9

IT23023

Method Overriding :-

Method overriding is a feature i java that allows a subdelays to provide a new implement for a method that is already define in ~~it~~ its superclass. How it works in ~~inter~~ inheritence -

i) When a subclass overrides a method, the subclass version of the method gets executed, even if the method called on a superclass reference holding a superclass object.

ii) This process is called runtime polymorphism, because the method call is resolved at runtime.

iii) Overriding enables customised behavior for subclass object while maintaining a consistent interface.

~~D~~ What happened when a subclass overrides a method -

1. Subclass method executes, replacing the superclass method.

IT23023

2. Runtime polymorphism determines method execution at runtime.

3. Superclass method is hidden unless called using `super`.

4. Overriding rules apply.

5. `super` call can call the overriden superclass method.

The `super` keyword is used to call an overridden method from the superclass. This allows the subclass to extend or modify the behavior without completely replacing it.

1. Potential issue when overriding methods :-

i) Visibility Restriction - Cannot reduce access (eg. Public → Private)

ii) Exception Limitation → Cannot throw broader exceptions.

iii) final state method → `final` methods can't be overridden. static method are hidden, not overriden.

17/2/2023

2. Issue with constructor :-

i) Constructor cannot be overridden because they are not inherited.

ii) Super() must be used for superclass initialization.

- If the superclass has a parameterized constructor the subclass must explicitly call.

```
super(arguments);
```

- If no explicit `super()` is used, Java inserts a default constructor call, which may cause an error if the superclass lacks a no arguments constructor.

17.2.2023

10

Difference between Static and Non-Static members in Java.

Feature	Static Members	Non Static members
Defination	Belong to the class and are shared among all objects	Belong to the individual objects each instance has is own copy
Access	Accessed using class name of instance	Accessed only through an object of the class
Memory Allocation	Stored in the method area.	Stored in the heap memory.
Invocation	Can be called without creating an object	Required object creation before calling
Usage	Used for constants, utility methods and shared properties	Used for object specific behavior and instance data
Properties	Static members are shared and constant for all	Non static member are unique to each instance

1727023

Check if a number or string is palindrome:-

Program-

```
import java.util.Scanner;
public class palindromeCheck{
        static boolean Pal(String s){
         return s.equalsIgnoreCase(new StringBuilder(s).
                                    reverse().toString());
        }

        static boolean Pal (int n){
            int rev=0, temp=n;
            while (temp>0){
             rev = rev* 10 + temp % 10;
             temp /=10;
             } return n == rev; }
    Public static void main (String[] args){
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      System.out.println (n+(Pal(n)?"is": is not").
                                  +"a palindrome");

      String s = sc.nextLine();
      System.out.println(s+(Pal(n)?"is": "is not ")+"a palindrome");
      sc.close();
    }
}
```

IT23023

11

Class Abstraction :-

Class abstraction is the process of simplifying complex reality by modifying only the essential attributes and behavior of an object. while hiding unnessessary details. It focuses on what an object does rather than how it does.

ex :- A `Vehicle` class may define as abstract method start but each subclass (eg. car, bike) provides its own implementation

Class Encapsulation :-

Encapsulation is the but building of data (attributes) and methods (behaviors) that operate on that data within a single unit. It also involves controlling the access to the internals data of an object, preventing direct modification from the outside. This is often achive through access modifiers like private, protected and public

Ex :- A `Bank Account` class has a `Balance` variable marked `private` and users can access it only through `getBalance()` and `deposit()` methods. Those method can include to logic to validate the operations and maintain the integrity of the data.

IT23023

Difference between abstract Class and Interface:-

Abstract class	Interface
1. Abstract class can have both abstract and concrete methods.	1. Interface contains only abstract method.
2. Can have instance variable with any access modifier	2. Can have only public static final constants.
3. Can have constructors	3. Cannot have constructors.
4. Can include both implemented and non-implemented methods	4. All method are public and abstract by default.
5. Example: abstact class Anim	5. Example :- Interface Anim

1723023

12

Program :-

```
import java.util.Scanner;
Class Baseclass {
    void printResult (String msg, object result){
        System.out.println(msg + result);
    }
}
class Sumclass extends Baseclass {
    double subseries () {
        double sum = 0.0;
        for(double i = 1.0; i >= 1.0; i -= 0.1){
            sum += i;
        } return sum; }
}

Class DivisorMultipleClass extends Baseclass {
    int gcd (int a, int b){
        while (b != 0){
            int temp = b;
            b = a % b;
            a = temp; }
        return a; }
}
```

