OBJECT ORIENTED PROGRAMMING WITH JAVA

Java Static Scope Rule

Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Static Scope Rule in Java

Static scope rule

```
class Box{
    float x = 10.0;
    float y = 20.0;
    float w = 15.0;

    float area(){
        return(2*(x*y + x*w + y*w));
    }
}
```

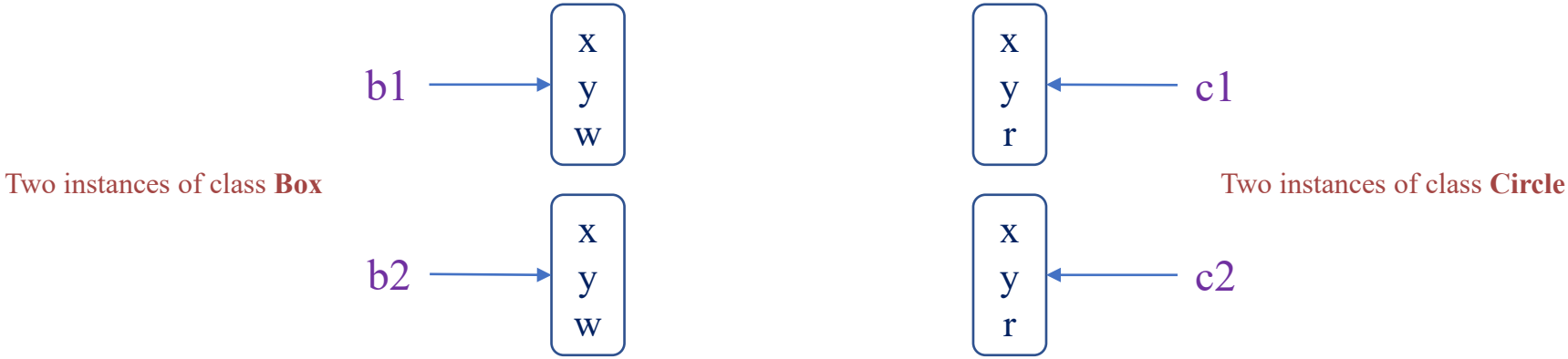
```
class Circle{
   float x = 0.0;
   float y = 0.0;
   float r = 5.0;

   float area(){
       return(((22/7)*r*r));
    }
}
```

```
class GeoClass{
     float x = 50;
     float y = 60;
     public static void main(String args[]){
     Box b = new Box();
     Circle c = new Circle();
     System.out.println("GeoClass Data: x = " + x);
     System.out.println("Box Data: x = " + b.x);
     System.out.println("Box Area: " + b.area);
     System.out.println("Circle Data: x = " + c.x);
     System.out.println("Circle Area: " + c.area);
    }
}
```

Static scope rule : Another example

```
class StaticScope {
     public static void main(String args[]) {
          int x; // known to all code within main
          x = 10;

          if(x == 10) {         // start new scope
               int y = 20;      // known only to this block
               System.out.println("x and y: " + x + " " + y);
               x = y * 2;       // x and y both are known here.
           }
           y = 100;           // Error! y is now no more known here
           System.out.println("x is " + x); // x is still known here.
     }
}
```

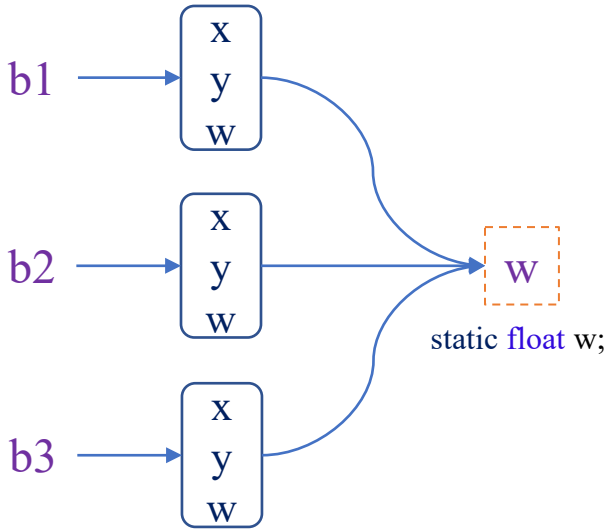
Instance variable versus Class variable



- In class **`Box`** and class **`Circle`**, we declared three variables (x, y, w) and (x, y, r), respectively.
- Such a variable is called **instance variable**.
- They are so called because each instance of the class, say, **`Circle`**, has its own copy.

Instance variable versus Class variable

- Java does not allow global variables.
- Every variable in Java must be declared inside a class.
- The keyword **`static`** is used to make a variable just like global variable.
- A variable declared with **`static`** keyword is called **class variable.**
- It acts like a global variable, that is, there is only one copy of the variable associated with the class.
  That is, one copy of the variable regardless of the number of instances of the class.



Three instances of class **Box**

Static variable : An example

```
public class Circle{
   static int circlecount = 0; // class variable
   public double x,y,r; // instance variables
   public Circle(double x, double y, double r){
      this.x = x; this.y = y; this.r = r;
      circlecount++;
   }
   public Circle(double r){
      this(0.0,0.0,r);
      circlecount++;
   }
   public Circle(Circle c){
      this(c.x,c.y,c.r);
      circlecount++;
   }
   public Circle(){
      this(0.0,0.0,0.1);
       circlecount++;
   }
```

```
   public double circumference(){
       return (2*3.14159*r);
    }
    public double area(){
        return(3.14159*r*r);
    }
    public static void main(String args[ ]){
       Circle c1 = new Circle();
       Circle c2 = new Circle(5.0);
       Circle c3 = new Circle(c1);
       System.out.println("c1#\" + c1.circlecount + "c2#\" +
                          c2.circlecount + "c3#\" + c3.circlecount);
    }
}
```

Declaring static method : An example

```
// A class method and instance method
public class Circle{
     public double x,y,r;
     // All constructors are here.
     // An instance method. Return the bigger of two circles.
     public Circle bigger(Circle c){
        if(c.r>r) return c;
        else return this;
     }
     // A class method: Return the bigger of two classes.
     public static Circle bigger (Circle a, Circle b) {
         if (a.r > b.r) return a;
         else return b:
     }

public static void main(String args[]){
          Circle a = new Circle (2.0);
          Circle b = new Circle (3.0);
          Circle c = a.bigger (b);         // Call of the instance method
          Circle d = Circle.bigger (a,b); // Call of the class method
    }
}
```

Nested Class in Java

Nested class in Java

- In Java, a class can be defined inside a class. Let us look at the following example.

```
class Circle{
    static double x,y,r;
    Circle(double r){
        this.r = r;
    }
// Following is the nested class
   public static class Point{
      double x, y;
      void display(){
           System.out.println("(x,y): (" + this.x + "," + this.y + ")");
      }
      Point(double a, double b){
          this.x = a;
          this.y = b;
       }
  }
```

Continued to the next slide……

Nested class in Java

- In Java, a class can be defined inside a class. Let us look at the following example.

```
        public boolean isInside(Point p){
           double dx = p.x - x;
           double dy = p.y - y;
           double distance = Math.sqrt((dx*dx)+(dy*dy));
           if(distance < r) return true;
           else return false;
        }
        public static void main(String args[]){
                 Circle a = new Circle (2.0);
                 Point pa = new Point (1.0,2.0);
                 pa.display();
                 System.out.println("Is the points (1,2) inside the circle with radius 2 :"+a.isInside(pa));
                 Circle b = new Circle (1.0);
                 Point pb = new Point (3.0,3.0);
                 System.out.println("Is the point (3,3) inside the circle with radius 1 :"+b.isInside(pb));
    }
}
```

Recursive Programs in Java

Recursion in Java : Calculation of *n*!

