

# Session 05

# Interface and Inheritance

(<https://docs.oracle.com/javase/tutorial/java/landI/>)

# Objectives

- Benefits of OO implementation:  
Inheritance, Polymorphism
- Working with Interfaces.
- Working with Abstract Methods and Classes.
- Anonymous Classes
- Enum Type
- Practice walkthroughs

# Implementing Object-Oriented Relationships

- 3 common relations in classes:

- “is a/ a kind of”

A is a B : A is con , B is Cha

- “has a”

bieu dien properties

- association

quan he 1vs1, 1vsnhieu, nhieu vs nhieu

- Examples:

- Student is a person

home

- “A home is a house that has a family and a pet.”

class

- An invoice contains some products and a product can be contained in some invoices

cach nhan dien ten Class thong qua cac thuoc tinh

# Implementing Object-Oriented Relationships...

The relation “is a” is implemented as a sub-class

Classes Professor, Student are sub-classes of the class Person  
Sub-classes inherit the structure of super class

Person
- String name, address
- String birthDate
+ String getName();
+ void setName(String n);
.....

The relation “has a” is implemented as reference

is a

Professor
- String department
+ String getDepartment();
+ void setDepartment(String d);

teach

is a

Student
- String studentId, majorField
- String degreeSought
+ String getStudentId();
+ void setStudentID(String id)
....

The class Professor has the field Student[] students

The class Student has the field Professor pr

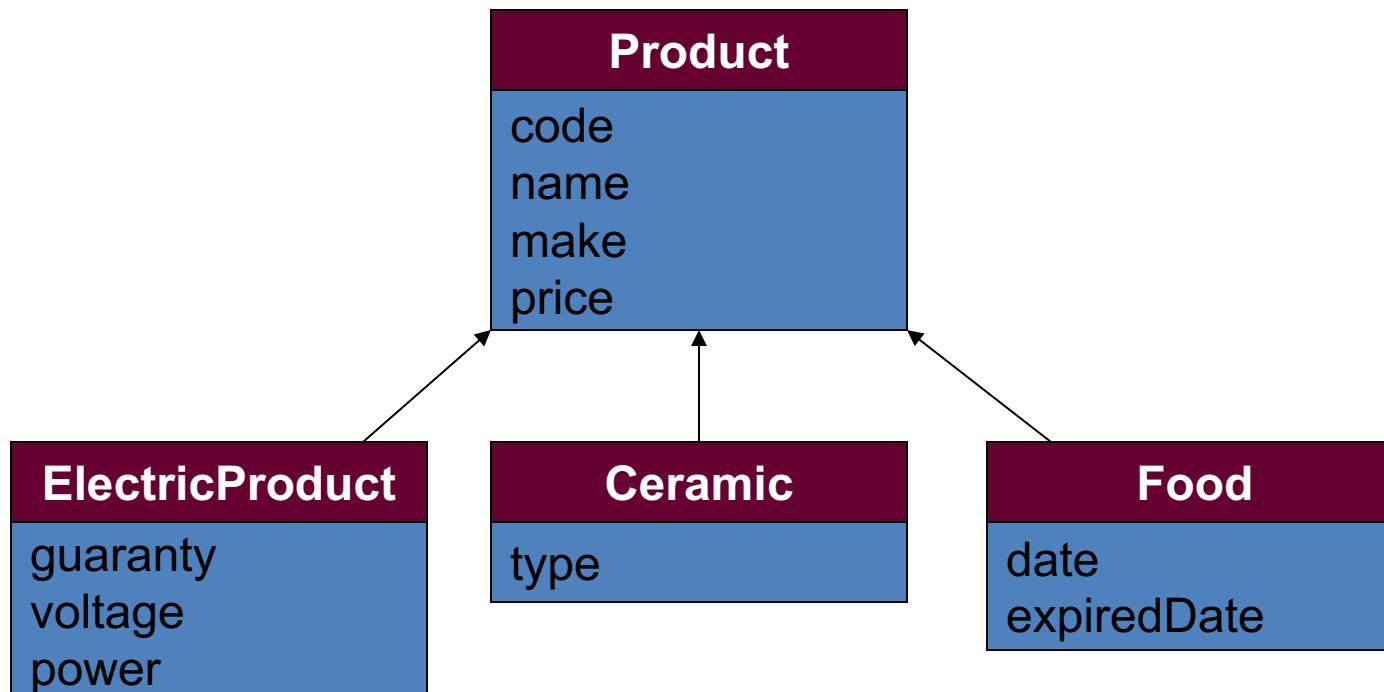
# Inheritance

- There are some sub-classes from one super class → An inheritance is a relationship where objects share a common structure: the structure of one object is a sub-structure of another object.
- The **extends** keyword is used to create sub-class.
- A class can be directly derived from **only one** class ( Java is a single-inherited OOP language).
- If a class does not have any superclass, then it is implicitly derived from Object class.
- Unlike other members, constructor cannot be inherited ( constructor of **super** class can not initialize sub-class objects)

