



# JAVA INHERITANCE

**BY: PRASAD PUJAR**

**ASST. PROF**

**DEPARTMENT OF COMPUTER SCIENCE &  
ENGINEERING**

**KLS GOGTE INSTITUTE OF TECHNOLOGY, BELAGAVI-  
590008**

# Inheritance in Java

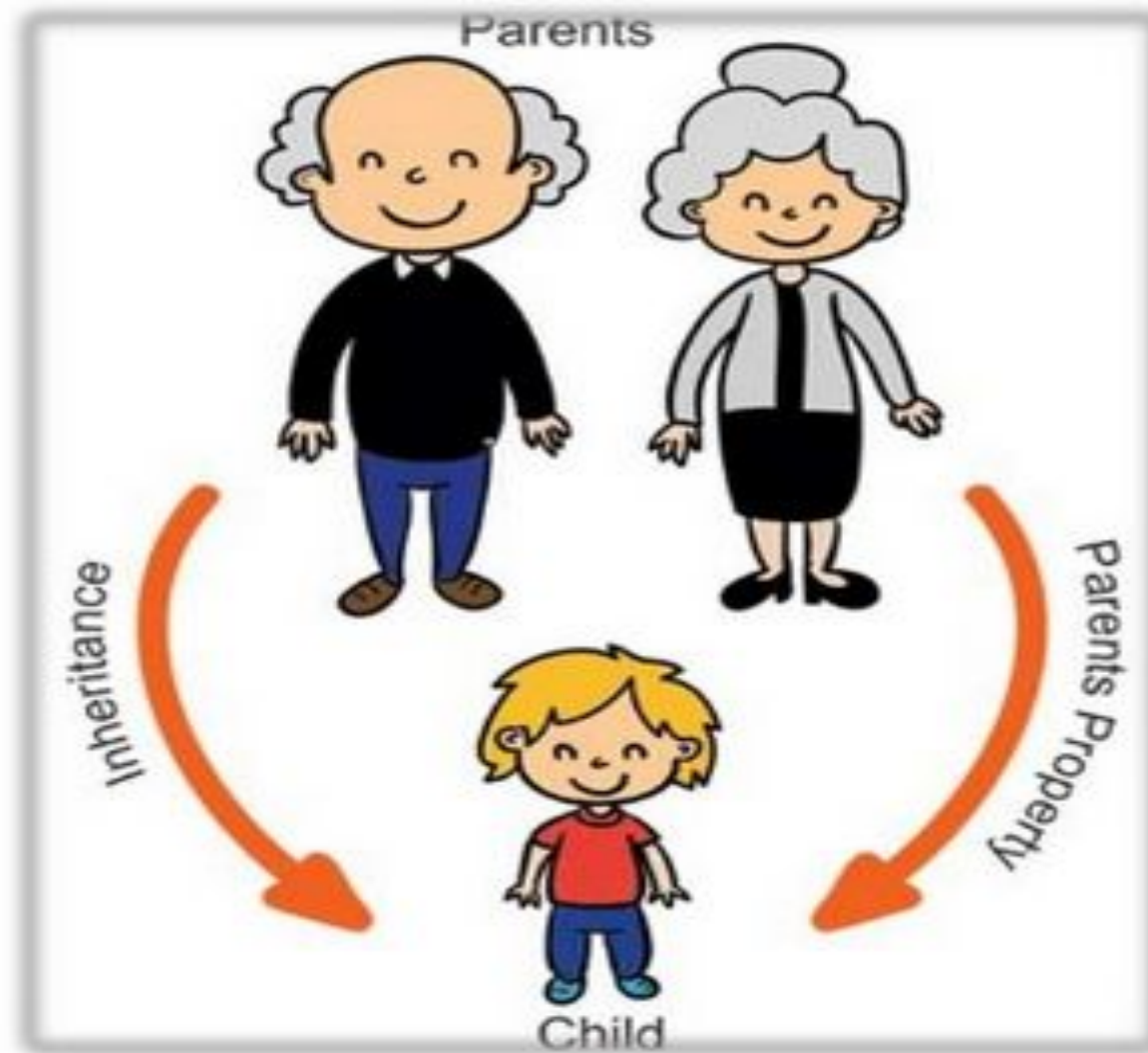
- **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of parent object.
- When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as Super class(Parent) and Sub class(child) in Java language.
- Inheritance defines is-a relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.