IT23023

```
Class NumberConversionClass extends Baseclass {
        string toBinary (int num) {
            return Integer.toBinary string (num); }

String to Hex (int num) {
        return Integer.toHex String (num).to upperCase (); }

String toOctal (int num) {
        return Integer.toOctal string (num); }
    }

Class CustomPrintClass {
    void pr (String msg)
     System.Out.Println (" >>" + msg); }
  }

Public class MainClass {
    public static void main (String[] args) {
    Scanner se = new Scanner (System.in);
    sub SumClass sumobj = new SumClass ();
  DivisonMultipleClass dmobj = new DivisonMultipleClass();
  NumberConversionClass no obj = new NumberConversionclass ();
  CustomPrintClas printobj = new CustomPrintClass ();
  double sum = sumobj.sumSeries ();
```

```
sumobj.printResult("Sum of the series:"+sum);
int num1 = sc.nextInt(), num2 = sc.nextInt();
sumobj.printResult("GCD:"+dmobj.gcd(num1,num2));
sumobj.printResult("LCM:"+dmobj.lcm(num1,num2));
System.out.println("Enter a decimal number of conversion:");
int num = sc.nextInt();
printObj.pr("Binary:"noobj.tobinary(num));
printobj.pr("Hexadecimal:"noobj.toHex(num));
printobj.pr("Octal:"noobj.tooctal(num));
sc.close(); }
}
```

IT23023

13

Java program:-

```
import java.util.Date;
class GeometricObject{
    private String color;
    private boolean filled;
    private Date dateCreated;
    public GeometricObject(){
        this.color = "White";
        this.filled = false;
        this.dateCreated = new Date(); }
    Public Geometric object(string color, boolean
                                        filled){
        this.color = color;
        this.filled = filled;
        this.dateCreated = new Date();
    }
    public string getColor(){
            return color; }
    Public void setColor(string color){
            this.color = color;
    }
```

IT23023

```
public boolean isFilled() {
        Return filled; }
public void setFilled(boolean filled){
        this.filled = filled; }

class circle extends GeometricObject {
        private double radius;
        public circle () {
                super (){ ;
                this.radius = 1.0;
        }
        public circle (double radius) {
                super();
                this.radius = radius; { ;
        public double getRadius(){
                return radius; }
        public void setRadius (double radius){
                this.radius = radius;
        public double getArea () {
                return Math.PI * radius * radius; }
```

M23023

```
public. class main {
    public static void main (String[] args) {
    Circle circle = new Circle (1.0, "blue", true);
      circle. printCircle ();
  System.out.println ("Area" + circle.getArea());
  System.out.println ("Perimeter" + circle.getPerimeter());
  System.out.println ("Diameter" + circle.getDiameter());

  Rectangle rectangle = new Rectangle (4.0, 6.0, "Red", true);
  system.out.println (rectangle.toString());
  System.out.println ("Area" + rectangle.getArea());
  System.out.println ("Perimeter" + rectangle.getPerimeter());

    }

}
```

IT23023

14

The BigInteger class in java is part of the Java.math package and is used to handle arbitrarily large integers. Unlike primitive data types like int or long, which have fixed size limits, BigInteger can store integers of virtually any size limited only by the available memory. This makes it ideal for calculations involving very large numbers such as factorial of large integers, cryptographic algorithm or precise mathmatical calculations.

Java Program :-

```
import java.math.BigInteger;
import java.util.Scanner;

public class factorialCalculator {
    public static BigInteger factorial(int n){
    BigInteger result = BigInteger.ONE;
      for(int i=2; i<=n; i++){
    result = result.multiply(BigInteger.valueOf(i));
    }
```

IT23023

```
public static void main (String [] args){
        Scanner scanner = new Scanner (System.in);
    System.out.println("Enter a number");
      int num = scanner.nextInt();
      if(number<0){
          system.out.println("Number is invalid");
        }
      else{
    BigInteger factorialResult = factorial (num);
    System.out.print("Factorial of "+num+ "is:");
    System.out.println (factorialResult);
      }
       Scanner.close();
      }
}
```

F123023

15

Abstract classes :-

- purpose :- provide a partial implementation and allow sub classes to complete it.
- Fields :- Can have instance variables.
- Methods :- Can have abstract (no body) and concrete (with body) methods.
- Inheritence :- Single inheritage.
- use case :- Preffered when you need to share code or maintain state among subclasses.

