
UNIT 2 POLYMORPHISM

Structure

Page no.

- 2.0 Introduction
- 2.1 Objectives
- 2.2 Introduction to Polymorphism
- 2.3 Advantages of Polymorphism
- 2.4 Types of Polymorphism
- 2.5 Method of Overloading
- 2.6 Method of Overriding
- 2.7 Abstract Class
- 2.8 Application of Abstract Class
- 2.9 Summary
- 2.10 Solutions/Answer to Check Your Progress
- 2.11 References/Further Reading

2.0 INTRODUCTION

Polymorphism literally means “having many forms” and is used as a feature in object-oriented programming that provides one interface for a general class of actions. The specific action is implemented to address the exact nature of the situation.



Figure 1: An example of polymorphism

Polymorphism in Java is the ability of an entity to take many forms as per requirement. As shown in figure 1, a man standing in the middle is only one, but he takes multiple roles like a dad for his children, an employee, a salesperson, and many more. This is known as polymorphism because one person taking many roles.

Another example may be the saving of two contact numbers of the same person. What we do for this? We save these two numbers with the same name or in other words both number can be saved in one contact name only. That is also a kind of polymorphism. In a similar fashion, in java programming, one object can take multiple forms depending on the context of the program. In this unit, you will learn the concept of polymorphism and an important concept known as abstract class.

2.1 OBJECTIVES

After reading this unit, you will be able to:

- ... Explain the basics of polymorphism,
- ... Write the program on method overloading and method overriding,
- ... Differentiate between overloading and overriding,
- ... Use the abstract class and its application in programming, and
- ... Explain the difference between abstract class and interface.

2.2 INTRODUCTION TO POLYMORPHISM

The objective behind polymorphism in Java is to facilitate you for using the methods inherited by inheritance to perform different tasks. The object can take many forms. If there are one or more classes or objects related to each other by inheritance in a program, then polymorphism occurs. So, here you can say that the goal of polymorphism is communication, but it follows a different approach.

As you know, inheritance represents the IS-A relationship, so any java object is polymorphic if it can pass more than one IS-A test.

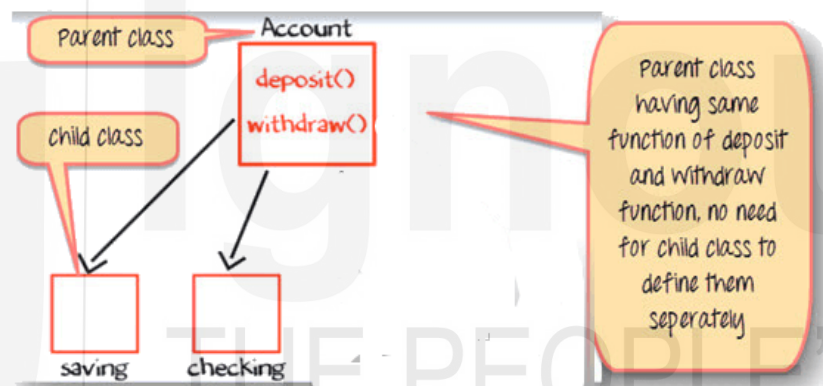


Figure 2 A scenario of inheritance

In figure 2, Account is the parent class with two methods: deposit() and withdraw(). Also, Account class has two child classes: Savings and Current, and both the child classes performs the same operation of deposit and withdraw. So the child classes will not define the methods again and use the inherited one.

The concept of polymorphism in Java makes it possible to perform a single action in different ways with the idea of reusability. In figure 3, it is shown that all having speak() method in their interface, but each one of them performs it differently. All the descendants have performed with the same heads but with different method bodies.



Figure 3: Single action in different ways

Let us discuss the programming aspects of polymorphism of Java with the help of a simple example in which “Animal” is a parent class and “Cat” is a child class. Both the parent and child classes are using the same method makeSound().

Programming Example 1:

```
class Animal
{
    /*Animal is a parent class*/
    void makeSound()
    {
        System.out.println("Now Speak!");
    }
}
class Cat extends Animal
{
    /*Cat is a child class*/
    void makeSound()
    {
        System.out.println("Meow");
    }
}
class PolymorphismExample
{
    public static void main(String args[])
    {
        Animal a = new Animal(); /*creating object*/
        Cat d = new Cat();        /*creating object*/
        a.makeSound();
        d.makeSound();
    }
}
```

Output:

Now Speak!
Meow

In the above example program, you can see that both the classes are using the same method in different ways.

