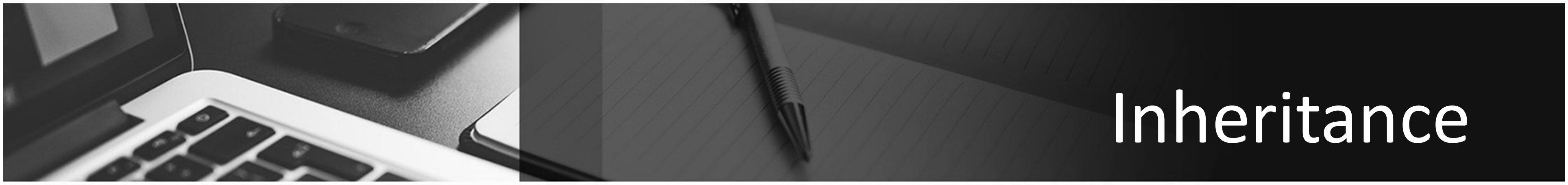
OOPS

Lecture 13-14-15: Inheritance

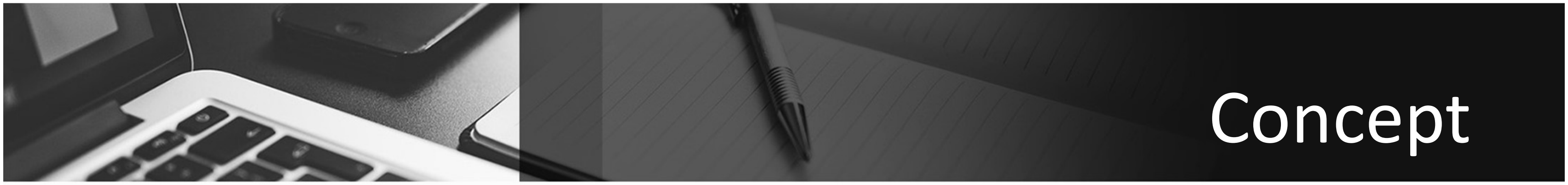
Dr. Anjali
Assistant Professor

Atal Bihari Vajpayee
Indian Institute of Information Technology
and Management (ABV-IIITM), Gwalior
(An Institute of National Importance, Ministry of Education, Government of India)



Inheritance

- Objects of different kinds (classes) have their own unique behavior.
- Objects of different kinds often **share** similar behavior too.
- For example:
  - Student, Professor, Software Engineer, Chemical Engineer, Physicist, Guitarist, Drummer
  - Each has distinct actions to perform, but they also have many features and behavior in common



Concept

- *Inheritance*: you can create new classes that are built on existing classes. Through the way of inheritance, you can reuse the existing class's methods and fields, and you can also add new methods and fields to adapt the new classes to new situations
- Subclass and superclass
- Subclass and superclass have a IsA relationship: an object of a subclass IsA(n) object of its superclass

Why not just copy-and-paste?

- Say I have an **Employee** class and want to create an **HourlyEmployee** class that adds info about wages. Why not copy-and-paste, then modify?

1. Fixing bugs: what if one were wrong?
2. Maintenance: what if **Employee** changes?
3. Code-reuse: would code that takes an **Employee** as a parameter also take an **HourlyEmployee**?

The Basics of Inheritance

- Inheritance allows you to **reuse** methods that you've already written to create more specialized versions of a class.
- Java keyword: **extends**.
  - public class HourlyEmployee **extends** Employee.
  - We say that an HourlyEmployee **IS-A** Employee
- Employee is said to be the parent/base class (or **superclass**), and HourlyEmployee is called a child/derived class (or **subclass**).
- HourlyEmployee receives copies of all of the non-private methods and variables present in Employee.

