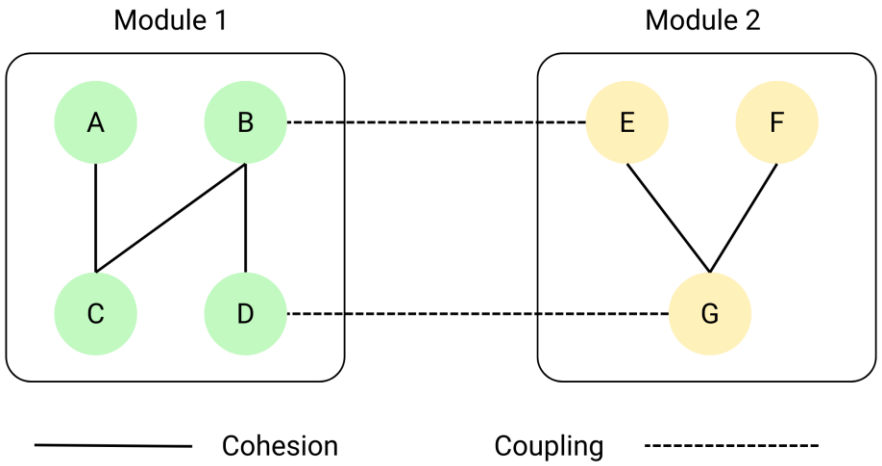
Difference between method overloading and method overriding in java

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.

4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Cohesion and Coupling in OOPs

- Cohesion and coupling are common concepts in designing modular software systems.
- Cohesion and Coupling are are used to measure the code quality and ensure it is maintainable and scalable.



Coupling

- **Coupling** is the indication that shows the relationship between modules or we can say the interdependence between modules.
- There are two types of coupling -
  - **Tight Coupling** (*a bad programming design*)
  - **Loose Coupling** (*a good programming design*)

Tight Coupling (*a bad programming design*)

- Tight coupling means that modules are closely connected.

- In case of tight coupling, changes in one module can affect others.
- In case of tight coupling, the classes and objects are dependent on each other and hence it reduces the re−usability of the code.
- If a class A has some ***public* data members** and another class B is accessing these data members ***directly using the dot operator***(which is possible because data members were declared **public**), the two classes are said to be **tightly coupled**.
- Such *tight coupling* between two classes leads to the ***bad designing***.

- **Example:**
  - Class **A** has an instance variable, *empName*, which is declared **public**.
  - Class **A** has two **public** *getter and setter* methods which check for a valid access and valid setting of data member - *empName*. The checking condition as follows:
    - A valid setting of the data member, *empName* i.e. *it cannot be set to a **null** value*.
    - A valid access of the data member, *empName* i.e. *it is only accessed when its value is **not null***.
  - Class **B** creates an object of class **A** and directly sets the value of its data member, *empName*, to **null** and directly accesses its value because it was declared **public**.
  - Thus, the validity checks for the data member, *empName*, which were implemented within **getName()** and **setName()** methods of class A are *bypassed*.
  - It shows that class **A** is *tightly coupled* to class **B**, and it is a *bad design*.

- **Coding Example of Tight Coupling**

```
class A {
        public String empName;      //public data member of A class

        //Checking a valid setting of instance variable, "empName"
        public void setName(String name){
                if (name==null)
                        System.out.println("You can't initialized name to a null");
                else
                        empName = name;
        }

        //Checking a valid access of instance variable, "empName"
        public String getName(){
                if(empName != null)
```

```
                return empName;
        else
                return "not initiaized";
    }

}

class B {
        public static void main(String[] args) {

            //Creating an object of class A
            A obj = new A();

            //Directly setting the value of data member " empName " of class A,
            //due to tight coupling between the class A and B
            obj. empName = null;

            //Direct access of data member " empName" of class A,
            //due to tight coupling between two classes
            System.out.println("Name is " + obj. empName);
        }
}
```

Output:

Name is null

Loose Coupling (*a good programming design*)

- Loose coupling means that modules are independent and changes in one module have minimal impact on others.
- Loose coupling is frequently used because changing one class does not affect another class.
    - Hence, it reduces dependencies on a class. Consequently, we can easily reuse it.
- Loose coupling avoid unnecessary dependency.
- A good application designing is creating an application with loosely coupled classes by following **encapsulation**.
    - By declaring data members of a class with the **private** access modifier, which disallows other classes to directly access these data members, and forcing them to call the ***public* getter, setter methods** to access these *private* data members.