- Factorial calculation of *n*, an integer value is defines as follows.

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots 3 \times 2 \times 1$$

$$= 1 \times 2 \times 3 \times \ldots (n\text{-}2) \times (n\text{-}1) \times n$$

Note: $0! = 1$

```
public class SimpleFactorial{
    int n;

    public static void main(String[] args) {
       int facto = 1;
       n = Integer.parseInt(args[0]);
       if ((n == 0) || (n == 1) {
          System.out.println("Factorial of " + n + ": " + facto);
          return;
        }
        for(int i = 1; i <= n, i++)
            facto = facto * i;
        System.out.println("Factorial of " + n + ": " + facto);
        return;
   }
}
```

Recursion in Java : Calculation of *n*!

- Recursive definition of $n!$

$$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots 3 \times 2 \times 1$$

$$= n \times (n\text{-}1)! \qquad \text{with } 0! = 1$$

```
public class RecursiveFactorial{
  int n;
  int factorial(int n) {
      if (n == 0)
          return(1);
      else
          return(n*factorial(n-1));
   }

   public static void main(String[] args) {
       Recursive x = new RecursiveFactorial();
       x.n = Integer.parseInt(args[0]);
       System.out.println("Factorial of "+ n + ": " + x.factorial(x.n));
   }
}
```

Recursion in Java : Fibonacci sequence

- Following is a series of numbers called the Fibonacci sequence.

0 1 1 2 3 5 8 13 21

- Recursive definition of n-th Fibonacci number.

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_0 = F_1 = 1$$

```
public class SimpleFibonacci{
   int n;

   public static void main(String[] args) {
       n = Integer.parseInt(args[0]);
       int fibo1 = 0, fibo2 = 1;
       System.out.print(fibo1 + " " + fibo2);
       while (n > 1) {
         fibo = fibo1 + fibo2;
         System.out.print(" " + fibo);
         fibo1 = fibo2; fibo2 = fibo; n++;
       }
   }
}
```

Recursion in Java : Fibonacci sequence

- Recursive definition of n-th Fibonacci number.

$$F_n = F_{n-1} + F_{n-2} \quad \text{with } F_0 = F_1 = 1$$

```
class RecursiveFibonacci {
    int n;
    int fibonacci(int n){
       if (n == 0)
           return 0;
       else if (n == 1)
               return 1;
            else
               return(fibonacci(n-1) + fibonacci(n-2));
     }
     public static void main(String args[]){
         Fibonacci x = new RecursiveFibonacci();
         x.n = Integer.parseInt(args[0]);
         for(int i = 0; i <= x.n; i++){
               System.out.println(x.fibonacci(i));
     }
   }
}
```

Recursion in Java : GCD calculation

Greatest common divisor (GCD) calculation is as follows.

GCD(35, 15) = 5
GCD(10, 50) = 10
GCD(11, 0) = 11
GCD(8, 8) = 8
GCD(1, 13) = 1

For any two integers *m* and *n* (such that $m < n$), the *GCD (m,n)* calculation can be defined recursively is as follows.

GCD(m, n) = GCD(n,m) if m>n;
GCD(m, n) = m, if m = 0;
GCD(m, n) = 1, if m = 1;
GCD(m, n) = m, if m = n;
GCD(m, n) = GCD(m, n%m);

Recursion in Java : GCD calculation

For any two integers *m* and *n* (such that $m < n$), the *GCD (m,n)* calculation can be defined recursively is as follows.

```
GCD(m, n) = GCD(n,m) if m>n;
GCD(m, n) = n, if m = 0;
GCD(m, n) = 1, if m = 1;
GCD(m, n) = m, If m = n;
GCD(m, n) = GCD(m, n%m);
```

```java
public class RecursiveGCD {
    int m, n;
    int gcd(int m, int n){
        if(m>n) return gcd(n,m);
        if(m==n)return m;
        if(m==0)return n;
        if(m==1)return 1;
        return gcd(m,n%m);
    }

    public static void main(String[] args) {
        RecursiveGCD g = new RecursiveGCD();
        g.m = Integer.ParseInt(args[0]);
        g.n = Integer.ParseInt(args[1]);
        System.out.printf("GCD of %d and %d is %d.", g.m, g.n, g.gcd(g.m, g.n));
    }
}
```