# Inheritance in Java

- Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
- **Why use Inheritance ?**
- For Method Overriding (used for Runtime Polymorphism).
- It's main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

# Inheritance in Java



# Inheritance in Java

- **Important points**

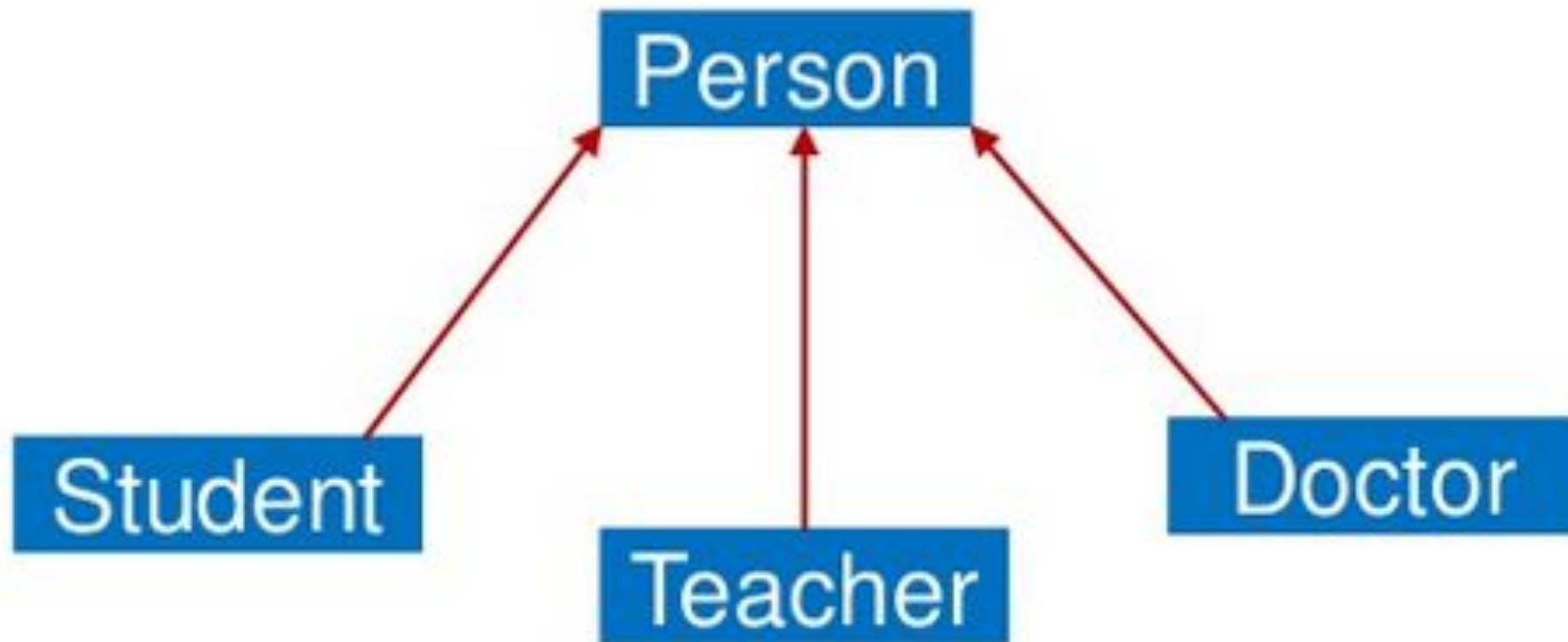
- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.
- The data members and methods of a class are known as features.
- The concept of inheritance is also known as re-usability or extendable classes or sub classing or derivation.