- **Example:**
  - Class **A** has an instance variable, *name*, which is declared **private**.
  - Class **A** has two **public** *getter and setter* methods which check for a valid access and valid setting of data member - *empName*. The checking condition as follows:
    - A valid setting of the data member, *empName* i.e. *it cannot be set to a **null** value*.
    - A valid access of the data member, *empName* i.e. *it is only accessed when its value is **not null***.
  - Class **B** creates an object of class A, calls the **getName()** and **setName()** methods and their ***implemented checks are properly executed*** before the value of instance member, *empName*, is accessed or set.
  - It shows that class **A** is *loosely coupled* to class **B**, which is a *good programming design*.

- **Coding Example of Loose Coupling**

```
class A {

        //data member "empName" is declared private to implement loose coupling.
        private String empName;

        //Checking a valid setting of data member, empName
        public void setName(String name) {
                if (name == null)
                        System.out.println("You can't initialize name to a null");
                else
                        empName = name;
        }

        //Checking a valid access to data member, empName
        public String getName() {
                if(empName != null)
                        return empName;
                else
                        return "not initiaized";
        }

}
```

```
class B {
        public static void main(String[] args) {

                //Creating an object of class A
                A obj = new A();

                //Calling setter method, as the direct access of "empName"
                // is not possible i.e. loose coupling between classes
                obj.setName(null);

                //Calling getter method, as the direct access of "empName"
                // is not possible i.e. loose coupling
                System.out.println("Name is " + obj.getName());
        }
}
```

Output:

```
You can't initialized name to a null
Name is not initiaized
```

Cohesion

- **Cohesion** refers to the extent to which a class is defined to do a **specific specialized task**.
- **Cohesion** is an indication that shows the relationship within modules.
- Cohesion is basically the dependency between internal elements of modules like methods and internal modules.
- Cohesion provides the information about the functional strength of the modules.
- There are two types of cohesion -
    - **Low cohesion** (*a bad programming design*)
    - **High Cohesion** (*a good programming design*)

Low cohesion (*a bad programming design*)

- When a class is designed to do **many different tasks** rather than focus on a *single specialized task*, this class is said to be a "*low cohesive*" class.
- Low cohesive classes are said to be *badly designed*, as it requires a lot of work at creating, maintaining and updating them.
- **Example:**
    - A class **PlayerDatabase** which is performing **many different tasks** like connecting to a database, printing the information of all the players, printing information of a single player, printing all the events, printing all the rankings and finally closing all opened database connections.
    - Such a class is not easy to create, maintain and update, as it is involved in performing many different tasks i.e. *a programming design to avoid*.

- **Example of a Low Cohesion Class**

```
class PlayerDatabase {

        public void connectDatabase();
        public void printAllPlayersInfo();
        public void printSinglePlayerInfo();
        public void printRankings();
        public void printEvents();
        public void closeDatabase();
}
```

High Cohesion (*a good programming design*)

- *A high cohesive* classes, which are targeted towards a ***specific specialized task,*** rather than performing many different purposes.
- *These* classes are not only easy to create but also easy to maintain and update.
- The greater the cohesion, the better will be the program design.
- High cohesion will allow us to reuse the classes and methods.
- **Example:**
  - Created several different classes, where each class is performing a specific specialized task, which leads to an easy creation, maintenance and modification of these classes.
  - Classes created by following this programming design are said to performing a cohesive role and are termed as *high cohesion classes*.
  - It is an appropriate programming design to follow while creating an application.

- **Example of a High Cohesion Class**

```
class PlayerDatabase {
        ConnectDatabase connectD= new connectDatabase();
        PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
        PrintRankings rankings = new PrintRankings();
        CloseDatabase closeD= new CloseDatabase();
        PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();
}


class ConnectDatabase {
        //connecting to database.
}
```

```
class CloseDatabase {
        //closing the database connection.
}

class PrintRankings {
        //printing the players current rankings.
}

class PrintAllPlayersInfo {
        //printing all the players information.
}

class PrintSinglePlayerInfo {
        //printing a single player information.
}
```

Difference between Cohesion and Coupling

- The following table highlights all the important differences between cohesion and coupling –

Cohesion	Coupling
Cohesion is the measure of degree of relationship between elements of a module.	Coupling is the measure of degree of relationship between different modules.
It represents relationships within the module or we can say, it is an intra-module concept.	It represent the relationships between the modules or we can say, it is an inter module concept.
It represents the functional strength of the modules.	It represents the interdependency among the modules.
A module with high cohesion contains elements that are tightly related to each other and united in their purpose.	Two modules have high coupling (or tight coupling), if they are closely connected and dependent on each other.
A module is said to have low cohesion if it contains unrelated elements.	Modules with low coupling among them work mostly independently of each other.
When modules are highly cohesive, high quality software is built.	When the modules are loosely coupled, it results in high quality software.