ACHARYA

Dept of AIML

Module 3.

Chapter: Inheritance

Content: Inheritance Basics, Using super, Creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

By. Mohammed Tahir Mirji

ACHARYA

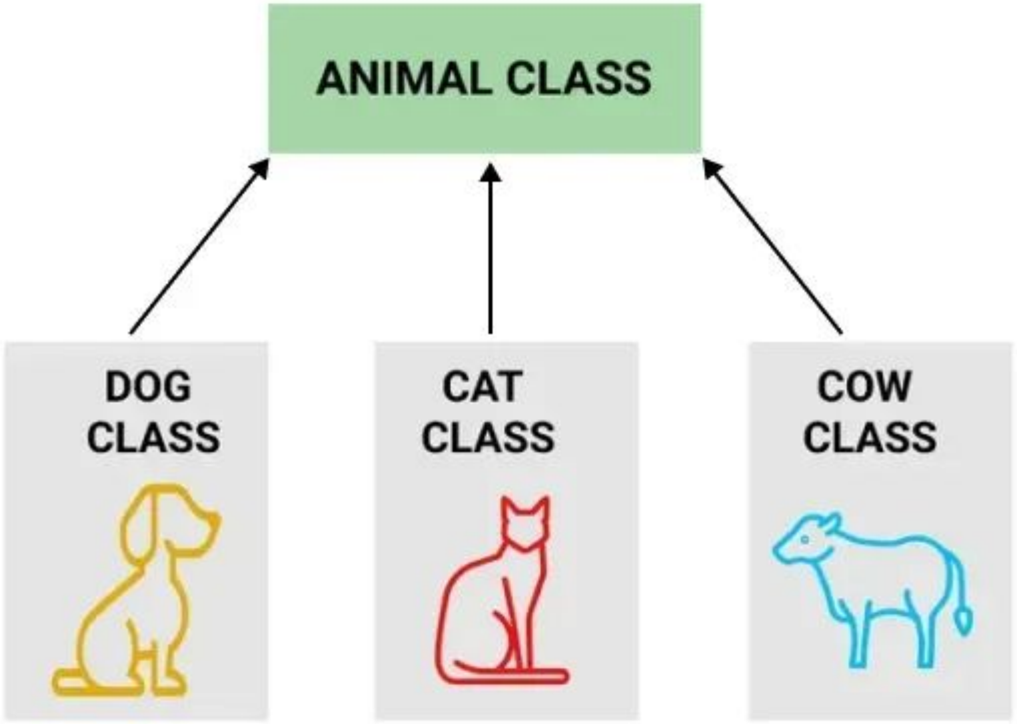
Dept of AIML

Content

- Inheritance Basics
- Using super
- Creating a Multilevel Hierarchy
- When Constructors Are Executed
- Method Overriding
- Dynamic Method Dispatch
- Using Abstract Classes
- Using final with Inheritance
- Local Variable Type Inference and Inheritance
- The Object Class

Inheritance in Java

It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.



Ex: Animal is the base class and Dog, Cat and Cow are derived classes that extend the Animal class.

Inheritance in Java

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

// Child class
class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    } }

// Child class
class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    } }
// Child class
class Cow extends Animal {
    void sound() {
        System.out.println("Cow moos");
    } }
```

```
// Main class
public class M {
    public static void main(String[]
args) {
        Animal a;
        a = new Dog();
        a.sound();

        a = new Cat();
        a.sound();

        a = new Cow();
        a.sound();
    } }
```

Output

```
Dog barks
Cat meows
Cow moos
```

ACHARYA

Dept of AIML

Using super

The super keyword in Java is a reference variable that is used to refer to the parent class when we are working with objects.

```
// Base class vehicle
class Vehicle {
    int maxSpeed = 120;
}

// sub class Car extending vehicle
class Car extends Vehicle {
    int maxSpeed = 180;

    void display()
    {
        // print maxSpeed from the vehicle class
        // using super
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}
```

```
// Driver Program
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output

```
Maximum Speed: 120
```

Use of super with Methods in Java

```
// superclass Person
class Person {
    void message()
    {
        System.out.println("This is person class\n");
    }
}

// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }

void display()
    {
        // will invoke or call current
        // class message() method
        message();

        // will invoke or call parent
        // class message() method
        super.message();
    }
}

// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Dept of AIML

Use of super with Constructors in Java

```
// superclass Person
class Person {
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

// subclass Student extending the Person class
class Student extends Person {
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}
```