2.3 ADVANTAGES OF POLYMORPHISM

The major benefit of polymorphism is the reusability of codes which saves a lot of time and effort. Existing old classes and codes that were once tested and already implemented can be reused wherever required by using the concept of polymorphism, and the new functionality in the existing system can be added using the same interface.

Another benefit of polymorphism is that multiple data types(double, float, int, long, etc) can be stored in a single variable, making it easier to search for and implement these variables. You can use the single variable name to represent commonality in the features.

Also, Coupling between two different methods, i.e. the degree of interaction that violates the principle of hiding information, can be reduced by polymorphism.

2.4 TYPES OF POLYMORPHISM

In general, Java programming supports types of polymorphism which are as follows.

- ... Static Polymorphism
- ... Dynamic Polymorphism

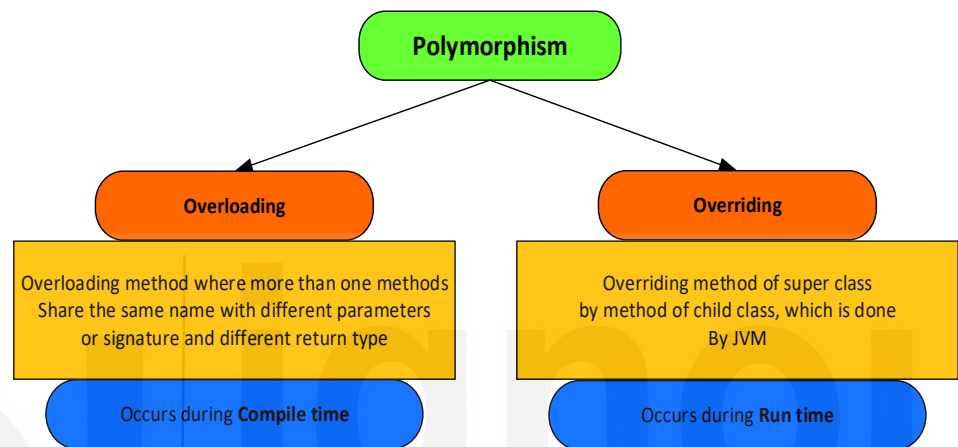


Figure 4. Types of Polymorphism

Static polymorphism is also known as **compile-time polymorphism** because compiler resolves the polymorphism during the compilation of the program by checking the method signatures. Static polymorphism can be achieved through **method overloading** with the same name but different parameters., which means the name of the methods is the same but the signature of the methods is different.

Dynamic polymorphism is also known as **run time polymorphism** because JVM(Java Virtual Machine) resolves the call to an overridden method at the runtime of the program. Dynamic polymorphism can be achieved by method overriding with the same name and parameters but in different classes. In the coming sections, you will learn more about method overloading and method overriding in detail.

2.5 OVERLOADING OF METHOD

Method overloading is one of the ways through which polymorphism is supported by Java. Method overloading is a concept of declaring multiple methods with the same name and different parameters of the same or different data types in the same class. Method overloading is compile-time or static polymorphism.

When an overloaded method is invoked, Java uses the number of parameters or data types of the parameters as its guide to distinguish which version of the overloaded method is to be called. Let us take an example of a method `sum(int x, int y)` having two parameters is different from the argument list of a method `sum(int x, int y, int z)` having three parameters.

There are three ways to overload a method:

1. There are different number of parameters in the argument list of the methods.

Example: sum(int, int)
 sum(int, int, int)

2. There are different data types of the parameters in the argument list.

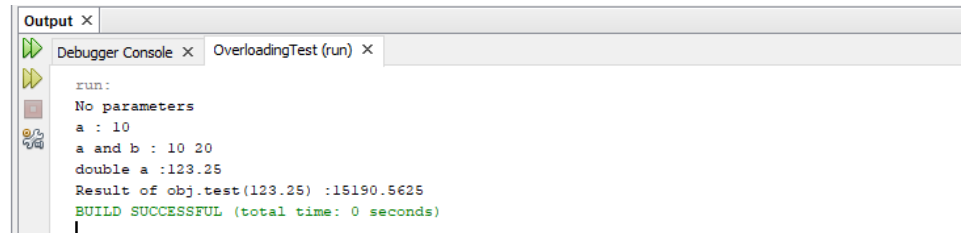
Example: sum(float, float)
 sum(float, int)