# Inheritance...

- **How to construct a class hierarchy? → Intersection**

- Electric Products< code, name, make, price, guaranty, voltage, power>
- Ceramic Products < code, name, make, price, type >
- Food Products < code, name, make, price , date, expiredDate >



# Inheritance...: “super” Keyword

- Constructors Are **Not** Inherited
- **super(...)** for Constructor Reuse
  - `super(arguments); //invoke a superclass constructor`
  - The call **must be the first statement in the subclass constructor**
- Replacing the Default Parameterless Constructor

# Inheritance...: “super” Keyword

- We use the Java keyword super as the qualifier for a method call:  
*super. methodName(arguments);*
- Whenever we wish to invoke the version of method methodName that was defined by our superclass.
- **super()** is used to access the superclass's constructor. And It must be the first statement in the constructor of the subclass.

# Inheritance...

```
1  public class Rectangle {
2      private int length = 0;
3      private int width = 0;
4      // Overloading constructors
5      public Rectangle() // Default constructor
6      {
7          public Rectangle(int l, int w)
8          {   length = l>0? l: 0;   width= w>0? w: 0;
9          }
10     // Overriding the toString method of the java.lang.Object class
11     public String toString()
12     {
13         return "[" + getLength() + "," + getWidth() + "]";
14     }
15     // Getters, Setters
16     public int getLength() { return length; }
17     public void setLength(int length) { this.length = length; }
18     public int getWidth() { return width; }
19     public void setWidth(int width) { this.width = width; }
20     public int area() { return length*width; }
}
```

# Inheritance...

```
1  public class Box extends Rectangle {  
2      private int height=0; // additional data  
3      public Box() { super(); }  
4      public Box (int l, int w, int h)  
5      { super(l, w); // Try swapping these statements  
6          height = h>0? h: 0;  
7      }  
8      // Additional Getter, Setter  
9      public int getHeight() { return height; }  
10     public void setHeight(int height)  
11         { this.height = height; }  
12     // Overriding methods  
13     public String toString()  
14     { return "[" + getLength() + "," +  
15         getWidth() + "," + getHeight() + "]";  
16     }  
17     public int area(){  
18         int l = this.getLength();  
19         int w = this.getWidth();  
20         int h = this.getHeight();  
21         return 2*(l*w + w*h + h*l);  
22     }  
23     // additional method  
24     public int column()  
25         return this.getLength()*this.getWidth()*height;  
26     }  
27 }
```

```
1  public class Demo_1 {  
2      public static void main (String[] args)  
3      { Rectangle r= new Rectangle(2,5);  
4          System.out.println("Rectangle: " + r.toString());  
5          System.out.println(" Area: " + r.area());  
6          Box b= new Box(2,2,2);  
7          System.out.println("Box " + b.toString());  
8          System.out.println(" Area: " + b.area());  
9          System.out.println(" Volumn: " + b.volumn());  
10     }  
11 }
```

## Output - Chapter06 (run)

```
run:  
Rectangle: [2,5]  
Area: 10  
Box [2,2,2]  
Area: 24  
Volumn: 8  
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Overriding and Hiding Methods (1)

- **Overriding a method:** An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.
  - Use the **@Override** annotation that instructs the compiler that you intend to override a method in the superclass (you may not use it because overriding is default in Java).
- **Hiding a method:** Re-implementing a static method implemented in super class

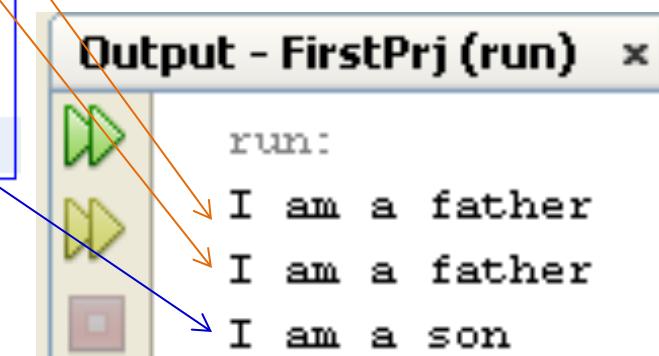
# Overriding and Hiding Methods (1)

## Method Hiding Java Code

