Experiment No 4

60009210105
Amitesh
Sawarkar

D 12

Aim: -To Implement Inheritance and Interface in Java
Theory: -

1. Inheritance in Java

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
              .....
}
```

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).

The extends keyword is used to perform inheritance in Java.

For example,

```
class Animal {
  // methods and fields
}

// use of extends keyword
// to perform inheritance
class Dog extends Animal {

  // methods and fields of Animal
  // methods and fields of Dog
}
```

In the above example, the Dog class is created by inheriting the methods and fields from the

Animal class. Here, Dog is the subclass and Animal is the superclass.

is-a relationship

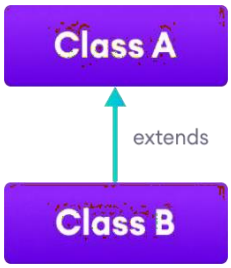
In Java, inheritance is an is-a relationship. That is, we use inheritance only if there exists an

is-a relationship between two classes. For example,
Car is a Vehicle
Orange is a Fruit
Surgeon is a Doctor
Dog is an Animal
Here, Car can inherit from Vehicle, Orange can inherit from Fruit, and so on.
super Keyword in Java Inheritance

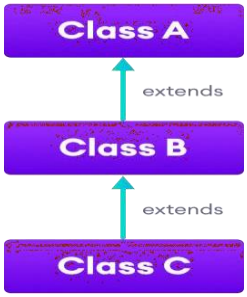
Super Keyword in Inheritance
Previously we saw that the same method in the subclass overrides the method in superclass. In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.
Example

```
class Animal {

  // method in the superclass
  public void eat() {
  System.out.println("I can eat");
  }
}
// Dog inherits  Animal
class Dog extends Animal {
  // overriding the eat() method
  @Override
  public void eat() {
    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }
  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}

class Main {
  public static void main(String[] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // call the eat() method
    labrador.eat();
    labrador.bark();
  }
}
```

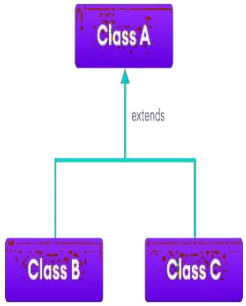
2. Types of Inheritance in Java
1. Single Inheritance
In single inheritance, a single subclass extends from a single superclass. For example,



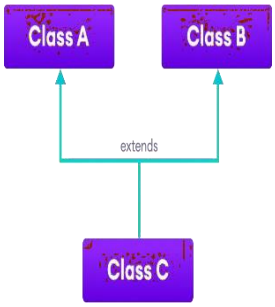
2. Multilevel Inheritance
In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



3. Hierarchical Inheritance
In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



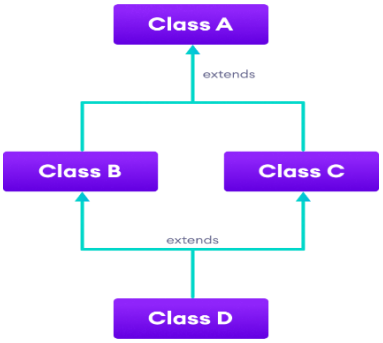
4. Multiple Inheritance
In multiple inheritance, a single subclass extends from multiple super classes. For example,



Note: Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces. To learn more, visit Java implements multiple inheritance.

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



Here, we have combined hierarchical and multiple inheritance to form a hybrid inheritance.

2. Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods. The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body. Java Interface also **represents the IS-A relationship**. **It cannot be instantiated just like the abstract class**. Since Java 8, we can have default and static methods in an interface. Since Java 9, we can have private methods in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

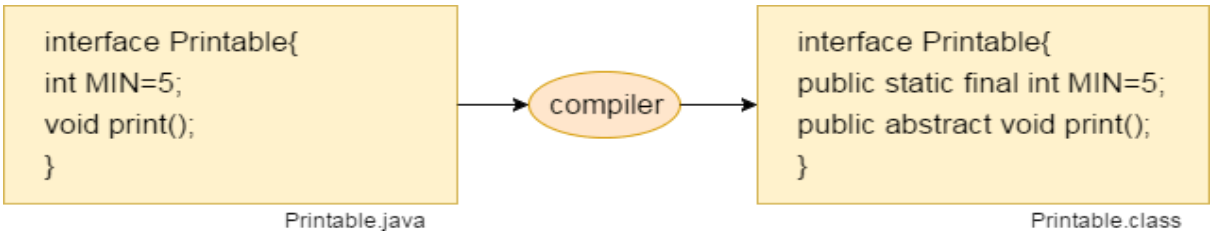
Syntax:

```
interface <interface name> {

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

Internal addition by the compiler

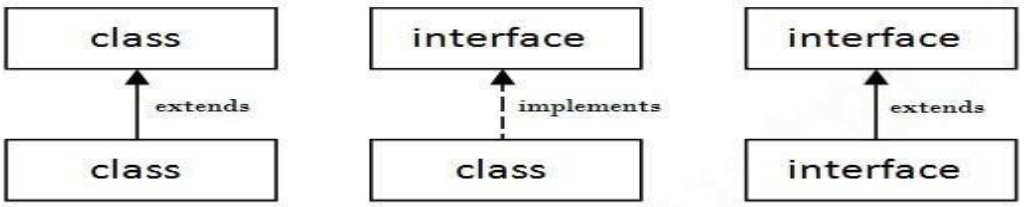
The **Java compiler adds public and abstract keywords before the interface method**. Moreover**, it adds public, static and final keywords before data members**.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.



The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable {
void print ();
}
class A6 implements printable {
public void print () {System.out.println("Hello");}
public static void main (String args []) {
A6 obj = new A6();
obj.print ();
 }
}
```

3. Polymorphism in Java

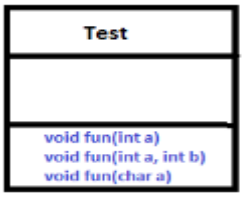
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real-life Illustration: Polymorphism

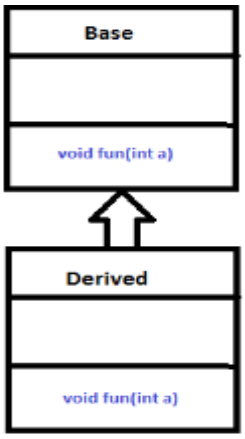
A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, so it means many forms.

Types of polymorphism
In Java polymorphism is mainly divided into two types:
Compile-time Polymorphism
Runtime Polymorphism



Overloading



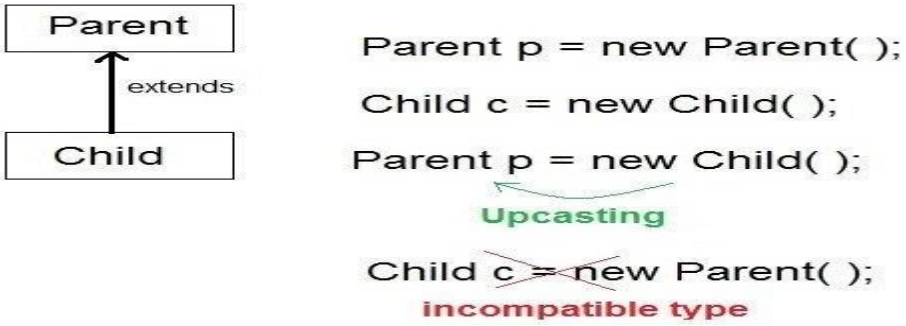
Overriding

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.

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. **Method overriding**, on the other hand, 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**.

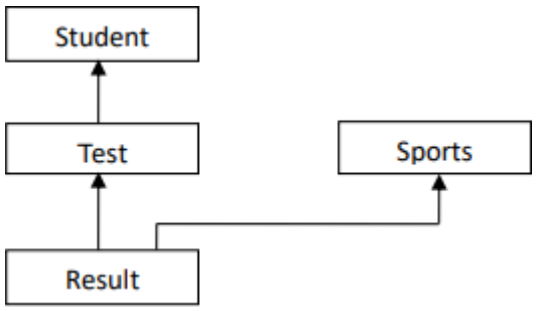
Dynamic method dispatch using base class and interface reference in Java

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



4. Lab Assignments to complete in this session

i. WAP to implement three classes namely Student, Test and Result. Student class has member as rollno, Test class has members as sem1_marks and sem2_marks and Result class has member as total. Create an interface named sports that has a member score (). Derive Test class from Student and Result class has multiple inheritances from Test and Sports. Total is formula based on sem1_marks, sem2_mark and score.



