

Chapter 3

Inheritance and Polymorphism

Inheritance method overriding

- **Inheritance** is one of the basic concept in object orientation in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities.
- With inheritance, programmers save time during program development by reusing proven and debugged high-quality software.
- This also increases the likelihood that a system will be implemented effectively.

- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
- The existing class is called the **superclass**, and the new class is the **subclass**.
- A subclass normally adds its own fields and methods.
- Therefore, a subclass is more specific than its superclass and represents a more specialized group of objects.
- Typically, the subclass exhibits the behaviors of its superclass and additional behaviors that are specific to the subclass.

- ***Syntax for inheritance***

class *SubClassName* **extends** *SuperClassName*{}

- In java the extend keyword is used to show inheritance in a class.
- A subclass can call a constructor defined by its superclass by use of the following form of **super**:

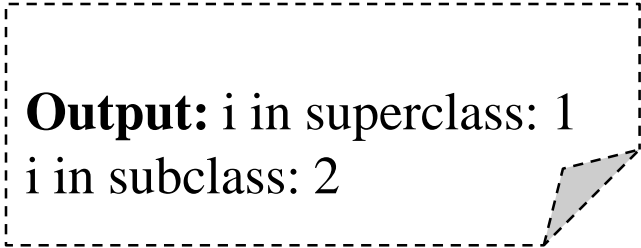
super(parameter-list);

- Here, *parameter-list* specifies any parameters needed by the constructor in the superclass.
- **super()** must always be the first statement executed inside a subclass constructor.
- There is a second form of super that acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:
 super.member
- Here, member can be either a method or an instance variable.

- This form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Example

```
class A {  
    int i;  
}  
  
// Create a subclass by extending class A.  
class B extends A {  
    int i; // this i hides the i in A  
    B(int a, int b) {  
        super.i = a; // i in A  
        i = b; // i in B  
    }  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}  
  
class UseSuper {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2);
```

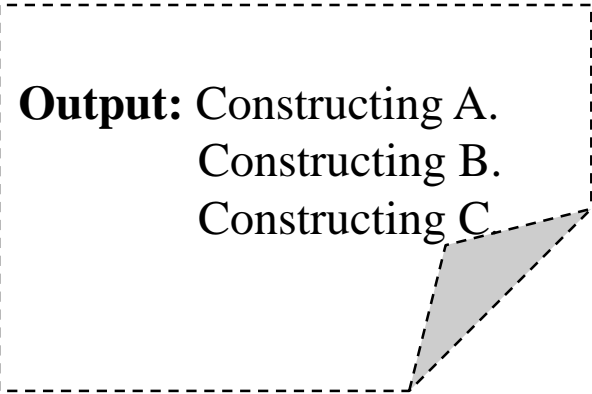


Output: i in superclass: 1
i in subclass: 2

- When a subclass object is created, whose constructor is executed first, the one in the subclass or the one defined by the superclass?
- For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa?
 - The answer is that in a class hierarchy, constructors are called **in order of derivation, from superclass to subclass**.
 - Further, since **super()** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super()** is used.

Example: Demonstrate when constructors are called

```
class A {  
    A() {  
        System.out.println("Constructing A.");  
    }  
}  
// Create a subclass by extending class A.  
class B extends A {  
    B() {  
        System.out.println("Constructing B.");  
    }  
}  
// Create another subclass by extending B.  
class C extends B {  
    C() {  
        System.out.println("Constructing C.");  
    }  
}  
class OrderOfConstruction {  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```



Output: Constructing A.
Constructing B.
Constructing C

- **Example:** Suppose we want to maintain a class roster for a class whose enrolled students include both undergraduate and graduate students.
- For each student, we record her or his name, three test scores, and the final course grade.
- The final course grade, either pass or no pass, is determined by the following formula:

Type of Student	Grading Scheme
Undergraduate	Pass if $(\text{test1} + \text{test2} + \text{test3})/3 \geq 70$
Graduate	Pass if $(\text{test1} + \text{test2} + \text{test3})/3 \geq 80$

- We solve this question by defining three classes.
- The first is the Student class to incorporate behavior and data common to both graduate and undergraduate students.
- The second and third classes are the GraduateStudent class to incorporate behavior specific to graduate students and the UndergraduateStudent class to incorporate behavior specific to undergraduate students.
- The Student class is defined as:

```
class Student {  
    protected final static int NUM_OF_TESTS = 3;  
    protected String name;  
    protected int[] test;  
    protected String courseGrade;  
    public Student( ) {  
        this("No Name");  
    }  
    public Student(String studentName) {  
        name = studentName;  
        test = new int[NUM_OF_TESTS];  
        courseGrade = "****";  
    }  
    public String getCourseGrade( ) {  
        return courseGrade;  
    }  
    public String getName( ) {  
        return name;  
    }  
}
```

```
public int getTestScore(int testNumber) {  
    return test[testNumber-1];  
}  
public void setName(String newName) {  
    name = newName;  
}  
public void setTestScore(int testNumber, int testScore) {  
    test[testNumber-1] = testScore;  
}  
public void computeCourseGrade( ) {  
    //do nothing - override this method in the subclass  
    Method overriding is discussed in next section  
}}
```

```
class GraduateStudent extends Student {  
    public void computeCourseGrade() {  
        int total = 0;  
        for (int i = 0; i < NUM_OF_TESTS; i++) {  
            total += test[i];  
        }  
        if (total/NUM_OF_TESTS >= 80) {  
            courseGrade = "Pass";  
        } else {  
            courseGrade = "No Pass";  
        }  
    }  
}
```

```
class UndergraduateStudent extends Student {  
    public void computeCourseGrade() {  
        int total = 0;  
        for (int i = 0; i < NUM_OF_TESTS; i++) {  
            total += test[i];  
        }  
        if (total/NUM_OF_TESTS >= 70) {  
            courseGrade = "Pass";  
        } else {  
            courseGrade = "No Pass";  
        }  
    }  
}
```

Method Overriding

- In a class hierarchy, when a method in a subclass has the same return type and signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

Example

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + "
            " + j);
    }
}
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
}
```

```
// display k – this overrides
    show() in A
    void show() {
        System.out.println("k: " +
            k);
    }
}
class Override {
    public static void
        main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls
            show() in B
    }
}
```

Overloading vs Overriding: Difference between Method Overloading and Method Overriding

Following are the key differences between method overloading and overriding in Java.

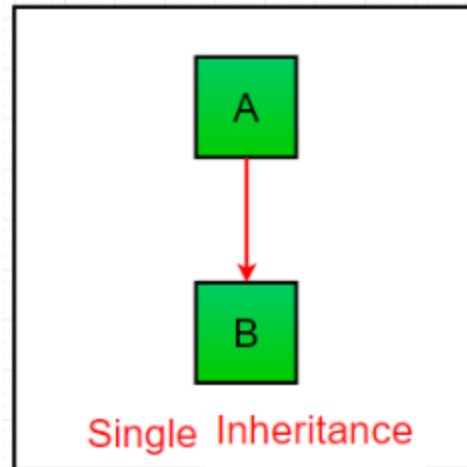
Method Overloading	Method Overriding
<ul style="list-style-type: none">•It is used to increase the readability of the program•It is performed within the same class•Parameters must be different in case of overloading•Is an example of compile-time polymorphism•Return type can be different but you must change the parameters as well.	<ul style="list-style-type: none">•Provides a specific implementation of the method already in the parent class•It involves multiple classes•Parameters must be same in case of overriding•It is an example of runtime polymorphism•Return type must be same in overriding

Types of Inheritance

- There are different types of inheritance which is supported by Java.

