OBJECT ORIENTED PROGRAMING - INHERITANCE

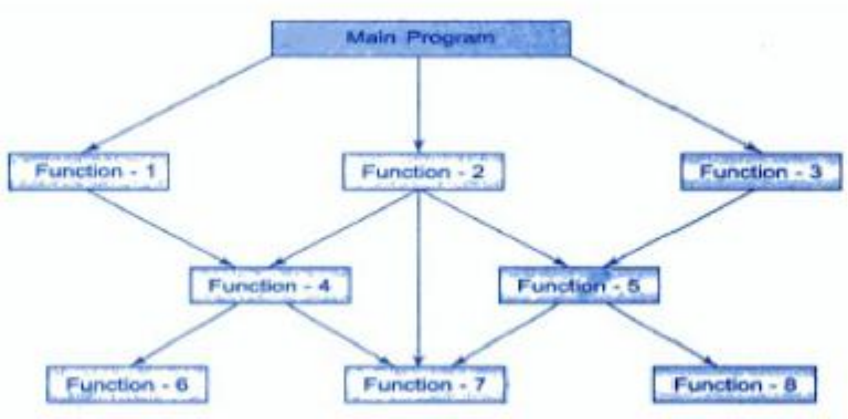
Tam Hung Phan

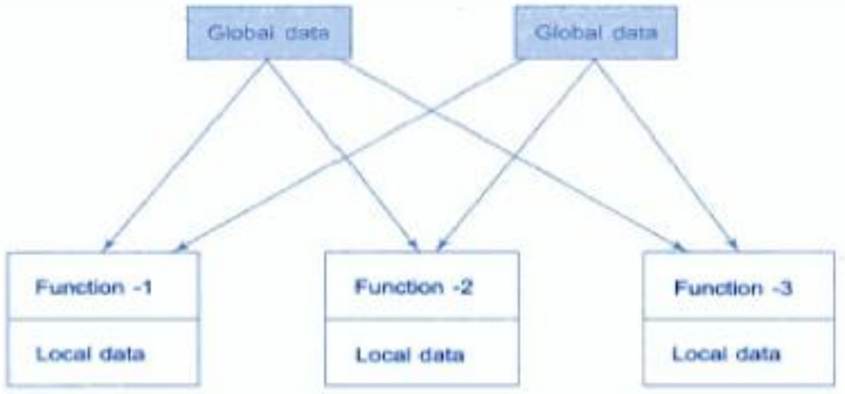
Content

- Introduction
  - Structured programming
  - Concept of OOP
  - Classes & Objects
  - Features of OOP
    - Abstraction & Encapsulation
    - Inheritance
      - Multiple inheritance
      - Abstract class
      - Interface
    - Polymorphism

Procedure Oriented Programming

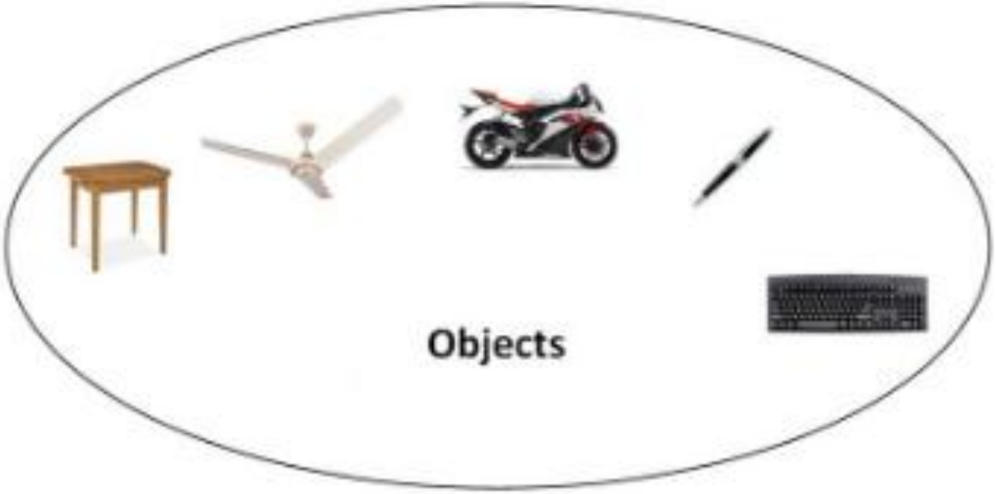
- ***Set of subroutines***” or a “***set of functions***”
- Global data can be accessed from anywhere.
- Less *reusable*.
- *Separate* the data structures (variables) and algorithms (functions)
- COBOL, FORTRAN and C





Object Oriented Programming

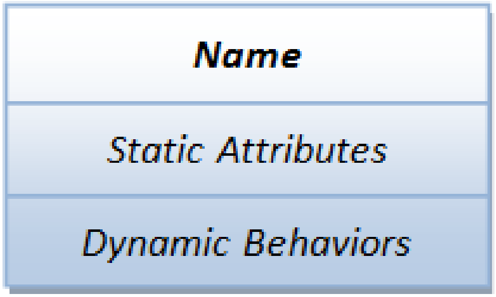
- Object means a real word entity
- The basic unit of OOP is a class which encapsulates both the properties and operations
- Easier to reuse these classes
- Combines the data structures and algorithms
- Data Hiding so provides more security
- Some concept:
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation



CLASSES & OBJECTS

- Class
  - Is blueprint from which objects are created.
  - Define data and action of objects.
- Object
  - Consist of data and actions.
  - Objects are instances of classes.
  - In most cases we interact with object through its methods.

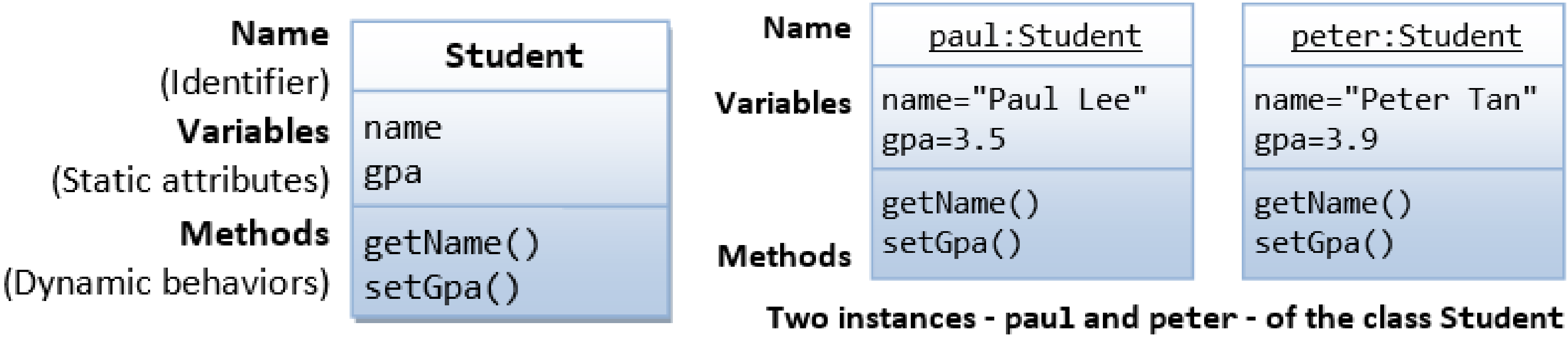
A Class is a 3-Compartment Box Encapsulating Data and Operations



Name

Static Attributes

Dynamic Behaviors



Name (Identifier)

Variables (Static attributes)

Methods (Dynamic behaviors)

Student

name
gpa

getName()
setGpa()