Sample Classes

Superclass

```
public class Person{
        private String name;

        public Person ( ) {
                name = “no_name_yet”;
        }

        public Person ( String initialName ) {
                this.name = initialName;
        }

        public String getName ( ) {
                return name;
        }

        public void setName ( String newName ) {
                name = newName;
        }}
```

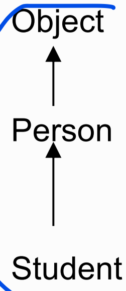
Subclass

```
public class Student extends Person {
        private int studentNumber;

        public Student ( ) {
                super( ); //person() is invoked
                studentNumber = 0;
        }
        public Student (String initialName, int
        initialStudentNumber) {
                super(initialName);
                studentNumber =
        initialStudentNumber;
        }
        public int getStudentNumber ( ) {
                return studentNumber;
        }
        public void setStudentNumber (int
        newStudentNumber ) {
               studentNumber = newStudentNumber;
        }
PSVM()
{Student s1=new Student();//
 s1.setName(“abc”);
}
}
```

Classes hierarchy

- Every class is an extended (inherited) class, whether or not it’s declared to be. If a class does not declared to explicitly extend any other class, then it implicitly extends the `Object` class
- Class hierarchy of previous example



Object

Person

Student

Fields/Methods in Extended Classes

- An object of an extended class contains two sets of variables and methods

1. fields/methods which are defined locally in the extended class
2. fields/methods which are inherited from the superclass

?

☞ What are the fields for a `Student` object in the previous example

Constructors in extended classes

- ✓ A constructor of the extended class can invoke one of the superclass’s constructors by using the `super` method.
- ✓ If no superclass constructor is invoked explicitly, then the superclass’s no-arg constructor

```
super( )
```

  is invoked automatically as the first statement of the extended class’s constructor.
- ✓ Constructors are not methods and are NOT inherited.

Three phases of an object’s construction

- When an object is created, memory is allocated for all its fields, which are initially set to be their default values. It is then followed by a three-phase construction:
  - invoke a superclass’s constructor
  - initialize the fields by using their initializers and initialization blocks
  - execute the body of the constructor

- The invoked superclass’s constructor is executed using the same three-phase constructor. This process is executed recursively until the `Object` class is reached

To Illustrate the Construction Order. . .

```
class X {
      protected int xOri = 1;
      protected int whichOri;
      public X() {
            whichOri = xOri;
      }
}
    Y objectY = new Y();
```

```
class Y extends X {
      protected int yOri = 2;
      public Y() {
              //super();
              whichOri = yOri;
      }
}
```

Step	what happens	xOri	yOri	whichOri
0	fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	Object constructor invoked	0	0	0
4	X field initialization	1	0	0
5	X constructor executed	1	0	1
6	Y field initialization	1	2	1
7	Y constructor executed	1	2	2

Inheritance Example 2

```
class Cleanser {
    private String activeIngredient;
    public void dilute(int percent)     {// water-down}
    public void apply(DirtyThing d) {// pour it on}
    public void scrub(Brush b)          {// watch it work}
}
public class Detergent extends Cleanser {
    private String specialIngredient;
    public void scrub(Brush b) {
        // scrub gently, then
        super.scrub(b);  // the usual way
    }
    public void foam() { // make bubbles}
}
```

Access Control, Again

- **Detergent** does indeed have an **activeIngredient**, but it’s not accessible.
- If **Detergent** need to access it, it must be either
  - made **protected** (or friendly) in **Cleanser**, or
  - be accessible through get and set methods in **Cleanser**.
- You can’t inherit just to get access!

What Is A **Detergent** Object?

- An object of type **Cleanser**, having all the members of **Cleanser**.
- An object of type **Detergent**, having all the additional members of **Detergent**.

Subclasses and Constructors

- Think of a **Detergent** object as containing a **Cleanser** *sub-object*.
- So, that sub-object has to be constructed when you create a **Detergent** object.
- The **Cleanser** object has to be created *first*, since constructing the remaining **Detergent** part might rely on it.
- "Always call the base class constructor first."