```
1 package com.test;
2
3 class Parent {
4
5     public static void foo() {
6         System.out.println("Inside foo method in parent class");
7     }
8
9     public void bar() {
10        System.out.println("Inside bar method in parent class");
11    }
12 }
13
14 class Child extends Parent {
15     // Hiding
16     public static void foo() {
17         System.out.println("Inside foo method in child class");
18     }
19
20     // Overriding
21     public void bar() {
22         System.out.println("Inside bar method in child class");
23     }
24 }
25
26 public class Code {
27
28     public static void main(String[] args) {
29         Parent p = new Parent();
30         Parent c = new Child();
31         System.out.println("*****Method Hiding*****");
32         p.foo(); // This will call method in parent class
33         c.foo(); // This will call method in parent class
34         System.out.println("*****Method overriding*****");
35         p.bar(); // This will call method in parent class
36         c.bar(); // This will call method in child class
37     }
38 }
39 }
```

# Overriding and Hiding Methods (2)

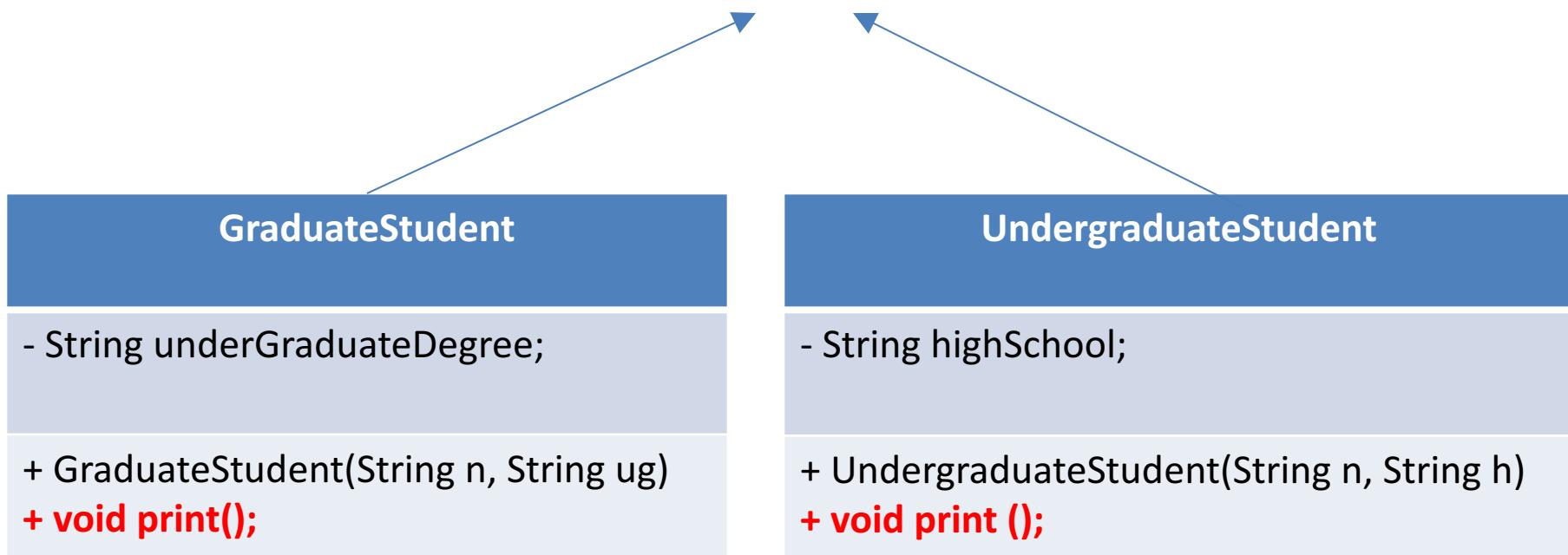
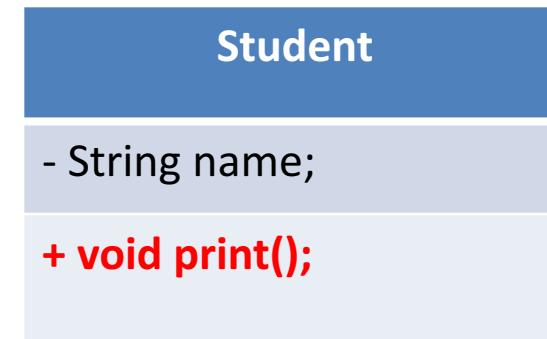
```
class Father1 {  
    public static void m(){  
        System.out.println("I am a father");  
    }  
}  
  