Name

Variables

Methods

paul:Student

name="Paul Lee"
gpa=3.5

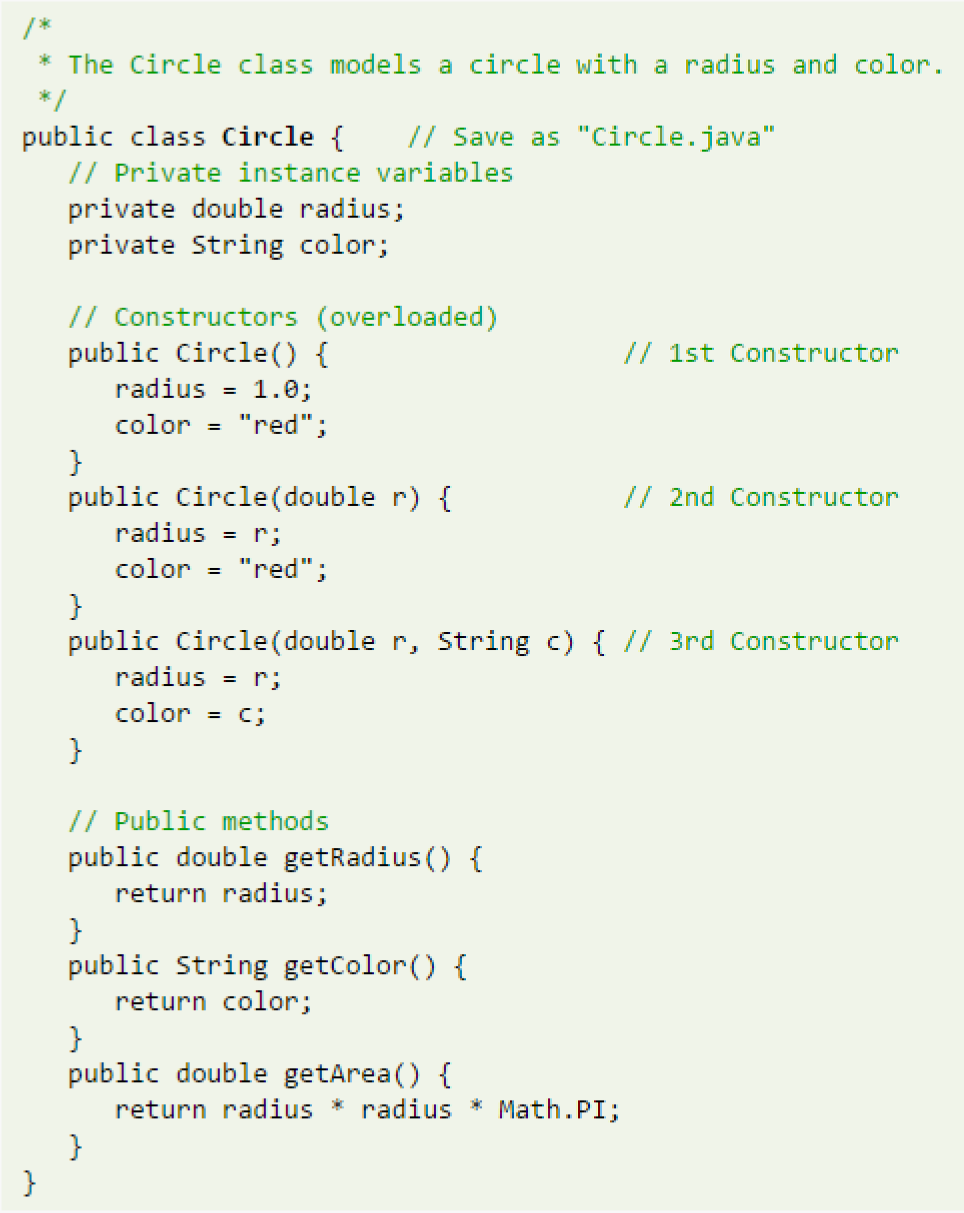
getName()
setGpa()

peter:Student

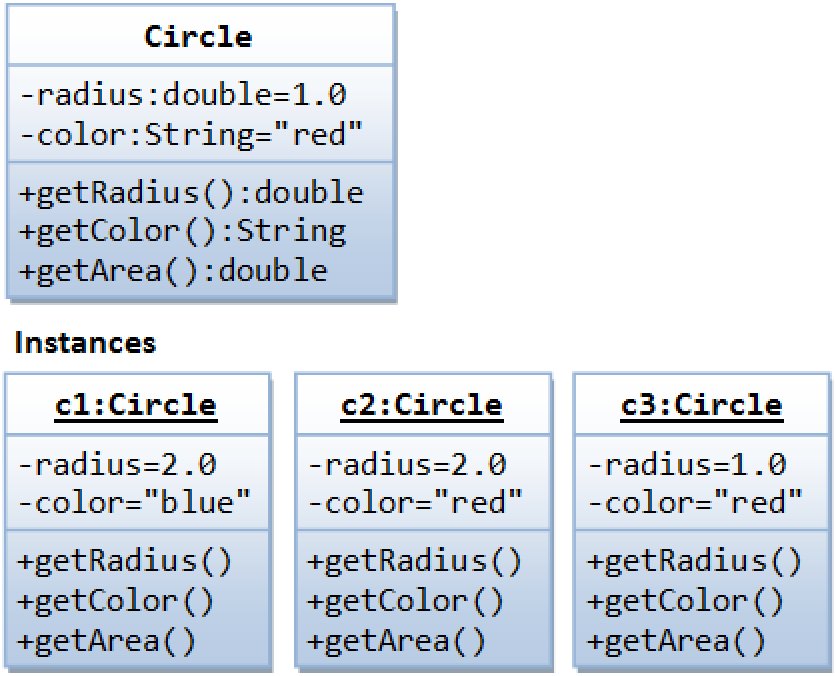
name="Peter Tan"
gpa=3.9

getName()
setGpa()