3. There is a different sequence of data types of the parameters.

Example: sum(float, int)
 sum(int, float)

Here is a simple example to illustrate the concept of method overloading.

```
/*-----Program to demonstrate method overloading-----*/
package overloadingtest;
class DemoOverload
{
void test( )
{
System.out.println("No parameters");
}
/*-----overload test for one integer parameter-----*/
void test(int a)
{
System.out.println("a : " + a);
}
/*-----overload test for two integer parameters-----*/
void test(int a, int b)
{
System.out.println("a and b : " + a + " " + b);
}
/*-----overload test for a double parameter-----*/
double test(double a)
{
System.out.println("double a : " + a);
return a*a;
}
}
class MyOverloading
{
public static void main(String args[])
{
DemoOverload obj = new DemoOverload( );
double result;
/*-----call all versions of test( )-----*/
obj.test( );
obj.test(10);
obj.test(10,20);
result = obj.test(123.25);
System.out.println("Result of obj.test(123.25) : " + result);
}
}
```

Output:


```

Output ×
Debugger Console × OverloadingTest (run) ×

run:
No parameters
a : 10
a and b : 10 20
double a :123.25
Result of obj.test(123.25) :15190.5625
BUILD SUCCESSFUL (total time: 0 seconds)

```

In the above program, test() is overloaded four times. The first version of test() takes no parameters, the second version takes one parameter of integer type, the third version takes two parameters of integer types, and the last version takes one parameter of a double data type with the double return type. However, return type does not play any role in method overloading.

CHECK YOUR PROGRESS-1

Q1. What is method overloading?

Q2. What is compile-time polymorphism?

Q3 Write a program to demonstrate the overloading of three methods with the same name.

2.6 OVERRIDING OF METHOD

In a class hierarchy, when a method in the subclass has the same name and parameters as in its superclass, that method is said to override the method in superclass. When an overridden method is called from its subclass, then always the version of subclass

method will be referred to, and the version of the method of superclass will remain hidden. This mechanism provides an opportunity for the subclass to have its own specific implementations of the overridden methods.

Overriding the method in Java is one of the ways to achieve runtime polymorphism.

Consider the following example:

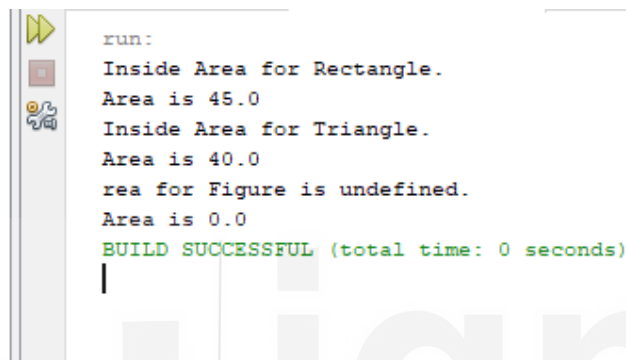
```
class Figure
{
    /*Using runtime polymorphism*/
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    double area( )
    {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}
class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a, b);
    }
    double area( )
    {
        /*Override area for Rectangle*/
        System.out.println("Inside Area for Rectangle.");
        return dim1*dim2;
    }
}
class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a, b);
    }
    double area( )
    {
        /*Override area for Right Triangle*/
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class FindAreas
{
    public static void main(String args[])
    {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
    }
}
```

```

        Figure ref;
        ref = r;
        System.out.println("Area is " + ref.area());
        ref = t;
        System.out.println("Area is " + ref.area());
        ref = f;
        System.out.println("Area is " + ref.area());
    }
}

```

Output of the program:



```

run:
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
rea for Figure is undefined.
Area is 0.0
BUILD SUCCESSFUL (total time: 0 seconds)

```

In the above example, there is a superclass called Figure that stores the dimensions of two-dimensional objects. It also defines the method `area()` that calculates the area of an object. The two subclasses: Rectangle and Triangle are derived from the superclass Figure. Both the subclasses override the method `area()` to return their respective areas.

