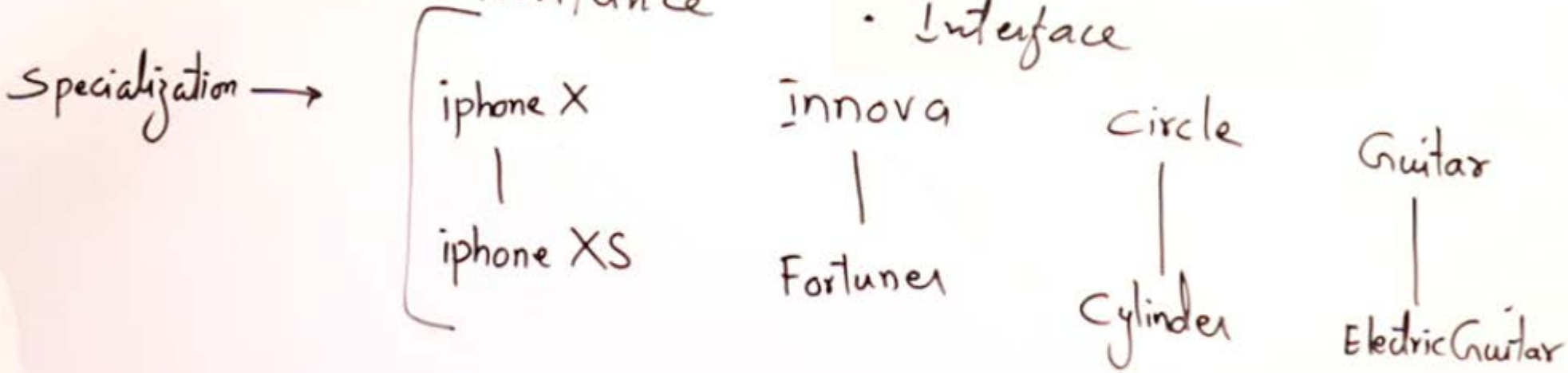
Inheritance

- Inheritance
- Abstract classes
- Interface



--> * Generalization mean, " group of class " ko kisi common name sai call karna.
* ya ek Bottom up (mtlb nicha sai upper ki traf) approach hota hai.
* ham interface approach Sai, generalization ko achieve kar sakta hai.

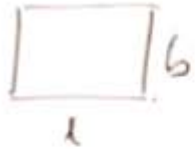
--> * Specalization mean, kisi existence class ka upgrade version jis mai kuch purana and kuch new features add ho.
* ya ek Top - down (upper sai nicha ki traf) approch hai.
* Specalization achieve using " inheritance ".
* new Class is derived from an existing Super Class.

Inheritance

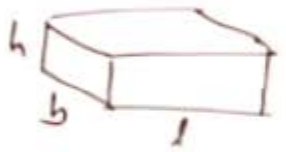
circle

cylinder

Rectangle



Cuboid

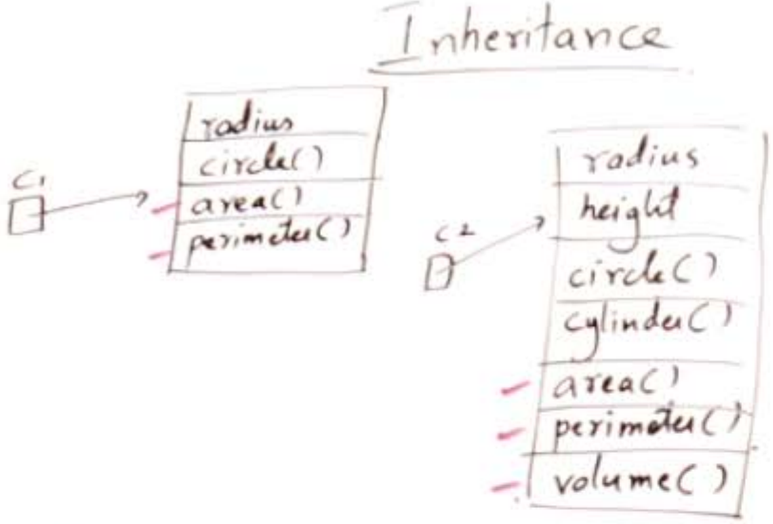


- properties
- Methods

--> Process of acquiring the features of existing class to new class called inheritance.