Two instances - paul and peter - of the class Student



```java
/*
 * The Circle class models a circle with a radius and color.
 */
public class Circle {    // Save as "Circle.java"
   // Private instance variables
   private double radius;
   private String color;

   // Constructors (overloaded)
   public Circle() {                    // 1st Constructor
      radius = 1.0;
      color = "red";
   }
   public Circle(double r) {            // 2nd Constructor
      radius = r;
      color = "red";
   }
   public Circle(double r, String c) { // 3rd Constructor
      radius = r;
      color = c;
   }

   // Public methods
   public double getRadius() {
      return radius;
   }
   public String getColor() {
      return color;
   }
   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```



Circle

-radius:double=1.0
-color:String="red"

+getRadius():double
+getColor():String
+getArea():double

Instances

c1:Circle

-radius=2.0
-color="blue"

+getRadius()
+getColor()
+getArea()

c2:Circle

-radius=2.0
-color="red"

+getRadius()
+getColor()
+getArea()

c3:Circle

-radius=1.0
-color="red"

+getRadius()
+getColor()
+getArea()

CLASSES & OBJECTS

- Access modifier:
  - This is the way to specify the accessibility of a class and its members with respective to other classes.
  - Access modifiers support for OOP features
  - Used at 2 levels:
    - ✓ Top – level for Class & Interface.
    - ✓ Member – level

CLASSES & OBJECTS

- Top – level for Class & Interface:
  - Public
  - Package/Default modifiers

Access modifier	Scope
Public	Inside and outside the package
Package/default	Just inside the package

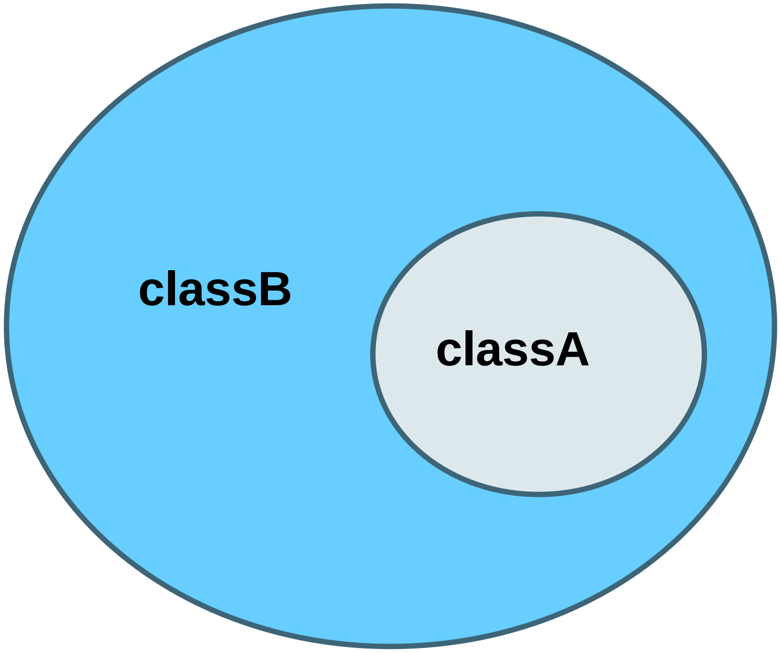
CLASSES & OBJECTS

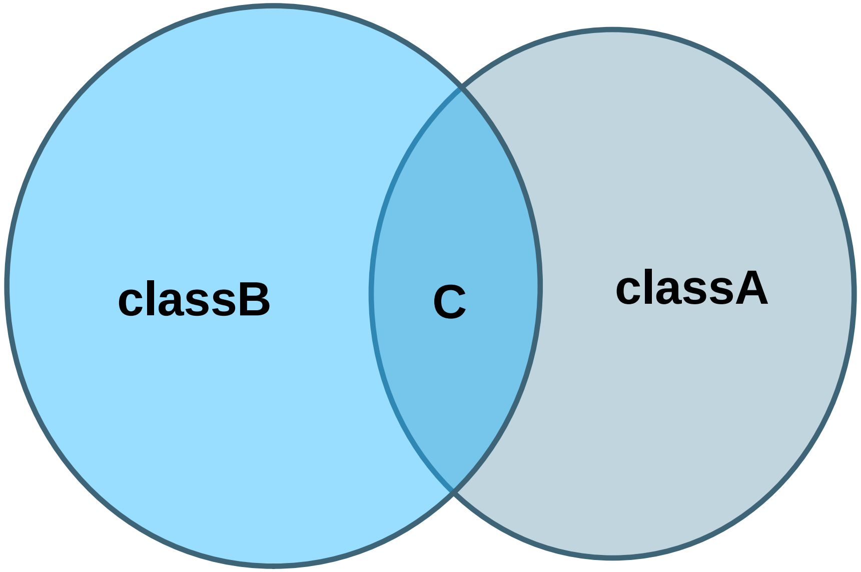
- Object member - Level

Access	public	protected	private
Same class (base)	Yes	Yes	Yes
Derived class	Yes	Yes	No
Outside classes	Yes	No	No

INHERITANCE

There are many cases that an object acquires some/all properties/methods of another object.

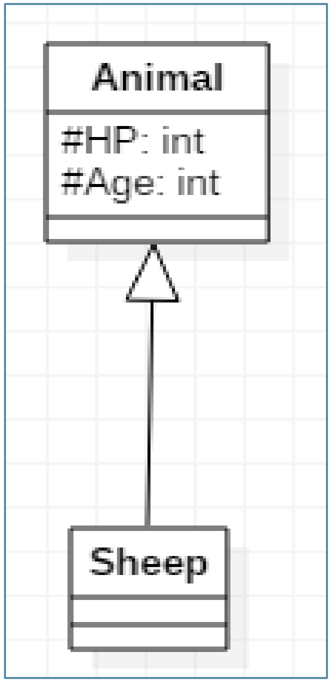




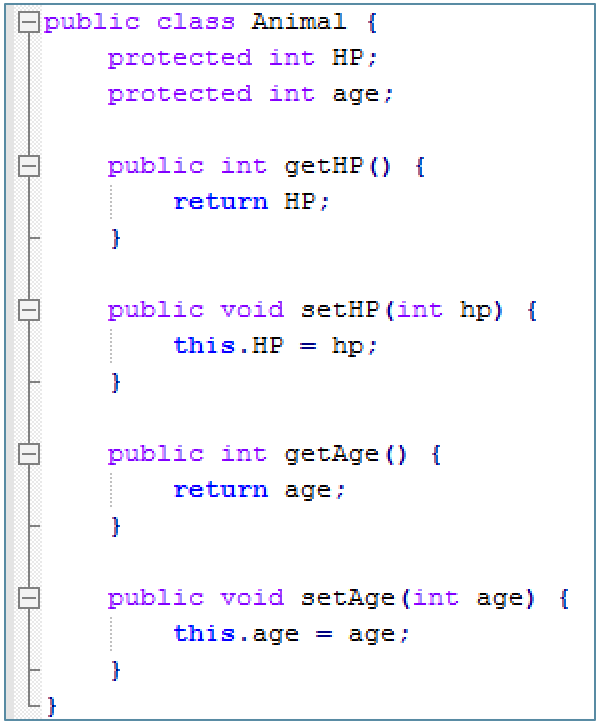
INHERITANCE

Define a new class base on existing classes.

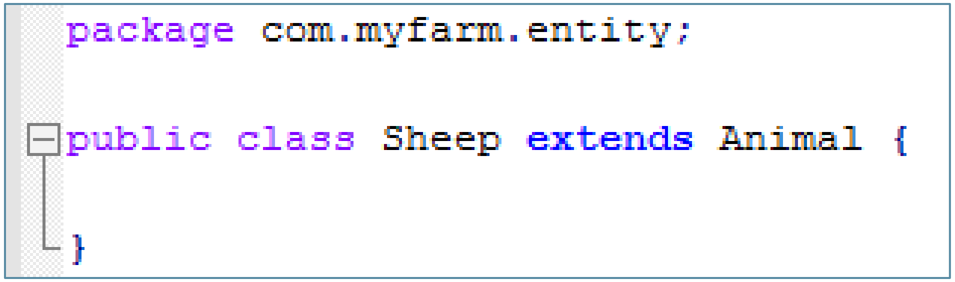
- Existing class: base class
- New class: derived class



INHERITANCE



```java
public class Animal {
    protected int HP;
    protected int age;

    public int getHP() {
        return HP;
    }

    public void setHP(int hp) {
        this.HP = hp;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

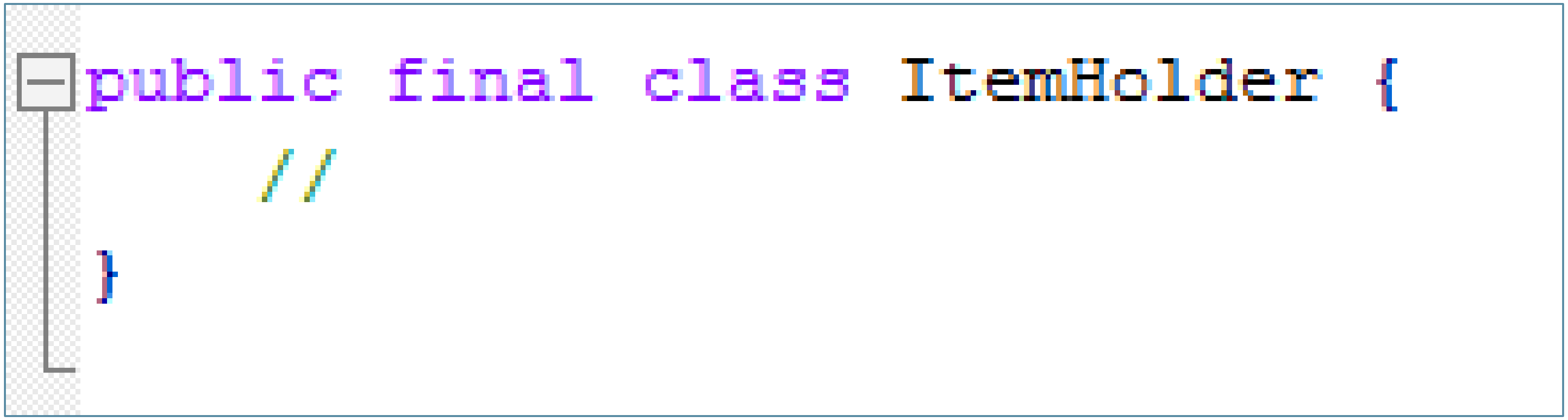


```java
package com.myfarm.entity;

public class Sheep extends Animal {

}
```

INHERITANCE

The ***final*** keyword in Java

- Sometime we don’t want a specific class to be a super class.
- Authors control the use of their code.

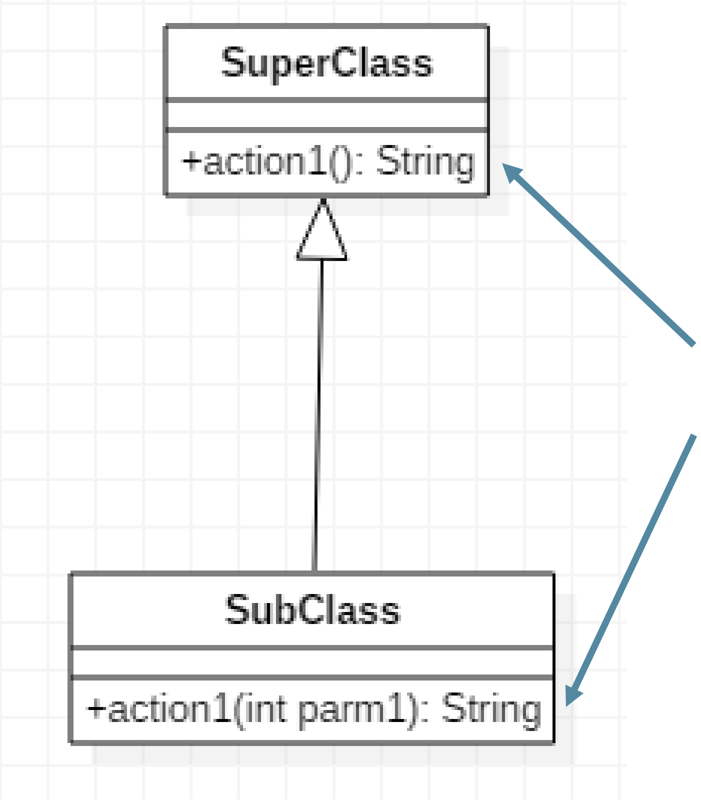


```
public final class ItemHolder {
      //
}
```

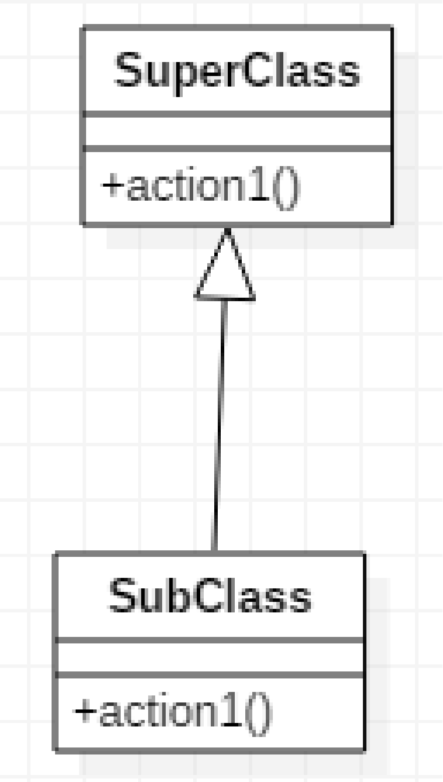
ACCESS SCOPE

Access	public	protected	private
Same class (base)	Yes	Yes	Yes
Derived class	Yes	Yes	No
Outside classes	Yes	No	No

METHOD OVERLOADING



same name but not the parameter list, type of parameters or return type.



