

# OOPJ

---

**Deepak Upadhyay**  
**Sr. Cyber Security Trainer, IT**  
**&**  
**Artificial Intelligence Trainer**





## CHAPTER-2

### Fundamentals – II, Unit - 6



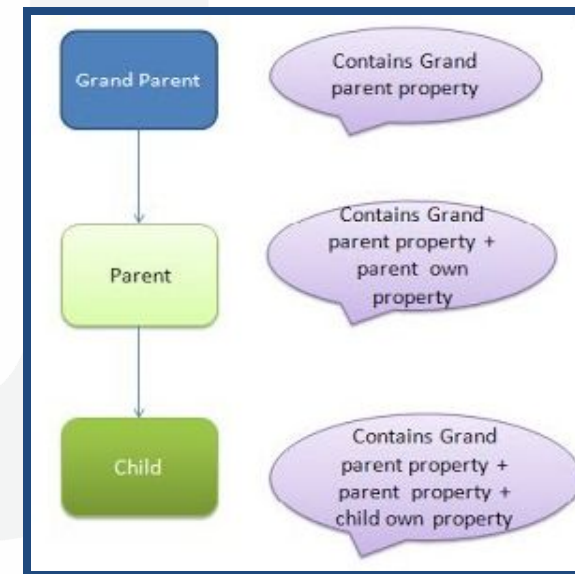


# Inheritance

- Inheritance is an important pillar of OOP.
- **It is the mechanism in java by which one class is allow to inherit the features (i.e. fields and methods) of another class.**
- Terminologies need to understand prior to inheritance.

**(1) Super Class**

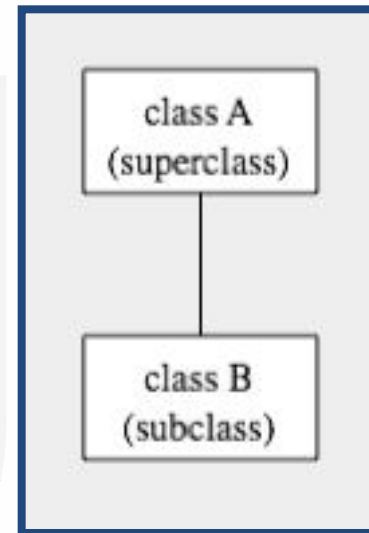
**(2) Sub Class**





# Inheritance

- **Super Class:** The class whose features are inherited is known as super class **(or a base class or a parent class)**.
- **Sub Class:** The class that inherits the other class is known as sub class **(or a derived class, extended class, or child class)**.  
The subclass can add its own fields and methods in addition to the superclass fields and methods.





# Inheritance

## Definition

- Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.
- Inheritance represents the **IS-A relationship**, also known as **parent-child relationship**.
- Keyword used for inheritance is 'extends'.

## Syntax

```
class Subclass-name extends Superclass-name
{
//methods and fields
}
```







# Inheritance

// A simple example of inheritance.

// Create a superclass.

```
class A {  
    int i, j;  
    void showij() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

// Create a subclass by extending class A.

```
class B extends A {  
    int k;  
    void showk() {  
        System.out.println("k: " + k);  
    }  
    void sum() {  
        System.out.println("i+j+k: " + (i+j+k));  
    }  
}
```

```
class SimpleInheritance {  
    public static void main(String args[]) {  
        A superOb = new A();  
        B subOb = new B();
```

// The superclass may be used by itself.

```
        superOb.i = 10;  
        superOb.j = 20;  
        System.out.println("Contents of superOb: ");  
        superOb.showij();  
        System.out.println();
```

// The subclass has access to all public members of its superclass.

```
        subOb.i = 7;  
        subOb.j = 8;  
        subOb.k = 9;  
        System.out.println("Contents of subOb: ");  
        subOb.showij();  
        subOb.showk();  
        System.out.println();  
        System.out.println("Sum of i, j and k in subOb:");  
        subOb.sum();  
    }  
}
```





# Inheritance

// A simple example of inheritance.

// Create a superclass.

```
class A {  
    int i, j;  
    void showij() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

// Create a subclass by extending class A.

```
class B extends A {  
    int k;  
    void showk() {  
        System.out.println("k: " + k);  
    }  
    void sum() {  
        System.out.println("i+j+k: " + (i+j+k));  
    }  
}
```

```
class SimpleInheritance {  
    public static void main(String args[]) {  
        A superOb = new A();  
        B subOb = new B();
```

// The superclass may be used by itself.

```
        superOb.i = 10;  
        superOb.j = 20;  
        System.out.println("Contents of superOb: ");  
        superOb.showij();  
        System.out.println();
```

// The subclass has access to all public members of its superclass.

```
        subOb.i = 7;  
        subOb.j = 8;  
        subOb.k = 9;  
        System.out.println("Contents of subOb: ");  
        subOb.showij();  
        subOb.showk();  
        System.out.println();  
        System.out.println("Sum of i, j and k in subOb:");  
        subOb.sum();  
    }  
}
```

Contents of superOb:  
i and j: 10 20

Contents of subOb:  
i and j: 7 8  
k: 9

Sum of i, j and k in subOb:  
i+j+k: 24



# Inheritance

## Types of inheritance

- (1) Single inheritance
- (2) Multilevel inheritance
- (3) Hierarchical inheritance
- (4) Multiple inheritance (Through interfaces)
- (5) Hybrid inheritance (Through interfaces)



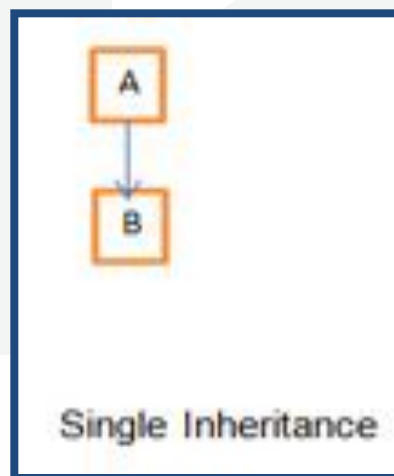


# Inheritance

## Types of inheritance

### (1) Single inheritance

- In single inheritance, subclasses inherit the features of one superclass.
- In image below, the class A serves as a base class for the derived class B.



```
public class A{  
.....  
}  
public class B extends A{  
.....  
}
```





# Inheritance

## Types of inheritance

### (2) Multilevel inheritance

- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.
- In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.
- In Java, a class cannot directly access the grandparent's members.



Multilevel Inheritance

```
public class A{.....}  
public class B extends A{.....}  
public class C extends B{.....}
```

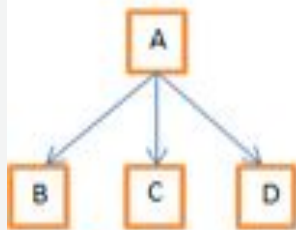


# Inheritance

## Types of inheritance

### (3) Hierarchical inheritance

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.
- In below image, the class A serves as a base class for the derived class B, C and D.



Hierarchical Inheritance

```
public class A{.....}  
public class B extends A{.....}  
public class C extends A{.....}  
tends A{.....}
```

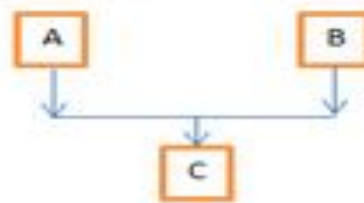


# Inheritance

## Types of inheritance


### (4) Multiple inheritance (Through interfaces)

- In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.
- Java does not support multiple inheritance with classes.
- In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



Multiple Inheritance

```
public class A { ..... }  
public class B { ..... }  
public class C extends A,B {  
    .....  
} // Java does not support multiple Inheritance
```

A large blue 'X' mark is placed over the code snippet, indicating that this syntax is not supported in Java.

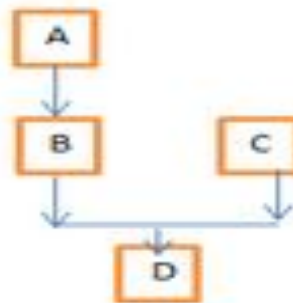


# Inheritance

## Types of inheritance

### (5) Hybrid inheritance (Through interfaces)

- It is a mix of two or more of the above types of inheritance.
- Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.
- In java, we can achieve hybrid inheritance only through Interfaces.