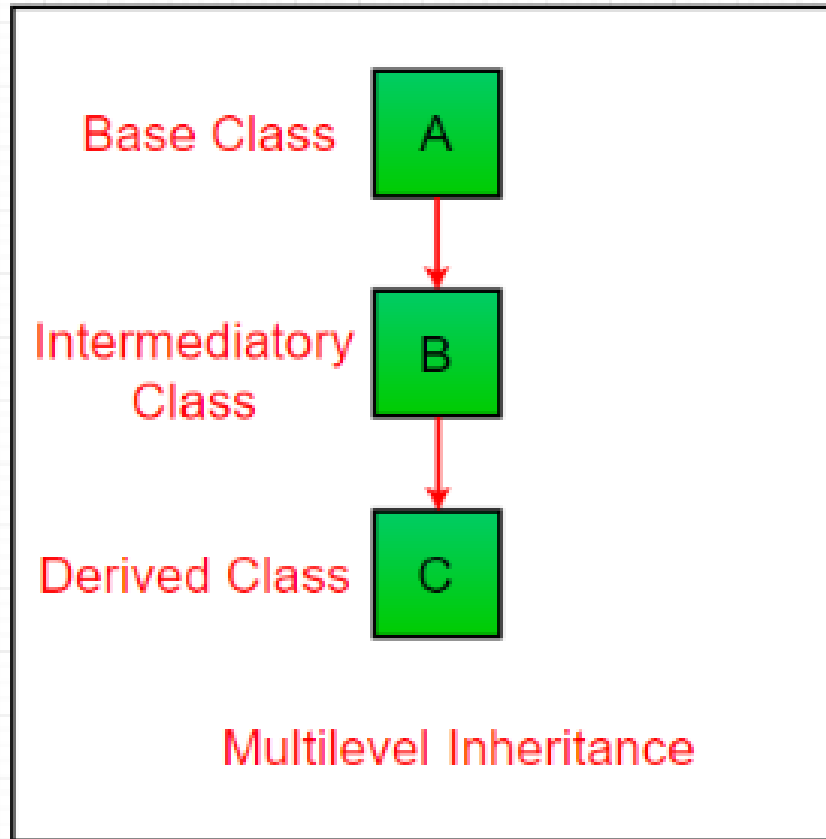
a) Single Inheritance:

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



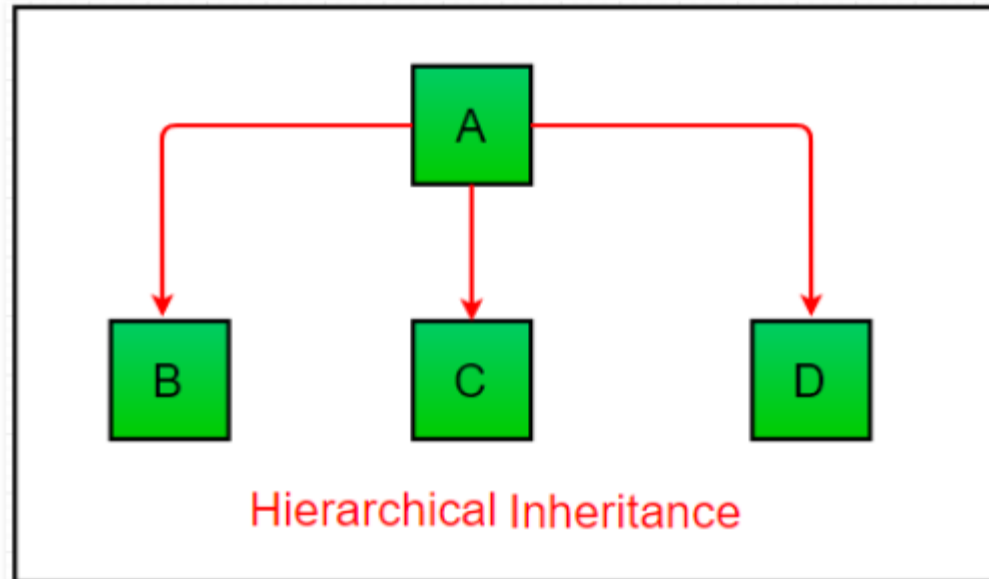
b) Multilevel Inheritance:

- 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 figure, 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.