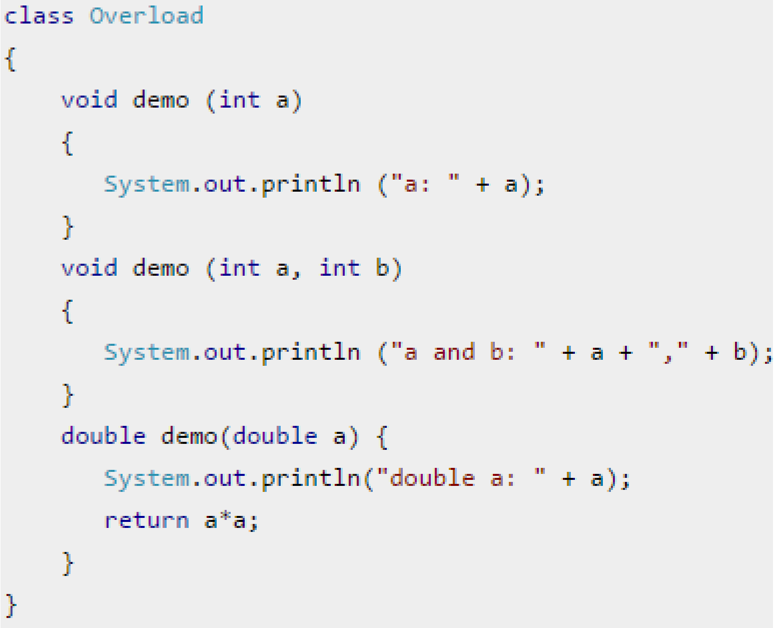
How about overriding???

METHOD OVERLOADING

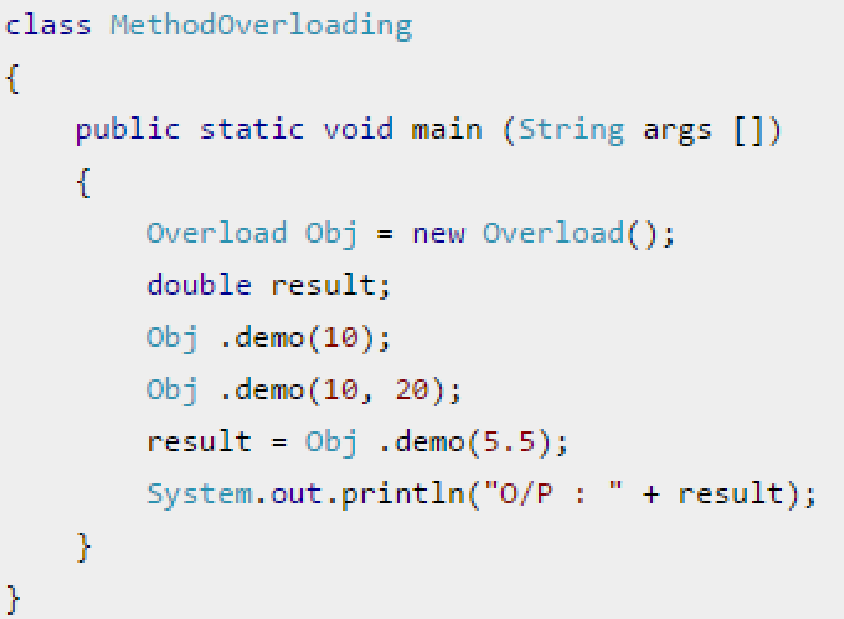
Rules for Method Overloading

- Overloading can take place in the same class or in its sub-class.
- Constructor in Java can be overloaded
- Overloaded methods must have a different argument list.
- Overloaded method should always be the part of the same class(can also take place in sub class), with same name but different parameters.
- The parameters may differ in their type or number, or in both.
- They may have the same or different return types.
- It is also known as compile time polymorphism.

METHOD OVERLOADING

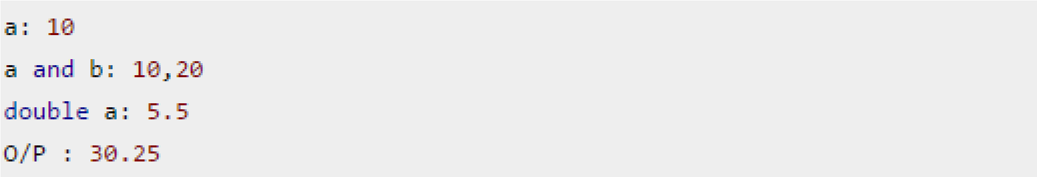


```
class Overload
{
    void demo (int a)
    {
       System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
       System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
       System.out.println("double a: " + a);
       return a*a;
    }
}
```



```
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

Output:

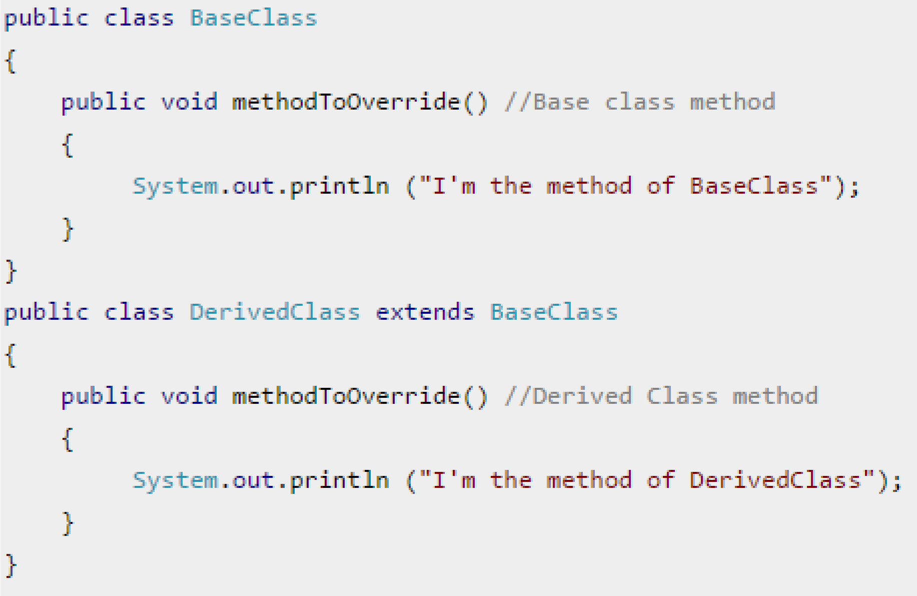