Hybrid Inheritance





# Inheritance

## Advantages

- Inheritance allows us to inherit all the properties of base class and can access all the functionality of inherited class. Which means it promotes reusability of code. In other word, it reduces code redundancy.

## Limitations

- Improper use of inheritance or less knowledge about it may lead to wrong solutions. Also sometimes, data members in base class are left unused which may lead to memory wastage.





# Inheritance

## Question

(1) A superclass can have any number of subclasses. But a subclass can have only one superclass. True or false? Justify it.

- **True.** This is because Java does not support multiple inheritance with classes. (Although with interfaces, multiple inheritance is supported by java.)



# Inheritance

## Question

Do it ...

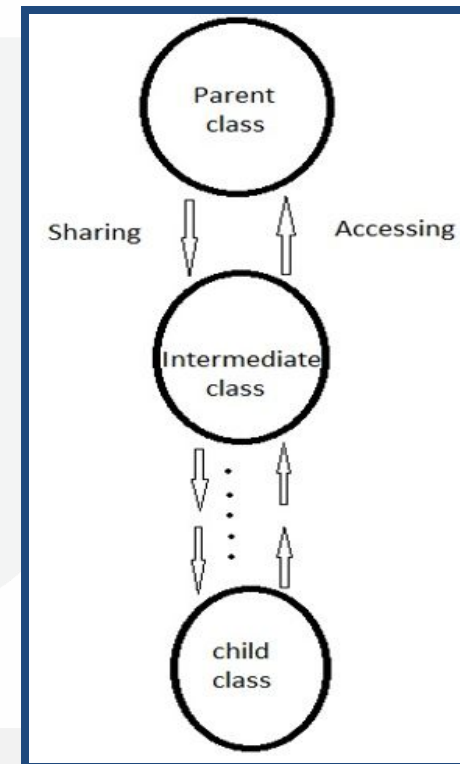
(2) We can not declare any new method in the subclass, that are not in the superclass. Is it true?





## class Hierarchy

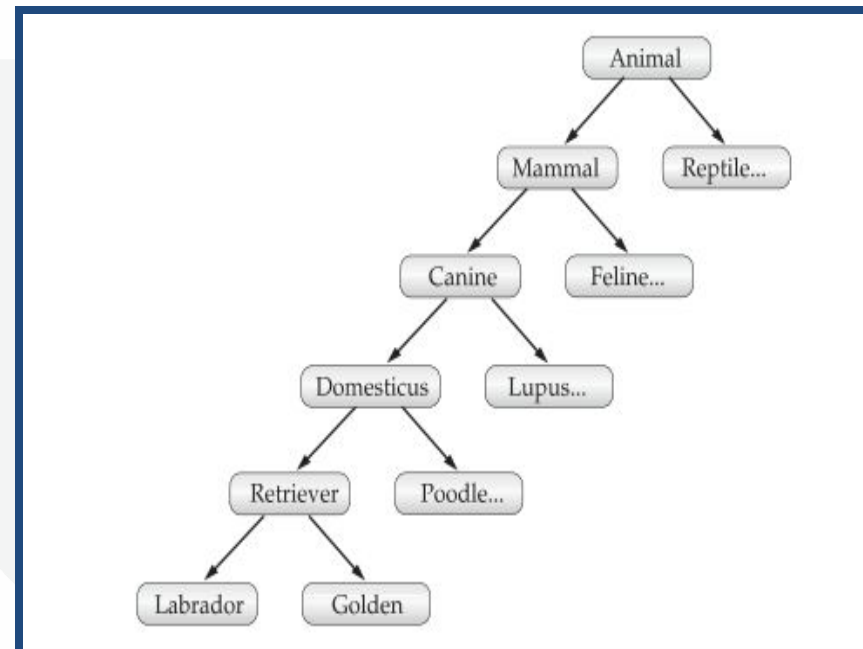
- Class hierarchy are basically, how classes are inter-related to each in certain order.
- The classes form a class hierarchy, or inheritance tree, which can be as deep as needed.
- The hierarchy of classes in Java has one root class, called Object class, which is superclass of any class.
- In general, the further down in the hierarchy a class appears, the more specialized its behaviour.





## class Hierarchy

- Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals.
- A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

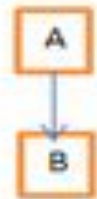




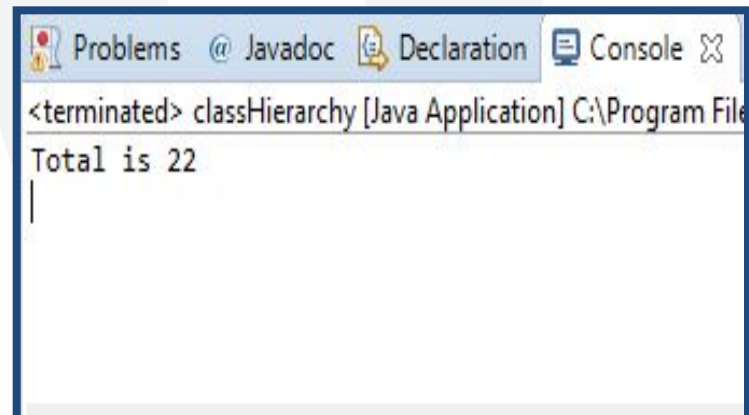
## class Hierarchy

```
class A {  
    int i;  
    int j;  
    void setij(int x, int y) {  
        i = x;  
        j = y;  
    }  
}  
  
class B extends A {  
    int total;  
    void sum() {  
        total = i + j;  
    }  
}
```

```
class classHierarchy {  
    public static void main(String args[]) {  
        B subOb = new B();  
        subOb.setij(10, 12);  
        subOb.sum();  
        System.out.println("Total is " + subOb.total);  
    }  
}
```



Single Inheritance





# Polymorphism

- Polymorphism is considered as one of the important features of Object Oriented Programming.
- **Polymorphism allows us to perform a single action in different ways.**
- In other words, polymorphism allows you to define one interface and have multiple implementations.
- Polymorphism is derived from 2 Greek words:
  - Poly- The word "poly" means many
  - Morphs-"morphs" means forms.
- **So, polymorphism means many forms.**





# Polymorphism

## Real life example of polymorphism (1)

- A person at the same time can have different characteristic.
- Like a man, at the same time is a father, a husband, an employee, etc.
- So, the same person posses different behaviour in different situations. This is called polymorphism.



# Polymorphism

## Real life example of polymorphism (2)



In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son





# Polymorphism

## Practical Example (3)

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

```
class Pig extends Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
}
```

```
class Dog extends Animal {  
    public void animalSound() {  
        System.out.println("The dog says: bow wow");  
    }  
}
```

```
class MyMainClass {  
    public static void main(String[] args) {  
        Animal myAnimal = new Animal(); // Create a Animal object  
        Animal myPig = new Pig(); // Create a Pig object  
        Animal myDog = new Dog(); // Create a Dog object  
        myAnimal.animalSound();  
        myPig.animalSound();  
        myDog.animalSound();  
    }  
}
```

```
The animal makes a sound  
The pig says: wee wee  
The dog says: bow wow
```



