

### Lesson Objectives

After completing this lesson, participants will be able to -

- Understand concept of Inheritance and Polymorphism
- Implement inheritance in java programs
- Implement different types of polymorphism




### Outline:

#### Lesson 6: Inheritance and Polymorphism

- 6.1: Inheritance
- 6.2: Using super keyword
- 6.3: InstanceOf Operator
- 6.4: Method & Constructor overloading
- 6.5: Method overriding
- 6.6: @override annotation
- 6.7: Using final keyword
- 6.8: Best Practices

6.1: Inheritance  
What is Inheritance?



**Basic TV**

**Smart TV**

Smart TV's are inherited from basic television which apart from multimedia functionality of TV allows us to do more like streaming video contents from Internet.

### What is Inheritance?

If you look at the slide example of how televisions are evolved in last decade, you will find two major differences between these two models. Basic TV's are used to watch programs typically they are streamed from set top box. On the other hand, the most advanced "smart TVs" are similar in function to "smart phones" and some tablets. These TVs go beyond providing access to web-based media, or streaming from content stored on your home computer. Smart TVs have the built in computing power that allows you to do many of the same things you can do with a smart phone or tablet such as web browsing, use of web-based services like Skype, and interactive access to social media sites.

Apart from the functional **enhancement**, there is slight **alteration** from hardware point of view. The position of sound boxes is altered and in smart TV's like to hide them from front view.

Smart TV's are "**inherited**" from Basic TV for two reasons:

1. To make enhancement and/or
2. To do alteration

This is basis of inheritance in OOPs where one class we can extend to either enhance its functionality or alter its behavior.

## 6.1: Inheritance

## What is Inheritance?

Inheritance allows programmers to reuse of existing classes and make them extendible either for enhancement or alteration

Allows creation of hierarchical classification

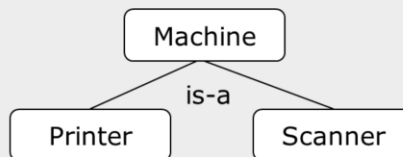
Advantage is reusability of the code:

- A class, once defined and debugged, can be used to create further derived classes

Extend existing code to adapt to different situations

Inheritance is ideal for those classes which has "is-a" relationship

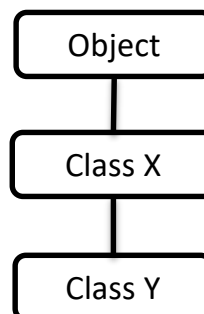
"Object" class is the ultimate superclass in Java



It is one of the fundamental mechanisms for code reuse in object-oriented programming. Inheritance allows new classes to be derived from existing classes. In Java, inheritance specifies how different is the sub class from its parent class. Thus, we can add new variables, methods and also modify the inherited methods. To inherit a class, you simply **extend** the super class into the subclass. Only single level inheritance is possible in Java.

**Superclass and Subclass:**

Any class **preceding** a specific class in the class hierarchy is said to be super class. On the other hand Any class **following** specific class in the hierarchy is called as subclass. All classes in Java are by default extensible. All All Java classes that do not explicitly extend a parent class automatically extend the **java.lang.Object** class.



In the above example, class Y is subclass. Class X is superclass of Y but subclass of Object class. **Object** class is the ultimate superclass in Java.

6.2: Using super keyword

### Using super Keyword

The super keyword is used to refer instance of its direct superclass

There are two uses of super keyword:

- Calling parent class constructor
- Call member of parent class

When you create instance of child class by calling its one of the constructor, it invokes immediate parent class default constructor which in turn calls its parent-parent class constructor. This process continues until constructor of Object class is called.

This constructor call chain will break if there is no default constructor available on parent class. To avoid this, you can call non-default constructor of parent class by using super keyword.

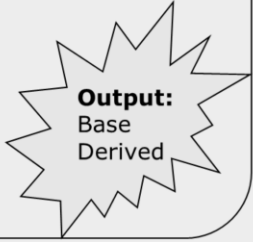
Kindly note when you call parent class constructor using super keyword, it must be written as first line of child constructor.

The other use of super keyword is to call to access the members of parent class. The member can be accessed by using syntax `super.memberName` or `super.memberMethod()`.

## 6.2: Using super keyword

## Inheritance : Example

```
class Base {  
    public void baseMethod() {  
        System.out.println("Base");  
    }  
}  
class Derived extends Base {  
    public void derivedMethod() {  
        super. baseMethod ();  
        System.out.println("Derived");  
    }  
}  
class Test {  
    public static void main(String args[]){  
        Derived derived=new Derived();  
        derived. derivedMethod();  
    }  
}
```



**Output:**  
Base  
Derived

### 6.3: InstanceOf Operator

#### instanceOf Operator

The instanceof operator compares an object to a specified type  
Checks whether an object is:

- An instance of a class.
- An instance of a subclass.
- An instance of a class that implements a particular interface.
- Example : The following returns true:

```
new String("Hello") instanceof String;
```

The instanceof operator is used to make a test whether the given object belongs to specified type. Consider the below example. The if statement returns true here as the child object is type of its superclass.

```
class Ticket{  
}  
class ConfirmedTicket extends Ticket {  
}  
...  
...  
ConfirmedTicket tkt= new ConfirmedTicket();  
If(tkt instanceof Ticket) {  
    //some processing  
}
```

### 6.3: InstanceOf Operator Demo

#### Inheritance

- Basic Inheritance
- Using super Keyword
- Use of instanceof keyword

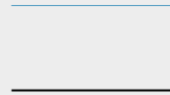
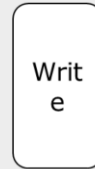




## 6.4: Polymorphism

## What is Polymorphism?

Poly meaning "many" and morph means "forms"  
It's capability of method to do different things based on the object used for invoking method



Polymorphism also enables an object to determine which method implementation to invoke upon receiving a method call

Java implements polymorphism in two ways

- Method Overloading
- Method Overriding

There are two ways in which polymorphism is implemented in Java.