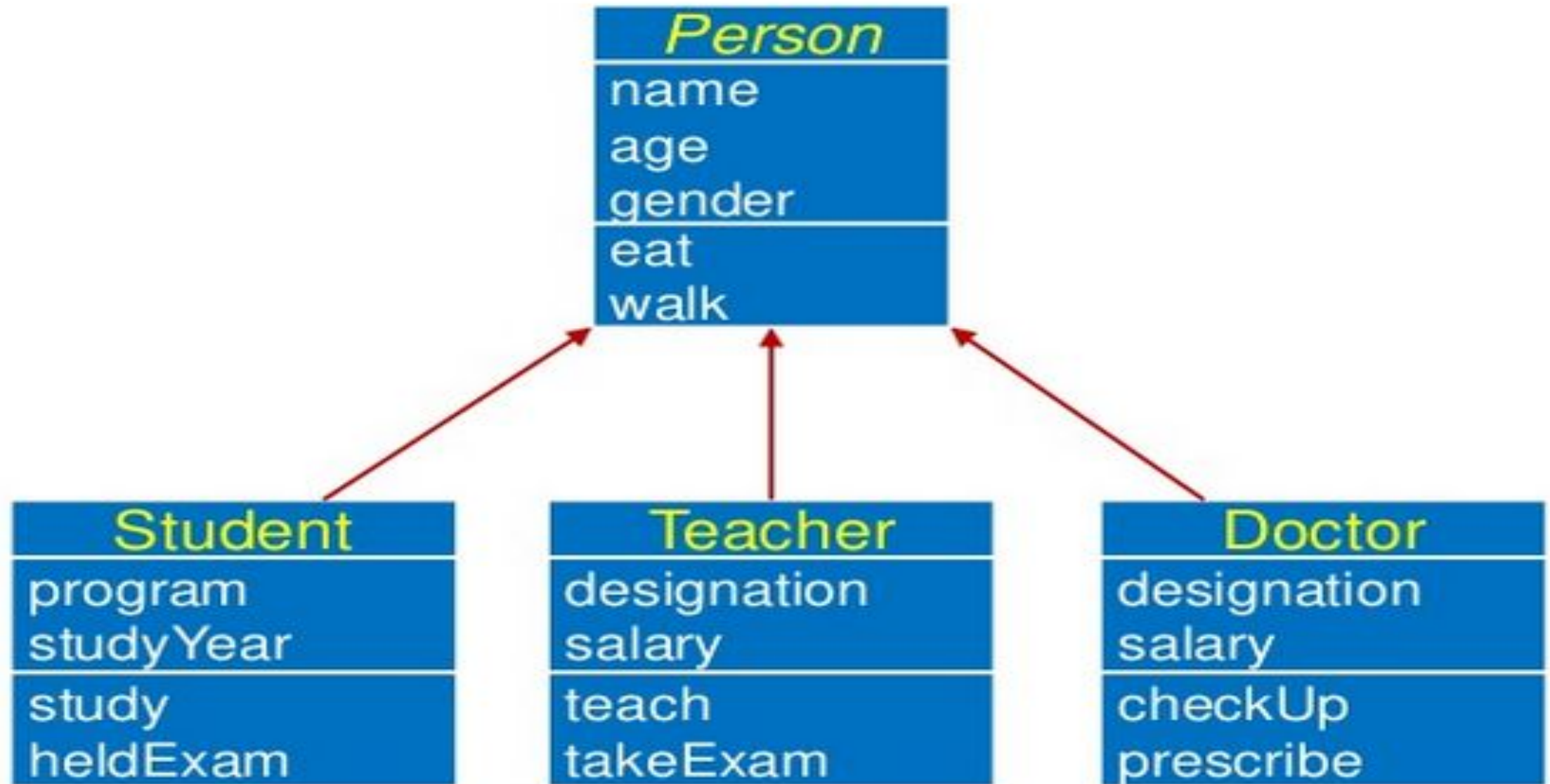
# Inheritance in Java

No	Term	Definition
1	Inheritance	<b>Inheritance</b> is a process where one object acquires the properties of another object
2	Subclass	Class which inherits the properties of another object is called as <b>subclass</b>
3	Superclass	Class whose properties are inherited by subclass is called as <b>superclass</b>
4	Keywords Used	extends and implements

## Example – Inheritance



## Example – “IS A” Relationship





## **Reuse with Inheritance**

- Main purpose of inheritance is reuse
- We can easily add new classes by inheriting from existing classes
  - Select an existing class closer to the desired functionality
  - Create a new class and inherit it from the selected class
  - Add to and/or modify the inherited functionality

# Inheritance in Java

- Inheritance in Java
- Inheritance in Java is done using
  - **extends** – In case of Java class and abstract class
  - **implements** – In case of Java interface.
- What is inherited
  - In Java when a class is extended, sub-class inherits all the **public, protected** and **default (Only if the sub-class is located in the same package as the super class)** methods and fields of the super class.
- What is not inherited
  - **Private** fields and methods of the super class are not inherited by the sub-class and can't be accessed directly by the subclass.
  - Constructors of the super-class are not inherited. There is a concept of constructor chaining in Java which determines in what order constructors are called in case of inheritance.