Recursion in Java

- What this program does for you?.

```
public class RecursionExample{
  static int count = 0;
  static void p(){
      count++;
      if(count <= 5){
          System.out.println("Hello " + count);
          p();
      }
  }

   public static void main(String[] args) {
        p();
    }
}
```

```
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

Questions to think…

- How information can be hided in Java ?
- How Java manages a large program to be developed?

Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Inheritance in Java

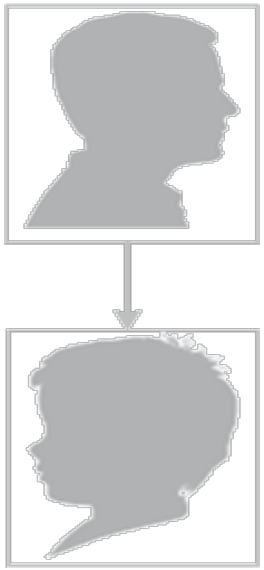
Debasis Samanta

Department of Computer Science & Engineering

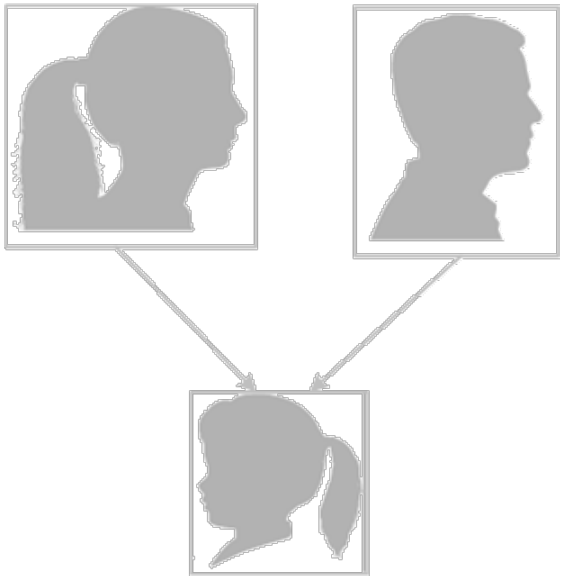
Indian Institute of Technology Kharagpur

Inheritance Concept

Concept of inheritance

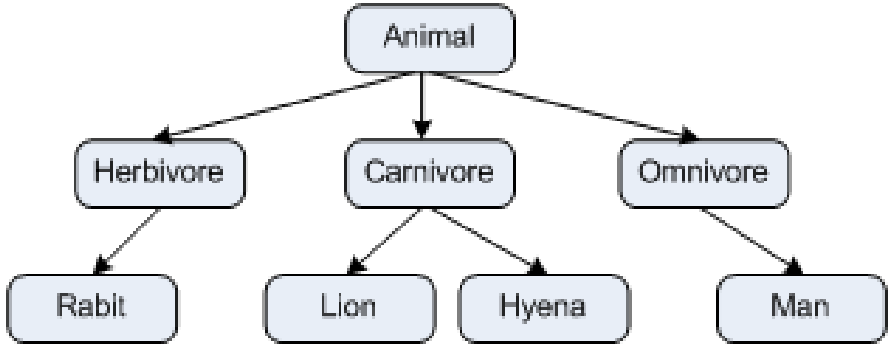


Single inheritance

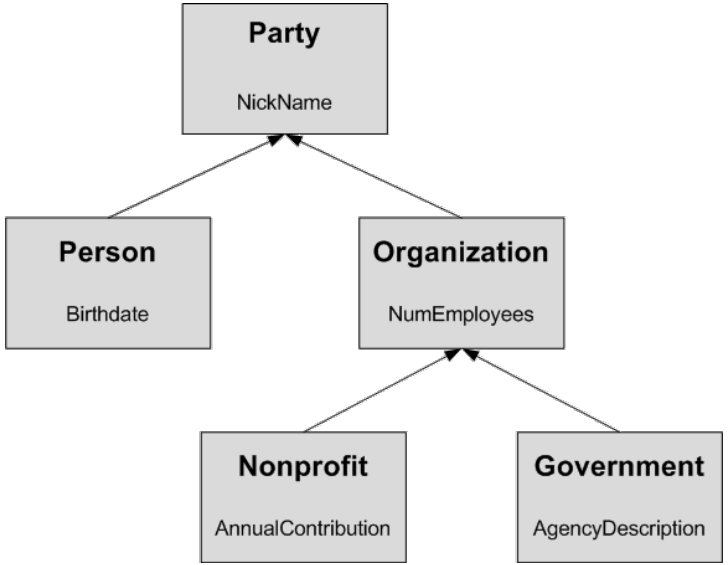


Multiple inheritance

Concept of inheritance



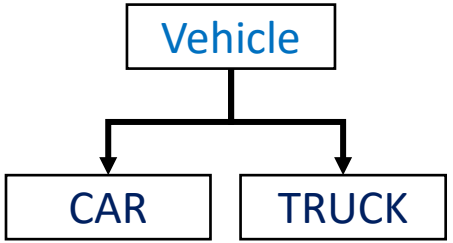
Single multi-level inheritance



Class hierarchy

Inheritance in Java

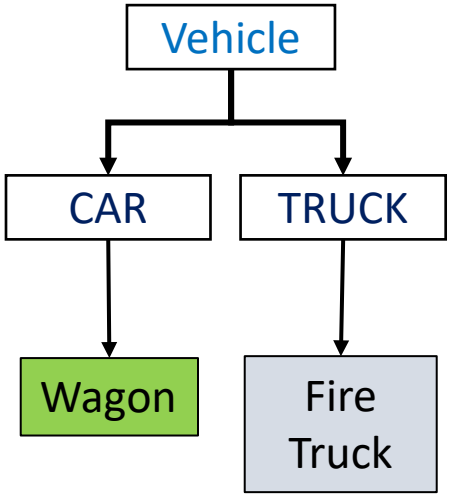
- Inheritance is one of the cornerstone of object-oriented programming because it allows the creation of hierarchical classification.
- Using inheritance, one can create a general class that include some common set of items.
- This class then can be used to create more specific classes which has all the items from the base class, in addition to some items of its own.



Terms used in inheritance

- **Superclass:** A class that is inherited is called a superclass.
- **Subclass:** The class that does inheriting is called a subclass.
  - A subclass is a specialized version of a superclass
  - It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements (i.e., variables and methods)

- **Reusability:** It is a mechanism which facilitates you to reuse the data and methods of the existing class when one create a new class.
  - One can use the same data and methods already defined in the previous class.

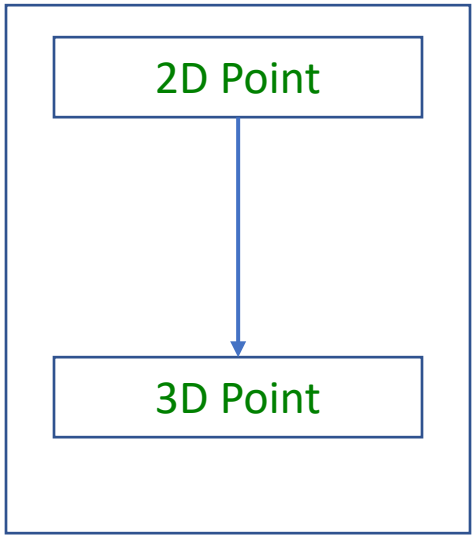


Inheritance syntax

- The **`extends`** keyword is used to define a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

```
class <Subclass-name> extends <Superclass-name>  {
     //data and methods in this sub-class
}
```

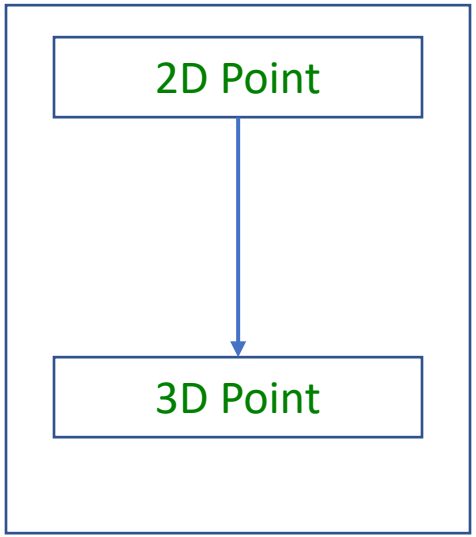
Example of a simple Inheritance




```
class Point2D{
            int x;
            int y;
            void display(){
                        System.out.println ("x="+x+"y="+y);
            }
}
```