# Polymorphism

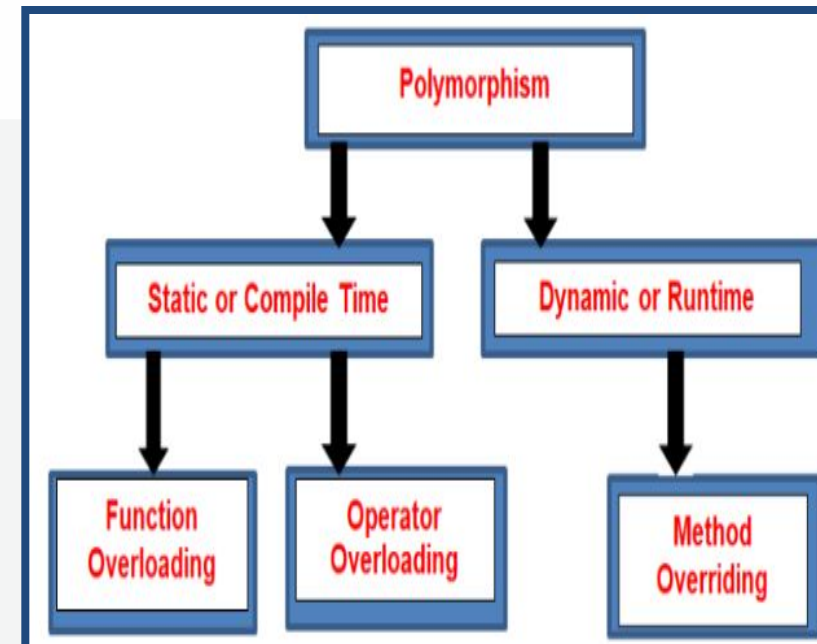
## Types of Polymorphism

### (1) Compile time Polymorphism

- It is also known as static polymorphism.
- It is achieved by...
  - Function/ Method Overloading
  - Operator Overloading

### (2) Runtime Polymorphism

- Also known as Dynamic Method Dispatch.
- It is a process in which a function call to the overridden method is resolved at Runtime, and is achieved by...
  - Method Overriding





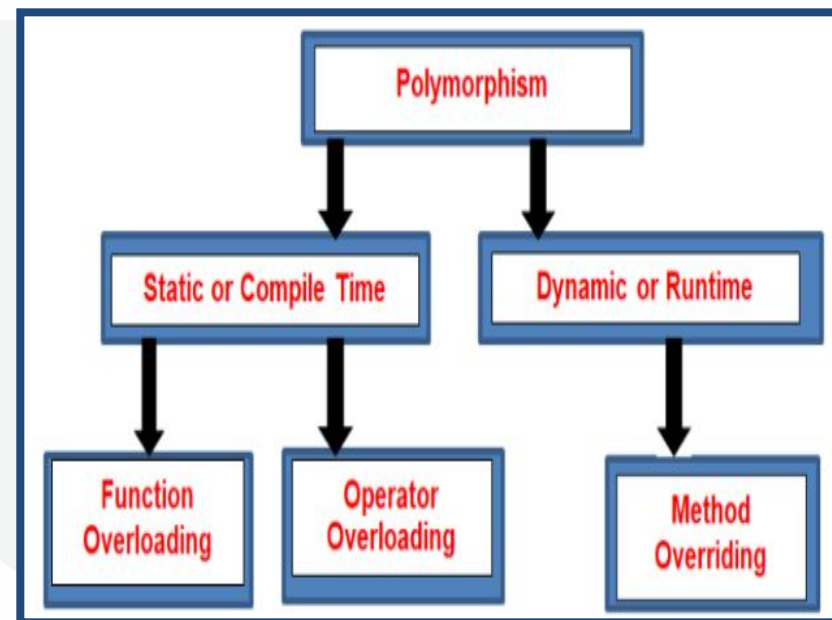
# Polymorphism

## Types of Polymorphism

### (1) Compile time Polymorphism

#### Function(Method) Overloading:

- When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**.
- Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.





# Polymorphism

## (1) Compile time Polymorphism

### Function(Method) Overloading (Ex 1):

// Java program for Method overloading (different types of arguments)

```
class MultiplyFun {
```

```
    // Method with 2 parameter
```

```
    static int Multiply(int a, int b)
```

```
    {
```

```
        return a * b;
```

```
    }
```

```
    // Method with the same name but 2 double parameter
```

```
    static double Multiply(double a, double b)
```

```
    {
```

```
        return a * b;
```

```
    }
```

```
}
```

```
class Main {
```

```
    public static void main(String[] args)
```

```
    {
```

```
        System.out.println(MultiplyFun.Multiply(2, 4));
```

```
        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
```

```
    }
```

```
}
```

8

34.65







# Polymorphism

## (1) Compile time Polymorphism

### Function(Method) Overloading (Ex 2):

// Java program for Method overloading (different numbers of arguments)

```
class MultiplyFun {  
  
    // Method with 2 parameter  
    static int Multiply(int a, int b)  
    {  
        return a * b;  
    }  
  
    // Method with the same name but 3 parameter  
    static int Multiply(int a, int b, int c)  
    {  
        return a * b * c;  
    }  
}
```

```
class Main {  
    public static void main(String[] args)  
    {  
        System.out.println(MultiplyFun.Multiply(2, 4));  
  
        System.out.println(MultiplyFun.Multiply(2, 7, 3));  
    }  
}
```

```
8  
42
```



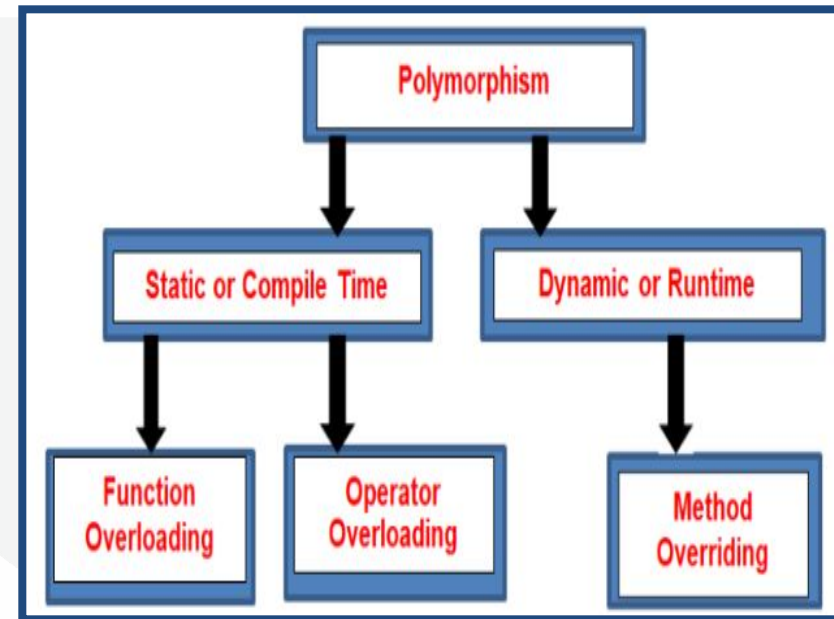


# Polymorphism

## (1) Compile time Polymorphism

### Operator Overloading:

- Java also provides an option to overload operators.
- For example, we can make the operator ('+') for string class to concatenate two strings.
- In Java, only "+" operator can be overloaded:
  - To add integers
  - To concatenate strings





# Polymorphism

## (1) Compile time Polymorphism

### Operator Overloading (Ex 1):

// Java program for Operator overloading

```
class OperatorOVERDDN {  
    void operator(String str1, String str2)  
    {  
        String s = str1 + str2;  
        System.out.println("Concatinated String - "+ s);  
    }  
  
    void operator(int a, int b)  
    {  
        int c = a + b;  
        System.out.println("Sum = " + c);  
    }  
}
```

```
class Main {  
    public static void main(String[] args)  
    {  
        OperatorOVERDDN obj = new OperatorOVERDDN();  
        obj.operator(2, 3);  
        obj.operator("joe", "now");  
    }  
}
```

```
Sum = 5  
Concatinated String - joenow
```



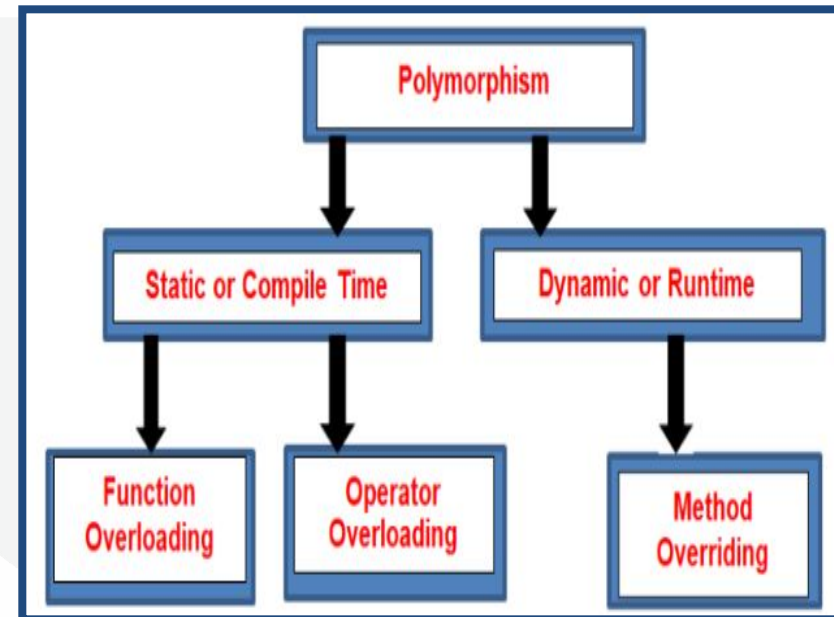


# Polymorphism

## (2) Runtime time Polymorphism

### Method Overriding:

- It occurs when a derived class has a definition for one of the member functions of the base class.
- That base function is said to be **overridden**.