```java
interface Sports {
    int score();
}

class Student {
    int rollno;

    public Student(int rollno) {
        this.rollno = rollno;
    }
}

class Test extends Student {
    int sem1_marks;
    int sem2_marks;

    public Test(int rollno, int sem1_marks, int sem2_marks) {
        super(rollno);
        this.sem1_marks = sem1_marks;
        this.sem2_marks = sem2_marks;
    }
}

class Result extends Test implements Sports {
    int sportsScore;

    public Result(int rollno, int sem1_marks, int sem2_marks, int sportsScore) {
        super(rollno, sem1_marks, sem2_marks);
        this.sportsScore = sportsScore;
    }

    public int total() {

        return sem1_marks + sem2_marks + score();
    }
```

```java
    @Override
    public int score() {

        return sportsScore;
    }
}

public class Q1 {
    Run | Debug
    public static void main(String[] args) {
        int studentRollno = 123;
        int sem1Marks = 85;
        int sem2Marks = 92;
        int sportsScore = 10;

        Result resultInstance = new Result(studentRollno, sem1Marks, sem2Marks, sportsScore);

        int totalMarks = resultInstance.total();
        System.out.println("Roll No: " + resultInstance.rollno);
        System.out.println("Semester 1 Marks: " + resultInstance.sem1_marks);
        System.out.println("Semester 2 Marks: " + resultInstance.sem2_marks);
        System.out.println("Sports Score: " + resultInstance.score());
        System.out.println("Total Marks: " + totalMarks);



    }
}
```

```
Roll No: 123
Semester 1 Marks: 85
Semester 2 Marks: 92
Sports Score: 10
Total Marks: 187
```

ii. Write an abstract class program to calculate area of circle, rectangle and triangle

```java
package Question2;

abstract class Shape {
    public abstract double area();
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public double area() {
        return length * width;
    }
}
```

```java
class Triangle extends Shape {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    @Override
    public double area() {
        return 0.5 * base * height;
    }
}

class Q2 {
    Run | Debug
    public static void main(String[] args) {

        Circle circle = new Circle(radius:5.0);
        Rectangle rectangle = new Rectangle(length:4.0, width:6.0);
        Triangle triangle = new Triangle(base:3.0, height:8.0);
        System.out.println("Area of Circle: " + circle.area());
        System.out.println("Area of Rectangle: " + rectangle.area());
        System.out.println("Area of Triangle: " + triangle.area());
    }
}
```

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
Area of Triangle: 12.0
```

iii. Create the Account class Account.java and write a main method in a different class to briefly experiment with some instances of the Account class.

Using the Account class as a base class, write two derived classes called SavingsAccount and CurrentAccount. A SavingsAccount object, in addition to the attributes of an Account object, should have an interest variable and a method which adds interest to the account. A CurrentAccount object, in addition to the attributes of an Account object, should have an overdraft limit variable. Ensure that you have overridden methods of the Account class as necessary in both derived classes.

Now create a Bank class, an object of which contains an array of Account objects. Accounts in the array could be instances of the Account class, the SavingsAccount class, or the CurrentAccount class. Create some test accounts (some of each type).

Write an update method in the bank class. It iterates through each account, updating it in the following ways: Savings accounts get interest added (via the method you already wrote); CurrentAccounts get a letter sent if they are in overdraft.

The Bank class requires methods for opening and closing accounts, and for paying a dividend into each account. Hints:

Note that the balance of an account may only be modified through the deposit(double) and withdraw(double) methods.