```
class Point3D extends Point3D{
            int z;
            void display(){
                        System.out.println("x="+x+"y="+y+"z="+z);
            }
}
```

Example of a simple Inheritance

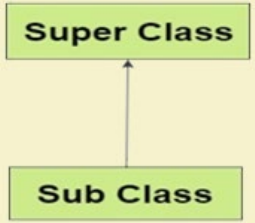



```
class simpleSingleInheritance{
    public static void main(String arge[]){
        Point2D new P1();
        Point3D new P2();
        P1.x = 10;
        P1.y = 20;
        System.out.println("Point2D P1 is" + P1.display());
        // Initializing Point3D
        P2.x = 5;
        P2.y = 6;
        P3.z = 15;
        System.out.println("Point3D P2 is" + P2.display());
    }
}
```

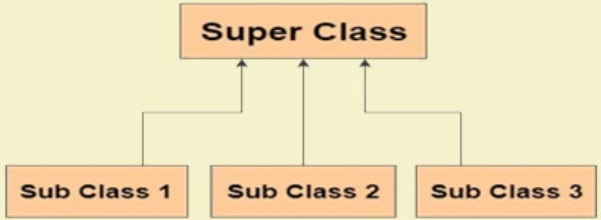
Types of Inheritances

Inheritance types

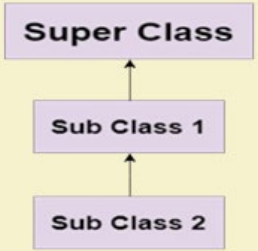
Single inheritance



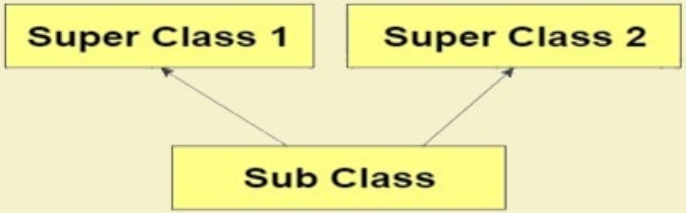
Multiple single inheritance



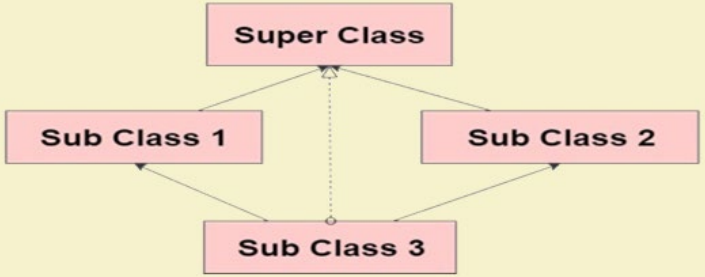
Multilevel single inheritance



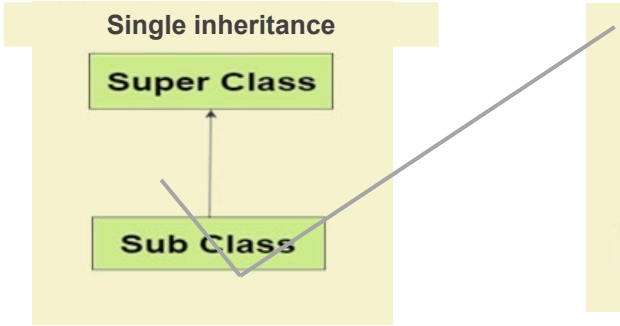
Multiple inheritance

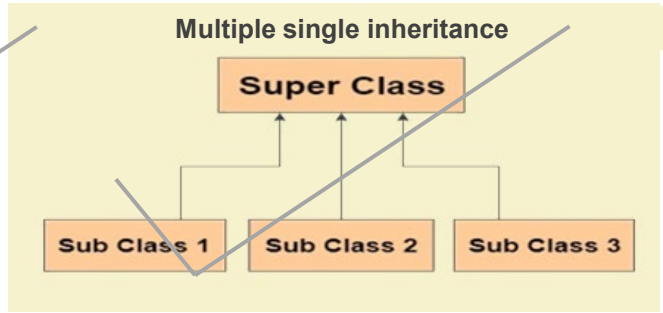


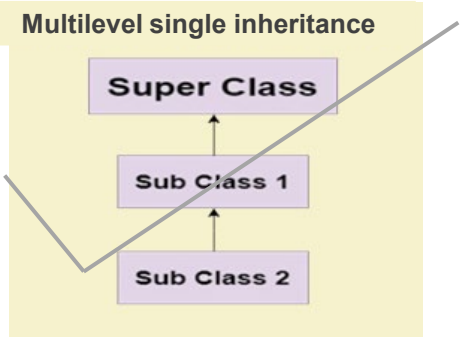
Hybrid inheritance

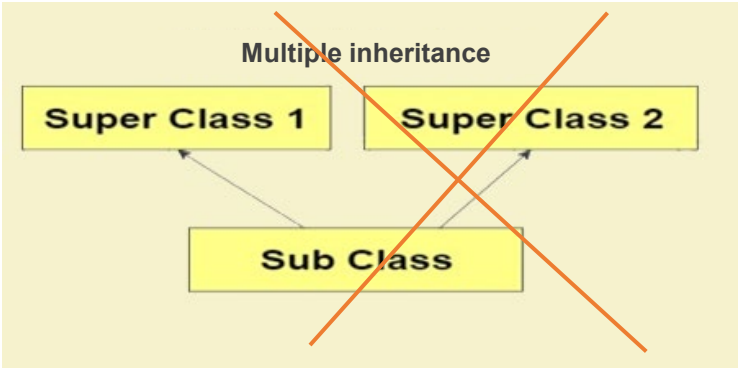


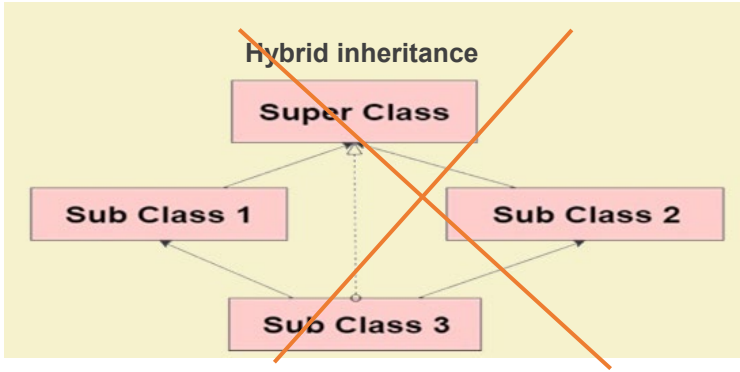
Inheritance types



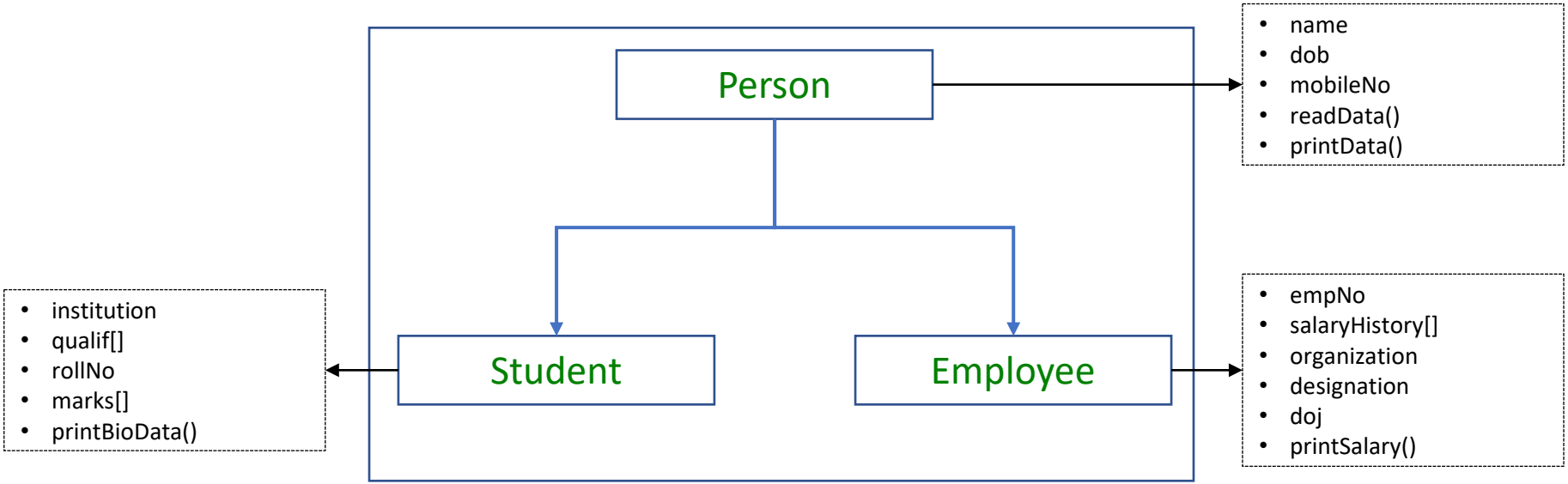








Single inheritance : An example



Single Inheritance in Java

Single Inheritance : Person class