# Polymorphism

## (2) Runtime time Polymorphism

### Method Overriding (Ex 1):

// Java program for Method overriding

```
class Parent {  
    void Print()  
    {  
        System.out.println("parent class");  
    }  
}  
  
class subclass1 extends Parent {  
    void Print()  
    {  
        System.out.println("subclass1");  
    }  
}
```

```
class subclass2 extends Parent {  
    void Print()  
    {  
        System.out.println("subclass2");  
    }  
}
```

```
class TestPolymorphism3 {  
    public static void main(String[] args)  
    {  
        Parent a;  
        a = new subclass1();  
        a.Print();  
  
        a = new subclass2();  
        a.Print();  
    }  
}
```

subclass1  
subclass2



# Polymorphism

## Advantages

- It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.
- Single variable can be used to store multiple data types.
- Easy to debug the codes.





## Static Binding

- The binding which can be resolved at compile time by compiler is known as static or early binding.
- Binding of all the static, private and final methods is done at compile-time .

### WHY???

- Static binding is better performance wise (no extra overhead is required).
- Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class.
- Hence compiler doesn't have any difficulty to determine object of local class.
- That's the reason binding for such methods is static.





## Static Binding

### Static Binding (Ex):

```
public class NewClass {  
    public static class superclass {  
        static void print()  
        {  
            System.out.println("print in superclass.");  
        }  
    }  
    public static class subclass extends superclass {  
        static void print()  
        {  
            System.out.println("print in subclass.");  
        }  
    }  
}
```

```
public static void main(String[] args)  
{  
    superclass A = new superclass();  
    superclass B = new subclass();  
    A.print();  
    B.print();  
}
```

```
print in superclass.  
print in superclass.
```



## Dynamic Binding

- In Dynamic binding compiler doesn't decide the method to be called.
- **Overriding is a perfect example of dynamic binding.**
- In overriding both parent and child classes have same method .





# Dynamic Binding

## Dynamic Binding (Ex):

```
public class NewClass {  
    public static class superclass {  
        void print()  
        {  
            System.out.println("print in superclass.");  
        }  
    }  
  
    public static class subclass extends superclass {  
        //Override  
        void print()  
        {  
            System.out.println("print in subclass.");  
        }  
    }  
}
```

```
public static void main(String[] args)  
{  
    superclass A = new superclass();  
    superclass B = new subclass();  
    A.print();  
    B.print();  
}
```

```
print in superclass.  
print in subclass.
```



# Static Binding & Dynamic Binding

## Important points

- Private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
- Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e. at run time





## Final Keyword

- The final keyword in java is used to **restrict the user**.
- final is a non-access modifier applicable **only to a variable, a method or a class**.
- When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant.

**Final Variable** → **To create constant variables**

**Final Methods** → **Prevent Method Overriding**

**Final Classes** → **Prevent Inheritance**





## Final Keyword

### Final Variable

- final variables are nothing but constants.
- We cannot change the value of a final variable once it is initialized.

### Syntax

```
final int a=5;
```

**Final Variable** → To create constant variables

**Final Methods** → Prevent Method Overriding

**Final Classes** → Prevent Inheritance





## Final Keyword

### Final method

- A final method cannot be overridden.
- Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

### Example

```
Class A{  
    final void hi(){ }  
}
```

```
Class B extends A{  
    void hi(){ } //error, can't override the final method  
}
```

Final Variable	→	To create constant variables
Final Methods	→	Prevent Method Overriding
Final Classes	→	Prevent Inheritance



## Final Keyword

### Final classes

- We cannot extend a final class.

### Example

```
final class A
```

```
{
```

```
}
```

```
class B extends A //generate error, cause final class can't inherited
```

```
{
```

```
}
```

Final Variable	→	To create constant variables
Final Methods	→	Prevent Method Overriding
Final Classes	→	Prevent Inheritance





## Final Keyword

- It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples

```
// a final variable
```

```
final int THRESHOLD = 5;
```

```
// a blank final variable
```

```
final int THRESHOLD;
```

```
// a final static variable PI
```

```
static final double PI = 3.141592653589793;
```

```
// a blank final static variable
```

```
static final double PI;
```





## Abstract Class

- A class that is declared using “abstract” keyword is known as abstract class.
- It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with **abstract methods (without an implementation, without braces, and followed by a semicolon)**.
- An abstract class can not be instantiated (you are not allowed to create object of Abstract class).

### Example

```
abstract class Shape    //An abstract class
{
    int color;
    abstract void draw(); // An abstract function }
```





## Abstract Class

- In Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {  
    abstract void fun();  
}  
class Derived extends Base {  
    void fun() { System.out.println("Derived fun() called"); }  
}  
class Main {  
    public static void main(String args[]) {  
        // Uncommenting the following line will cause compiler error as the  
        // line tries to create an instance of abstract class.  
        // Base b = new Base();  
  
        // We can have references of Base type.  
        Base b = new Derived();  
        b.fun();  
    }  
}
```

Derived fun() called





## Abstract Class

- an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

Base Constructor Called  
Derived Constructor Called





## Abstract Class

- In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Base fun() called







## Abstract Class

- **Abstract classes can also have final methods (methods that cannot be overridden)**