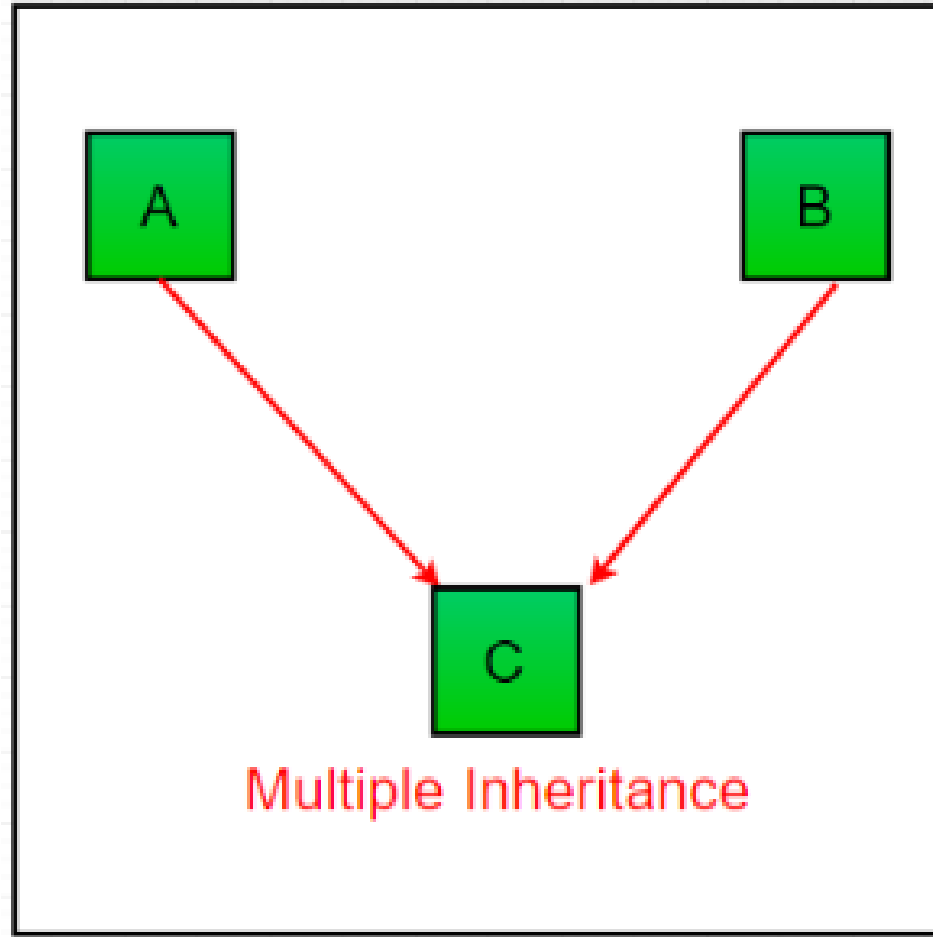
c) Hierarchical Inheritance:

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



d) Multiple Inheritance (Through Interfaces):

- In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes.
- Please note that 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.



Polymorphism

- Polymorphism is a concept by which we can perform a single action by different ways.
- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms.
 - So polymorphism means many forms.

- There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.
- If you overload method in java, it is the example of compile time polymorphism.
- Here, we will focus on runtime polymorphism in java which is achieved by method overriding.

Runtime Polymorphism in Java

- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

- Let's first understand the upcasting before Runtime Polymorphism.
- ***Upcasting***
 - When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:
 - **class A{}**
 - **class B extends A{}**
 - **A a=new B();//upcasting**

Example of Java Runtime Polymorphism

- In this example, we are creating two classes Bike and Splendar.
- Splendar class extends Bike class and overrides its run() method.
- We are calling the run method by the reference variable of Parent class.
- Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike{  
void run(){  
    System.out.println("running");  
} }  
class Splender extends Bike{  
void run(){  
    System.out.println("running safely with 60km");  
}  
public static void main(String args[]){  
    Bike b = new Splender();//upcasting  
    b.run();  
}  
}
```

Output: running safely with
60km.

Example 2

- Consider a scenario; Bank is a class that provides method to get the rate of interest. But, rate of interest may differ according to banks.
- For example, Abyssinia, Awash and CBE banks are providing 8.4%, 7.3% and 9.7% rate of interest.