```
a: 10
a and b: 10,20
double a: 5.5
O/P : 30.25
```

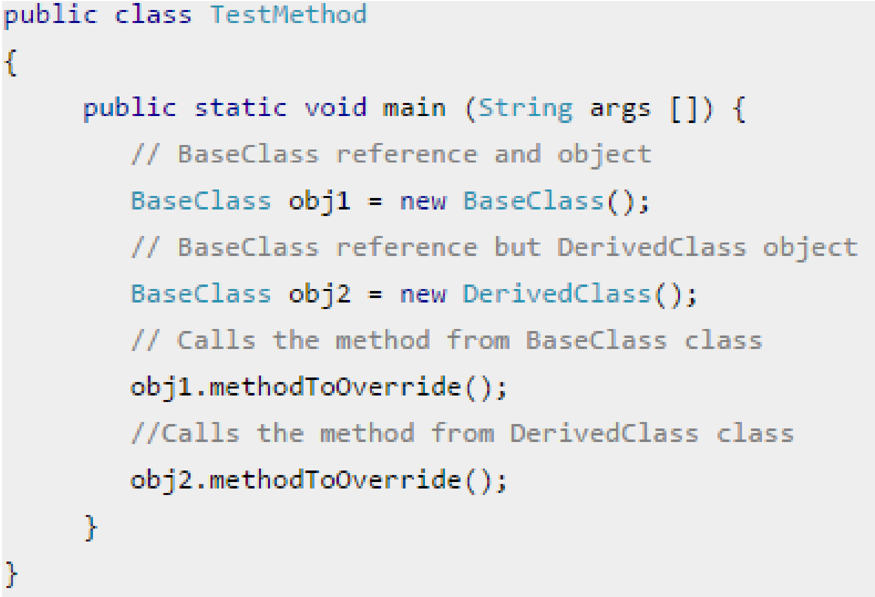
Method Overriding

- Child class has the same method as of base class
- **Rules for Method Overriding:**
  - applies only to inherited methods
  - object type (NOT reference variable type) determines which overridden method will be used at runtime
  - Overriding method can have different return type
  - Overriding method must not have more restrictive access modifier
  - Abstract methods must be overridden
  - Static and final methods cannot be overridden
  - Constructors cannot be overridden
  - It is also known as Runtime polymorphism.

Method Overriding

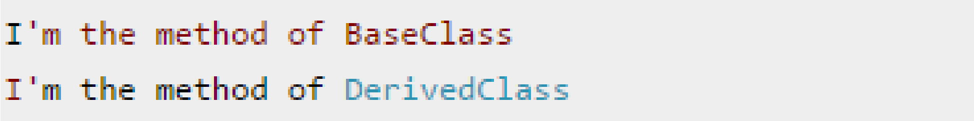


```
public class BaseClass
{
    public void methodToOverride() //Base class method
    {
         System.out.println ("I'm the method of BaseClass");
    }
}
public class DerivedClass extends BaseClass
{
    public void methodToOverride() //Derived Class method
    {
         System.out.println ("I'm the method of DerivedClass");
    }
}
```



```
public class TestMethod
{
     public static void main (String args []) {
        // BaseClass reference and object
        BaseClass obj1 = new BaseClass();
        // BaseClass reference but DerivedClass object
        BaseClass obj2 = new DerivedClass();
        // Calls the method from BaseClass class
        obj1.methodToOverride();
        //Calls the method from DerivedClass class
        obj2.methodToOverride();
     }
}
```

Output:

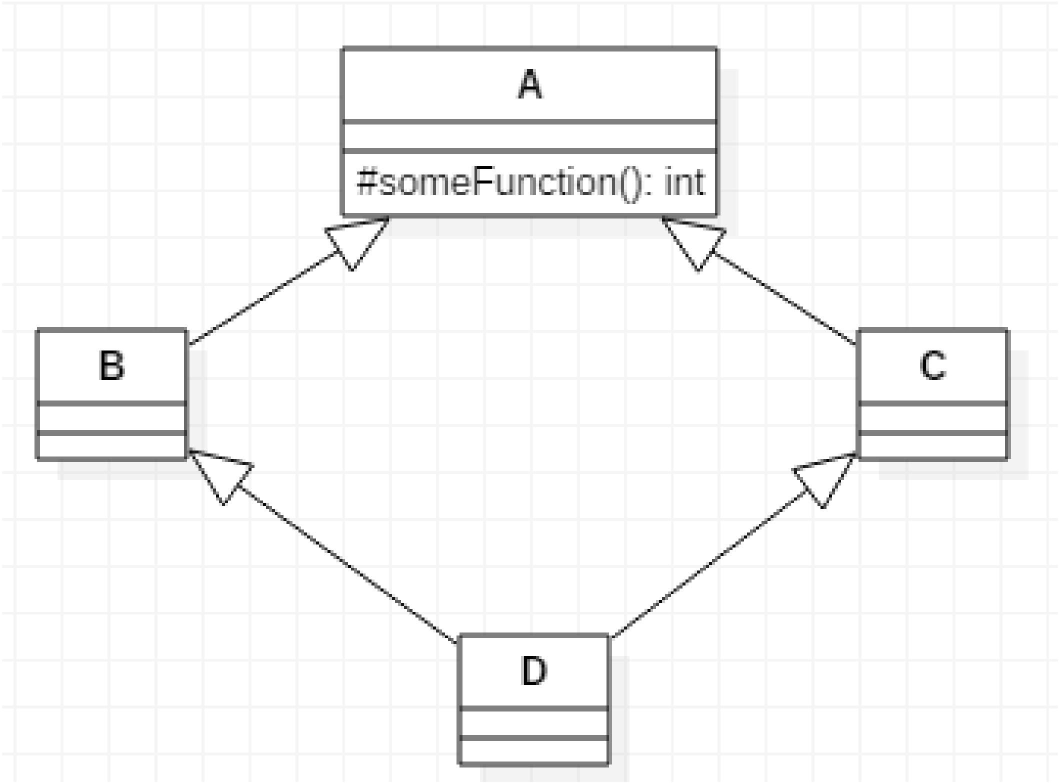


```
I'm the method of BaseClass
I'm the method of DerivedClass
```

MULTIPLE INHERITANCE

- A class can inherit from more than one class.
- Some languages those support multiple inheritance:
  - C++
  - Common Lisp
  - Perl
- Java does not support multiple inheritance, but we can overcome this by using *interface*.

DIAMOND PROBLEM



- Java does not allow multiple inheritance → we will not face this problem.
- For C++, we solve this problem by **virtual** inheritance.

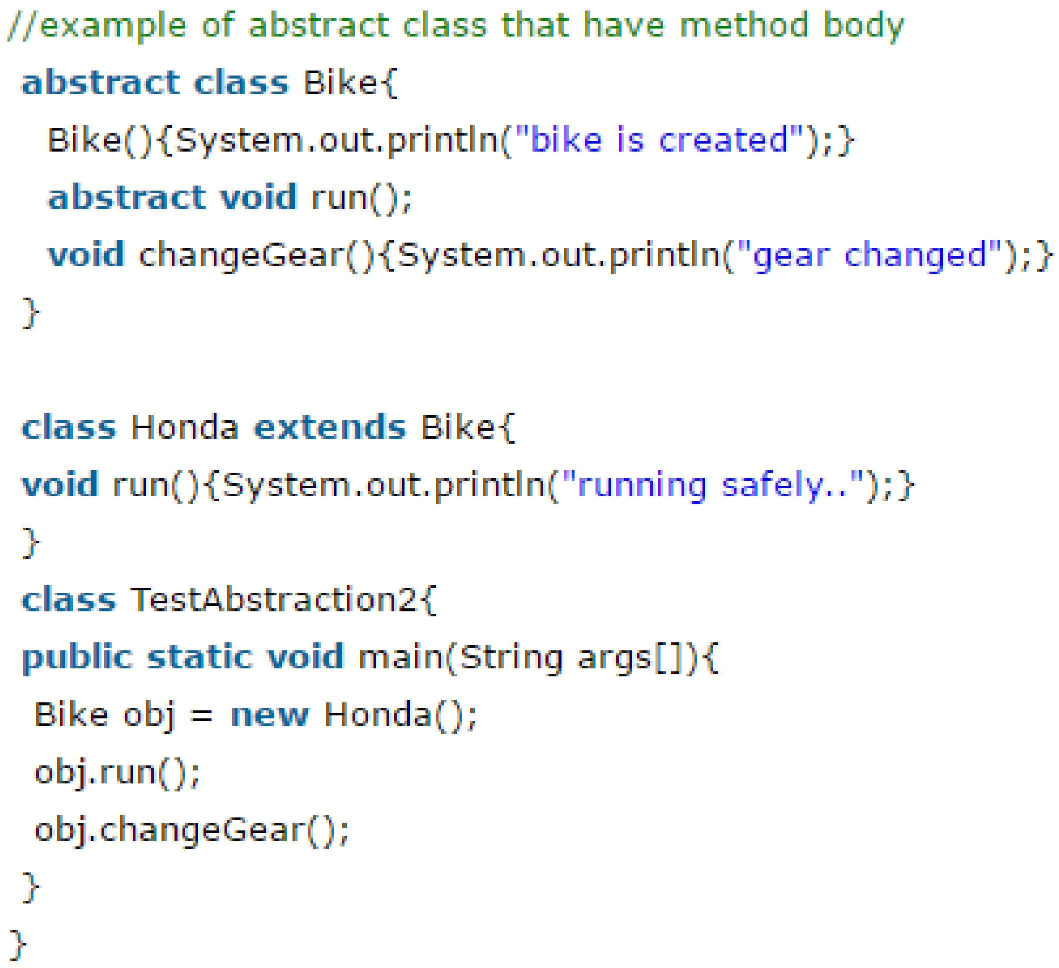
Abstraction

- In java, we use abstract class and interface to achieve abstraction.

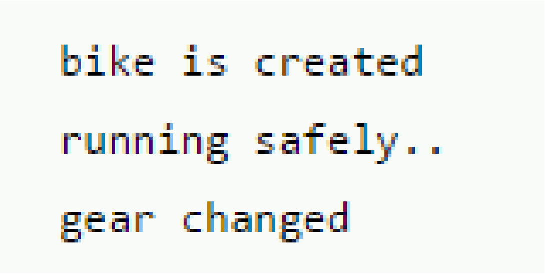
Abstract class

- Abstract classes may or may not contain *abstract methods* ie., methods with out body (public void get();)
- But, if a class have at least one abstract method, then the class **must**be declared abstract.
- If a class is declared abstract it cannot be instantiated.
- To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class you have to provide implementations to all the abstract methods in it.
- An abstract class can have data member, abstract method, method body, constructor and even main() method

Abstract class