The Account class should not need to be modified at all.
Be sure to test what you have done after each step.

```java
package Question3.Banking;

class Account {
    private int accountNumber;
    private String accountHolder;
    protected double balance;

    public Account(int accountNumber, String accountHolder, double balance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public String getAccountHolder() {
        return accountHolder;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited $" + amount + " into Account #" + accountNumber);
        }
    }
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn $" + amount + " from Account #" + accountNumber);
        } else {
            System.out.println("Insufficient funds in Account #" + accountNumber);
        }
    }

    @Override
    public String toString() {
        return "Account #" + accountNumber + " (" + accountHolder + "): $" + balance;
    }
}
```

Question3 > Banking > Bank.java > {} Question3.Banking

```java
package Question3.Banking;
import java.util.ArrayList;
import java.util.List;

public class Bank {
    private List<Account> accounts;

    public Bank() {
        accounts = new ArrayList<>();
    }

    public void openAccount(Account account) {
        accounts.add(account);
        System.out.println("Opened a new account: " + account);
    }

    public void closeAccount(Account account) {
        accounts.remove(account);
        System.out.println("Closed account: " + account);
    }

    public void payDividend(double dividend) {
        for (Account account : accounts) {
            account.deposit(dividend);
        }
    }
    public void update() {
        for (Account account : accounts) {
            if (account instanceof SavingsAccount) {
                ((SavingsAccount) account).addInterest();
            } else if (account instanceof CurrentAccount) {
                if (account.getBalance() < 0) {
                    System.out.println("Letter sent for overdraft on Current Account #" + account.getAccountNumber());
                }
            }
        }
    }
}
```

Question3 > Banking > J CurrentAccount.java > {} Question3.Banking

```java
package Question3.Banking;

class CurrentAccount extends Account {
    private double overdraftLimit;

    public CurrentAccount(int accountNumber, String accountHolder, double balance, double overdraftLimit) {
        super(accountNumber, accountHolder, balance);
        this.overdraftLimit = overdraftLimit;
    }

    @Override
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance + overdraftLimit) {
            balance -= amount;
            System.out.println("Withdrawn $" + amount + " from Current Account #" + getAccountNumber());
            if (balance < 0) {
                System.out.println("Account #" + getAccountNumber() + " is in overdraft.");
            }
        } else {
            System.out.println("Insufficient funds in Current Account #" + getAccountNumber());
        }
    }

    @Override
    public String toString() {
        return super.toString() + " (Overdraft Limit: $" + overdraftLimit + ")";
    }
}
```

Question3 > Banking > J Main.java > {} Question3.Banking

```java
package Question3.Banking;

public class Main {
    Run | Debug
    public static void main(String[] args) {

        SavingsAccount savingsAccount = new SavingsAccount(accountNumber:1, accountHolder:"John", balance:1000, interestRate:3.5);
        CurrentAccount currentAccount = new CurrentAccount(accountNumber:2, accountHolder:"Alice", balance:2000, overdraftLimit:500);
        Account normalAccount = new Account(accountNumber:3, accountHolder:"Bob", balance:1500);

        Bank bank = new Bank();
        bank.openAccount(savingsAccount);
        bank.openAccount(currentAccount);
        bank.openAccount(normalAccount);

        savingsAccount.deposit(amount:500);
        currentAccount.withdraw(amount:1000);
        normalAccount.withdraw(amount:200);

        bank.payDividend(dividend:50);

        bank.update();

        normalAccount.toString();
    }
}
```