```
class Bank{  
  float getRateOfInterest(){return 0;}  
}  
  
class Abyssinia extends Bank{  
  float getRateOfInterest(){return 8.4f;}  
}  
  
class Awash extends Bank{  
  float getRateOfInterest(){return 7.3f;}  
}  
  
class CBE extends Bank{  
  float getRateOfInterest(){return 9.7f;}  
}
```



```
class TestPolymorphism{  
public static void main(String args[]){  
    Bank b;  
    b=new Abyssinia ();  
    System.out.println("Abyssinia  
        Rate of Interest: "+b.getRateOfInterest());  
    b=new Awash ();  
    System.out.println("Awash  
        Rate of Interest: "+b.getRateOfInterest());  
    b=new CBE ();  
    System.out.println("CBE  
        Rate of Interest: "+b.getRateOfInterest());  
} }
```

Output:

Abyssinia Rate of Interest: 8.4

Awash Rate of Interest: 7.3

CBE Rate of Interest: 9.7

Java Runtime Polymorphism with Data Member

- Method is overridden not the data members, so runtime polymorphism can't be achieved by data members.
- In the example given below, both the classes have a data member speed limit; we are accessing the data member by the reference variable of Parent class which refers to the subclass object.
- Since we are accessing the data member which is not overridden, hence it will access the data member of Parent class always.

Example

```
class Bike{  
    int speedlimit=90;  
}  
class Honda3 extends Bike{  
    int speedlimit=150;  
  
    public static void main(String args[]){  
        Bike obj=new Honda3();  
        System.out.println(obj.speedlimit);//90  
    }  
}
```

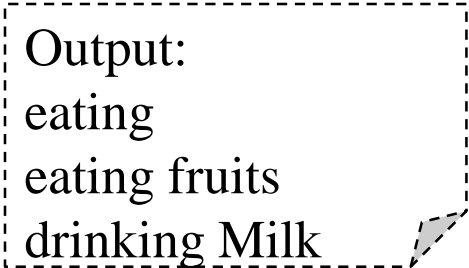


Output:
90

Java Runtime Polymorphism with Multilevel Inheritance

Let's see the simple example of Runtime Polymorphism with multilevel inheritance.

```
class Animal{  
void eat(){System.out.println("eating");} }  
class Dog extends Animal{  
void eat(){System.out.println("eating fruits");}  
}  
class BabyDog extends Dog{  
void eat(){System.out.println("drinking milk");}  
public static void main(String args[]){  
    Animal a1,a2,a3;  
    a1=new Animal();  
    a2=new Dog();  
    a3=new BabyDog();  
    a1.eat();  
    a2.eat();  
    a3.eat();  
}
```



Output:
eating
eating fruits
drinking Milk

Abstract classes and interfaces

- A class that is declared with abstract keyword, is known as abstract class in java.
- It can have abstract and non-abstract methods (method with body).
- Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
 - It shows only important things to the user and hides the internal details
 - for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.
- There are two ways to achieve abstraction in java
 - Abstract class
 - Interface

Abstract class in Java

- A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

abstract class A{}

- An abstract class's purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design.

Abstract method

- A method that is declared as **abstract** and does not have implementation is known as abstract method.
- **Example abstract method**
 - **abstract void** printStatus();//no body and abstract

- A class that contains any abstract methods must be explicitly declared abstract even if that class contains some concrete(nonabstract) methods.
- Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods.
- Constructors and static methods cannot be declared abstract. Constructors are not inherited, so an abstract constructor could never be implemented

- Though non-private static methods are inherited, they cannot be overridden.
- Since abstract methods are meant to be overridden so that they can process objects based on their types, it would not make sense to declare a static method as abstract.

Example