# Inheritance in Java

- Syntax of Inheritance

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class.

# Inheritance in Java

- **extends** Keyword
- **extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

```
class Super {  
    ....  
    ....  
}  
class Sub extends Super {  
    ....  
    ....  
}
```



# Inheritance in Java

- Please Note:
- In inheritance Parent Class and Child Class having multiple names, List of the name are below:
- **Parent Class = Base Class = Super Class**
- **Child Class = Derived Class = Sub Class**

# Simple Example of Inheritance

```
class Parent
{
    public void p1() {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {
    public void c1() {
        System.out.println("Child method");
    }

    public static void main(String[] args) {
        Child cobj = new Child();
        cobj.c1(); //Calling method of Child class
        cobj.p1(); //Calling method of Parent class
    }
}
```



## Access Control and Inheritance

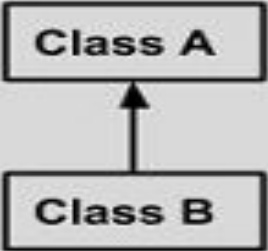
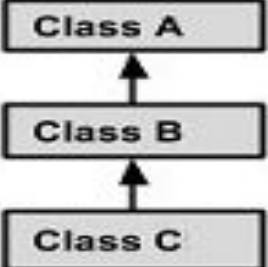
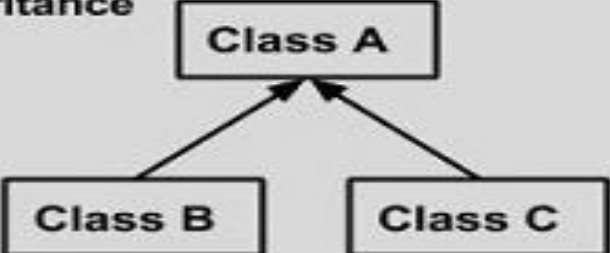
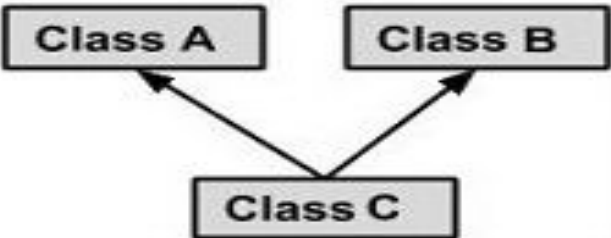
- A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the members of derived classes should be declared private in the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

## Access Control and Inheritance

- The following rules for inherited methods are enforced:
  1. Methods declared public in a superclass also must be public in all subclasses.
  2. Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
  3. Methods declared private are not inherited at all, so there is no rule for them.



<p><b>Single Inheritance</b></p>  <pre> graph BT     B[Class B] --&gt; A[Class A] </pre>	<pre> public class A {     ..... } public class B extends A {     ..... } </pre>
<p><b>Multi Level Inheritance</b></p>  <pre> graph BT     C[Class C] --&gt; B[Class B]     B --&gt; A[Class A] </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends B {.....} </pre>
<p><b>Hierarchical Inheritance</b></p>  <pre> graph BT     B[Class B] --&gt; A[Class A]     C[Class C] --&gt; A </pre>	<pre> public class A { .....} public class B extends A {.....} public class C extends A {.....} </pre>
<p><b>Multiple Inheritance</b></p>  <pre> graph BT     C[Class C] --&gt; A[Class A]     C --&gt; B[Class B] </pre>	<pre> public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance </pre>

## Method Overriding in Java

- Declaring a method in **subclass** which is already present in **parent class** is known as method overriding.

OR

- In other words If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

OR

- In other words If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Without Inheritance Method Overriding is not Possible

## Example of Method Overriding

```
class Vehicle{  
    void run(){  
        System.out.println("Vehicle is running");  
    }  
class Bike extends Vehicle{  
    void run(){  
        System.out.println("Bike is running safely");  
    }  
public static void main(String args[]){  
    Bike obj = new Bike();  
    obj.run();  
}  
}
```



## Example of Method Overriding

```
class Vehicle{  
    void run(){  
        System.out.println("Vehicle is running");  
    }  
class Bike extends Vehicle{  
    void run(){  
        System.out.println("Bike is running safely");  
    }  
  
public static void main(String args[]){  
    Bike obj = new Bike();  
    obj.run();  
}
```

Method name  
are same

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

# Method Overriding in Java

- **Advantage of Java Method Overriding**

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism

- **Rules for Method Overriding**

- method must have same name as in the parent class.
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).