Question3 > Banking > J SavingsAccount.java > {} Question3.Banking

```java
package Question3.Banking;

class SavingsAccount extends Account {
    private double interestRate;

    public SavingsAccount(int accountNumber, String accountHolder, double balance, double interestRate) {
        super(accountNumber, accountHolder, balance);
        this.interestRate = interestRate;
    }

    public void addInterest() {
        double interestAmount = balance * interestRate / 100;
        deposit(interestAmount);
        System.out.println("Interest added to Account #" + getAccountNumber() + ": $" + interestAmount);
    }

    @Override
    public String toString() {
        return super.toString() + " (Interest Rate: " + interestRate + "%)";
    }
}
```

```
Opened a new account: Account #1 (John): $1000.0 (Interest Rate: 3.5%)
Opened a new account: Account #2 (Alice): $2000.0 (Overdraft Limit: $500.0)
Opened a new account: Account #3 (Bob): $1500.0
Deposited $500.0 into Account #1
Withdrawn $1000.0 from Current Account #2
Withdrawn $200.0 from Account #3
Deposited $50.0 into Account #1
Deposited $50.0 into Account #2
Deposited $50.0 into Account #3
Deposited $54.25 into Account #1
Interest added to Account #1: $54.25
```

iv. Create a class called Employee whose objects are records for an employee. This class will be a derived class of the class Person which you will have to copy into a file of your own and compile. An employee record has an employee's name (inherited from the class Person), an annual salary represented as a single value of type double, a year the employee started work as a single value of type int and a national insurance number, which is a value of type String. Your class should have a reasonable number of constructors and accessor methods, as well as an equals method. Write another class containing a main method to fully test your class definition

Question4 > Employee > employee > J Employee.java > {} Question4.Employee.employee

```java
package Question4.Employee.employee;
import Question4.Employee.person.*;

public class Employee extends Person {
    private double annualSalary;
    private int startYear;
    private String nationalInsuranceNumber;

    public Employee(String name, double annualSalary, int startYear, String nationalInsuranceNumber) {
        super(name);
        this.annualSalary = annualSalary;
        this.startYear = startYear;
        this.nationalInsuranceNumber = nationalInsuranceNumber;
    }

    public double getAnnualSalary() {
        return annualSalary;
    }

    public int getStartYear() {
        return startYear;
    }

    public String getNationalInsuranceNumber() {
        return nationalInsuranceNumber;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        Employee other = (Employee) obj;
        return Double.compare(other.annualSalary, annualSalary) == 0
                && startYear == other.startYear
                && nationalInsuranceNumber.equals(other.nationalInsuranceNumber)
                && getName().equals(other.getName());
    }

    @Override
    public String toString() {
        return "Employee: " + getName() +
                "\nAnnual Salary: $" + annualSalary +
                "\nStart Year: " + startYear +
                "\nNational Insurance Number: " + nationalInsuranceNumber;
    }
}
```

Question4 > Employee > person > Person.java > {} Question4.Employee.person

```java
package Question4.Employee.person;

public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Question4 > Employee > Main.java > {} Question4.Employee

```java
package Question4.Employee;

import Question4.Employee.employee.*;

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Employee emp1 = new Employee(name:"John Doe", annualSalary:60000, startYear:2015, nationalInsuranceNumber:"NI12345");
        Employee emp2 = new Employee(name:"Jane Smith", annualSalary:55000, startYear:2017, nationalInsuranceNumber:"NI54321");

        System.out.println(x:"Employee 1 Details:");
        System.out.println("Name: " + emp1.getName());
        System.out.println("Annual Salary: $" + emp1.getAnnualSalary());
        System.out.println("Start Year: " + emp1.getStartYear());
        System.out.println("National Insurance Number: " + emp1.getNationalInsuranceNumber());

        System.out.println("\nAre Employee 1 and Employee 2 the same? " + emp1.equals(emp2));

        System.out.println("\nEmployee 1 Details:\n" + emp1);
        System.out.println("\nEmployee 2 Details:\n" + emp2);
    }
}
```

```
Employee 1 Details:
Name: John Doe
Annual Salary: $60000.0
Start Year: 2015
National Insurance Number: NI12345

Are Employee 1 and Employee 2 the same? false

Employee 1 Details:
Employee: John Doe
Annual Salary: $60000.0
Start Year: 2015
National Insurance Number: NI12345

Employee 2 Details:
Employee: Jane Smith
Annual Salary: $55000.0
Start Year: 2017
National Insurance Number: NI54321
```