```
package abstractClass;
public abstract class AbstractClass {
    public AbstractClass(){
        System.out.println("super class constructor");
    }
    public static void staticDraw(){
        System.out.println("static draw method");
    }
    public abstract void draw();
}
```

```
package abstractClass;
public class Rectangle extends AbstractClass {
    public Rectangle(){
        System.out.println("sub class constructor");
    }
    @Override
    public void draw(){
        System.out.println("drawing rectangle...");
    }
}
```

```
package abstractClass;
public class TestAbstract {
    public static void main(String args[]){
        AbstractClass r=new Rectangle();
        r.draw();
        AbstractClass.staticDraw();
        r.staticDraw();//but better to use class reference
    }
}
```

```
super class constructor
sub class constructor
drawing rectangle...
static draw method
static draw method
```

Example

```
abstract class Bank{  
  abstract int getRateOfInterest();  
}  
class Abyssinia extends Bank{  
  int getRateOfInterest(){return 7;}  }  
class CBE extends Bank{  
  int getRateOfInterest(){return 8;}  
}  
  class TestBank{  
    public static void main(String args[]){  
      Bank b;  
      b=new Abyssinia();  
      System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  
  
      b=new CBE();  
      System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  
  
    }  
  }
```

Output:

Rate of Interest is: 7 %

Rate of Interest is: 8 %

- **Rule: If there is any abstract method in a class, that class must be abstract.**

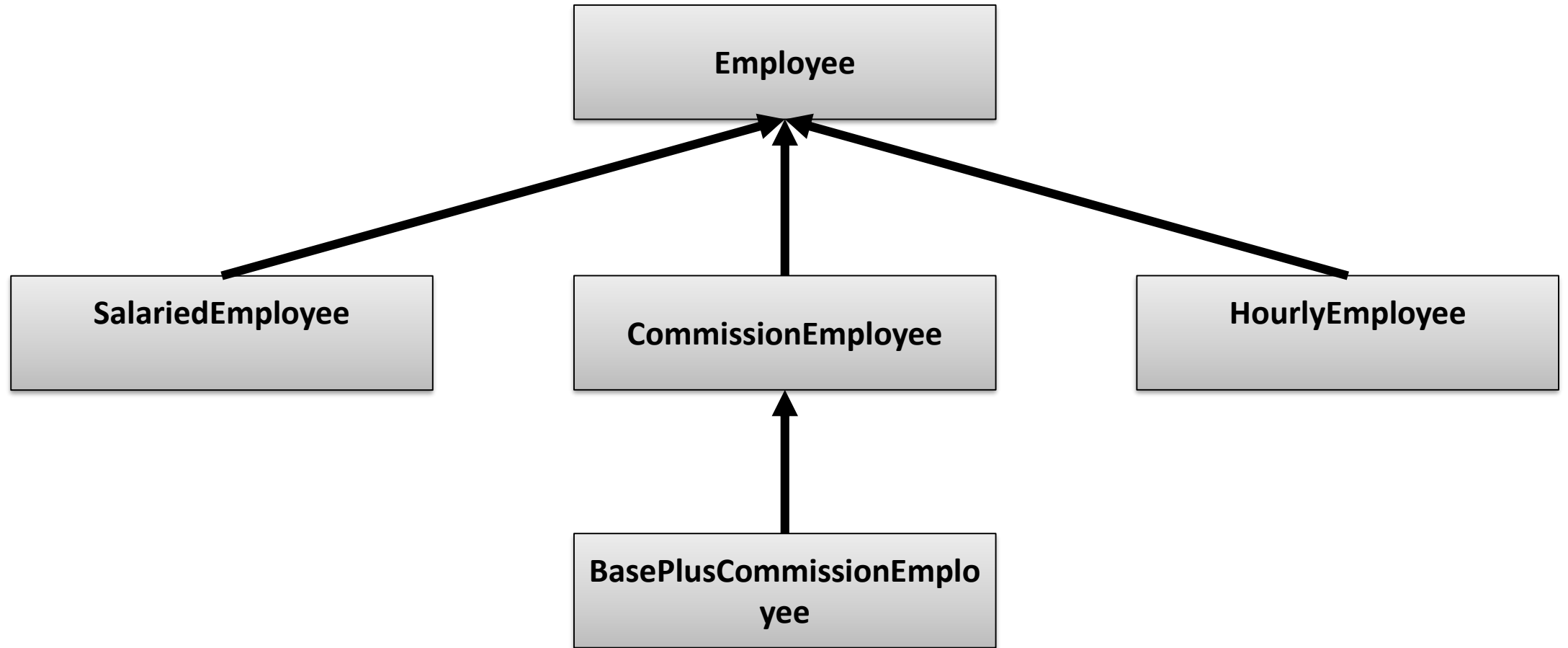
```