Subclasses and Constructors

```
class Cleanser {
    private String activeIngredient;
    int id;
    Cleanser() {
        System.out.println(“Cleanser constructor”);
    }
}
public class Detergent extends Cleanser {
    private String specialIngredient;
    Detergent() {
    System.out.println(“Detergent constructor”);
    }
    public static void main(String[] args) {
        Detergent d = new Detergent();
    }
}
```

Subclasses and Constructors

```
class Cleanser {
    private String activeIngredient;
    Cleanser(String active) {
        activeIngredient = active;
    }
}
public class Detergent extends Cleanser {
    private String specialIngredient;
    Detergent(String active, String special) {
        //super(active);  // what if this isn’t here?
        specialIngredient = special;
    }
}
```

protected Members in Inheritance

```
class Animal {
  protected String name;
  Animal(){}
Animal(String name)
{this.name=name;}
  protected void display() {
    System.out.println("I am an animal.");
  }
}
class Dog extends Animal {
   int num_l;
    Dog(){}
  Dog(int num_l, String name){
   super(name); this.num_l=num_l;}
  public void getInfo() {
    System.out.println("My name is " + name);
  }
}
```

```
public class Main extends Dog {
  public static void main(String[] args) {

    // create an object of the subclass
    Dog labrador = new Dog();
    Main m1= new Main();

    // access protected field and method
    // using the object of subclass
    labrador.name = "Rocky";
    labrador.display();
    labrador.getInfo();
    //m1.display();
      m1.getInfo();
}
}
```

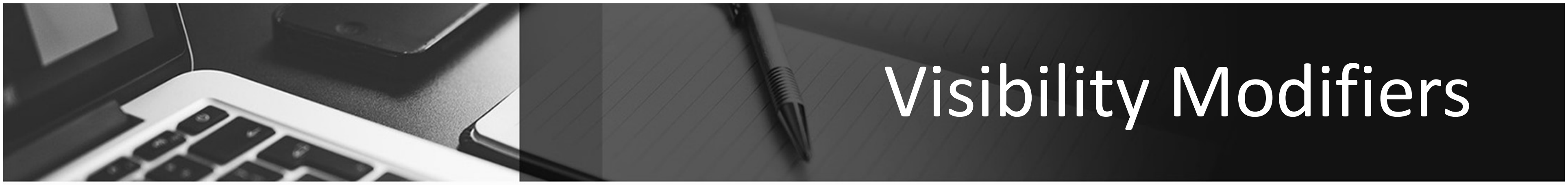
```
I am an animal.
My name is Rocky
I am an animal.
My name is null
```

Some Key Ideas in Inheritance

- Code reuse
- Overriding methods
- Protected visibility
- The “super” keyword

Code re-use

- The subclass **inherits** all the **public** and **protected** methods and fields of the superclass.
  - Constructors are not inherited
  - Constructors can be invoked by the subclass
- Subclass can add new methods and fields.



Visibility Modifiers

- **Public** – Accessible by any other class in any package.
- **Private** – Accessible only within the class.
- **Protected** – Accessible only by classes within the same package and any subclasses in other packages.
  - (For this reason, some choose not to use protected, but use private with accessors)
- Default (No Modifier) – Accessible by classes in the same package but not by classes in other packages.
  - **Use sparingly!**

The “super” Keyword

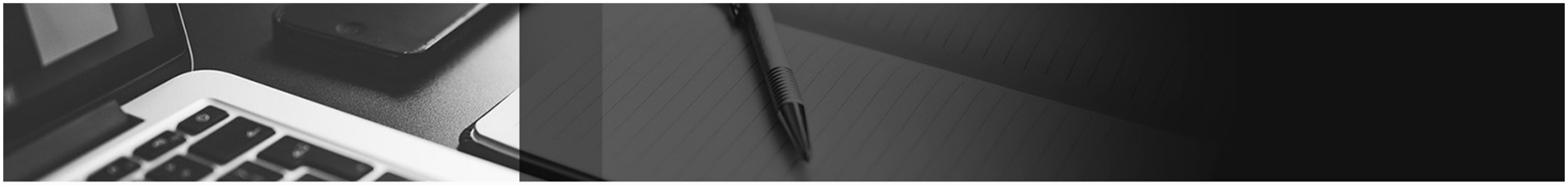
- It’s like the word “this,” only “super”:
- In a child class, “super” refers to its parent.
- Two uses:

1. ~~To~~ call a parent’s method, use **super.*methodName*(…)**
2. ~~To~~ call a parent’s constructor, use super(some parameter) from the child class’ constructor

- Reminder, still use *this* (super not needed) to access parent’s fields

Rules of using *super* in constructors

- A **super(…)** call must be the first line of the code of an object’s constructor if it is to be used.
- Instance variables cannot be passed along with the **super(…)** call. Only variables that are passed to the constructor that calls **super** may be passed to **super**.
- `super(…)` and `this(…)` cannot be used in the same constructor.

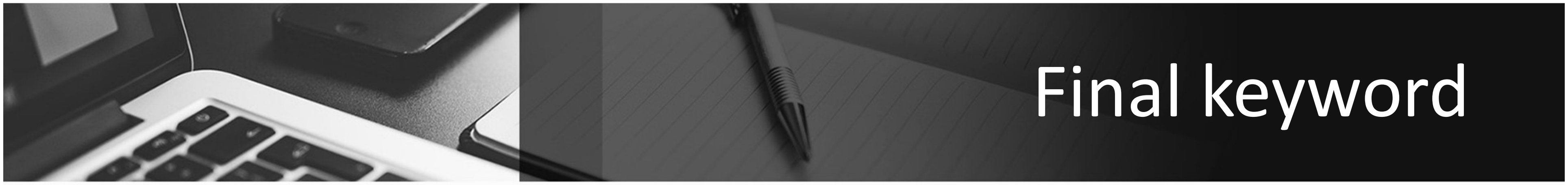


```
class Animal {
  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }}
// Dog inherits Animal
class Dog extends Animal {
  // overriding the eat() method
  @Override
  public void eat() {
    // call method of superclass
    super.eat();
    System.out.println("I eat pedigree");
  }
// new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }
}
```

```
class Cat extends Dog{
  // overriding the eat() method
  @Override
  public void eat() {
    // call method of superclass
    super.eat();
    System.out.println("I eat tuna");
  }

public class Main {
  public static void main(String[] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // call the eat() method
    labrador.eat();
    labrador.bark();
     Cat mia= new Cat();
      mia.eat();//
  }}
```

```
I can eat
I eat pedigree
I can bark
I can eat
I eat pedigree
I eat tuna
```



Final keyword

- **Every** object in Java extends java.lang.Object
- What does it mean for a field to be declared **final**?
  - **Final fields** can't be assigned a new value
  - **Final methods** cannot be overridden
  - **Final classes** cannot be extended
- There is only single inheritance in Java.
  - Subclass can be derived only from one superclass.

final Data (Compile-Time)

- For primitive types (**int**, **float**, etc.), the meaning is “this can’t change value”.

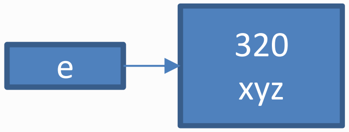
```
class Sedan {
      final int numDoors = 4;
```

- For references, the meaning is “this reference must always refer to the same object”.

```
        final Engine e = new Engine(300);
```

Note: e value cant be changed but we can change the instance fields

```
e.hp=320;
e= new Engine(200);//error
```



final Data (Run-Time)

- Called a “blank final;” the value is filled in during execution.