```java
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }

 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

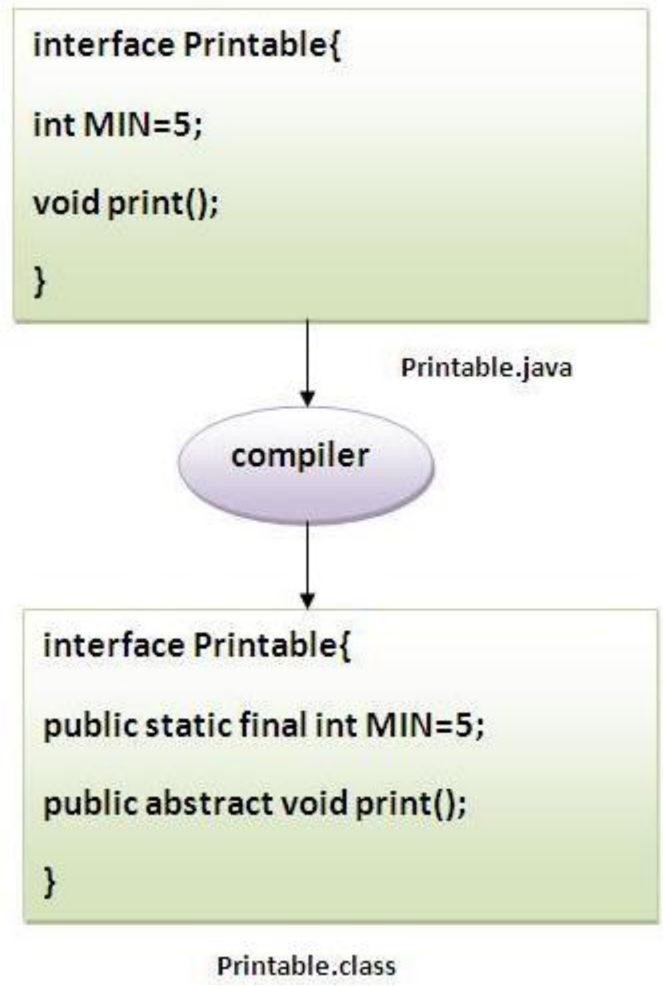


```
bike is created
running safely..
gear changed
```

Interface

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.
- The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

Interface



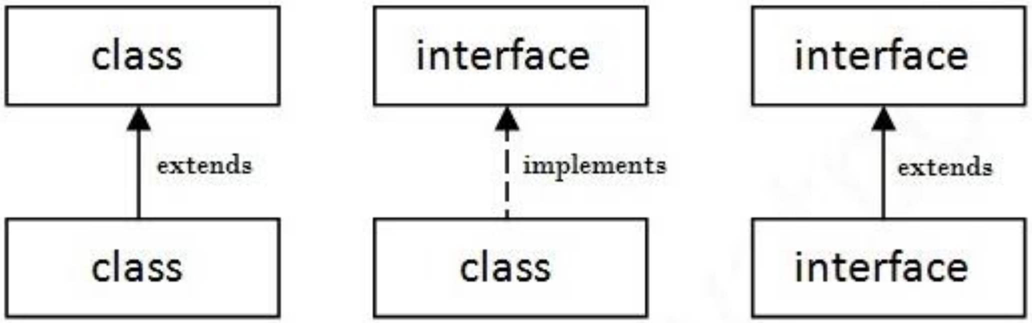
```
interface Printable{

int MIN=5;

void print();

}
```

Printable.java

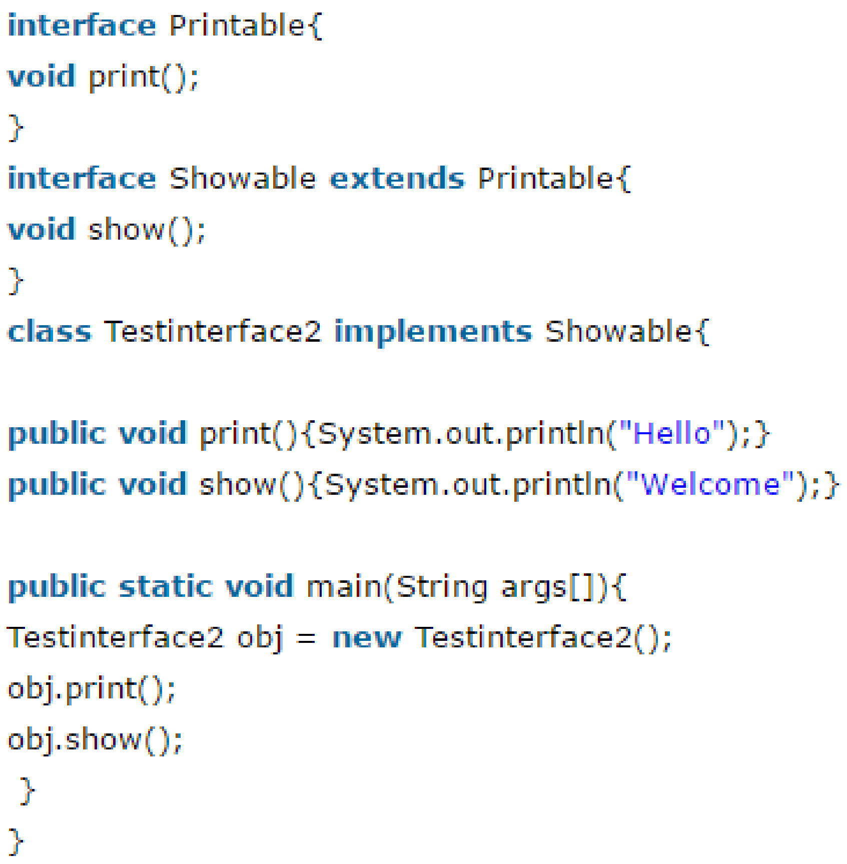
```
interface Printable{

public static final int MIN=5;

public abstract void print();

}
```

Printable.class

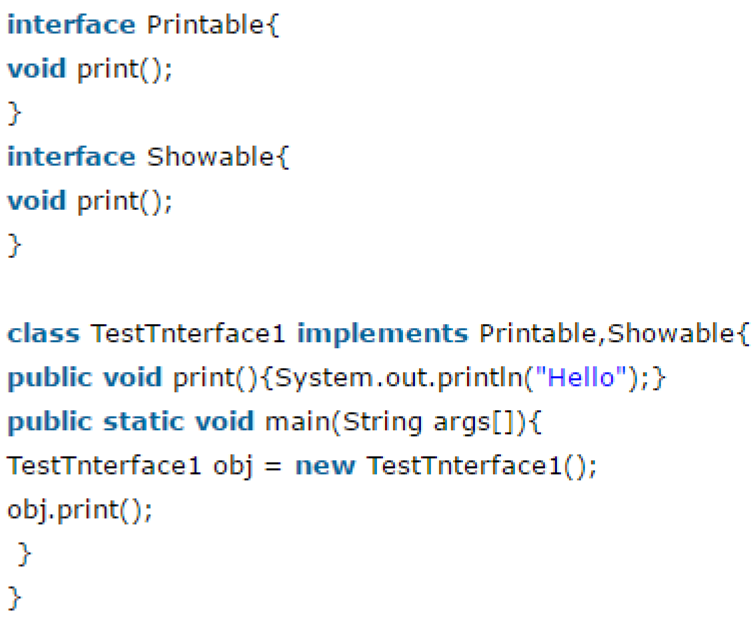
- An **interface** can **extend multiple interfaces**.
- A **class** can **implement multiple interfaces**.
- However, a **class** can only **extend a single class**.