```
// An abstract class with a final method
abstract class Base {
    final void fun() { System.out.println("Derived fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.fun();
    }
}
```

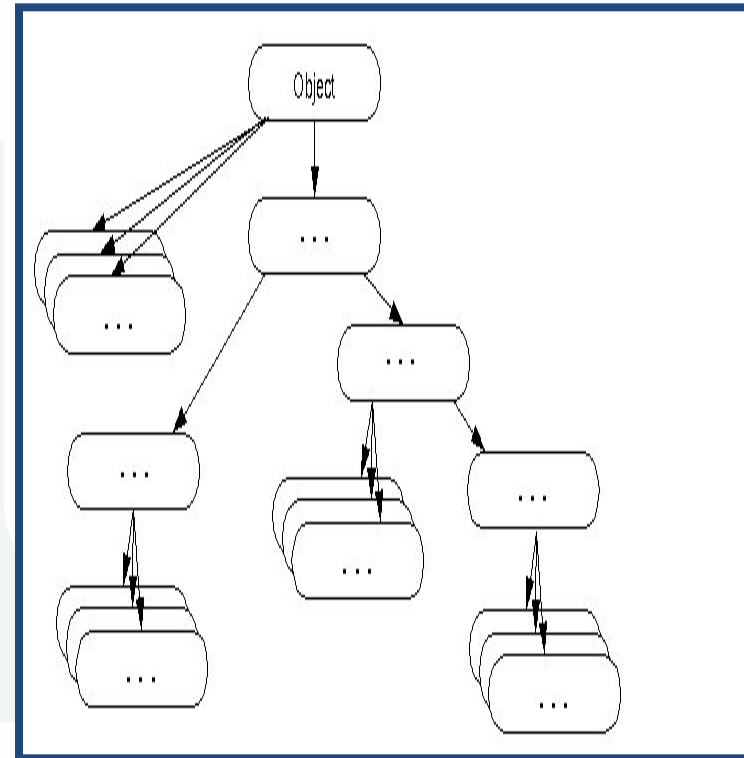
Derived fun() called





## Object Class

- The Object class is the parent class of all the classes in java by default.
- Object class is present in `java.lang` package.
- Every class in Java is directly or indirectly derived from the `Object` class.
- If a Class does not extend any other class then it is direct child class of `Object` and if extends other class then it is an indirectly derived.
- Therefore the Object class methods are available to all Java classes.
- Hence Object class acts as a root of inheritance hierarchy in any Java Program.





# Object Class

## Methods of Object Class

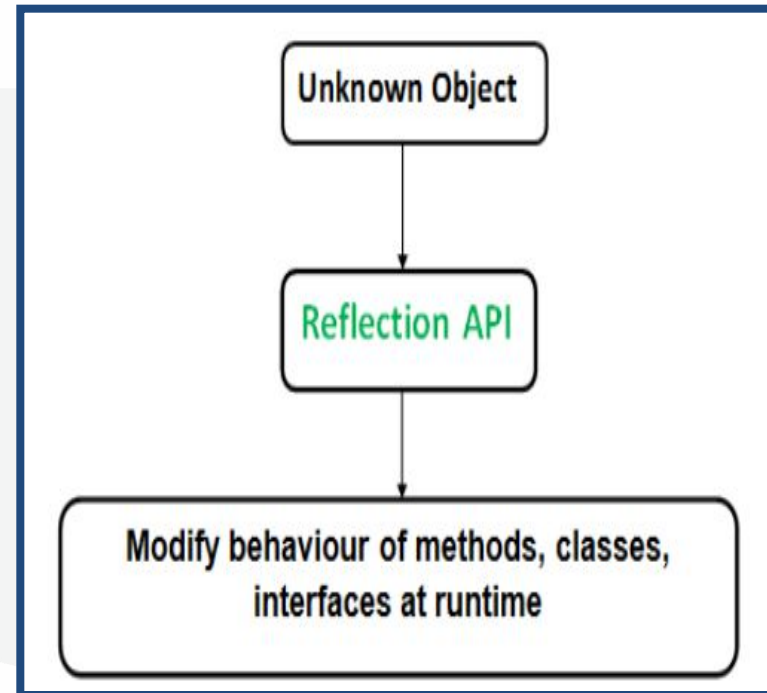
Method	Purpose
Class getClass()	returns the Class of an object .
String toString()	returns the string representation of this object.
boolean equals(Object obj)	compares the given object to this object.
void finalize()	is invoked by the garbage collector before object is being garbage collected.





## Reflection in java

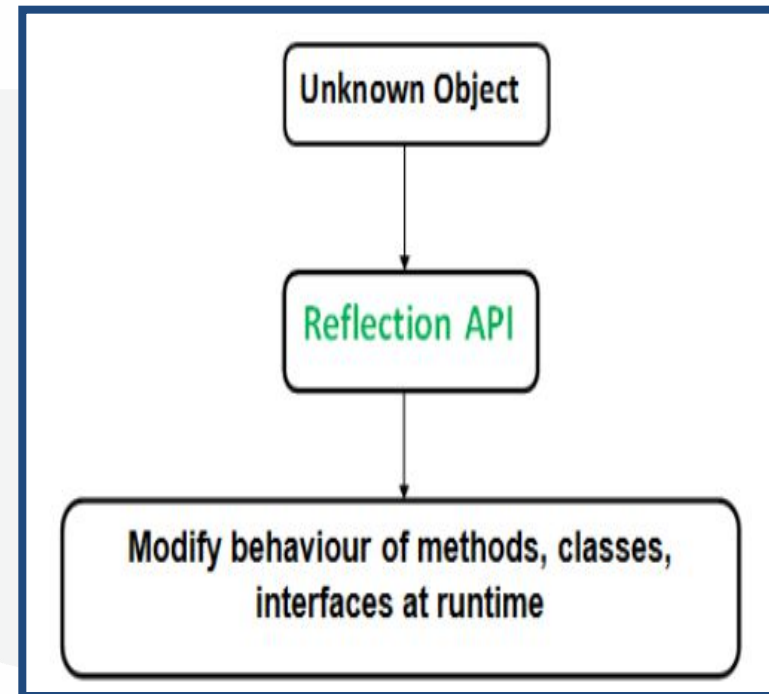
- Reflection is an API which is used to examine or modify the behaviour of methods, classes, interfaces at runtime.
- The required classes for reflection are provided under [java.lang.reflect](#) package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.





## Reflection in java

- Reflection can be used to get information about
- **Class** The `getClass()` method is used to get the name of the class to which an object belongs.
- **Constructors** The `getConstructors()` method is used to get the public constructors of the class to which an object belongs.
- **Methods** The `getMethods()` method is used to get the public methods of the class to which an object belongs.



# Reflection in java

## Advantages of Using Reflection

- Extensibility Features
- Debugging and testing tools

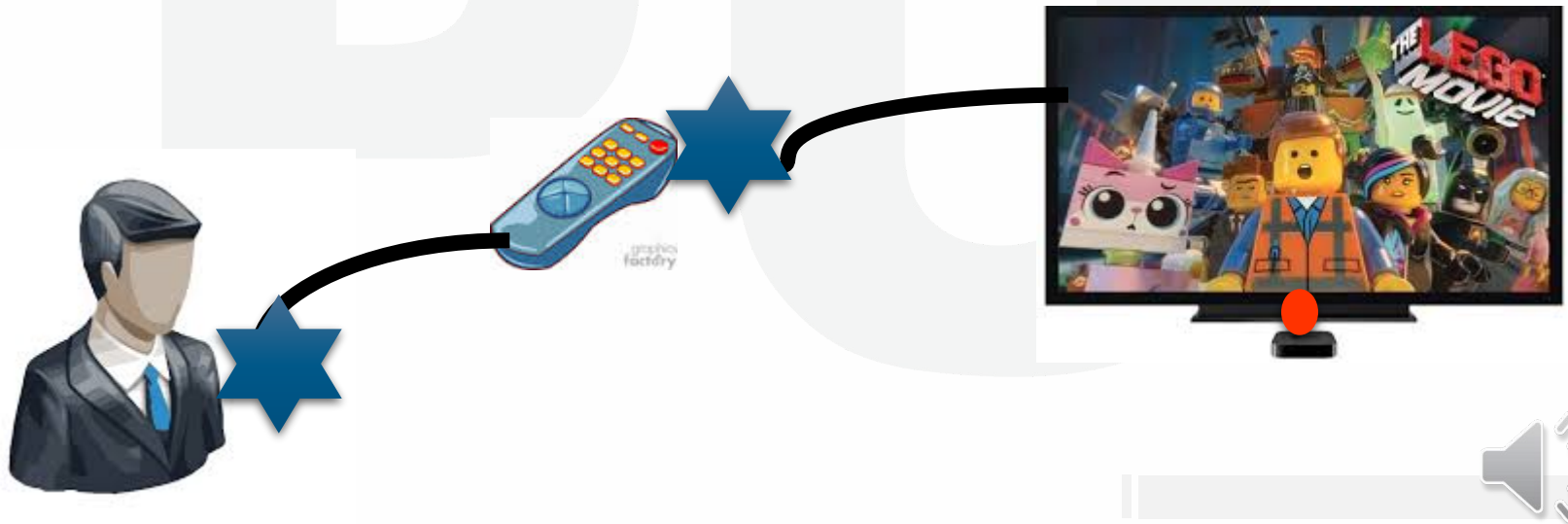
## Drawbacks

- Performance Overhead
- Exposure of Internals



# Interfaces

- **In General**, An interface is a device or system that unrelated entities use to interact.
  - The English language is an interface between two people.
  - A remote control is an interface between you and a television.

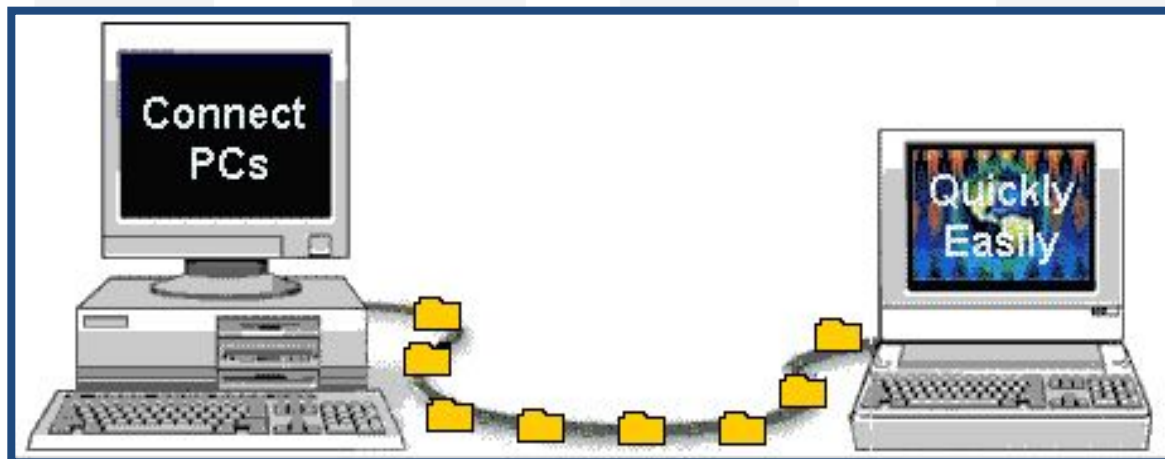




# Interfaces

- **In Computing,** An interface is a shared boundary across which two or more components of a computer system exchange information.

The exchange can be between software, hardware, humans and combinations of these.





# Interfaces

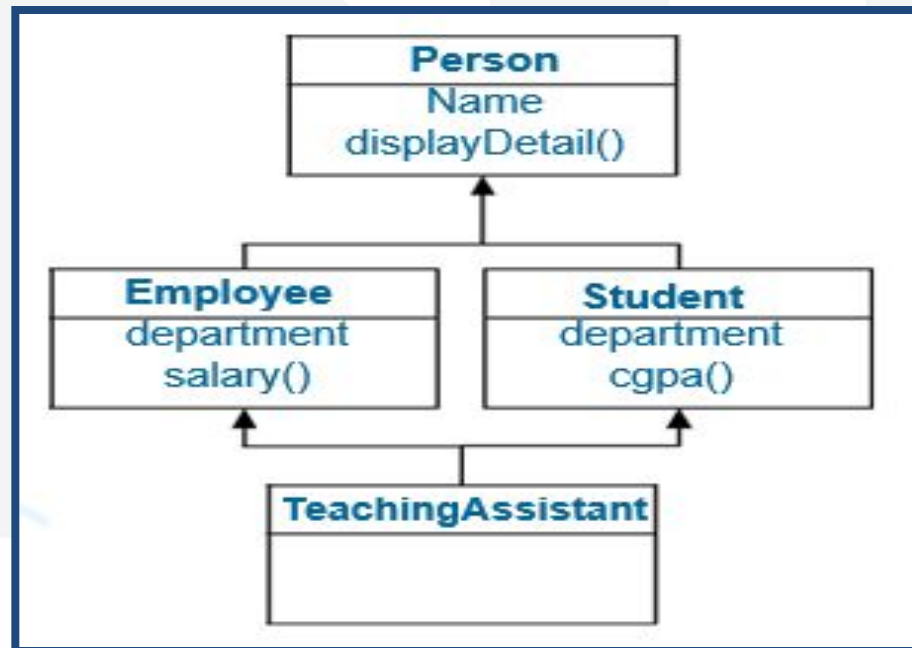
- **In Object oriented programming,** An interface is a common means for unrelated objects to communicate with each other.
- **In Java,**
  - An interface is a way through which unrelated objects use to interact with one another.
  - Using interface, you can specify what a class must do, but not how it does it.
  - It is not a class but a set of requirements for classes that implement the interface



# Interfaces

## Multiple Inheritance - example

For a teaching assistant, we want the properties from both Employee and Student.



# Interfaces

## Problems with Multiple Inheritance

Consider following declaration:

```
ta = new TeachingAssistant();  
ta.department;
```

**Name clash problem:** Which department does ta refers to?

**Combination problem:** Can department from Employee and Student be combined in Teaching Assistant?

**Selection problem:** Can you select between department from Employee and department from Student?

**Replication problem:** Should there be two departments in TeachingAssistant?

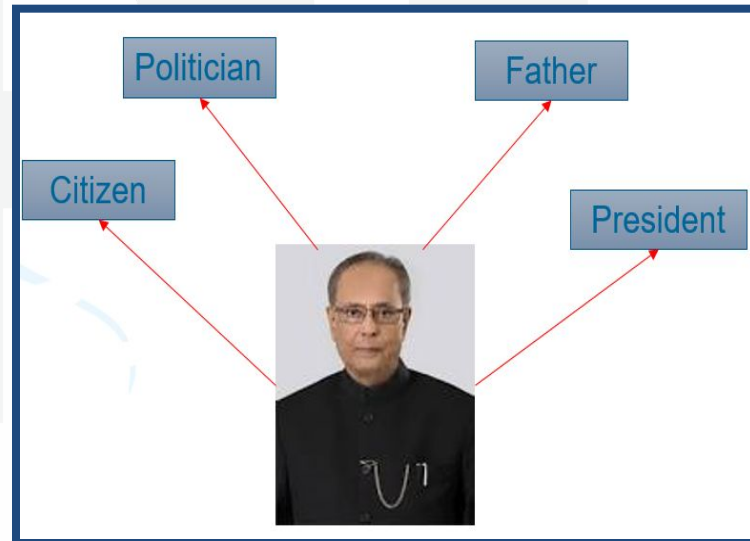


# Interfaces

## Why Interfaces are needed?

Multiple Inheritance in JAVA is not allowed – cannot extend more than one class at a time.

An object may need IS-A relationships with many type.



# Interfaces

## Solution for multiple Inheritance in java

```
public class Person extends Citizen implements Father,
Politician, President { }
```

```
public interface Politician {  
    public void joinParty();  
}
```

```
public interface President  
{public void winPoll();  
}
```

```
Public class Citizen {  
}
```

```
Public interface Father {  
    public void care();  
}
```

```
Public class Person {  
}
```





# Interfaces

## Java's Interface Concept

An interface defines a protocol of behaviour as a collection of method definitions (without implementation) and constants, that can be implemented by any class.

A class that implements the interface agrees to implement all the methods defined in the interface.

If a class includes an interface but does not implement all the methods defined by that interface, then that class must be declared as abstract.





# Interfaces

## Semantic Rules for Interfaces

### Instantiation

Does not make sense on an interface. Interfaces are not classes. You can never use the new operator to instantiate an interface.

```
public interface Comparable { . . . }
```

```
Comparable x = new Comparable( );
```



# Interfaces

## Semantic Rules for Interfaces

### Data Type

An interface can be used as a type, like classes. You can declare interface variables

```
class Employee implements Comparable { . . . }
```

```
Comparable x = new Employee( );
```





# Interfaces

## Semantic Rules for Interfaces

### Access modifiers

An interface can be public or “friendly” (default).

All methods in an interface are by default abstract and public.

- Static, final, private, and protected cannot be used.

All variables (“constants”) are public static final by default

- Private, protected cannot be used.





# Interfaces

## Syntax

- The Declaration of Interface consists of a keyword `interface`, its name, and the members.

```
interface InterfaceName {  
    // constant declaration  
    // method declaration  
    returtype methodname (argumentlist);  
}
```

- The Class that Implements Interface called as Implementation Class uses keyword 'implements'.