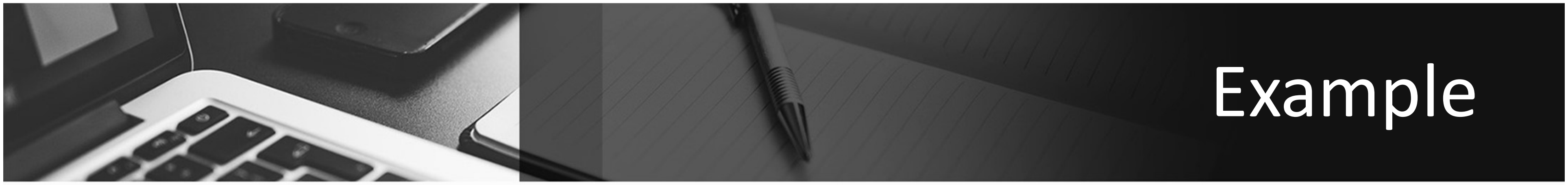
```
class Sedan {
   final int topSpeed; final int numDoors = 4;
   Sedan(int ts) {
      topSpeed = ts;
      // …
   }
}
class DragRace {
   Sedan chevy = new Sedan(120), ford = new Sedan(140);
   //! chevy.topSpeed = 150; //error
```

final Method Arguments

- Same idea:
  - a **final** primitive has a constant value
  - a **final** reference always refers to the same object.
- Note well: a **final** reference does *not* say that the object *referred to* can’t change

final Methods

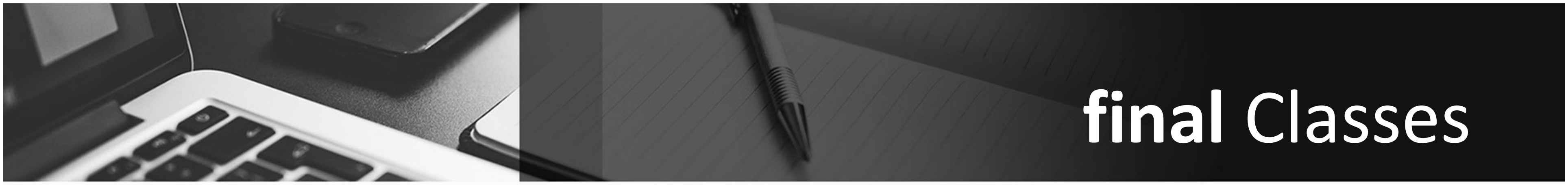
- **final** methods cannot be overridden in subclasses.
- **private** methods are implicitly **final**.



Example

```
public class FinalMethodExample {
  public final void display(){
    System.out.println("Hello welcome to Tutorialspoint");
  }
  public static void main(String args[]){
    new FinalMethodExample().display();
  }
  class Sample extends FinalMethodExample{
    public void display(){
      System.out.println("hi");
    }
  }
}
```

final methods cannot be overridden in subclasses.



final Classes

- These can’t be inherited from (ummm, “subclassed”?.
- All methods are implicitly **final**.

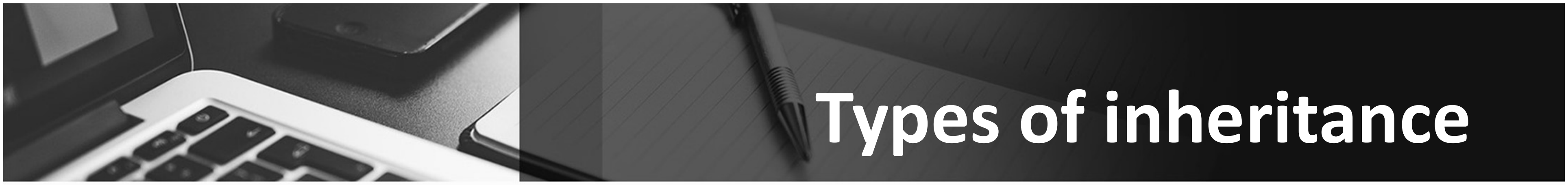
If you try to access a final class, Java will generate an error:

```
final class Vehicle {
  ...
}

class Car extends Vehicle {
  ...
}
```

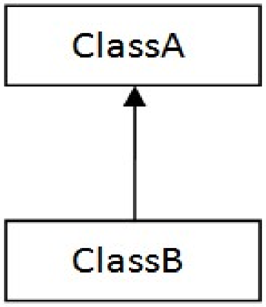
The output will be something like this:

```
Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
           ^
1 error)
```