```
// Driver Program
class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

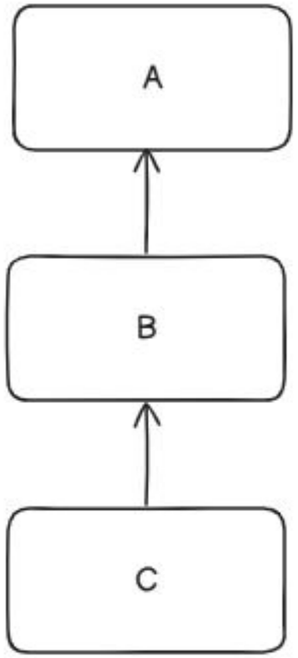
Output

```
Person class Constructor
Student class Constructor
```

Dept of AIML

Multilevel inheritance in Java

Multilevel inheritance is when a class inherits a class which inherits another class. An example of this is class C inherits class B and class B in turn inherits class A.



Multilevel inheritance in Java

```java
class A {
   void funcA() {
      System.out.println("This is class A");
   }
}
class B extends A {
   void funcB() {
      System.out.println("This is class B");
   }
}
class C extends B {
   void funcC() {
      System.out.println("This is class C");
   }
}
```

```java
public class Demo {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
      obj.funcC();
   }
}
```

Output

```
This is class A
This is class B
This is class C
```

When Constructors Are Executed?

In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.

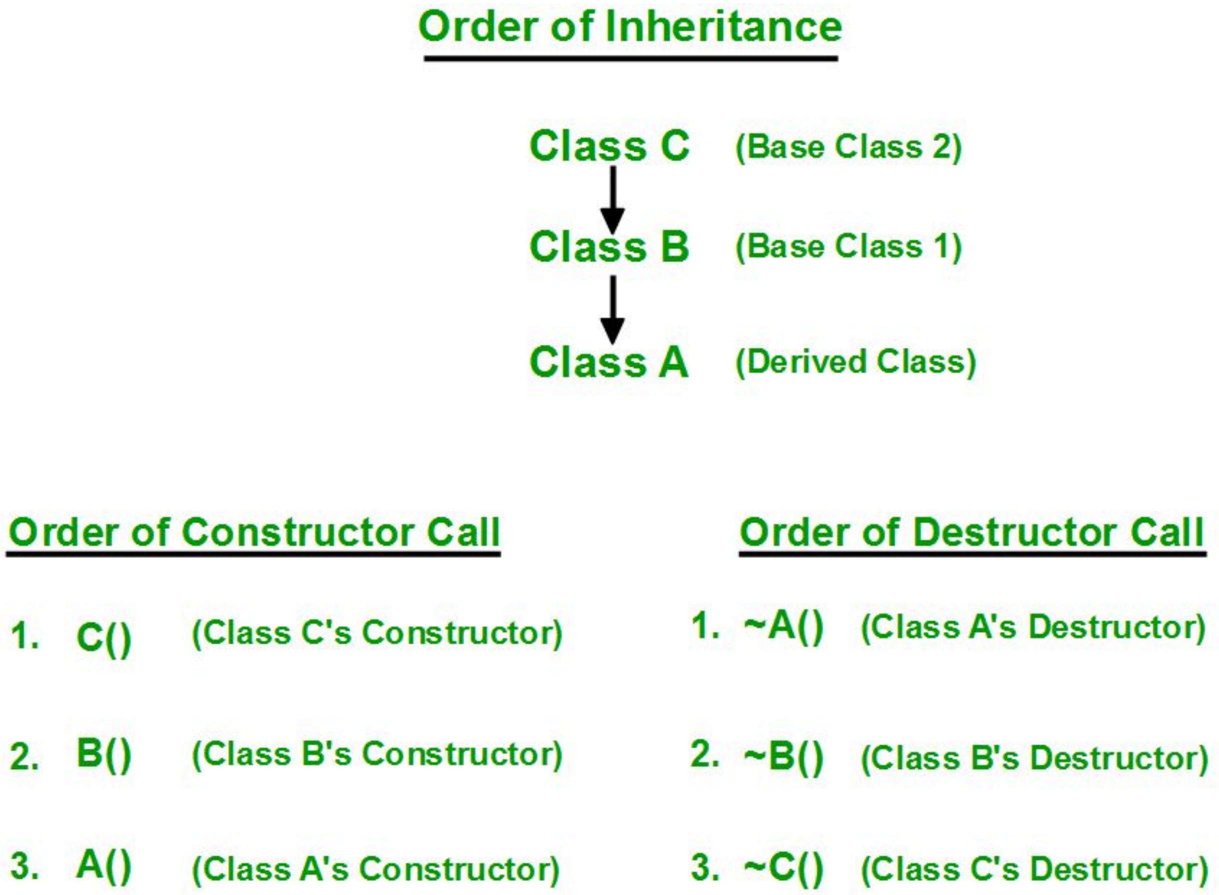
Execution Order

1. Object class constructor
2. Immediate superclass constructor
3. Subclass constructor

super() must be the first statement in a constructor

Dept of AIML

When Constructors Are Executed?