class Bike12{  abstract void run();  
}
```

Output: compile time error

- **Rule: If you are extending any abstract class that has abstract method, you must either provide the implementation of the method or make this class abstract.**

Lab work

- A company pays its employees on a weekly basis.
- The employees are of four types:
 - Salaried employees are paid a fixed weekly salary regardless of the number of hours worked,
 - Hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours,
 - commission employees are paid a percentage of their sales and
 - base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries.
- The company wants you to write an application that performs its payroll calculations polymorphically.



final Methods and Classes

- Variables can be declared final to indicate that they cannot be modified *after* they're initialized—such variables represent constant values.
- It's also possible to declare methods, method parameters and classes with the final modifier.
- A **final** method in a superclass *cannot* be overridden in a subclass—this guarantees that the final method implementation will be used by all direct and indirect subclasses in the hierarchy.
 - Methods that are declared private are implicitly final, because it's not possible to override them in a subclass.
 - Methods that are declared static are also implicitly final.
- A final method's declaration can never change, so all subclasses use the same method implementation, and calls to final methods are resolved at compile time—this is known as static binding

Interfaces

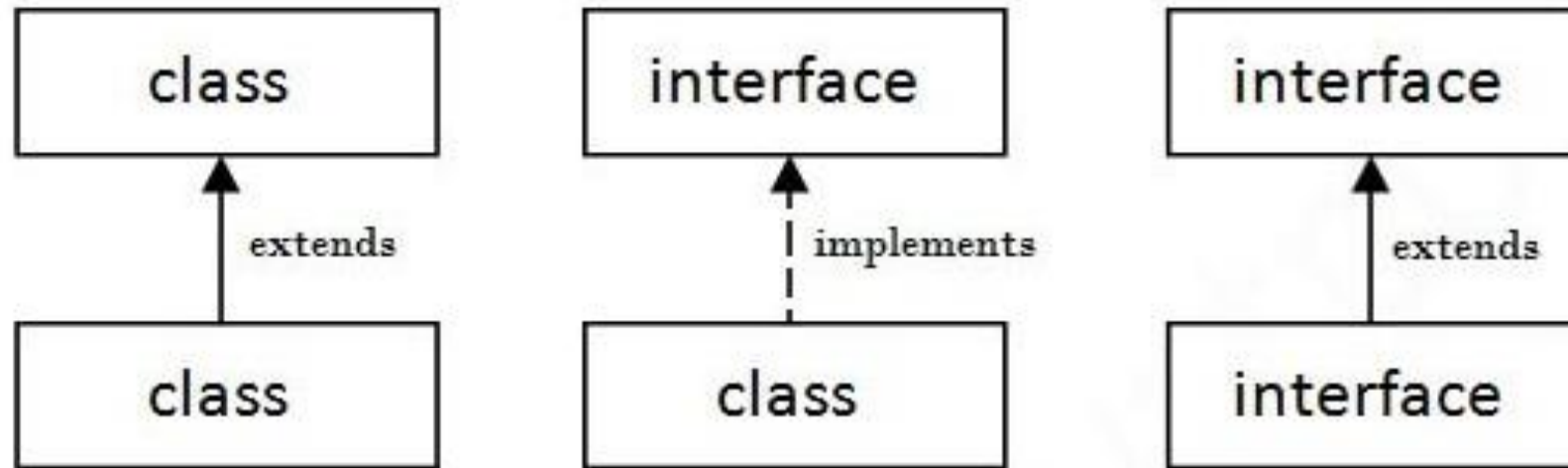
- An **interface in java** is a blueprint of a class.