```
class Person{
        String name;
        Date dob;
        int mobileNo;
        void readData(String n, Date d, int m){
              name = n;
              dob = d;
              mobileNo = m;
        }
        void printData(){
              System.out.println("Name : "+ name);
              dob.printDate();
              System.out.println("Mobile : "+ mobileNo);
        }
}
```

Single inheritance : Student class

```
class Person{
   String name;
   Date dob;
   int mobileNo;
   void readData(String n, Date d, int m){
      name = n;
      dob = d;
      mobileNo = m;
   }
   void printData(){
        System.out.println("Name : "+ name);
        dob.printDate();
        System.out.println("Mobile : "+ mobileNo);
   }
}
```

```
class Student extends Person{
        String institution;
        int[] qualif = new int[20];
        int rollNo;
        int[] marks = new int[5];

        void printBioData(){
            printData();
            System.out.println("Institution : "+ institution);
            System.out.println("Roll : "+ rollNo);
            for(int q=0; q<qualif.length;q++){
               System.out.println("Marks "+q+": "+ qualif[q]);
            }
            for(int m=0; m<marks.length;m++){
               System.out.print("Result "+m+": "+marks[m]);
            }
        }
}
```

Single Inheritance – employee

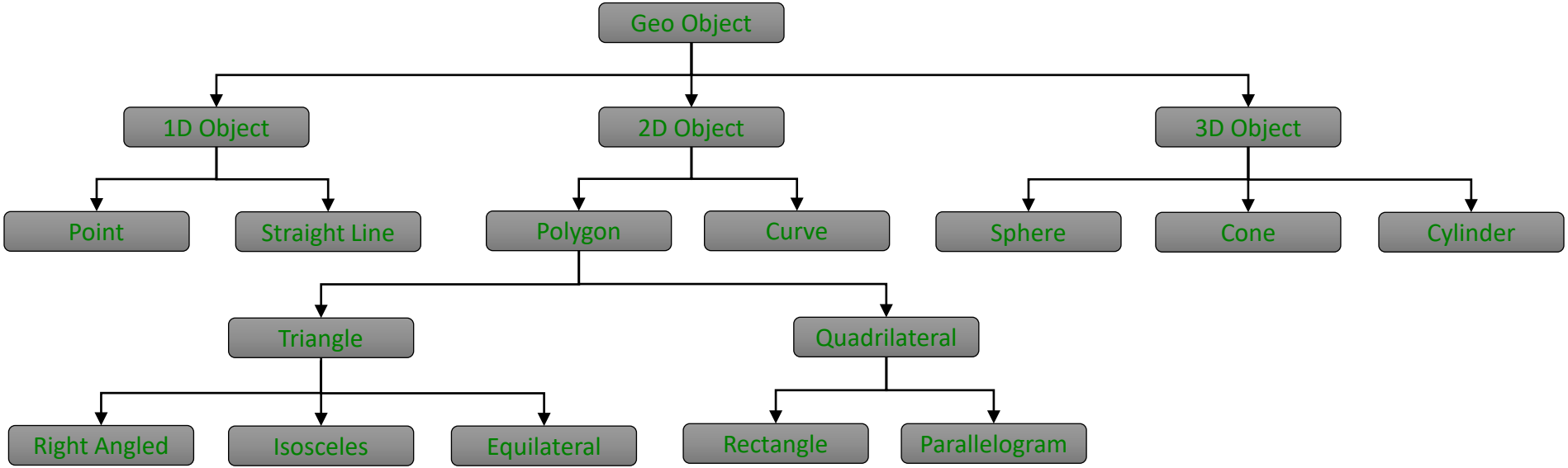
```
class Person{
   String name;
   Date dob;
   int mobileNo;
   void readData(String n, Date d, int m){
      name = n;
      dob = d;
      mobileNo = m;
   }
   void printData(){
        System.out.println("Name : "+ name);
        dob.printDate();
        System.out.println("Mobile : "+ mobileNo);
   }
}
```

```
class Employee extends Person{
        int empNo;
        int[] salaryHistory = new int[12];
        String organization;
        String designation;
        Date doj;
        void printSalary(){
            for(int s=0; s<salaryHistory.length;s++){
               System.out.println("Salary "+s+": "+salaryHistory[s]);
            }
        }
}
```

Single Inheritance : An example

```
class inheritanceDemo1{
      public static void main(String args[]){
          Person p = new Person();
          //Code with the objects p…
          Student s = new Student [100];
          //Code with the objects s…
          Employee e = new Employee[50];
          //Code with the objects e…

      }
}
```

Multilevel inheritance : An example



Method Overriding

Method overriding concept

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

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

Method overriding : An example

```
class Point2D{
        int x;
        int y;
        Point2D(int a, int b){
            x = a;
            y = b;
        }
        void display(){
            System.out.println("x = "+x+"y = "+y);
        }
}
```