Order of Inheritance

Class C (Base Class 2)

Class B (Base Class 1)

Class A (Derived Class)

Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

When Constructors Are Executed?

```
class A {
    A() {
        System.out.println("Constructor A");
    }
}
class B extends A {
    B() {
        System.out.println("Constructor B");
    }
}
```

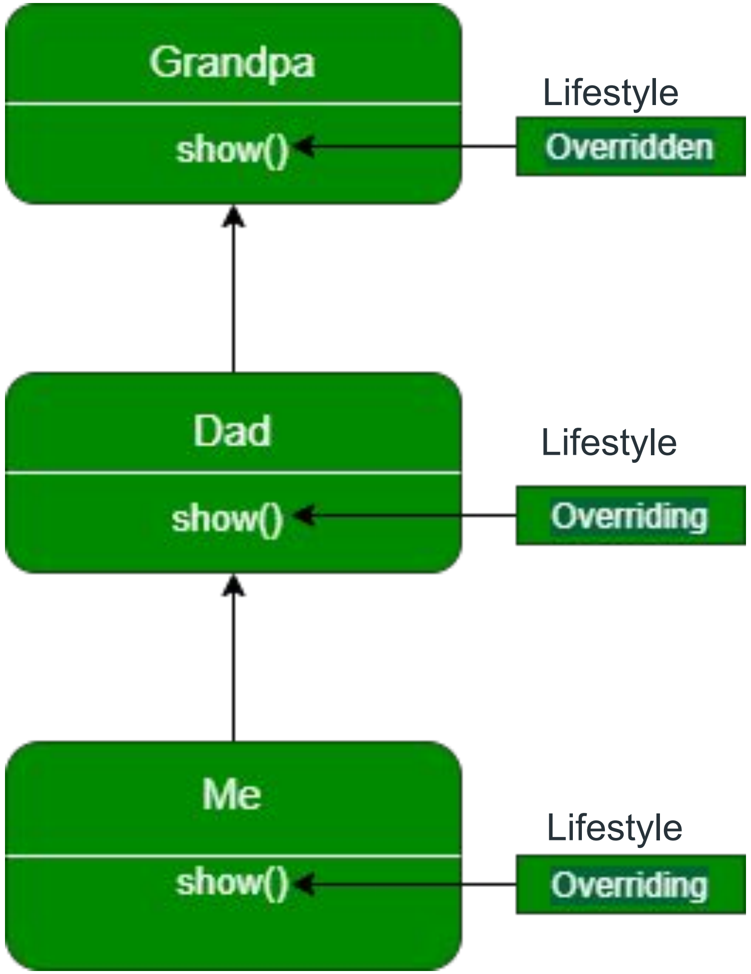
Output

```
Constructor A
Constructor B
```

Dept of AIML

Method Overriding in Java

Method overriding in Java is when a subclass implements a method that is already present inside the superclass.