Ex.: jaisa hmna circle mai height add kar dia to hma cyclinder mil gya.

Method to initialize inheritance:




```
class Test
{
    p.s.v. main( )
    {
      Circle c1=new Circle();
      Cylinder c2=new Cylinder();
      c1.area();
      c2.area();
      c2.volume();
}
```

```
class Circle
{
  private double radius;
  public Circle()
  {
     radius=0.0;
  }
  public double area(){}
  public double perimeter(){}
}
```

```
class Cylinder extends Circle
{
  private double height;
  public Cylinder()
  {
     height=0.0;
  }
  public double Volume(){}
}
```

--> "extends" keyword ka use kar ka ham method ko initialize karta hai.

Java

explain inheritance in detail with contrast of java.

Inheritance in Java

Inheritance is a fundamental concept of object-oriented programming (OOP) that allows a new class to acquire the properties (fields) and behaviors (methods) of an existing class. In Java, inheritance promotes **code reuse**, **polymorphism**, and **maintainability** by enabling classes to build on top of other classes.
When one class (the **subclass** or **child class**) inherits from another class (the **superclass** or **parent class**), it automatically gains access to the fields and methods of the superclass, and it can extend or override these as needed.

Key Concepts in Inheritance

1. **Superclass (Parent Class)**:
   - This is the class whose properties and behaviors are inherited by another class.
   - The superclass can have its own attributes and methods, which are passed on to the subclass.
2. **Subclass (Child Class)**:
   - This is the class that inherits from the superclass.
   - The subclass can inherit all the non-private fields and methods of the superclass, but it can also have additional attributes and methods of its own.
   - The subclass can also **override** methods from the superclass to provide its own implementation.

Syntax for Inheritance in Java

In Java, inheritance is implemented using the `extends` keyword:

```java
class Superclass {
    // Fields and methods of the superclass
}

class Subclass extends Superclass {
    // Additional fields and methods of the subclass
}
```

Example of Basic Inheritance:

```java
// Superclass
class Animal {
    String name;

    public void eat() {
        System.out.println(name + " is eating");
    }
}

// Subclass
class Dog extends Animal {
    // New method in the subclass
    public void bark() {
        System.out.println(name + " is barking");
    }
}

// Main class to demonstrate inheritance
public class Main {
    public static void main(String[] args) {
        // Creating an instance of Dog (subclass)
        Dog myDog = new Dog();
        myDog.name = "Buddy";  // Inherited field
        myDog.eat();  // Inherited method
        myDog.bark();  // Subclass-specific method
    }
}
```

Output:

```csharp
Buddy is eating
Buddy is barking
```

In this example, the `Dog` class inherits the `name` attribute and `eat()` method from the `Animal` class, while also introducing a new method `bark()`.

Types of Inheritance in Java

Java supports several types of inheritance:

1. **Single Inheritance**:
   - A subclass inherits from a single superclass.

   Example:

   ```java
   class A { }
   class B extends A { }
   ```

2. **Multilevel Inheritance**:
   - A class inherits from a subclass, which itself inherits from another class, creating a chain of inheritance.

Example:

```java
class A { }
class B extends A { }
class C extends B { }
```

3. **Hierarchical Inheritance**:
   - Multiple classes inherit from a single superclass.

   Example:

```java
class A { }
class B extends A { }
class C extends A { }
```

Multiple Inheritance (Not Supported Directly)

Java **does not support multiple inheritance** with classes. This means that a class cannot inherit from more than one superclass directly. This restriction exists to avoid the **diamond problem**, where multiple paths of inheritance could introduce ambiguity.
However, Java achieves **multiple inheritance of behavior** through **interfaces** (more on this below).