Interface :-

- purpose :- Define a contract that implementing classes must follow.
- fields : Only constants.
- Methods :- Abstract methods, default methods, Static methods.
- Inheritance :- Multiple Inheritage
- Use case :- Preffered when you need to define a behaviour that can be implemented by

unrelated classes.

When to use Abstract classes over interfaces:-

- Shared code
- state mangement.
- Constructor logic.

Ex:-

```
interface A {
        default void show() {
        system.out.print("A"); }
    }
interface B {
default void show() {System.out.println("B");}
    }
class C implements A,B {
    @Override.
    public void show() {
        A.super.show();
    }
}
```

17/2/2023

16

Polymorphism is the ability of a object to take on many forms. In java, it allows a single method on class to operate on objects of different types. It is achieved through method overriding and method overloading.

Dynamic method dispatch :- It is the mechanism by which a call to an overridden method is resolved at runtime rather than compile time. It is the foundation of runtime polymorphism in Java. Example of polymorphism using interface and method overriding

```
class Animal {
    void sound () {
        System.out.println ("Animal make a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound () {
        system.out.println ("Dog barks");
    }
}
```

IT23023

```
Class cat extends Animals {
    @Override
    void sound () {
        system.out.println ("Cat meows ");
    }
}
public class main {
    public static void main (String[] args) {
    Animal myAnimal = new Animal ();
    Animal myDog = new Dog();
    Animal myCat = new Cat();
myAnimal. sound ();
myDog. sound();
myCat. sound();
    }
}
```

IT23023

17

In java, both ArrayList and LinkedList are part of the java.util package and implement the List interface, but they differ significantly in how they store and manage elements, leading to different performance characteristic for common operations.

1: Underlying Data Structure :-

- ArrayList :-
  Array backed : It uses a dynamic array internally to store elements.
- Linked list : Each element is represented by a node containing a reference to both the previous and the next node.

2. Time complexities.

Operation	Arraylist	Linked List
Access	O(1)	O(n)
Insertion (at end)	O(1)	O(1)
Insertion (at index)	O(n)	O(n)
Deletion (at index)	O(n)	O(n)
search	O(n)	O(n)

PT23023

Access :-

- Arraylist :- Since it uses an array. It support random access. Given an index retriving the element takes constant time O(1).
- Linked list :- To access one an element, you must retrive the element takes constant time. O(1).

Insertion :-

- Arraylist :-
  - Insertion at end generally O(1)
  - Inserting at begining or middle require shifting the element making an operation O(n).

Linked List :-

- Inserting at the begining or end is O(1)
- Inserting in the middle requires finding the position first But the insertion itself is O(1)

IT23023

Deletion :-

- ArrayList :-
  Deletion from the end is O(1).
  Deletion from the middle or beginning is O(n).
- Linked list :-
  Deletion from the beginning or end is O(1).
  Deletion from the middle is O(n).

When to use each :-

- ArrayList :-
  * You need fast random access to elements.
  * The list is relatively static or only grows.
  * The list mostly involves appending elements.

LinkedList :-

* You need frequent insertion and deletion.
* You don't need fast random access.
* You expect work with a large number of elements.

K123023

18

Class CustomRandomGenerator

```
public class CustomRandomGenerator{
    public: int myRand (int i, int n){
    int [] random = new int [n+1];
    long currentTime = System.currentTimeMillis ();
    int t = 1;
    for( int j=1; j<=n ; j++){
        random [j] = t+=2;
    }
int ran = (int) (currentTime * random [i]) % 1000;
    return ran; }
    }
```

main class

```
import java.util.Scanner;
public class RandNumber{
    public static void main (String [] args){
    Scanner obj = new Scanner (System.in);
    int n = obj.nextInt();
CustomRandomGenerator obj2 = new CustomRandomGenerato();
```

ITE3023

```
for(int i = 1; i<=n; i++) {
    int num = obj2.myRand (i,n);
    if (num<0)
    System.out.println (-1*num);
    else
    System.out.println (num);
}
}
}
```