```
class Point3D extends Point3D{
        int z;
        Point3D(int c){
            z = c;
        }
        void display(){
            System.out.println("x="+x+"y="+y+"z="+z);
        }
}
```

```
class MethodOverridingTest{
    public static void main(String args[]){
        Point2D p = new Point2D(3.0, -4.0);
        p.display(); // Refers to the method in Point2D

        Point3D q = new Point3D(0.0);
        q.display(); // Refers to the method in Point3D

        Point2D x =(Point2D) q; // Cast q to an instance of class Point2D
        x.display();
        }
}
```

Note

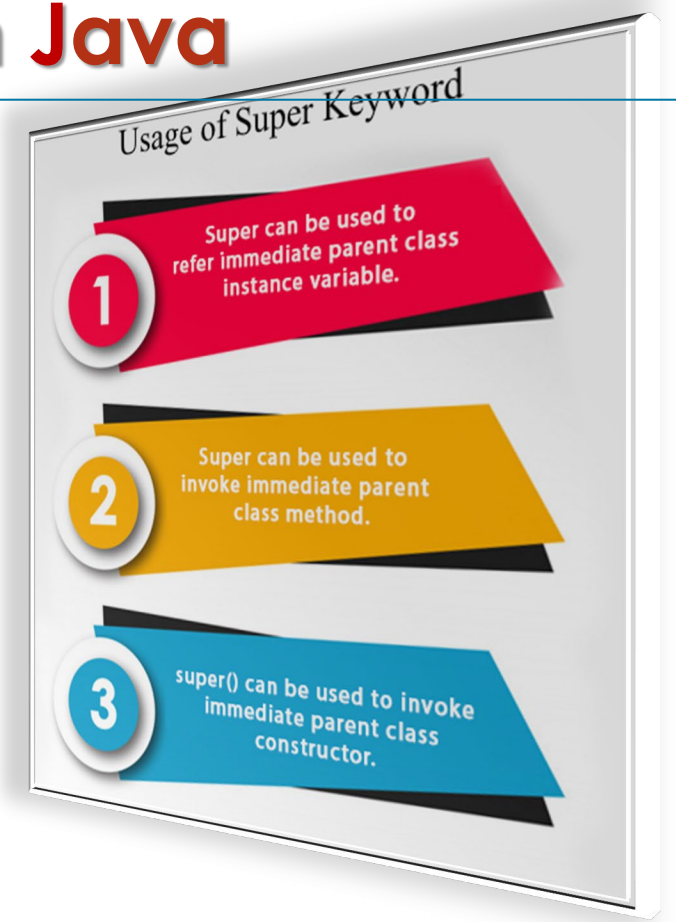
- A sub class object can reference a super class variable or method if it is not overridden.
- A super class object cannot reference a variable or method which is explicit to the sub class object.

`super` Keyword

super Keyword concept in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class members.

Whenever you create an instance of a sub class, an instance of its parent class is created implicitly, which is referred by **super** keyword.



super : Referring parent class instance variable

```
class Animal{
   String color="white";
}
class Dog extends Animal{
   String color = "black";
   void printColor(){
      System.out.println(color);
      System.out.println(super.color);
   }
}
class TestSuper1{
   public static void main(String args[]){
      Dog d = new Dog();
      d.printColor();
   }
}
```

```
black
white
```

Animal and **Dog** both classes have a common property color. If you print the color property, it will print the color of the current class by default. To access the parent property, you should use `super` keyword.

super : Invoking parent class method

```
class Animal{
   void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
   void eat(){System.out.println("eating bread...");}
   void bark(){System.out.println("barking...");}
   void work(){
           super.eat();
           bark();
           eat();}
   }
class TestSuper2{
   public static void main(String args[]){
      Dog d = new Dog();
      d.work();
   }
}
```

`Animal` and `Dog` both the classes have `eat()` method. If you call `eat()` method from `Dog` class, it will call the `eat()` method of `Dog` class by default because priority is given to local.

To call the parent class method, you need to use `super` keyword.

```
eating...
barking...
Eating bread...
```

super : Invoking parent class constructor

```
class Animal{
  Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
       super();
       System.out.println("dog is created");
    }
}

class TestSuper3{
   public static void main(String args[]){
      Dog d = new Dog();
   }
}
```

```
animal is created
dog is created
```

The super keyword can also be used to invoke the overloaded parent class constructor, if arguments are there, then they should be specified accordingly.

super : Invoking parent class constructor

```
class Point2D{
   double x, y;
   Point2D(){x = 0.0; y = 0.0} //Default initialization
   Point2D(double x, double y){thix.x = x; this.y = y;}
}
class Point3D extends Point2D{
   double z;
   Point3D(){super(); z = 0.0} //Default initialization
   Point3D(double x, double y, double z){
   super(x, y);
   this.z = z; }
}
class TestSuper4{
   public static void main(String args[]){
       Point3D p = new Point3D(2.0, 3.0, 4.0);
   }
}
```

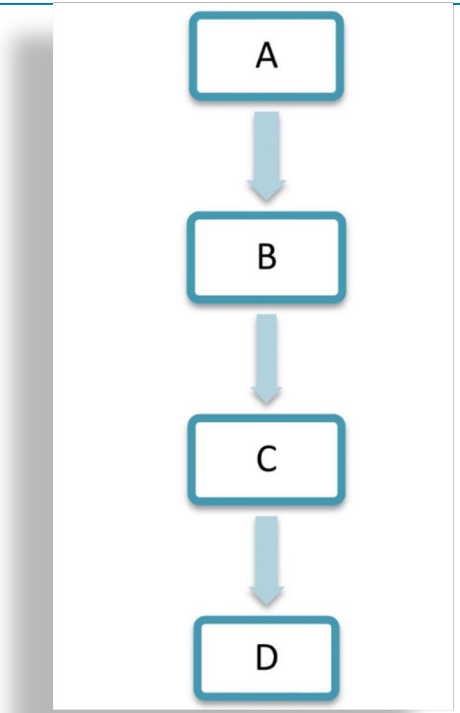
If there is a number of overloading constructors in the super class, then you have to define the super constructors matching with each constructor.

Dynamic Method Dispatch

Dynamic method dispatch concept

Dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. Also, it is called Runtime polymorphism.

In this process, an overridden method is called through the reference variable of a super class. The determination of the method to be called is based on the object being referred to by the reference variable.



super - refer parent class instance variable

```
class Bike{
    void run(){System.out.println("running");}
}
class Splendor extends Bike{
   void run(){System.out.println("running safely with 60km");
   }

   public static void main(String args[]){
       Splendor b1 = new Splendor();
       b1.run();
       Bike b2 = new Bike();
       b2.run();
       Bike b3 = new Splendor();//Up casting
       b3.run();
   }
}
```

For the `b3` object, we are calling the `run()` method by the reference variable of super class. Since it refers to the sub class object and sub class method overrides the super class method, the sub class method is invoked at runtime.