- It has constants and abstract methods.
- The interface in java is a mechanism to achieve abstraction.
- There can be only abstract methods in the java interface not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have methods and variables but the methods declared in interface contain only method signature, not body.

- Interface is declared by using **interface** keyword.
- It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.
- A class that implement interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

- As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.



Example: java interface which provides the implementation of Bank interface.

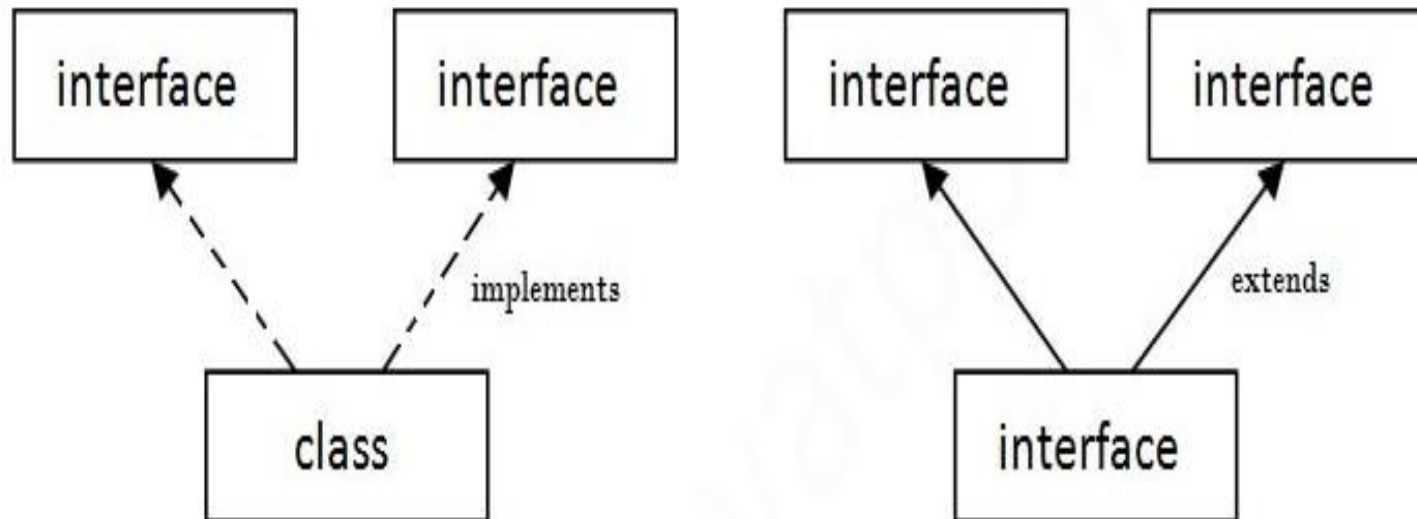
```
interface Bank{  
    float rateOfInterest(); }  
class Abyssinia implements Bank{  
    public float rateOfInterest(){return 9.15f;}  
}  
class CBE implements Bank{  
    public float rateOfInterest(){return 9.7f;}  
}  
class TestInterface2{  
    public static void main(String[] args){  
        Bank b=new Abyssinia();  
        System.out.println("ROI: "+b.rateOfInterest());  
    }  
}
```



Output: ROI: 9.15

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Example:

```
interface Printable{  
    void print(); }  
interface Showable{  
    void show();  
}  
class A7 implements Printable,Showable{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("Welcome");}  
    public static void main(String args[]){  
        A7 obj = new A7();  
        obj.print();  
        obj.show();  
    } }  
}
```



Output: Hello
Welcome

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
} }
```