Example of Multiple Inheritance (Not Allowed):

```java
// Not allowed in Java
class A { }
class B { }
class C extends A, B { }  // Compilation error
```

Method Overriding in Inheritance

In Java, when a subclass provides its own implementation of a method inherited from its superclass, it is known as **method overriding**. The overridden method in the subclass must have the same signature (method name, return type, and parameters) as the one in the superclass.

Example of Method Overriding:

```java
// Superclass
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}
```

```java
// Subclass overriding the method
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Create an Animal object
        Animal myDog = new Dog();        // Create a Dog object

        myAnimal.sound();  // Output: Animal makes a sound
        myDog.sound();     // Output: Dog barks
    }
}
```

Access Control in Inheritance

- **Private Members**: Private fields and methods of the superclass cannot be accessed or inherited by the subclass.
  Example:

  java

  ```java
  class Animal {
      private String name;
  }

  class Dog extends Animal {
      public void display() {
          // System.out.println(name);  // Error: name is private in Animal
      }
  }
  ```

- **Protected Members**: The `protected` modifier allows fields and methods to be inherited and accessed within the subclass, but they are not accessible by other classes outside the package.
- **Public Members**: Public fields and methods are fully accessible by the subclass, and they can be inherited and used directly.

Constructor in Inheritance

Constructors of the superclass are not inherited by the subclass, but the subclass can call the constructor of the superclass using the `super()` keyword. The `super()` call must be the first statement in the subclass constructor.

Example:

java

```java
class Animal {
    public Animal() {
        System.out.println("Animal is created");
    }
}

class Dog extends Animal {
```

```
    public Dog() {
        super();  // Calling the constructor of Animal
        System.out.println("Dog is created");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
    }
}
```

Output:

```
csharp
```

```
Animal is created
Dog is created
```

Inheritance and Polymorphism

Inheritance in Java supports **polymorphism**, where a subclass object can be treated as an object of its superclass. This allows Java to determine which method to invoke at runtime, leading to **runtime polymorphism**.

Example:

```
java
```

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Polymorphism
        myAnimal.makeSound();  // Output: Dog barks (runtime polymorphism)
    }
}
```

Inheritance with Interfaces (Multiple Inheritance of Behavior)

While Java doesn't allow multiple inheritance with classes, it allows a class to implement multiple interfaces, thus achieving **multiple inheritance of behavior**.

Example:

```java
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Duck implements CanFly, CanSwim {
    @Override
    public void fly() {
        System.out.println("Duck flies");
    }

    @Override
    public void swim() {
        System.out.println("Duck swims");
    }
}
```

In this example, the `Duck` class inherits behaviors from both the `CanFly` and `CanSwim` interfaces.

Advantages of Inheritance

1. **Code Reusability**: Inheritance allows a subclass to reuse the fields and methods of the superclass, reducing code duplication.
2. **Method Overriding**: Subclasses can provide specialized behavior by overriding methods from the superclass.
3. **Polymorphism**: Inheritance supports polymorphism, allowing objects of different classes to be treated as objects of a common superclass.
4. **Modularity**: Inheritance helps in dividing a complex problem into simpler sub-problems, improving code structure and design.

Disadvantages of Inheritance

1. **Tight Coupling**: Inheritance can create a strong dependency between the superclass and the subclass, which can make changes in the superclass ripple through to subclasses.
2. **Increased Complexity**: Deep inheritance hierarchies can be difficult to understand and maintain.
3. **Limited Flexibility**: Since Java does not support multiple inheritance with classes, there are limitations on how many classes a single class can extend.

Summary of Inheritance in Java

- Inheritance allows a subclass to acquire properties and behaviors from a superclass.
- Java supports single, multilevel, and hierarchical inheritance but does not support multiple inheritance with classes (it is achieved via interfaces).
- Inheritance enables code reuse, polymorphism, and method overriding.
- Private members of the superclass are not inherited, while protected and public members are accessible based on access control.