```
class classname implements InterfaceName {  
    ... }  
}
```

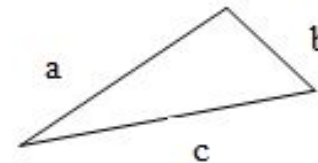
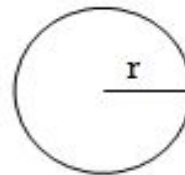
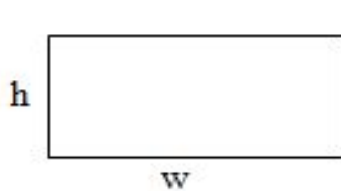




# Interfaces

## Example: An Interface for Shape Classes

- Creating classes to represent rectangles, circles, and triangles and compute their area and perimeter



- It may seem as there is an inheritance relationship here, because rectangle, circle, and triangle are all shapes.
- But code sharing is not useful in this case because each shape computes its area and perimeter in a different way.



# Interfaces

## Define Interface Shape

A better solution would be to write an interface called Shape to represent the common functionality (to compute an area and a perimeter) of all shapes:

```
public interface Shape {  
    public double getArea();  
    public double getPerimeter();  
}
```





# Interfaces

## Class Rectangle implements interface shape

```
public class Rectangle implements Shape {  
    private double width;  
    private double height;  
  
    // Constructs a new rectangle with the given dimensions.  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
    // Returns the area of this rectangle.  
    public double getArea() {  
        return width * height;  
    }  
    // Returns the perimeter of this rectangle.  
    public double getPerimeter() {  
        return 2.0 * (width + height);  
    }  
}
```





# Interfaces

## Class circle implements interface shape

```
public class Circle implements Shape {  
    private double radius;  
  
    // Constructs a new circle with the given radius.  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    // Returns the area of this circle.  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
  
    // Returns the perimeter of this circle.  
    public double getPerimeter() {  
        return 2.0 * Math.PI * radius;  
    }  
}
```





# Interfaces

## Class Triangle implements interface shape

```
public class Triangle implements Shape {  
    private double a;  
    private double b;  
    private double c;  
  
    // Constructs a new Triangle given side lengths.  
    public Triangle(double a, double b, double c) {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
  
    // Returns this triangle's area using Heron's formula.  
    public double getArea() {  
        double s = (a + b + c) / 2.0;  
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));  
    }  
  
    // Returns the perimeter of this triangle.  
    public double getPerimeter() {  
        return a + b + c;  
    }  
}
```

$$\text{area} = \sqrt{s(s-a)(s-b)(s-c)}$$
$$\text{where } s = \frac{a+b+c}{2}$$





# Interfaces

## Class Mensuration with main function

```
class Mensuration
{
    public static void main (String [] arg)
    {
        // Rectangle object
        Rectangle r = new Rectangle(10,20); // Rectangle object
        System.out.println("Area of Rectangle =" + (r.getArea()));
        System.out.println("Perimeter of Rectangle =" + (r.getPerimeter()));
        Circle c = new Circle(10); // Circleobject
        System.out.println("Area of Circle =" + (c.getArea()));
        System.out.println("Perimeter of Circle =" + (c.getPerimeter()));
        Triangle t = new Triangle(3,4,5); // Triangle object
        System.out.println("Area of Triangle =" + (t.getArea()));
        System.out.println("Perimeter of Triangle =" + (t.getPerimeter()));
    }
}
```





# Interfaces

## Output

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\HP>cd Desktop

C:\Users\HP\Desktop>cd Interview

C:\Users\HP\Desktop\Interview>java Mensuration
Area of Rectangle =200.0
Perimeter of Rectangle =60.0
Area of Circle =314.1592653589793
Perimeter of Circle =62.83185307179586
Area of Triangle =6.0
Perimeter of Triangle =12.0

C:\Users\HP\Desktop\Interview>
```





# Interfaces

## Extending Interface

One interface can inherit another interface using the *extends* keyword and not the *implements* keyword.

For example,

```
interface A extends B { }
```

Obviously, any class which implements a “sub-interface” will have to implement each of the methods contained in it’s “super-interface” also.





# Interfaces

## Abstract Class and Interface

Abstract class	Interface
A programmer uses an abstract class when there are some common features shared by all the objects.	A programmer writes an interface when all the features have different implementations for different objects.
Multiple inheritance not possible (Multiple "parent" interfaces)	Multiple inheritance possible (Only one "parent" class)
An abstract class contain both abstract and concrete(non abstract) method	An interface contain only abstract method
In abstract class, abstract keyword is compulsory to declare a method as an abstract	abstract keyword is optional to declare a method as an abstract in interface
An abstract class can have protected, public abstract method	An interface can have only public abstract method
Abstract class contain any type variable	Interface contain only static final variable (constant)



# Interfaces

## Benefits of Interfaces

Following concepts of object oriented programming can be achieved using Interface in JAVA.

1. Abstraction

2. Multiple Inheritance

3. polymorphism







# Interfaces

## Some of the most used Interfaces

### Iterator

To run through a collection of objects without knowing how the objects are stored, e.g., array.

### Cloneable

Used to make a copy of an existing object via the `clone()` method on the class `Object`. This interface is empty.

### •Serializable

Used to Pack a group of objects such that it can be send over a network or stored to disk. This interface is empty.

### •Comparable

The Comparable interface contains a `compareTo` method, and this method must take an `Object` parameter and return an integer





## Object Cloning

- Object cloning refers to creation of exact copy of an object.
- It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object.



# Object Cloning

Let's try...

**Using Assignment Operator to create copy of reference variable**

- In Java, there is no operator to create copy of an object.
- in Java, if we use assignment operator then it will create a copy of reference variable and not the object.





# Object Cloning

## Example

// Java program to demonstrate that assignment  
// operator only creates a new reference to same  
// object.

```
import java.io.*;
```

// A test class whose objects are cloned

```
class Test
{
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}
```

// Driver Class

```
class Main
{
    public static void main(String[] args)
    {
        Test ob1 = new Test();
        System.out.println(ob1.x + " " + ob1.y);
        // Creating a new reference variable ob2 pointing to same address as ob1
        Test ob2 = ob1;
        // Any change made in ob2 will be reflected in ob1
        ob2.x = 100;
        System.out.println(ob1.x+" "+ob1.y);
        System.out.println(ob2.x+" "+ob2.y);
    }
}
```

```
10 20
100 20
100 20
```





## Object Cloning

- The object cloning is a way to create an **exact copy of an object**.
- For this purpose, the **clone()** method of an object class is used to clone an object.
- The **Cloneable** interface must be implemented by a class whose object clone to create. If we do not implement Cloneable interface, clone() method generates **CloneNotSupportedException**.



# Object Cloning

## WHY???

- The clone() method saves the extra processing task for creating the exact copy of an object.
- If we perform it by using the new keyword, it will take a lot of processing to be performed, so we can use object cloning.

## Syntax

**protected Object clone() throws CloneNotSupportedException**





# Object Cloning

## (1) Usage of clone() method -Shallow Copy

```
// A Java program to demonstrate shallow copy using clone()
import java.util.ArrayList;
// An object reference of this class is contained by Test2
class Test
{
    int x, y;
}
// Contains a reference of Test and implements clone with shallow copy.
class Test2 implements Cloneable
{
    int a;
    int b;
    Test c = new Test();
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

```
// Driver class
public class Main
{
    public static void main(String args[]) throws CloneNotSupportedException
    {
        Test2 t1 = new Test2();
        t1.a = 10;
        t1.b = 20;
        t1.c.x = 30;
        t1.c.y = 40;

        Test2 t2 = (Test2)t1.clone();
        // Creating a copy of object t1 and passing it to t2
        t2.a = 100;
        // Change in primitive type of t2 will not be reflected in t1 field
        t2.c.x = 300;
        // Change in object type field will be reflected in both t2 and t1(shallow copy)
        System.out.println(t1.a + " " + t1.b + " " + t1.c.x + " " + t1.c.y);
        System.out.println(t2.a + " " + t2.b + " " + t2.c.x + " " + t2.c.y);
    }
}
```

10	20	300	40
100	20	300	40







# Object Cloning

## (1) Usage of clone() method -Shallow Copy

- In the above example, `t1.clone` returns the shallow copy of the object `t1`.
- Shallow copy is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e. object Y will point to same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
- Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.
- Shallow copies are cheap and simple to make.



# Object Cloning

## (2) Usage of clone() method -Deep Copy

- If we want to create a deep copy of object X and place it in a new object Y then new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other.
- A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.
- A deep copy occurs when an object is copied along with the objects to which it refers.





# Object Cloning

## (2) Usage of clone() method -Deep Copy