## Difference between Overloading and Overriding

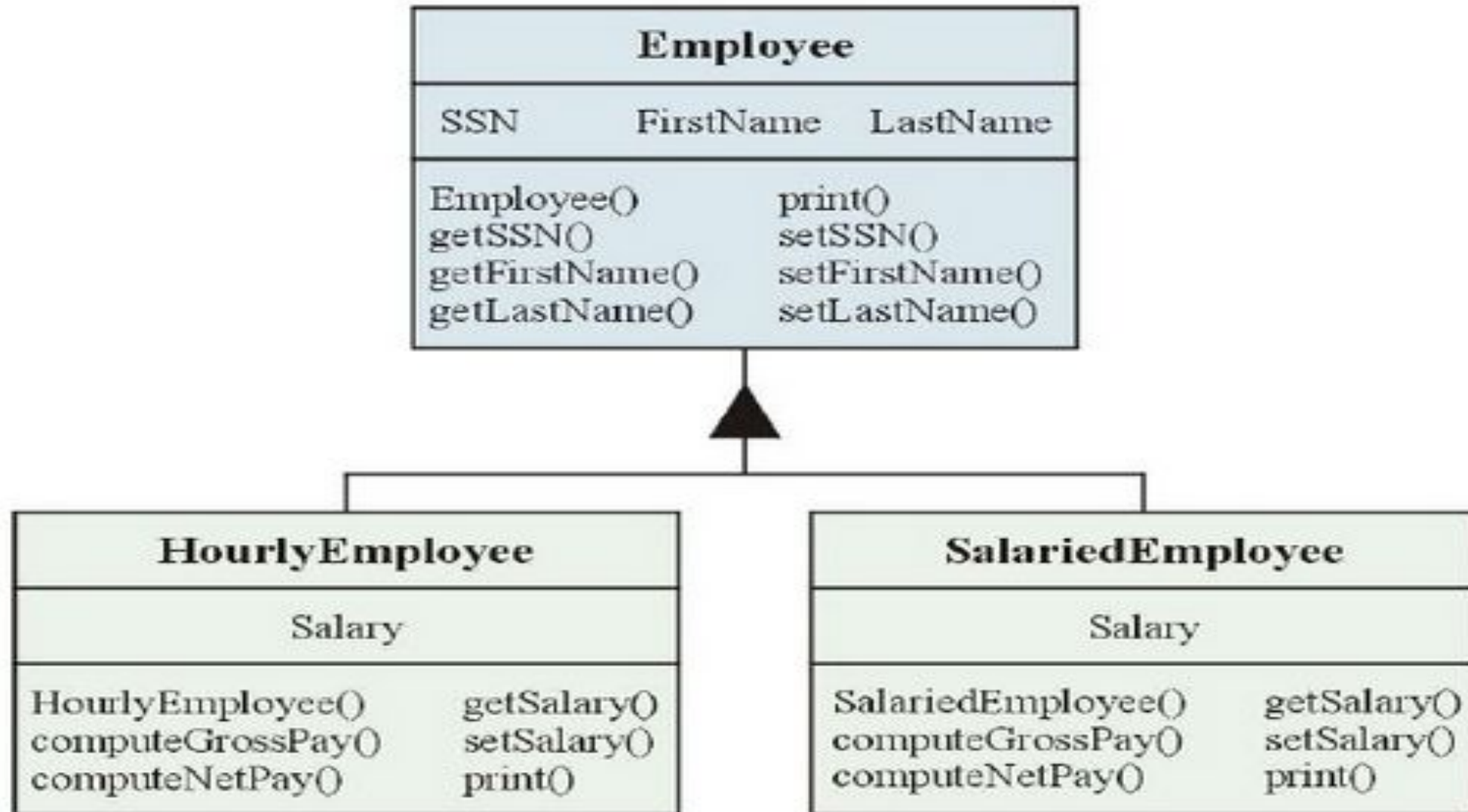
Overloading	Overriding
Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
Method signature must be different.	Method signature must be same.
Private, static and final methods can be overloaded.	Private, static and final methods can not be override.