Certain rules must be followed while overriding the methods:

1. While overriding child class method signature and parent class method signatures must be the same; otherwise, it will generate compilation error.
2. The return types of overridden method and overriding method must be the same.
3. Methods declared final cannot be overridden.
4. Static methods are bounded with class. Hence, it cannot be overridden.
5. The access level of methods cannot be more restrictive than the overridden method's access level.

2.7 ABSTRACT CLASS

Sometimes, there are certain situations where you want to create a superclass that only defines the generalized form that will be shared by all of its subclasses, leaving the details to be filled by the subclasses. In Java, an abstract method is a solution to this problem.

Abstract methods have only their declaration in the superclass. They are not defined in the superclass. These methods are overridden by the subclasses. The general syntax to declare abstract method is:

```
abstract type name(parameter-list);
```

If any class has one or more abstract methods, it must be declared as abstract. This is done by simply using the **abstract** keyword. The abstract keyword is placed in front

of the **class** keyword at the beginning of the class declaration. Instance of an abstract class cannot be directly created using **new** operator because its object cannot be created. It is also called an incomplete class as it is not fully defined. Abstract constructors or abstract static methods cannot be declared. Any subclass of an abstract class must either implement all the abstract methods of superclass or declare itself **abstract**.

You need to consider few key points for the abstract class.

- ... An abstract class is always declared with an abstract keyword.
- ... It can contain both abstract as well as non-abstract methods.
- ... You cannot instantiate the abstract class.
- ... Also, it can have final method if needed, which forces the subclasses not to change it.

Let us consider a simple example of a class with abstract method, followed by a class that implements that method.

```
abstract class Bank
{
    /*abstract class declaration*/
    abstract int getRateOfInterest( );
    /*abstract method declaration*/
}
class SBI extends Bank
{
    int getRateOfInterest( )
    {
        /*overriding abstract method*/
        return 7;
    }
}
class PNB extends Bank
{
    int getRateOfInterest( )
    {
        /*overriding abstract method*/
        return 8;
    }
}
class TestBank
{
    public static void main(String args[])
    {
        Bank b;    /*cannot be directly instantiated with new operator*/
        b = new SBI( );
        System.out.println("Rate of Interest is: " + b.getRateOfInterest() + "%");
        b = new PNB( );
        System.out.println("Rate of Interest is: " + b.getRateOfInterest() + "%");
    }
}
```

Output:

Rate of Interest is: 7%
Rate of Interest is: 8%

The above example consists of one superclass Bank which is declared abstract with an abstract method `getRateOfInterest()`. SBI and PNB are two subclasses, overriding the method `getRateOfInterest()`.

Let's take an example-based on employee and department to discuss the abstract class and abstract method.

```
abstract class Employee
{
    final int salary = 20000;
    public void display()
    {
        System.out.println("This is a method to display the employee information.");
    }
    abstract public void Department();
}
public class Teacher extends Employee
{
    public static void main(String args[])
    {
        Teacher obj = new Teacher();
        obj.display();
        obj.Department();
        //obj.salary=30000;
    }
    /*public void Department()
    {
        System.out.println("Computer Science Department");
    }
    */
}
```

Output of the program:

```
run:
java.lang.ExceptionInInitializerError Caused by: java.lang.RuntimeException:
Uncompilable source code - Overridingtest.Teacher is not abstract and does not
override abstract method Department() in Overridingtest.Employee
    at Overridingtest.Teacher.<clinit>(Teacher.java:12)
Exception in thread "main"
C:\Users\DELL\AppData\Local\NetBeans\Cache\8.2rc\executor-snippets\run.xml:53:
Java returned: 1
BUILD FAILED (total time: 2 seconds)
```

In this example, you can see that output of the program is throwing an error because the abstract method in Employee class must be defined in derived classes. However, it is also defined but commented. That is the reason it is throwing an error. If you remove the comment and method is part of the code, then it will execute properly. Try to do it as a practice exercise.

2.8 APPLICATION OF ABSTRACT CLASS

In this section, you will see an application of Abstract class along with the uses of Interface. Interface is also having similar uses as abstract class, and it is defined as a collection of methods. More about interface you will learn in unit 4 of this course. If you need to know more about interface in detail please refer unit 4 of this block then return back here to proceed further. Before discussing the application, let's have a look on the comparative chart of Abstract class and interface.