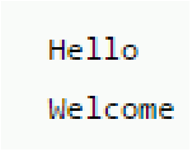
Interface



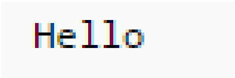
```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 }
}
```



```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestTnterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestTnterface1 obj = new TestTnterface1();
obj.print();
 }
}
```

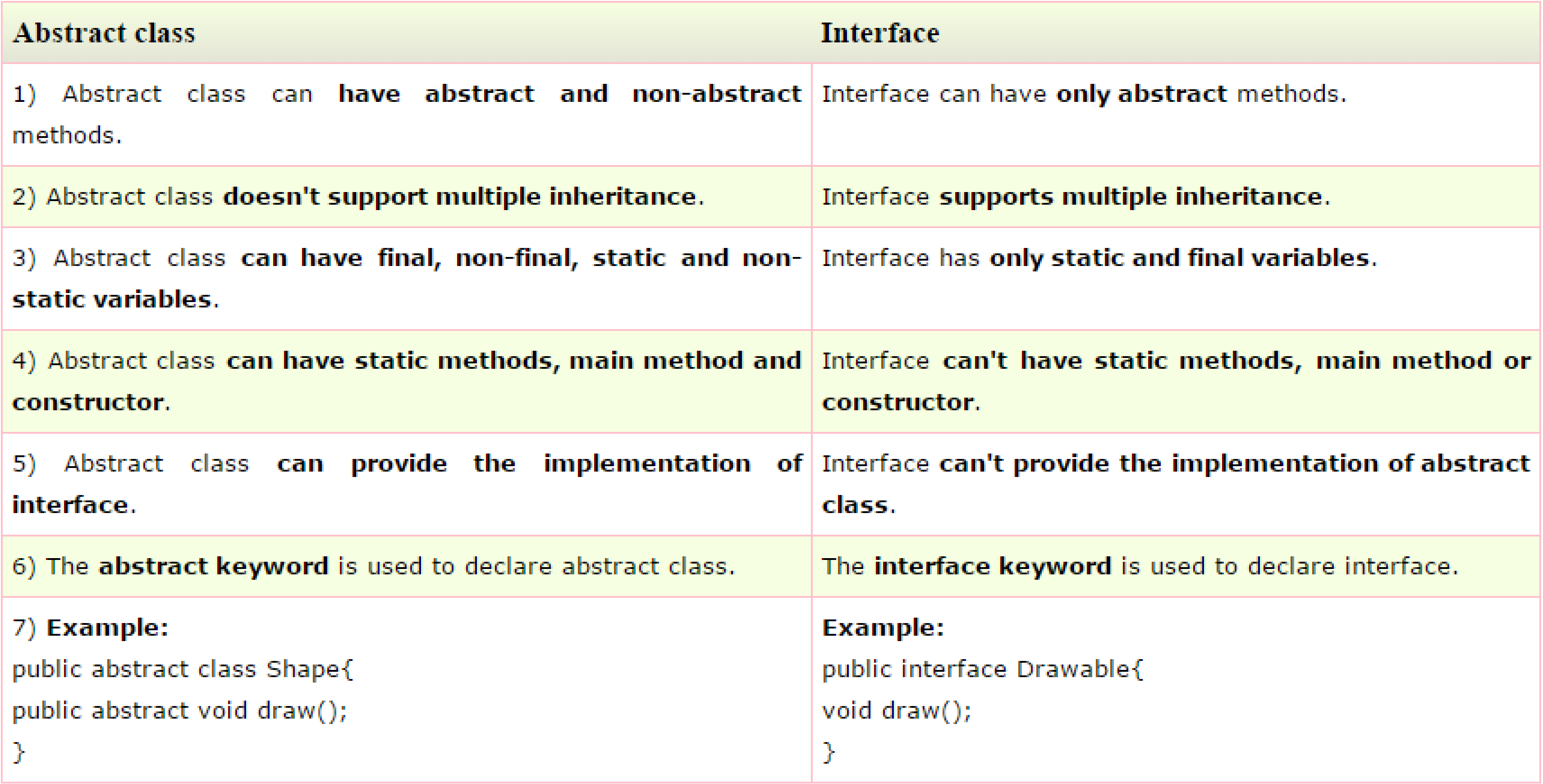


```
Hello
Welcome
```



```
Hello
```

Difference between abstract class and interface

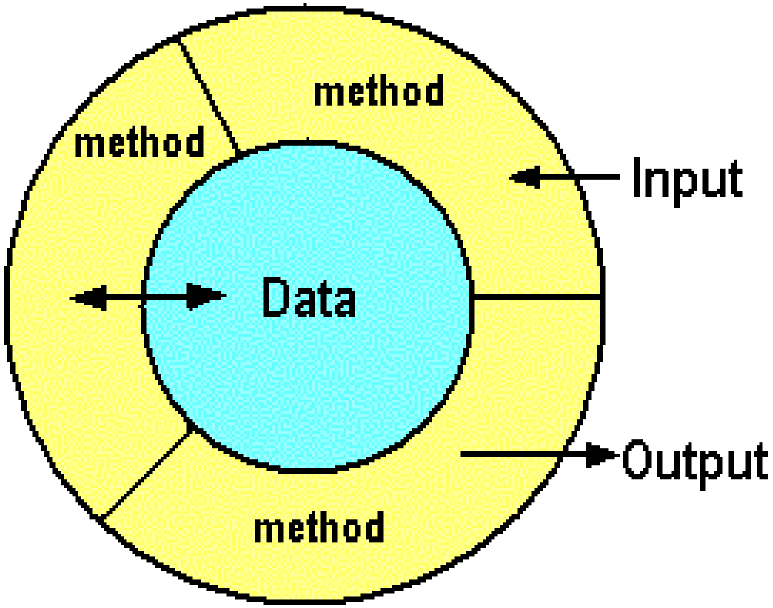


Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

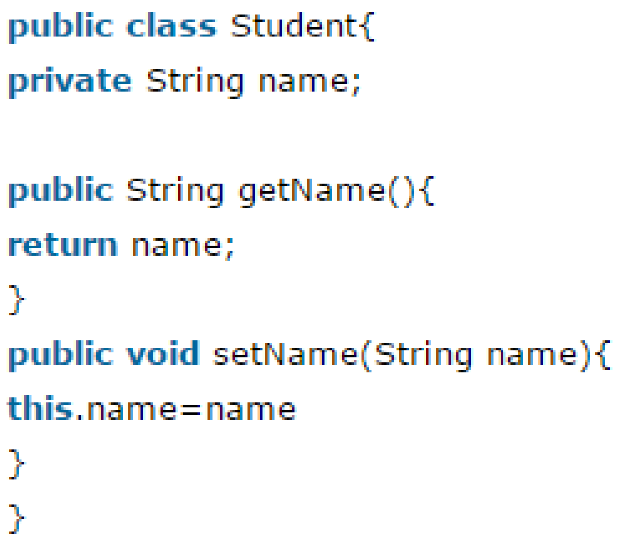
7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Encapsulation

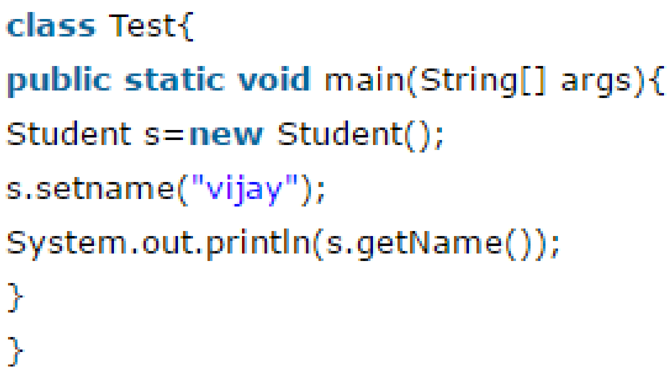
- Just methods of an object can access its own data.
- This is used to enforce the principle of data hiding.
- Handle with the visibility of object’s members.
- Implement “Abstraction”.