```
running safely with 60km
Running
running safely with 60km
```

Dynamic binding in Java: Example

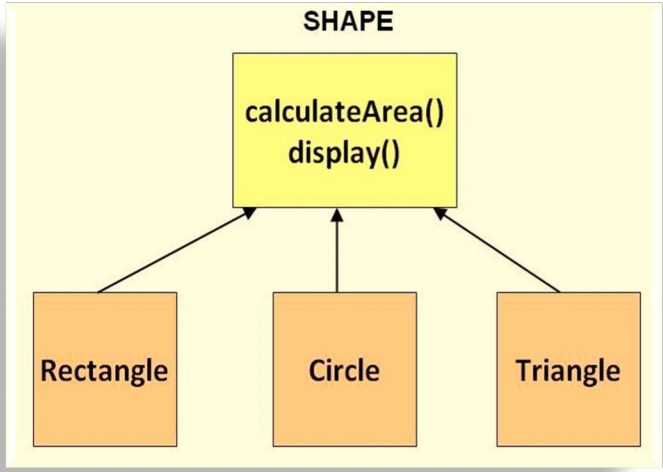
```
class A {
        void callMe (  ) {
            System.out.println ( "I am from A ") ;
        }
}

class B  extends A {
        void callMe (  ) {
           System.out.println ( "I am from B ");
        }
}

class Who {
        public void static main (String args [  ] ) {
           A  a = new B ( ) ;
           a.callMe();
           B b = new B( );
           b.callMe();
        }
}
```

I am from B
I am from B

Abstract class in Java

Abstract concept

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.
- A class which is declared with the **`abstract`** keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (i.e., method with the body only without its definition).



Abstract concept

Points to remember

- An abstract class must be declared with an **`abstract`** keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the sub class not to change the body of the method.

Abstract class in Java : Example

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

 class Honda extends Bike{
     void run(){
         System.out.println("Running safely");
     }

    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
   }
 }
```

Running safely

Here, `Bike` is an abstract class that contains only one abstract method `run()`.

Its implementation is provided by the `Honda` class.

Note:
An abstract method should be defined in it sub class.

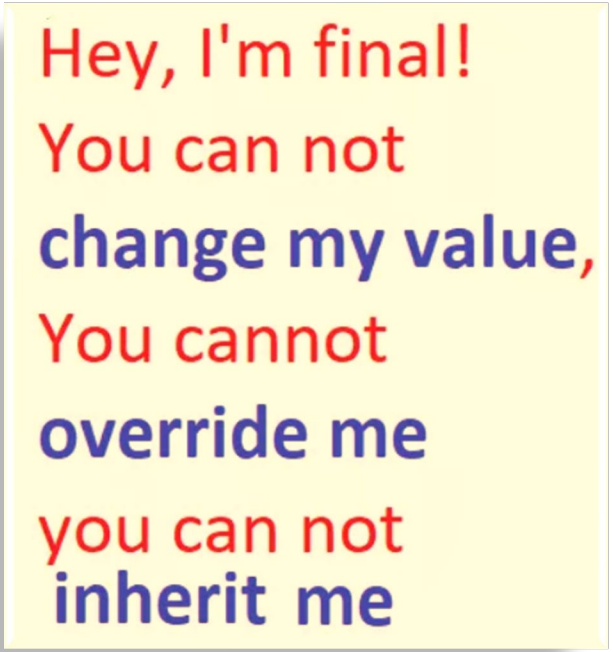
`final` Keyword in Java

`final` keyword concept

The **final** keyword in Java is used to restrict the access of an item from its super class to a sub class. The Java `final` keyword can be used in many context.

- Variable : a variable cannot be accessed in sub class
- Method : a method cannot called from a sub class object
- Class : a class cannot be sub classed.

Note:
If you make any class as `final`, you cannot extend it.



Final class in inheritance : An Example

```
final class Bike{}

class Honda1 extends Bike{
  void run(){
    System.out.println("Running safely with 100kmph");
  }

   public static void main(String args[]){
      Honda1 honda = new Honda1();
      honda.run();
   }
}
```

Extending a class which is declared as **`final`** will cause compile time error.

Question to think…

- Can we inherit a class from other class which is defined in other package?
- How information access can be restricted in a class?


Thank You

OBJECT ORIENTED PROGRAMMING WITH JAVA

Information Hiding in Java

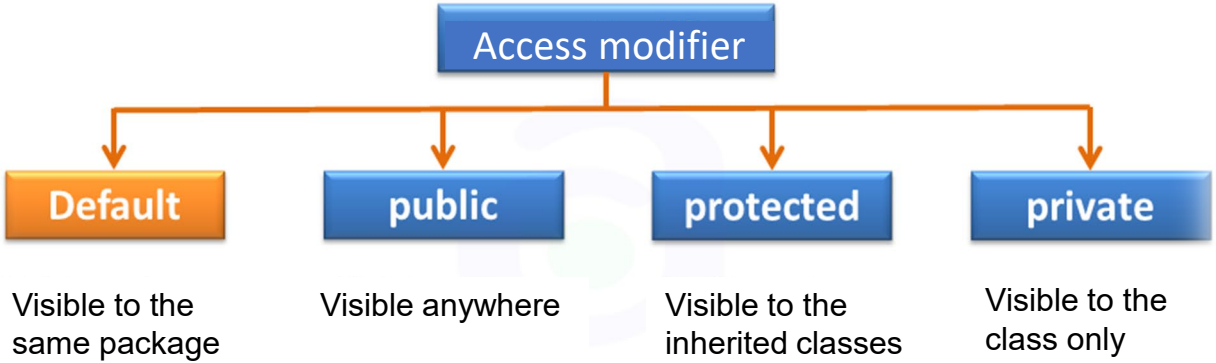
Debasis Samanta
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Access Modifiers in Java

Concept of access modifiers

The **access modifiers** in Java specify accessibility (scope) of a data member, method, constructor or class.

Type of access modifiers :



Access levels of modifiers

Access levels / Modifier	Class	Package	Sub class	Everywhere
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Default access modifier

If you don't use any modifier, it is treated as **default** by default.

The default modifier is accessible only within a package.

Default access modifier : An Example

```
//Save this program as A.java in a sub-directory "pack1"

class A {
   void msg(){System.out.println("Hi! I am in Class A");
  }
}
```

```
/* Save this program as B.java in another sub-directory
"pack2"  */

   class B{
      public static void main(String args[]){
           A obj = new A();    //Compile Time Error
           obj.msg();          //Compile Time Error
      }
   }
```

Here, two classes are with default access modifier. If they reside in two different directories, then the `class A` is not accessible to `class B` and vice-versa.

However, if they reside in the same directory, then there will be no error!