Abstract Class	Interface
1. Any class can be marked as abstract by using the "abstract" keyword. It can have both abstract and non-abstract methods.	1. Interface is defined as with the help of interface keyword. It can contain only abstract methods.
2. It does not support multiple inheritance.	2. Multiple inheritance can be achieved (implemented) using interface.
3. In abstract class, you can extend java class and implements many java interfaces.	3. Only one java interface can be extended.
4. Members of abstract class can be public, private, and protected etc.	4. Member of java interface are by default public.

Let's have a look on the program given below, in this program, you can see that Payee is an interface and SalaryEmployee, and CommissionEmployee are the two classes. These classes implement the interface Payee.

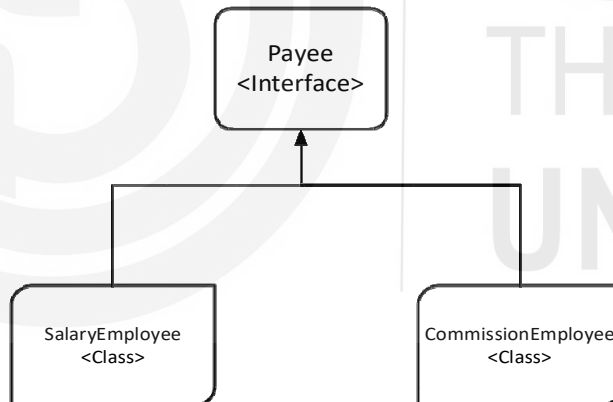


Figure 5. A scenario of interface implementation

```
/*-----Payee Interface-----*/
```

```
package Jtest;
public interface Payee
{
    String Ename( );
    Double EgrossPayment( );
    Integer EbankAccount( );
}
```

The class PaymentSystem is implemented to store the payee's information with the help of ArrayList.

```

/*-----Payment System class-----*/
package Jtest;

import java.awt.List;
import java.util.ArrayList;

public class PaymentSystem
{
    private ArrayList<Object> payees;
    public PaymentSystem( )
    {
        payees = new ArrayList<>( );
    }
    public void addPayee(Payee payee)
    {
        if(!payees.contains(payee))
        {
            payees.add(payee);
        }
    }
    public void processPayments( )
    {
        for(Object payee: payees)
        {
            Double grossPayment = ((Payee) payee).EgrossPayment( );
            System.out.println("Paying to"+((Payee) payee).Ename( ));
            System.out.println("tGrosst"+grossPayment);
            System.out.println("tTransferred to account:"+ ((Payee)
            payee).EbankAccount());
        }
    }
}

```

The SalaryEmployee class is to set the name, bank account details and gross wage of the employee by implementing the Payee interface.

```

/*-----SalaryEmployee class-----*/
package Jtest;

public class SalaryEmployee implements Payee
{
    private String name;
    private Integer bankAccount;
    protected Double grossWage;
    public SalaryEmployee (String name, Integer bankAccount, Double grossWage)
    {
        this.name = name;
        this.bankAccount = bankAccount;
        this.grossWage = grossWage;
    }
    public Integer EbankAccount()
    {
        return bankAccount;
    }
    public String Ename()
    {
        return name;
    }
}

```

```
}  
public Double EgrossPayment()  
  
{  
    return grossWage;  
}  
}
```

The CommissionEmployee class sets the information of the employee by implementing the Payee interface. It has also implemented some addition methods which are doCurrentCommission and giveCommission.

```
/*-----CommissionEmployee Class-----*/  
package Jtest;  
  
public class CommissionEmployee implements Payee  
{  
    private String name;  
    private Integer bankAccount;  
    protected Double grossWage;  
    private Double grossCommission = 0.0;  
    public CommissionEmployee(String name, Integer bankAccount, Double grossWage)  
    {  
        this.name = name;  
        this.bankAccount = bankAccount;  
        this.grossWage = grossWage;  
    }  
    public Integer EbankAccount()  
    {  
        return bankAccount;  
    }  
    public String Ename()  
    {  
        return name;  
    }  
  
    public Double EgrossPayment()  
    {  
        return grossWage + doCurrentCommission();  
    }  
  
    public Double doCurrentCommission()  
    {  
        Double commission = grossCommission;  
        grossCommission = 0.0;  
        return commission;  
    }  
    public void giveCommission (Double amount)  
    {  
        grossCommission +=amount;  
    }  
}
```