Encapsulation

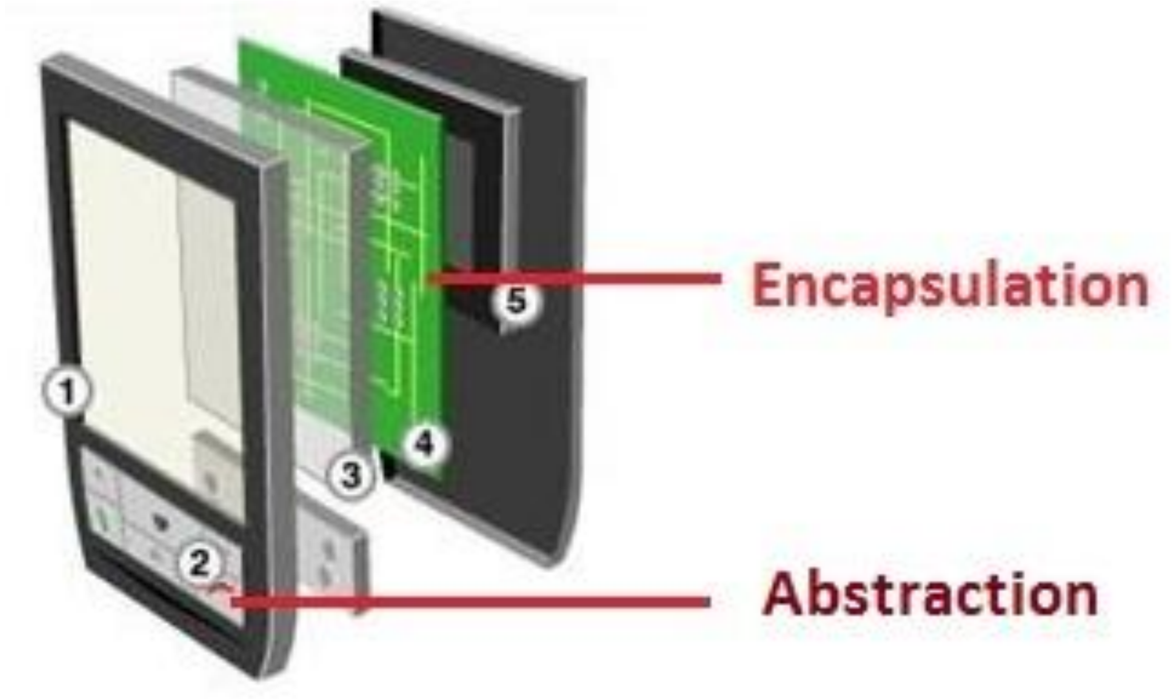


```
public class Student{
private String name;

public String getName(){
return name;
}
public void setName(String name){
this.name=name
}
}
```



```
class Test{
public static void main(String[] args){
Student s=new Student();
s.setname("vijay");
System.out.println(s.getName());
}
}
```

Encapsulation vs Abstraction



Abstract Classes

- May or may not include abstract methods.
- Cannot be instantiated

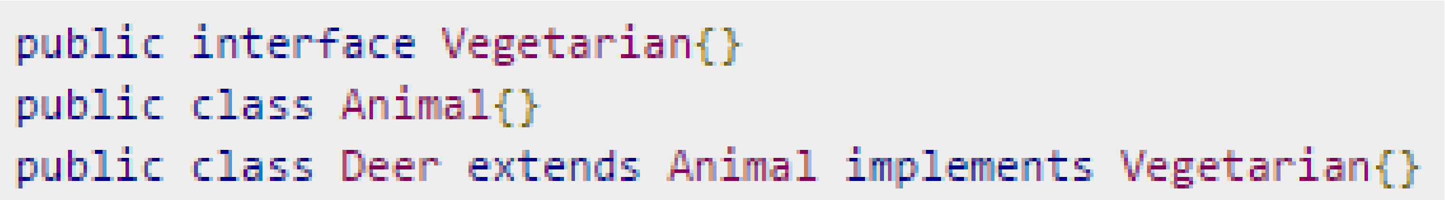
Interfaces

- A reference type in Java.
- Interfaces are not classes.
- Can contain only constant and method signatures.
- Cannot be instantiated.

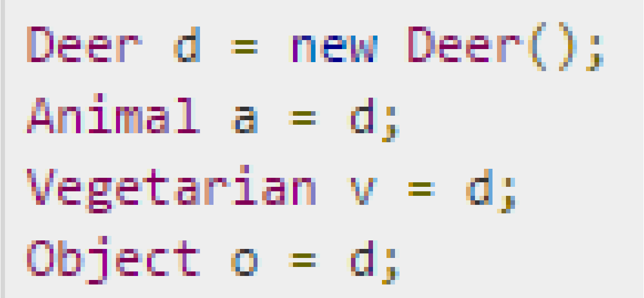
Polymorphism

- Object can be represented in many forms.
- A powerful mechanism in OOP to *separate the interface and implementation*
- Occurs when a parent class reference is used to refer to a child class object

Polymorphism



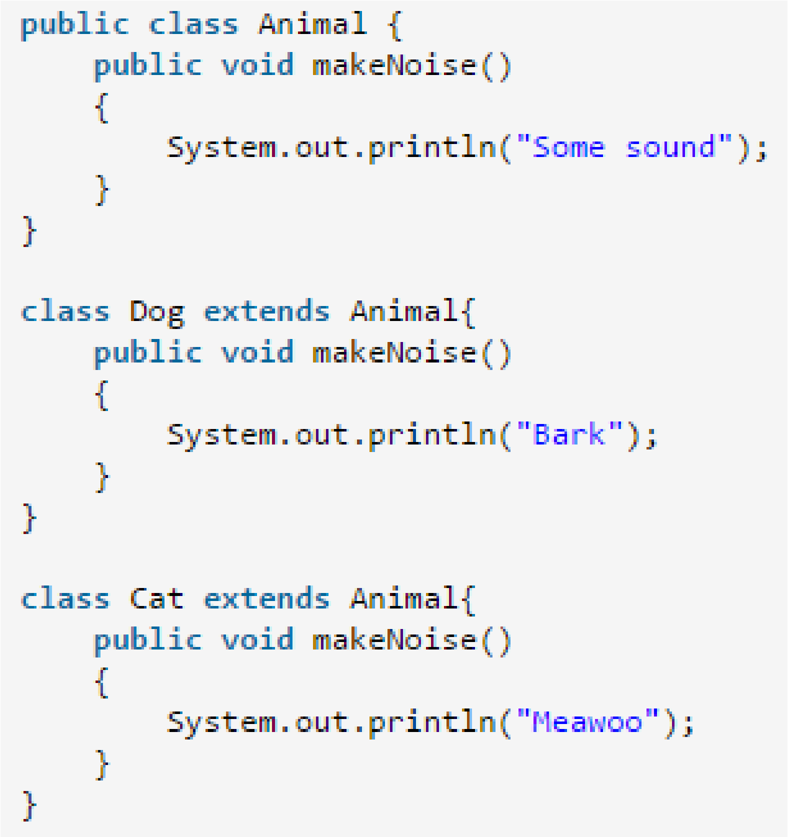
```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```



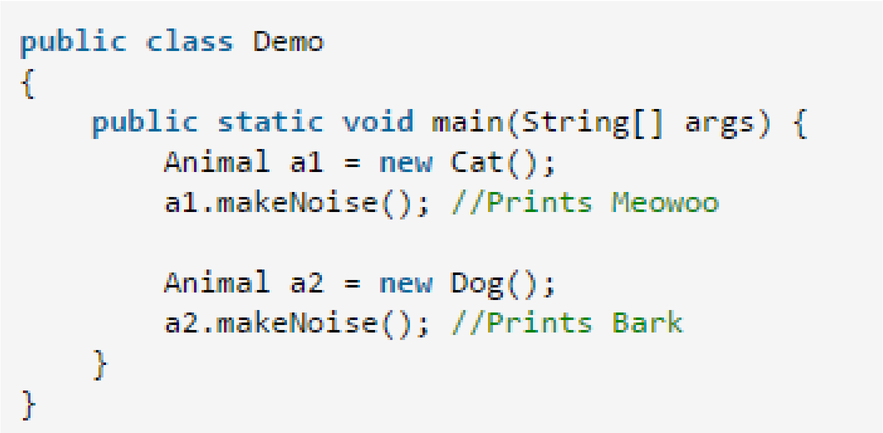
```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

✓ A Deer IS-A Animal
✓ A Deer IS-A Vegetarian
✓ A Deer IS-A Deer
✓ A Deer IS-A Object

Exercises



```java
public class Animal {
    public void makeNoise()
    {
        System.out.println("Some sound");
    }
}

class Dog extends Animal{
    public void makeNoise()
    {
        System.out.println("Bark");
    }
}

class Cat extends Animal{
    public void makeNoise()
    {
        System.out.println("Meawoo");
    }
}
```



```java
public class Demo
{
    public static void main(String[] args) {
        Animal a1 = new Cat();
        a1.makeNoise(); //Prints Meowoo

        Animal a2 = new Dog();
        a2.makeNoise(); //Prints Bark
    }
}
```

References

- Stephen Prata, **C++ Primer Plus**, 4thEdition, *Sams*, 2004.
- Bjarne Stroustrup, **The C++ Programming Language**, 4thEdition, Addison-Wesley, 2013
- Marshall Cline, **C++ FAQ Lite**, 2ndEdition, Addison-Wesley, 2000
- Robert L. Kruse, Alexander J. RybaData, **Data Structures and Program Design in C++**, *Prentice-Hall*, Inc., 2000
- Bruce Eckel, **Thinking in C++**,*Prentice Hall*, 1998
- http://faculty.orangecoastcollege.edu/sgilbert/book/03-4-ObjectOrientedConcepts-B/index.html