## Difference between Overloading and Overriding

Overloading	Overriding
Access modifiers point of view no restriction.	Access modifiers point of view not reduced scope of Access modifiers but increased.
Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
Overloading can be exhibited both are method and constructor level.	Overriding can be exhibited only at method label.
Overloading can be done at both static and non-static methods.	Overriding can be done only at non-static method.
For overloading methods return type may or may not be same.	For overriding method return type should be same.

# Inheritance Concepts - Example





# Inheritance Concepts – Employee Example

- Employee is a base class.
- HourlyEmployee and SalariedEmployee are derived classes that inherit all data and function members from Employee.
  - e.g., SSN and setLastName()
- Each derived class can add data and function members.
  - e.g., Salary and computeGrossPay()
- Different derived classes can defined the same members with different data types or implementations.
  - e.g., Salary can be a float in HourlyEmployee and an int in SalariedEmployee, computeGrossPay() can use a different algorithm in each derived class.



# Constructor Function Calls

- The derived class constructor function is called first - it instantiates its parameters then calls the base class constructor function.
- The base class constructor function instantiates its parameters and calls its base class constructor function.
- If a base class has no base classes, it executes the body of its constructor function
- When a base class constructor function terminates the nearest derived class constructor function executes.
- Calls and parameter instantiation go “up the hierarchy”, execution goes “down the hierarchy”

# Constructor Function Calls - Example

Order of call and execution for HourlyEmployee():

- HourlyEmployee() is called
- HourlyEmployee() instantiates its parameters and calls Employee()
- Employee() instantiates its parameters and calls Object()
- Object() instantiates its parameters, executes, and returns to Employee()
- Employee() executes and returns to HourlyEmployee ()
- HourlyEmployee() executes and returns



# Abstract Class

- An *abstract class* is a class that is declared abstract.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- Abstract class may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.



- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject
{
    // declare fields
    // declare non-abstract methods
    abstract void draw();
}
```

- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it does not, the subclass must also be declared abstract.

```
// A Simple demonstration of abstract.
abstract class A {
    abstract void callme();

    // concrete methods are still allowed in abstract classes
    void callmetoo() {
        System.out.println("This is a concrete method.");
    }
}

class B extends A {
    void callme() {
        System.out.println("B's implementation of callme.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}
```



```
// Using abstract methods and classes.
abstract class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // area is now an abstract method
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
```

```
class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created

        figref = r;
        System.out.println("Area is " + figref.area());

        figref = t;
        System.out.println("Area is " + figref.area());
    }
}
```



# The Object Class

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object, i.e. Object is a superclass of all other classes.
- A reference variable of type Object can refer to an object of any other class.
- Even arrays are implemented as classes, a variable of type Object can also refer to any array.
- The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object <i>object</i> )	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long <i>milliseconds</i> ) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i> )	Waits on another thread of execution.



# JAVA INTERFACES



# Inheritance

- ❑ OOP allows you to derive new classes from existing classes. This is called *inheritance*.
- ❑ Inheritance is an important and powerful concept in Java. Every class you define in Java is inherited from an existing class.
- ❑ Sometimes it is necessary to derive a subclass from several classes, thus inheriting their data and methods. In Java only one parent class is allowed.
- ❑ With *interfaces*, you can obtain effect of multiple inheritance.





# What is an Interface?

- ❑ An interface is a classlike construct that contains only constants and abstract methods.
- ❑ Cannot be instantiated. Only classes that implements interfaces can be instantiated. However, `final public static` variables can be defined to interface types.
- ❑ Why not just use abstract classes? Java does not permit multiple inheritance from classes, but permits implementation of *multiple interfaces*.



# What is an Interface?

- ❑ Protocol for classes that completely separates specification/behaviour from implementation.
- ❑ One class can implement many interfaces
- ❑ One interface can be implemented by many classes
- ❑ By providing the `interface` keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.





# Why an Interface?

- ❑ Normally, in order for a method to be called from one class to another, both classes need to be present at *compile time* so the Java compiler can check to ensure that the method signatures are compatible.
- ❑ In a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.
- ❑ Interfaces are designed to avoid this problem.
- ❑ Interfaces are designed to support dynamic method resolution at *run time*.



# Why an Interface?

- ❑ Interfaces are designed to avoid this problem.
- ❑ They disconnect the definition of a method or set of methods from the inheritance hierarchy.
- ❑ Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.
- ❑ This is where the real power of interfaces is realized.





# Creating an Interface

File InterfaceName.java:

```
modifier interface InterfaceName  
{  
    constants declarations;  
    methods signatures;  
}
```

Modifier is **public** or not used.

File ClassName.java:

```
modifier Class ClassName implements InterfaceName  
{  
    methods implementation;  
}
```


If a class implements an interface, it *should override* all the abstract methods declared in the interface.



# Creating an Interface

```
public interface MyInterface
{
    public void aMethod1(int i);    // an abstract methods
    public void aMethod2(double a);
    ...
    public void aMethodN();
}

public Class MyClass implements MyInterface
{
    public void aMethod1(int i) { // implementaion }
    public void aMethod2(double a) { // implementaion }
    ...
    public void aMethodN() { // implementaion }
}
```





# Creating a Multiple Interface

```
modifier interface InterfaceName1 {  
    methods1 signatures;  
}  
modifier interface InterfaceName2 {  
    methods2 signatures;  
}  
...  
modifier interface InterfaceNameN {  
    methodsN signatures;  
}
```



# Creating a Multiple Interface

If a class implements a multiple interfaces, it *should override* all the abstract methods declared in all the interfaces.

```
public Class ClassName implements InterfaceName1,  
                                   InterfaceName2, ..., InterfaceNameN  
{  
    methods1 implementation;  
    methods2 implementation;  
    ...  
    methodsN implementation;  
}
```





# Example of Creating an Interface

```
// This interface is defined in  
// java.lang package  
public interface Comparable  
{  
    public int compareTo(Object obj);  
}
```




# Comparable Circle

```
public interface Comparable  
{  
    public int compareTo(Object obj);  
}
```


---

```
public Class Circle extends Shape  
                implements Comparable {  
    public int compareTo(Object o) {  
        if (getRadius() > ((Circle)o).getRadius())  
            return 1;  
        else if (getRadius() < ((Circle)o).getRadius())  
            return -1;  
        else  
            return 0;  
    }  
}
```



# Comparable Cylinder

```
public Class Cylinder extends Circle  
                implements Comparable {  
    public int compareTo(Object o) {  
        if (findVolume() > ((Cylinder)o).findVolume())  
            return 1;  
        else if (findVolume() < ((Cylinder)o).findVolume())  
            return -1;  
        else  
            return 0;  
    }  
}
```





# Generic findMax Method

```
public class Max
{
    // Return the maximum between two objects
    public static Comparable findMax(Comparable ob1,
                                     Comparable ob2)
    {
        if (ob1.compareTo(ob2) > 0)
            return ob1;
        else
            return ob2;
    }
}
```



# Using Interface

This is one of the key features of interfaces.

- The method to be executed is looked up dynamically at *run time*, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the "callee."





# Using Interface

Objective: Use the `findMax()` method to find a maximum circle between two circles:

```
Circle circle1 = new Circle(5);  
Circle circle2 = new Circle(10);  
Comparable circle = Max.findMax(circle1, circle2);  
System.out.println(((Circle)circle).getRadius());  
// 10.0  
System.out.println(circle.toString());  
// [Circle] radius = 10.0
```



## Extending interface

An interface can extend another interface, similarly to the way that a class can extend another class.

The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

**Note :** Any class that implements an interface **must implement all methods defined by that interface**, including any that inherited from other interfaces.



<http://www.java2all.com>



# extending interfaces

- One interface may inherit another interface.
- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {  
    void myMethod1(...) ;  
}  
interface MyInterface2 extends MyInterface1 {  
    void myMethod2(...) ;  
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.



## Example: Interface Inheritance 1

- Consider interfaces A and B.

```
interface A {  
    void meth1();  
    void meth2();  
}
```

B extends A:

```
interface B extends A {  
    void meth3();  
}
```

## Example: Interface Inheritance 2

- MyClass must implement all of A and B methods:

```
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```



## variables in interface

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:
  - 1) declare an interface with variables initialized to the desired values
  - 2) include that interface in a class through implementation
- As no methods are included in the interface, the class does not implement
- anything except importing the variables as constants.

## Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;  
interface SharedConstants {  
    int NO = 0;  
    int YES = 1;  
    int MAYBE = 2;  
    int LATER = 3;  
    int SOON = 4;  
    int NEVER = 5;  
}
```