```java
// Constructor in heritance //

class parent
{
    public parent()
    {
        System.out.println(x:"Parent Constructor.");
    }
}

class child extends parent
{
    public child()
    {
        System.out.println(x:"Child Constructor.");
    }
}

class grandchild extends child
{
    public grandchild()
    {
        System.out.printf(format:"Grand child Constructor.\n");
    }
}

class inheritancepracc
{
    Run | Debug
    public static void main(String[] args)
    {
        grandchild gc = new grandchild();
    }
}
```

--> Jab bhi chain of constructor available ho and haam " grandchild - class " ko call karat hai to usa sai uper ka jitna hi constructor ho wo sabhi call hota hai top to bottom series mai.

Output

```
PS D:\Java> cd "d:\Java\6 month Java\" ; if ($?) { javac inheritancepracc.java } ; if ($?) { java inheritancepracc }
Parent Constructor.
Child Constructor.
Grand child Constructor.
PS D:\Java\6 month Java>
```

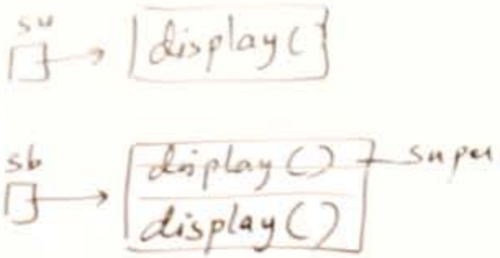
Method Overriding

Definition:
Redefine the method of "super - class" in "sub-class" called method overriding.

```
class Super
{
    public void display()
    {
        s.o.p("Hello");
    }
}
```

```
class Sub extends Super
{
    public void display()
    {
        s.o.p("Hello Welcome");
    }
}
```

Method Overriding




```
class Test
{
    p.s.v.main()
    {
        Super su=new Super();
        su.display(); ---- Hello

        Sub sb=new Sub();
        sb.display(); ---- Hello welcome
    }
}
```

```
class Super
{
    public void display()
    {
        s.o.p("Hello");
    }
}

class Sub extends Super
{
    public void display()
    {
        s.o.p("Hello Welcome");
    }
}
```

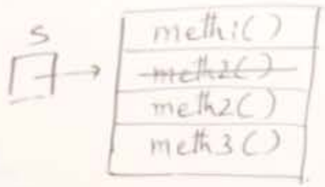
--> method overriding method ko initiate karna ka liya phela hma super class mai ek method bnana hota hai.
--> then same method ko same name sai sub class mai bnata hai. But ise mai uska content ko update kar deta hai.

--> Ham jab bhi mian class mai subclass ka function ko call karenga to memory mai dono method show karega but super class wala method over-shadow ho jayaga and update output show hoga.

```
3
4   class Car
5   {
6       public void start(){System.out.println("Car Started");}
7       public void accelerate(){System.out.println("Car is Accelerated");}
8       public void changeGear(){System.out.println("Car Gear Changed");}
9
10  }
11
12  class LuxaryCar extends Car
13  {
14      public void changeGear(){System.out.println("Automatic Gear");}
15      public void openRoof(){System.out.println("Sun Roof is Opened");}
16
17  }
18
19  public class OverridingExample
20  {
21
22      public static void main(String[] args)
23      {
24          Car c=new LuxaryCar();
25          c.start();
26          c.accelerate();
27          c.changeGear();
28          c.openRoof();
29      }
30
31  }
32
```

--> jab ham "extend" ka use kar ka koi program bnata hai to ham "super-class" ka "method" ko "sub-class" ka object bna ka use kar sakta hai. Lekin ise trah sirf "super-class" ka metod hi show hota hai aagar "sub-class" ka method ko access karna ka try karenga to error aayaga.

Dynamic Method Dispatch




```
class Test
{
    P.S.V.main()
    {
        Super s=new Sub();
        s.meth1();
        s.meth2();
        s.meth3();
    }
}
```

ref → Super s
object → new Sub()

s.meth1(); —— meth1

s.meth2(); —— Sub meth2

~~s.meth3();~~