In the above two, we can see that you can see that SalaryEmployee and CommissionEmployee abstract methods are implemented again and again because the classes that are using the Payee interface, they have to define the Payee methods.

```

/*-----Employee class-----*/
package Jtest;

public abstract class Employee implements Payee
{
    private String name;
    private Integer bankAccount;
    protected Double grossWage;
    public Employee(String name, Integer bankAccount, Double grossWage)
    {
        this.name = name;
        this.bankAccount = bankAccount;
        this.grossWage = grossWage;
    }
    public String name( )
    {
        return name;
    }
    public Integer bankAccount( )
    {
        return bankAccount;
    }
}

/*-----Main PaymentApplication class-----*/
package Jtest;

public class PaymentApplication
{
    public static void main(final String []args)
    {
        PaymentSystem paymentSystem = new PaymentSystem();
        CommissionEmployee raviSingh = new CommissionEmployee("Ravi Singh", 5000, 301.0);
        paymentSystem.addPayee(raviSingh);
        CommissionEmployee sksingh = new CommissionEmployee("Sunil Singh", 6000, 545.0);
        paymentSystem.addPayee(sksingh);
        SalaryEmployee maryBrown = new SalaryEmployee("Mary Brown", 7000, 500.0);
        paymentSystem.addPayee(maryBrown);
        SalaryEmployee susanWhite = new SalaryEmployee("Susan White", 8000, 555.0);
        paymentSystem.addPayee(susanWhite);

        raviSingh.giveCommission(40.0);
        raviSingh.giveCommission(35.0);
        raviSingh.giveCommission(45.0);

        sksingh.giveCommission(50.0);
        sksingh.giveCommission(51.0);
        sksingh.giveCommission(23.0);
        sksingh.giveCommission(14.5);
        sksingh.giveCommission(57.3);

        paymentSystem.processPayments();
    }
}

```

You can see the output of the program given below also visualize the implementation of abstract methods in the classes.

Output of the program:

Paying toRavi Singh
tGrosst421.0
tTransferred to account:5000
Paying toSunil Singh
tGrosst740.8
tTransferred to account:6000
Paying toMary Brown
tGrosst500.0
tTransferred to account:7000
Paying toSusan White
tGrosst555.0
tTransferred to account:8000

Now look at the scenario given in the figure,

/*-----Modified classes to reduce the code duplicity-----*/

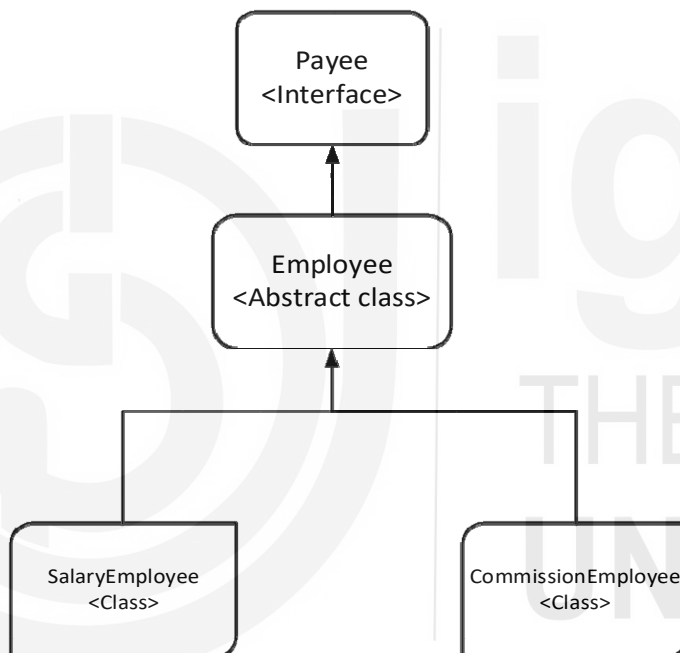


Figure 6 A Scenario to interface class

CHECK YOUR PROGRESS-2

Q1. Why do we use polymorphism in Java?

.....

.....

.....

.....

Q2. What do you mean by polymorphic parameters? Explain it with the help of suitable program.

.....

.....

.....

.....

Q3 Find the answers to the program given below.