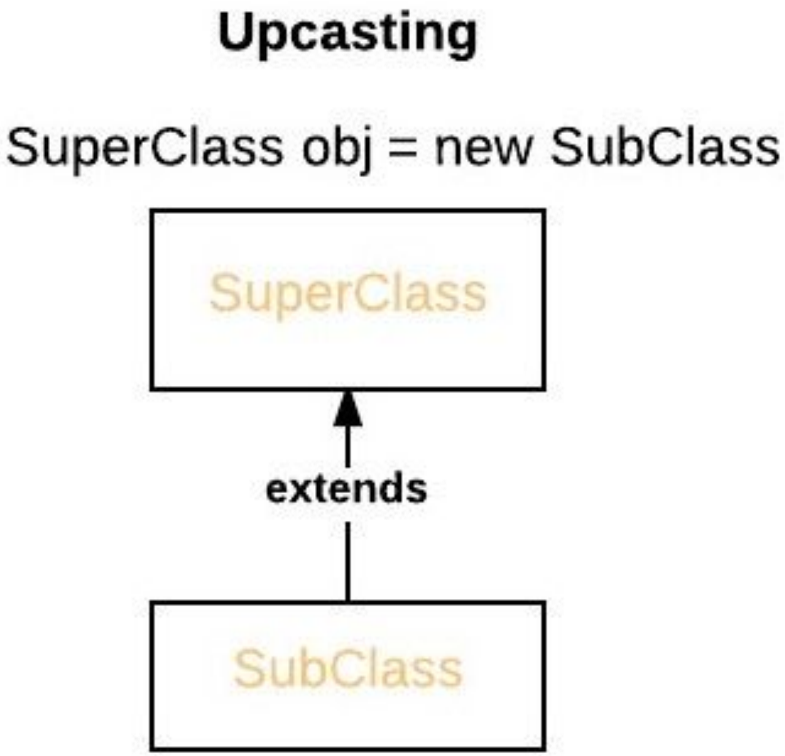


ACHARYA

Dept of AIML

Dynamic Method Dispatch in Java

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.



A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

Dynamic Method Dispatch in Java

- Mechanism by which a call to an overridden method is resolved at runtime
-
- Reference of superclass pointing to subclass object

```
Parent p;
p = new Child();
p.show();
```

Output

```
Child method
```

Dept of AIML

Dynamic Method Dispatch in Java

Why It Matters

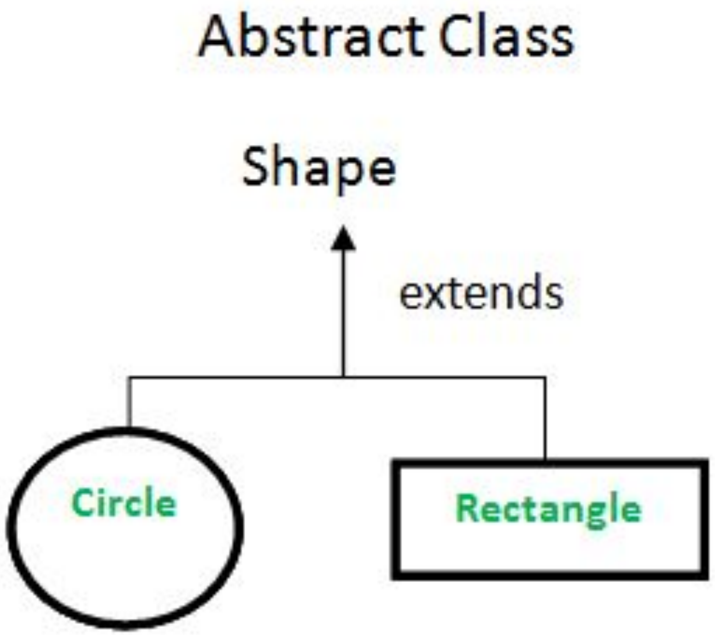
- Enables flexible and extensible code
- Core principle of polymorphism

Using Abstract Classes in Java

An abstract class cannot be instantiate

May contain:

- Abstract methods
-
- Concrete methods
-
- Constructors



- Abstract methods have no body
- A subclass must implement all abstract methods
- Can have constructors for initialization
- Used when behavior varies but structure remains same

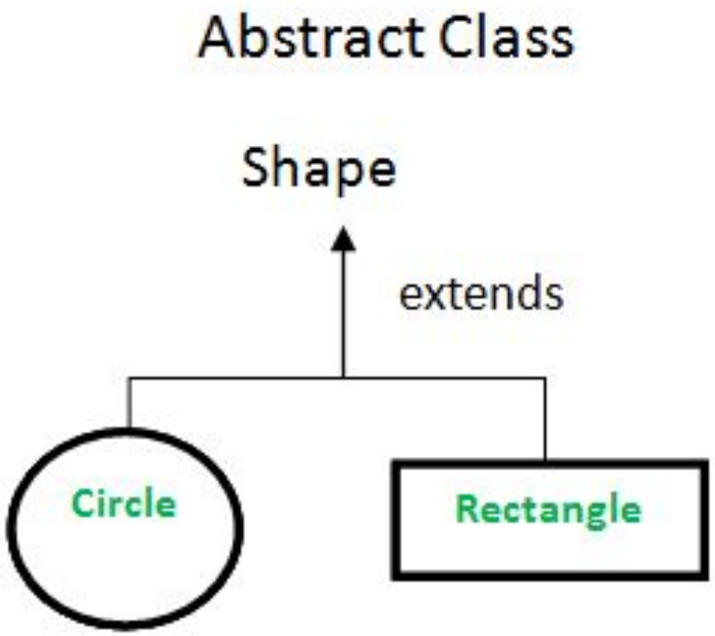
Using Abstract Classes in Java

```
abstract class Shape {
    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

Using Abstract Classes in Java

Java abstract class is a class that can not be instantiated by itself, it needs to be subclassed by another class to use its properties.



Using final with Inheritance in Java

final is a keyword in java used for restricting some functionalities.

We can declare variables, methods, and classes with the final keyword

`final` method → Cannot be overridden

`final` class → Cannot be inherited

`final` variable → Constant value

Purpose

Prevent modification

Ensure security and consistency

ACHARYA

Dept of AIML

Using final with Inheritance in Java