```
// A Java program to demonstrate deep copy using clone()
import java.util.ArrayList;
// An object reference of this class is contained by Test2
class Test
{
    int x, y;
}
// Contains a reference of Test and implements clone with deep copy.
class Test2 implements Cloneable
{
    int a, b;
    Test c = new Test();
    public Object clone() throws CloneNotSupportedException
    {
        // Assign the shallow copy to new reference variable t
        Test2 t = (Test2)super.clone();
        t.c = new Test();
        // Create a new object for the field c and assign it to shallow copy obtained,
        // to make it a deep copy
        return t;
    }
}
```

```
public class Main
{
    public static void main(String args[]) throws CloneNotSupportedException
    {
        Test2 t1 = new Test2();
        t1.a = 10;
        t1.b = 20;
        t1.c.x = 30;
        t1.c.y = 40;

        Test2 t3 = (Test2)t1.clone();
        t3.a = 100;

        // Change in primitive type of t2 will not be reflected in t1 field
        t3.c.x = 300;

        // Change in object type field of t2 will not be reflected in t1(deep copy)
        System.out.println(t1.a + " " + t1.b + " " + t1.c.x + " " + t1.c.y);
        System.out.println(t3.a + " " + t3.b + " " + t3.c.x + " " + t3.c.y);
    }
}
```

10	20	30	40
100	20	300	0



# Inner Classes

- An inner class is a class defined within another class, or even within an expression.

## (1) Nested Inner Class

- It can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
- Like class, interface can also be nested and can have access specifiers.



# Inner Classes

## Nested Inner Class Example

```
class Outer {  
    // Simple nested inner class  
    class Inner {  
        public void show() {  
            System.out.println("In a nested class method");  
        }  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Outer.Inner in = new Outer().new Inner();  
        in.show();  
    }  
}
```

In a nested class method





## Inner Classes

- we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself.
- This code will not work

```
class Outer {  
    void outerMethod() {  
        System.out.println("inside outerMethod");  
    }  
    class Inner {  
        public static void main(String[] args){  
            System.out.println("inside inner class Method");  
        }  
    }  
}
```

Error illegal static declaration in inner class Outer.Inner public static void main(String[] args) modifier 'static' is only allowed in constant variable declaration

# Inner Classes

## (2) Anonymous inner Classes

- Anonymous inner classes are declared without any name at all.
- They are created in two ways.
  - a) As subclass of specified type
  - b) As implementer of the specified interface







# Inner Classes

## a) As subclass of specified type

```
class Demo {  
    void show() {  
        System.out.println("i am in show method of super class");  
    }  
}  
  
class Flavor1Demo {  
    // An anonymous class with Demo as base class  
    static Demo d = new Demo() {  
        void show() {  
            super.show();  
            System.out.println("i am in Flavor1Demo class");  
        }  
    };  
  
    public static void main(String[] args){  
        d.show();  
    }  
}
```

i am in show method of super class  
i am in Flavor1Demo class





## Inner Classes

### b) As implementer of the specified interface

```
class Flavor2Demo {  
    // An anonymous class that implements Hello interface  
    static Hello h = new Hello() {  
        public void show() {  
            System.out.println("i am in anonymous class");  
        }  
    };  
    public static void main(String[] args) {  
        h.show();  
    }  
}  
  
interface Hello {  
    void show();  
}
```

i am in anonymous class





## Proxies

- Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’.
- A real world example....cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that’s exactly what the Proxy pattern does, “**Controls and manage access to the object they are protecting**”

### Types of Proxies

- (1) Remote proxy
- (2) Virtual proxy
- (3) Protection proxy
- (4) Smart Proxy





# Proxies

## (1) Remote proxy

- They are responsible for representing the object located remotely.
- Talking to the real object might involve marshalling and unmarshalling of data and talking to the remote object.
- All that logic is encapsulated in these proxies and the client application need not worry about them.





# Proxies

## (2) Virtual proxy

- These proxies will provide some **default and instant results** if the real object is supposed to take some time to produce results.
- These proxies initiate the operation on real objects and provide a default result to the application.
- Once the real object is done, these proxies push the actual data to the client where it has provided dummy data earlier.





## Proxies

### (3) Protection proxy

- If an application does **not have access to some resource** then such proxies will talk to the objects in applications that have access to that resource and then get the result back.
- A very simple real life scenario is our **college internet**, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.





# Proxies

## (4) Smart Proxy

- A smart proxy provides additional layer of security by **interposing** specific actions **when the object is accessed**.
- An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.







# Stream

- A mechanism that either consumes or produces information It is an abstract concept which is implemented in java. Ex. Water pipe
- Provides abstraction and consistent interface across all diverse types of input/output devices like disks, networks and various data structures.
- There are two kinds of Streams

**(1) InPutStream** – The InputStream is used to read data from a source.

**(2) OutPutStream** – The OutputStream is used for writing data to a destination.





# Stream

## Types of Stream

### (1) Byte Stream

- Java byte streams are used to perform input and output of 8-bit bytes.
- Though there are many classes related to byte streams but the most frequently used classes are, `FileInputStream` and `FileOutputStream`.
- Following constructor takes a file name as a string to create an input stream object to read the file.

```
InputStream f = new FileInputStream("C:/java/hello");
```

- Following constructor takes a file name as a string to create an input stream object to write the file.

```
OutputStream f = new FileOutputStream("C:/java/hello");
```





# Stream

## (1) Byte Stream

- Some important byte stream classes are...

Stream class	Description
<b>BufferedInputStream</b>	Used for Buffered Input Stream.
<b>BufferedOutputStream</b>	Used for Buffered Output Stream.
<b>DataInputStream</b>	Contains method for reading java standard datatype
<b>DataOutputStream</b>	An output stream that contain method for writing java standard data type
<b>FileInputStream</b>	Input stream that reads from a file
<b>FileOutputStream</b>	Output stream that write to a file.
<b>InputStream</b>	Abstract class that describe stream input.
<b>OutputStream</b>	Abstract class that describe stream output.
<b>PrintStream</b>	Output Stream that contain <code>print()</code> and <code>println()</code> method





# Stream

## Types of Stream

### (2) Character Stream

- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode.
- Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**.
- Though internally **FileReader** uses **FileInputStream** and **FileWriter** uses **FileOutputStream** but here the major difference is that **FileReader** reads two bytes at a time and **FileWriter** writes two bytes at a time.





# Stream

## (2) Character Stream

- Some important character stream classes are...

Stream class	Description
<b>BufferedReader</b>	Handles buffered input stream.
<b>BufferedWriter</b>	Handles buffered output stream.
<b>FileReader</b>	Input stream that reads from file.
<b>FileWriter</b>	Output stream that writes to file.
<b>InputStreamReader</b>	Input stream that translate byte to character
<b>OutputStreamReader</b>	Output stream that translate character to byte.
<b>PrintWriter</b>	Output Stream that contain <code>print()</code> and <code>println()</code> method.
<b>Reader</b>	Abstract class that define character stream input
<b>Writer</b>	Abstract class that define character stream output





# Stream

Some Standard streams provided by java are...

- **Standard Input** – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as `System.in`.
- **Standard Output** – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as `System.out`.
- **Standard Error** – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as `System.err`.





# I/O Programming

Based on Input/output streams Let's do I/O programming.

**Ex1: Read contents of file 'hello.txt'**

This will read from file hello.txt

```
import java.io.*;
class ReadHello{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("hello.txt");
            int i=0;
            while((i=fin.read())!=-1){
                System.out.println((char)i);
            }
            fin.close();
        } catch(Exception e){
            system.out.println(e);}
    }
}
```







# I/O Programming

Based on Input/output streams Let's  
do I/O programming.

**Ex2: Write in a file 'hello.txt'**

This will create file hello.txt and write  
Hello students.

```
import java.io.*;
public class Intellipaat{
    public static void main(String args[]){
        try{
            FileOutputStream fo=new FileOutputStream("hello.txt");
            String i="Hello students";
            byte b[]=i.getBytes(); //converting string into byte array
            fo.write(i);
            fo.close();
        }
        catch(Exception e){
            system.out.println(e);}
    }
}
```



# × ○ DIGITAL LEARNING CONTENT



## Parul<sup>®</sup> University



[www.paruluniversity.ac.in](http://www.paruluniversity.ac.in)