1. Method Overloading
2. Method Overriding

Method Overloading is compile time polymorphism where the same method name has different meanings. Method overriding on other hand, is kind of runtime polymorphism where a subclass defines method with the same signature as defined by its superclass.

#### 6.4: Polymorphism

### Method Overloading

Two or more methods within the same class share the *same* name. Parameter declarations are different  
You can overload Constructors and Normal Methods

```
class Box {  
    Box(){  
        //1. default no-argument constructor  
    }  
    Box(dbl dblValue){  
        // 2. constructor with 1 arg  
    }  
    public static void main(String[] args){  
        Box boxObj1 = new Box(); // calls constructor 1  
        Box boxObj2 = new Box(30); // calls constructor 2  
    } }
```

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. Here, the methods are said to be overloaded. When an overloaded method is invoked, Java determines which method to actually call by using the type and/or number of operators as its guide.

Refer the example above. Here, two constructors have been declared; one with no arguments, the second with a single argument

**Note:** In addition to overloading constructors, you can also overload normal methods.

## 6.4: Polymorphism

## Constructor Overloading

If class has more than one constructors, then they are called as overloaded constructors.

When constructors are overloaded, they must differ in

- Number of parameters
- Type of parameters
- Order of parameters



A same model car can be constructed in different ways as per the requirement. Few of them are basic, while the other differ in fuel type, color, engine power, CNG, A.C. and or other accessories.



Example:

```
class Car {  
    int noOfCylinders;  
    int noOfValves;  
    int enginePower;  
    boolean isPowerSteering;  
    Car(){  
        //1. default no-argument constructor  
        noOfCylinders = 3;  
        noOfValves = 4;  
        enginePower = 48;           //48 ps  
        isPowerSteering = false;  
    }  
    Car(boolean isPowerSteering){  
        // 2. constructor with 1 arg  
        this();  
        this.isPowerSteering = isPowerSteering;  
    }  
    Car(int enginePower,int noOfCylinders, int noOfValves){  
        // 3. constructor with // 3 args  
        this.noOfCylinders = noOfCylinders;  
        this.noOfValves = noOfValves ;  
        this.enginePower = enginePower;  
        this.isPowerSteering = true;  
    }  
}
```

### 6.5: Polymorphism Method Overriding

In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method

Overridden methods allow Java to support run-time polymorphism



Normal Swap Machine



Chip card Machine which **overrides** the card reading for better security

Example:

```
class SwipeMachine{
    void readCard (){
        // functionality to read normal cards
    }
}
class ChipCardMachine extends SwipeMachine{
    void readCard(){
        //functionality to read chip card
    }
}
public static void main(String args[]){
    SwipeMachine normal = new SwipeMachine();
    normal.readCard();        //reading normal swipe card
    normal = new ChipCardMachine();
    normal.readCard();        //reading chip based swipe card
}}
```

## 6.6: Override Annotation

## @Override Annotation

The @Override annotation informs the compiler that the element is meant to override an element declared in a superclass  
It applies to only methods

```
public class Employee {  
    @override  
    public String toString() {  
        //statements  
        return "EmpName is:"+empname;  
    }  
}
```

Above code will throw a compilation error as it is not overriding the toString method of Object Class

**@Override:**

The Compiler checks that a method with this annotation really overrides a method from the Super class or not

While it's not required to use this annotation while overriding a method, it helps to prevent errors. If a method marked with **@Override** fails to correctly override a method in one of its superclasses, the compiler generates an error.

Most commonly, it is useful when a method in the base class is changed to have a different parameter list. A method in a subclass that used to override the superclass method no longer does so due to the changed method signature. This can sometimes cause strange and unexpected behavior, especially while dealing with complex inheritance structures. The **@Override** annotation safeguards against this. **@Override** is useful in detecting changes in parent classes which has not been reported down the hierarchy. Without it, a method signature can be changed and altering its overrides can be forgotten. With **@Override**, the compiler catches it for you.

**Other annotations supported in Java SE:**

The **@SuppressWarnings** annotation instructs the compiler to suppress the warning messages it normally shows during compilation time.

**@Deprecated** marks an old method as deprecated. Which says this method must not be used anymore because in the future versions, this old method may not be supported.

## 6.6: Polymorphism Demo

- Polymorphism
- Method Overloading
  - Method Overriding



## 6.7: Using Final Modifier

## Final Modifier

Final Modifier : Can be applied to variables, methods and classes

Final variable:

- Behaves like a constant; i.e. once initialized, it's value cannot be changed
- Example: `final int i = 10;`

Final Method:

- Method declared as final cannot be overridden in subclasses
- Their values cannot change their value once initialized
- Example:

Final class:

```
class A {  
    public final int add (int a, int b) {  
        Return a+b; }  
}
```

A class or method cannot be abstract & final at the same time.

- Cannot be sub-classed at all
- Examples: *String* and *StringBuffer* class



6.7: Inheritance and Polymorphism  
**Lab**



**Lab 4: Inheritance and Polymorphism**





## Summary



In this lesson, you have learnt about:

- Inheritance
- Using super keyword
- InstanceOf Operator
- Method & Constructor overloading
- Method overriding
- @override annotation
- Using final keyword
- Best Practices



### Review Question

Question 1: Which of the following options enable parent class to avoid overriding of its methods.

- extends
- Override
- Final

Question 2: When you want to invoke parent class method from child, it should be written as first statement in child class method

- True/False

Question 3: Which of the following access specifier enables child class residing in different package to access parent class methods?

- private
- public
- Final
- Protected