```
final class A {
    // Cannot be inherited
}

class B {
    final void display() {
        System.out.println("Final method");
    }
}
```

ACHARYA

Dept of AIML

Local Variable Type Inference & Inheritance in Java

Type inference happens at **compile time**

Inheritance still follows **reference-type rules**

```
var obj = new Parent();
obj = new Child(); // Allowed
```

Dept of AIML

Local Variable Type Inference & Inheritance in Java

Local Variable Type Inference

Introduced using var

Compiler infers the type

Ex:

```
var obj = new Child();
```

Type inference happens at **compile time**

Inheritance still follows **reference-type rules**

The Object Class Java

Object class (in java.lang) is the root of the Java class hierarchy. Every class in Java either directly or indirectly extends Object. It provides essential methods like toString(), equals(), hashCode(), clone() and several others that support object comparison, hashing, debugging, cloning and synchronization

Every class implicitly extends Object

Common Methods

toString()

equals(Object obj)

Overriding Object Class Methods in Java

toString()

```
class Student {
    int id;
    String name;

    public String toString() {
        return id + " " + name;
    }
}
```

Dept of AIML

Overriding Object Class Methods in Java

Why Override?

- To provide meaningful object representation
- Improve debugging and logging

Dept of AIML

Interfaces

- Interfaces: Interfaces
- Default Interface Methods
- Use static Methods in an Interface
- Private Interface Methods.

ACHARYA

Dept of AIML

Interface

An Interface in Java is an abstract type that defines a set of methods a class must implement.

- An interface acts as a contract that specifies what a class should do, but not how it should do it.
- It is used to achieve abstraction and multiple inheritance in Java
- A class that implements an interface must implement all the methods of the interface. Only variables are public static final by default.

Interface

```
import java.io.*;
// Interface Declared
interface testInterface {

    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}
// Class implementing interface
class TestClass implements testInterface {

    // Implementing the capabilities of Interface
    public void display(){
      System.out.println("test");
    }
}
```

```
class B{

    public static void
main(String[] args){

        TestClass t = new
TestClass();
        t.display();
        System.out.println(t.a);
    }
}
```

```
test
10
```

Default Interface Methods

- The implementation of these methods has to be provided in a separate class.
- Interfaces can now have both abstract and default methods.
- Default methods provide backward compatibility without breaking existing code.

Default Interface Methods

```
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method");
    }
}
```

```
class TestClass implements
TestInterface
{
    // implementation of square
abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String
args[])
    {
        TestClass d = new
TestClass();
        d.square(4);

        // default method executed
        d.show();
    }
}
```

Use static Methods in an Interface

Static methods in interfaces have a complete implementation and cannot be overridden by implementing classes.

Key Features

1. Declared with the static keyword inside an interface.
2. Contain a complete definition and cannot be overridden.
3. Called using the interface name only (e.g., InterfaceName.methodName()).
4. The scope of the static method is limited to the interface in which it is defined

Use static Methods in an Interface

```
interface NewInterface {

    // static method
    static void hello()
    {
        System.out.println("New Static
Method Here");
    }

    // Public and abstract method of
Interface
    void overrideMethod(String str);
}
// Implementation Class
public class InterfaceDemo implements
NewInterface {

    public static void main(String[] args)
    {
        InterfaceDemo interfaceDemo = new
InterfaceDemo();

        // Calling the static method of interface
        NewInterface.hello();

        // Calling the abstract method of
interface
        interfaceDemo.overrideMethod("Hello,
Override Method here");
    }
    // Implementing interface method

    @Override
    public void overrideMethod(String str)
    {
        System.out.println(str);
    }
}
```

ACHARYA

Dept of AIML

Chapter Summary

- Constructors execute **top-down hierarchy**
- Method overriding enables **runtime polymorphism**
- Dynamic method dispatch resolves method calls at runtime
- Abstract classes define **incomplete behavior**
- `final` restricts inheritance and overriding
- `Object` class is the **ultimate superclass**