Default access modifier : Another example

```
    //Save this program as A.java
package pack1;        // It is a sub-directory "pack1"
class A {
  void msg(){System.out.println("Hi! I am in Class A");}
}
```

```
     //Save this program as B.java
package pack2;  //It is in a sub-directory "pack2"
import pack1.*;  /* Import all class files in pack1 */

class B{
    public static void main(String args[]){
         A obj = new A();    //Compile Time Error
         obj.msg();          //Compile Time Error
   }
}
```

Suppose, there are two packages, say `pack 1` and `pack2`.

Here, `class A` and `class B` are declared as `default` and we are accessing the `class A` from outside its package, since `class A` is `default`, so it cannot be accessed from any outside package.

Default access modifier : An Example

```
//Save this program as A.java in a directory, say temp.

class A {
   void msg(){System.out.println("Hi! I am in Class A");
  }
}
```

```
/* Save this program as B.java in the same directory */

  class B{
     public static void main(String args[]){
          A obj = new A();    //Okay. It is accessible.
          obj.msg();          //It is also accessible.
     }
 }
```

Here, two classes are with default access modifier, and thy are residing in the same directory (or may be in the same file). Hence, in this case, the `class A` is accessible to `class B` and vice-versa.

Public access modifier

The **public** access modifier is accessible everywhere.

It has the widest scope among all other modifiers.

public access modifier : An Example

```
//Save as A.java in a sub-directory say pack1
 package pack1;
 public class A{
      public void msg(){
         System.out.println("Class A: Hello Java!");
      }
 }
```

```
//Save as B.java  in another sub-directory say pack2
 package  pack2;
 import pack1.*;

 class B{
      public static void main(String args[]){
           A obj = new A();
           obj.msg();
        }
 }
```

The `class A` in package `pack1` is `public`, so this class can be accessed from any class outside to this package, for example, in this case, from `class B` in `pack2` .

```
Class A: Hello Java!
```

public access modifier : An example

```
public class A{
   public int data = 40;
   public void msg(){
         System.out.println("Class A: Hello Java!");
   }
}

public class B{
   public static void main(String args[]){
        A obj = new A();     //OK : Class A is public
        System.out.println(obj.data);
                             //OK : data is public
        obj.msg();           //OK: msg is public
   }
}
```

We have created two classes `class A` and `class B`. The `class A` contains public data member and public method and are accessible to `class B`.

Note:
It does not matter whether the `class A` and `class B` belong to the same directory or in the same program file.

Public access modifier : Another example

```
//Save this program as A.java
package pack1;        // It is a sub-directory "pack1"

public class A {
  void msg(){System.out.println("Hi! I am in Class A");}
}
```

```
//Save this program as B.java
package pack2;  // It is another sub-directory "pack2"
import pack1.*;  /* Import all classes in pack1 here

class B{
    public static void main(String args[]){
         A obj = new A();    //Okay program!
         obj.msg();          //This is now public
   }
}
```

When a class is `public`, all its member with default access specifier are also `public`.

Private access modifier

The **private** access modifier is accessible only within the class.

Private access modifier : An example

```
public class A{
   private int data = 40;
   public void msg(){
         System.out.println("Class A: Hello Java!");
   }
}

 public class B{
     public static void main(String args[]){
         A obj = new A();     //OK : Class A is public
         System.out.println(obj.data);
            //Compile Time Error : data is private
         obj.msg(); //OK : msg is public
     }
}
```

Private access modifier : An example

```
private class A{
   int data = 40;
   void msg(){
         System.out.println("Class A: Hello Java!");
   }
}

 public class B{
     public static void main(String args[]){
         A obj = new A();     //Error : Class A is public
         System.out.println(obj.data);
            //Compile Time Error : data is private
         obj.msg(); //Error : msg is private
     }
}
```

When a class is `private`, all its member with default access specifier are also `private`.

How, if a member in a public class is declared as public or protected?

Think about this…

```
public class A{
   private int data = 40;
   public void msg(){
         System.out.println("Hello Java!" + data);
   }
}

 public class B{
    public static void main(String args[]){
       A obj = new A();     //OK : Class A is public
       System.out.println(obj.data);//Compile Time Error : data is private
       obj.msg(); //Calls msg() method of class A, which in turns private data : Error!
    }
 }
```

private constructor : An example

```
public class A{
   private A(){
          //private constructor
   }
   void msg(){
         System.out.println("Class A: Hello Java!");
   }
}

public class Simple{
   public static void main(String args[]){
      A obj = new A();    //Compile Time Error!
   }
}
```

If you make any class constructor **private**, you can not create an instance of that class from outside the class.

Protected access modifier

The **`protected`** access modifier is accessible within a package or from outside a package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Protected access modifier : An Example

```
public class A{
    protected int i = 555;
    void msg(){
       System.out.println("Class A: Hello Java!" + i);
    }
}
```

```
   class B {
       public static void main(String args[]){
            A obj = new A();
            obj.msg(); // Error: Compilation error
       }
    }
```

Here, the protected data `i` is accessible to any methods in the same class. Alos, it is accessible to any of its sub class.

Here, `class A` is accessible to `class B` as it is declared `public`; however, any method in the `class B` (even they are in the same file or package) cannot access `protected` data of `class A` directly or indirectly.

Protected access modifier : An Example

```
public class A{
     public int i = 555;
     protected  void msg(){
          System.out.println("Hello Java! + i");
     }
}
```

```
class B extends A{
     public static void main(String args[]){
          B obj = new B();
          obj.msg();
     }
}
```

```
Hello Java! 555
```

Here, the `msg()` method of the `class A` is declared as `protected`, and hence, it can be accessed from outside the class only through inheritance.

What will happen if `i` is made private in `class A`?

Protected access modifier : Another Example

```
//Save as A.java  in a sub directory pack1
package pack1;

public class A{
    protected void msg(){
          System.out.println("Class A: Hello Java!");
    }
}
```

```
//Save as B.java  in another sub-directory pack2
package pack2;
import pack1.*;

class B extends A{
     public static void main(String args[]){
          B obj = new B();
          obj.msg();
     }
}
```

```
Hello Java!
```

We have created the two packages `pack1` and `pack2`.

The `class A` of `pack1` package is `public`, so can be accessed from outside the package. The method `msg()` of the `class A` is declared as `protected`, so it can be accessed from outside the class through inheritance.

Java access modifiers with method overriding

```
public class A{
   protected void msg(){
      System.out.println("Class A: Hello Java!");
   }
}

public class Simple extends A{
   void msg(){
       System.out.println("Class B: Welcome!");
                    //Compile Time Error
   }

   public static void main(String args[]){
       Simple obj = new Simple();
       obj.msg();
   }
}
```

If you are **overriding** any method, overridden method (i.e. declared in sub class) must not be more restrictive.

The **default** modifier is more restrictive than **protected**. That is why, there is compile time error.

What will happen if the `msg()` in class `Simple` is declared as `public` or `protected`?

Questions to think…

- How a package can be built ?
- Is it possible that two classes having the same name but in two different packages are to be used in another class outside the packages?

Thank You