```
class Super
{
    void meth1(){s.o.p("meth1");}
    void meth2()
    {
        s.o.p("Super meth2");
    }
}

class Sub extends Super
{
    void meth2()
    {
        s.o.p("Sub meth2");
    }
    void meth3(){s.o.p("meth3");}
}
```

--> Dynamic method dispatch useful for achieving runtime polymorphism using method overriding.

--> Dynamic method dispatch mai ham "super - class" ka reference ka use karta hai and "sub-class" ka object create karta hai.

--> agar ham "extends" ka use kar kai program bnata hai and dynamic method dispatch ka use karta hai to heap mai "sub-class" ka sabhi method load ho jayenga but ham only "super-class" ka method ko use kar sakta hai.

Do's and Don'ts of Overriding

- Signature must be same in method overriding.
- If the method name is different the method is not overridden but it is overloaded.
- Argument may be different but the parameter must be same.
- Return type must be same, if it is not same then the method is neither overridden nor overloaded.
- Final and static methods cannot be overridden.
- Method can be overridden with same or lenient (public, protected) access specifiers but the stricter(private) access specifiers cannot be used in sub class.

```java
package overloading;

class Test
{
    public int max(int a,int b)
    {
        return a>b?a:b;
    }

    public int max(int a,int b,int c)
    {
        if(a>b && a>c) return a;
        else if(b>c) return b;
        return c;
    }
}

public class Overloading
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.max(10,5);
        t.max(10,15,5);



    }

}
```

```java
package override;

class Super
{
    public void display()
    {
        System.out.println("Super Display");
    }
}

class Sub extends Super
{
    public void display()
    {
        System.out.println("Sub Display");
    }
}

public class Override
{
    public static void main(String[] args)
    {
        Super s=new Sub();
        s.display();

    }
}
```

➢ Polymorphism using Overloading and Overriding

- Polymorphism is one of the principles of Object-oriented-programming, polymorphism means one name different actions.
- Poly means 'many', morphism means 'forms'.
- Polymorphism is achieved using method overriding and overloading.
- In method overloading access specifiers, return types are same but number of parameters or type of parameters are different.
- In overloading number or type of argument will decides which method is to be called.
- Overloading is achieved in same class whereas overriding is achieved in inheritance.
- In method overriding signature is same but in overloading signatures must be different.
- Method calls are different in overriding it depends on object.
- Method overriding is used for **runtime polymorphism** and method overloading is used for **compile time polymorphism.**

Java

explain polymorphism in java.

Polymorphism in Java

Polymorphism is a core concept in object-oriented programming (OOP) that refers to the ability of an object to take on multiple forms. In Java, polymorphism allows one method or interface to be used for different types of objects, and the actual method that is executed is determined at runtime or compile time.
There are two types of polymorphism in Java:

1. **Compile-time Polymorphism (Static Polymorphism)**
2. **Runtime Polymorphism (Dynamic Polymorphism)**

Both types enable Java to achieve **flexibility** and **extensibility** in code, promoting easier maintenance and the ability to handle various objects in a unified manner.

---

1. Compile-time Polymorphism (Static Polymorphism)

Also known as **method overloading**, compile-time polymorphism occurs when multiple methods in the same class have the same name but different parameters (either in the number or type of arguments). The method to be executed is determined at compile-time based on the method signature.

Example of Method Overloading:

```java
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
```

```
        System.out.println(calc.add(5, 10));         // Calls add(int, int)
        System.out.println(calc.add(5, 10, 15));     // Calls add(int, int, int)
        System.out.println(calc.add(5.5, 10.5));     // Calls add(double, double)
    }
}
```

Output:

```
15
30
16.0
```

In the example above, the `add()` method is overloaded with different numbers and types of parameters. The compiler selects the appropriate method based on the arguments passed during the method call.

Key Points:

- The decision on which method to call is made at **compile time**.
- It provides flexibility by allowing multiple methods with the same name but different parameter types or counts.

---

2. Runtime Polymorphism (Dynamic Polymorphism)

Also known as **method overriding**, runtime polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method to be executed is determined at runtime, based on the object that is invoking the method, even though the reference might be of the superclass type.

Example of Method Overriding:

```java
// Superclass
class Animal {
    // Method to be overridden
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass 1
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

// Subclass 2
class Cat extends Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Superclass reference, superclass object
        Animal myDog = new Dog();        // Superclass reference, subclass object
        Animal myCat = new Cat();        // Superclass reference, subclass object

        myAnimal.sound();  // Output: Animal makes a sound
        myDog.sound();     // Output: Dog barks
        myCat.sound();     // Output: Cat meows
    }
}
```

Output:

```
css


Animal makes a sound
Dog barks
Cat meows
```

In this example, the `**sound()**` method is defined in the `**Animal**` class and overridden in both the `**Dog**` and `**Cat**` subclasses. The decision about which method to invoke is made at **runtime** based on the actual object (whether it's a `**Dog**` or `**Cat**`), even though the reference type is `**Animal**`.

Key Points:

- **Method overriding** allows a subclass to provide its specific implementation of a method that is already present in the superclass.
- The decision on which method to invoke is made at **runtime**, enabling **dynamic method dispatch**.
- Runtime polymorphism supports **inheritance** and **method overriding**.

Differences Between Compile-time and Runtime Polymorphism

Feature	Compile-time Polymorphism (Method Overloading)	Runtime Polymorphism (Method Overriding)
Method Resolution	At compile-time	At runtime
Method Signature	Methods must have the same name but different parameter lists	Methods must have the same name and signature
Overloading or Overriding	Overloading	Overriding
Inheritance Required	No, method overloading occurs within the same class	Yes, method overriding requires inheritance
Flexibility	Provides flexibility in defining methods with the same name	Supports dynamic behavior based on the object type
Example	`**add(int a, int b)**` and `**add(double a, double b)**`	`**sound()**` method in the `**Animal**` and `**Dog**` classes

Polymorphism with Interfaces

Polymorphism is also possible through **interfaces** in Java. Since an interface defines methods without implementation, different classes can implement the interface and provide their own specific implementations of the methods.

Example:

```java
// Interface
interface Animal {
    void sound();
}

// Class 1 implements the interface
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

// Class 2 implements the interface
class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Interface reference, Dog object
        Animal myCat = new Cat();  // Interface reference, Cat object

        myDog.sound();  // Output: Dog barks
        myCat.sound();  // Output: Cat meows
    }
}
```

In this case, both `Dog` and `Cat` implement the `Animal` interface, and at runtime, the appropriate `sound()` method is invoked based on the actual object.

Advantages of Polymorphism

1. **Flexibility**: Polymorphism allows a single interface to be used for a general class of actions, making the code more flexible.
2. **Code Reusability**: Methods can be reused by different objects, reducing code duplication.
3. **Extensibility**: New classes can be added with minimal changes to the existing code.
4. **Maintenance**: Polymorphism reduces the need for conditional statements and makes it easier to maintain and extend the codebase.

Example of Flexibility with Polymorphism:

```java
public class Main {
    public static void main(String[] args) {
        // Polymorphism in action
        Animal myAnimal = new Dog();   // Polymorphic reference
```

```
            makeAnimalSound(myAnimal);     // Calls Dog's sound method

            myAnimal = new Cat();          // Same reference, now it's a Cat
            makeAnimalSound(myAnimal);     // Calls Cat's sound method
        }

        public static void makeAnimalSound(Animal animal) {
            animal.sound();  // Dynamic dispatch: calls the appropriate method
        }
    }
```

In this example, the `makeAnimalSound()` method works with any object that is of type `Animal`. The method to be invoked is determined at runtime based on the actual object passed to it (`Dog` or `Cat`).

Summary

- **Polymorphism** in Java allows a single action to behave differently based on the object.
- **Compile-time polymorphism** (method overloading) resolves method calls at compile time based on method signatures.
- **Runtime polymorphism** (method overriding) resolves method calls at runtime based on the object type.
- Polymorphism enhances flexibility, maintainability, and code reuse, making it a powerful tool in object-oriented programming.