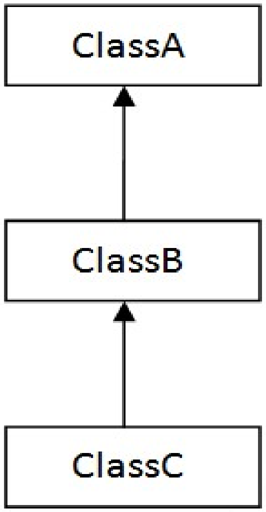


Types of inheritance

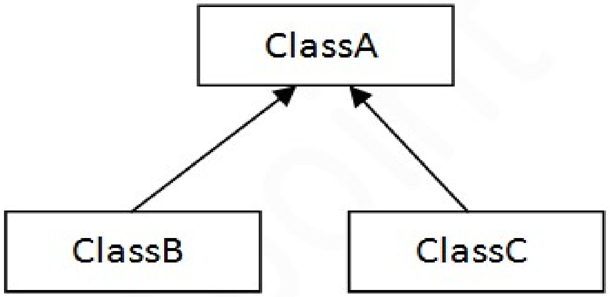
- Single Inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Multiple Inheritance (Not for classes in JAVA)
- Hybrid inheritance



1) Single

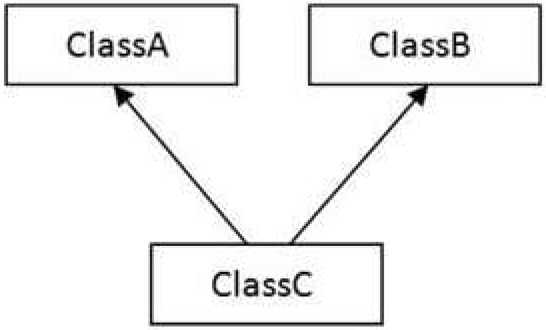


2) Multilevel

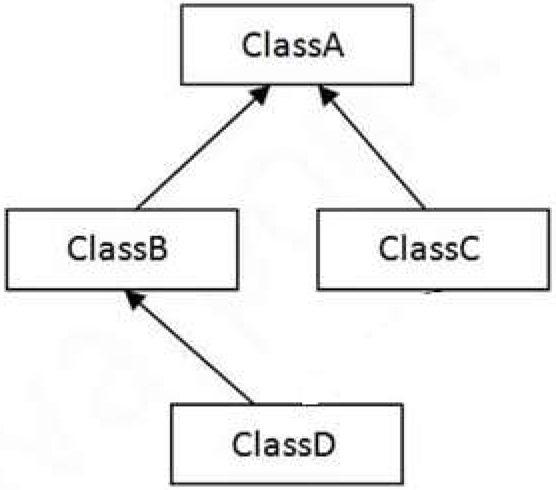


3) Hierarchical

[redacted]



4) Multiple



5) Hybrid

Note: Multiple inheritance is not supported in J
class.

Single Inheritance Example- TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
public class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

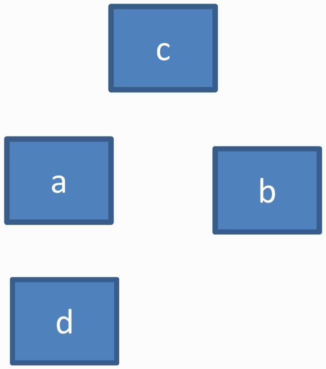
```
meowing...
eating...
```

Hybrid Inheritance Example

```
class C {
  public void disp() {
                System.out.println("C");
  }
}
class A extends C {
  public void disp() {
                System.out.println("A");
  }
}
class B extends C {
  public void disp() {
                System.out.println("B");
  }
}
```

```
class D extends A
{
  public void disp()
  {  super.disp();
                System.out.println("D");
  }
  public static void main(String args[]){

                D obj = new D();
                obj.disp();
  }
}
```



Here, we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.

Class A and B extends class C → Hierarchical inheritance

Class D extends class A → Single inheritance

Why multiple inheritance is not supported in java through classes?

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were
 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
} }
```

Output:
Compile Time Error