It return type, method name and argument list is the same.

```
class Demo
{
    public int myMethod(int num1, int num2)
    {
        System.out.println("First myMethod of class Demo");
        return num1+num2;
    }
    public int myMethod(int var1, int var2)
    {
        System.out.println("Second myMethod of class Demo");
        return var1-var2;
    }
}
class Sample4
{
    public static void main(String args[])
    {
        Demo obj1= new Demo();
        obj1.myMethod(10,10);
        obj1.myMethod(20,12);
    }
}
```

.....

.....

.....

Q3 Find the answers of the program given below.

Its return type is different, while method name and argument list is same.

```
class Demo2
{
    public double myMethod(int num1, int num2)
    {
        System.out.println("First myMethod of class Demo");
        return num1+num2;
    }
    public int myMethod(int var1, int var2)
    {
        System.out.println("Second myMethod of class Demo");
        return var1-var2;
    }
}
class Sample5
{
    public static void main(String args[])
    {

```



```
{
    Demo2 obj2= new Demo2();
    obj2.myMethod(10,10);
    obj2.myMethod(20,12);
}
```

.....

.....

.....

.....

2.9 SUMMARY

In this unit, we have discussed the important concept known as polymorphism. This unit also discussed the types of polymorphism. Also, the concepts of overloading and overriding have been explained with the help of example programs. Property inheritance is discussed with the help of extends and implements keywords. Furthermore, in the end, we have discussed, code reusability concept using abstract class instead of using the implement method of interface.

2.10 SOLUTION/ANSWER TO CHECK YOUR PROGRESS

Check you progress 1:

1. Answer 1.

Whenever several methods have the same names with:

- ... Different method signatures and different numbers or types of parameters.
- ... Same method signature but the different number of parameters.
- ... Same method signature and the same number of parameters but of a different data type.

It is determined at the compile time.

2. Answer 2.

Compile-Time Polymorphism: The best example of compile-time or static polymorphism is the method overloading. Whenever an object is bound with their functionality at compile time is known as compile-time or static polymorphism in java.

3. Answer 3.

The class Summation given below have the three methods with the same name sum.

```
public class Summation{
    public int SUM(int m , int n){
        return (m + n);
    }
    public int SUM(int m , int n , int p){
        return (m + n + p) ;
    }
}
```

```
public double SUM(double m , double n){
    return (m + n);
}
public static void main( String args[]){
    Summation ob = new Summation();
    ob.SUM(15,25);
    ob.SUM(15,25,35);
    ob.SUM(11.5 , 22.5);
}
}
```

Check you progress 2:

1.

Polymorphism in Java makes possible to write a method that can correctly process lots of different types of functionalities that have the same name. We can also gain consistency in our code by using polymorphism.

The list of advantages of polymorphism are as follows.

It provides reusability to the code. The classes that are written, tested and implemented can be reused multiple times. This, in turn, saves a lot of time for the coder. Also, the code can be changed without affecting the original code.

A single variable can be used to store multiple data values. The value of a variable inherited from the superclass into the subclass can be changed without changing that variable's value in the superclass or any other subclasses.

With lesser lines of code, it becomes easier for the programmer to debug the code.

2.

Parametric polymorphism allows a name of a parameter or method in a class to be associated with different types. Let's have a look at the example given below, in which *content* is defined as string once then after that defined as *Integer*.

```
public class Department extends Employee{
    private String content;
    public String setContentDelimiter(){
        int content = 100;
        this.content = this.content + content;
    }
}
```

Here, the local declaration of a parameter always overrides the global declaration of another parameter with the same name. To handle this issue, it is advisable to use global references such as this keyword to indicate the global variables within a local context.

Problem with the polymorphic parameter is, it suffers from variable hiding.

3. It will throw a compile time error because more than one method with the same name and argument list can not be defined in a same class.

3. Run this program you will observe that it will throw a compilation error, because you can not have more than one method with the same name and argument list in a class even though their return type is different. The point to note is that the method return type does not matter in case of overloading.

2.11 REFERENCES/FURTHER READINGS

- ... Herbert Schildt “Java The Complete Reference”, McGraw-Hill, 2017.
- ... Savitch, Walter, “ Java: An introduction to problem solving & programming” , Pearson Education Limited, 2019.
- ... Neil, O. , “Teach yourself JAVA”, Tata McGraw-Hill Education, 1999.
- ... Sarcar, Vaskaran. “The Concept of Inheritance”, In Interactive Object-Oriented Programming in Java, pp. 65-90. Apress, Berkeley, CA, 2020.