class Son1 extends Father1{  
    public static void m(){  
        System.out.println("I am a son");  
    }  
}  
  
public class HidingMethodDemo {  
    public static void main(String args[]){  
        Father1 obj= new Father1();  
        obj.m();  
        obj= new Son1();  
        obj.m();  
        Son1 obj2= new Son1();  
        obj2.m();  
    }  
}
```



# Polymorphism

- The ability of two or more objects belonging to ***different*** classes to respond to exactly the ***same*** message (method call) in different class-specific ways.
- ***Inheritance combined with overriding facilitates polymorphism.***

# Polymorphism...)



# Polymorphism...)

## - *HinhHoc*

## - *MainClass*

```
1 public class MainClass {  
2  
3     public static void main(String[] args) {  
4         HinHoc hinhHoc = new HinHoc();  
5         hinhHoc.tinhDienTich(); // Đoạn code |  
6  
7         // Có lúc hinhHoc đóng vai trò là Hin  
8         hinhHoc = new HinTron();  
9         hinhHoc.tinhDienTich(); // Đoạn code |  
10  
11        // Có lúc hinhHoc đóng vai trò là Hin  
12        hinhHoc = new HinChuNhat();  
13        hinhHoc.tinhDienTich(); // Đoạn code |  
14    }  
15}  
16 }
```

```
1 public class HinHoc {  
2  
3     public void tinhDienTich() {  
4         System.out.println("Chưa biết hình nào");  
5     }  
6 }
```

## - *HinTron*

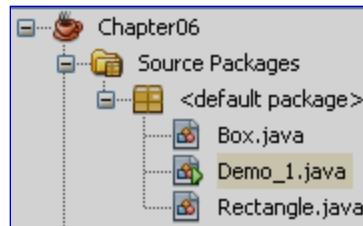
```
1 public class HinTron extends HinHoc {  
2  
3     @Override  
4     public void tinhDienTich() {  
5         System.out.println("Đây là Diện tích hình Tròn");  
6     }  
7 }  
8 }
```

## - *HinChuNhat*

```
1 public class HinChuNhat extends HinHoc {  
2  
3     @Override  
4     public void tinhDienTich() {  
5         System.out.println("Đây là Diện tích hình Chữ nhật");  
6     }  
7 }  
8 }
```

# Overriding Inherited Methods

Overridden method: An inherited method is re-written



```
1  public class Rectangle {
2      protected int length=0, width=0;
3      // Overloading methods
4      public void setValue(int l)
5      {   length = l>0?l:0; }
6      public void setValue(int l, int w)
7      {   length = l>0? l: 0;
8         width= w>0? w: 0; }
9
10     // Overriding the toString method of the java.lang.Object class
11     public String toString()
12     {   return "[" + length + "," + width + "]"; }
13
14 }
```

```
1  public class Box extends Rectangle {
2      int height=0;
3      public void set (int l, int w, int h)
4      {   super.setValue(l, w);
5         height = h>0? h: 0;
6     }
7
8     // Overriding the toString method
9     // of the Rectangle class
10    public String toString()
11    {   return "[" + length + "," + width +
12        "," + height + "]"; }
13 }
```

Output - Chapter06 (run)

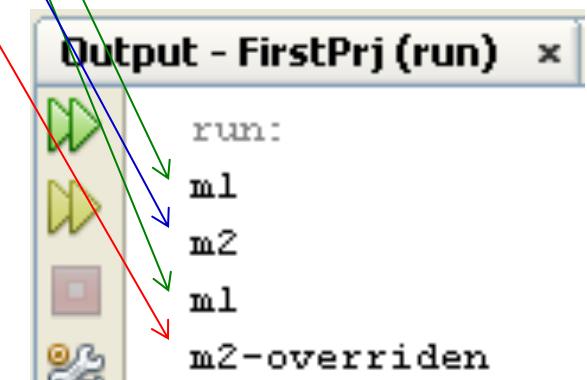
run:	[5,0]
	[10,20]
copy:	[5,10,15]

```
public class Demo_1 {
    public static void main (String[] args)
    {   Rectangle r= new Rectangle();
        r.setValue(5);
        System.out.println(r.toString());
        r.setValue(10,20);
        System.out.println(r.toString());
        Box b= new Box();
        b.set(5,10,15);
        System.out.println(b.toString());
    }
}
```

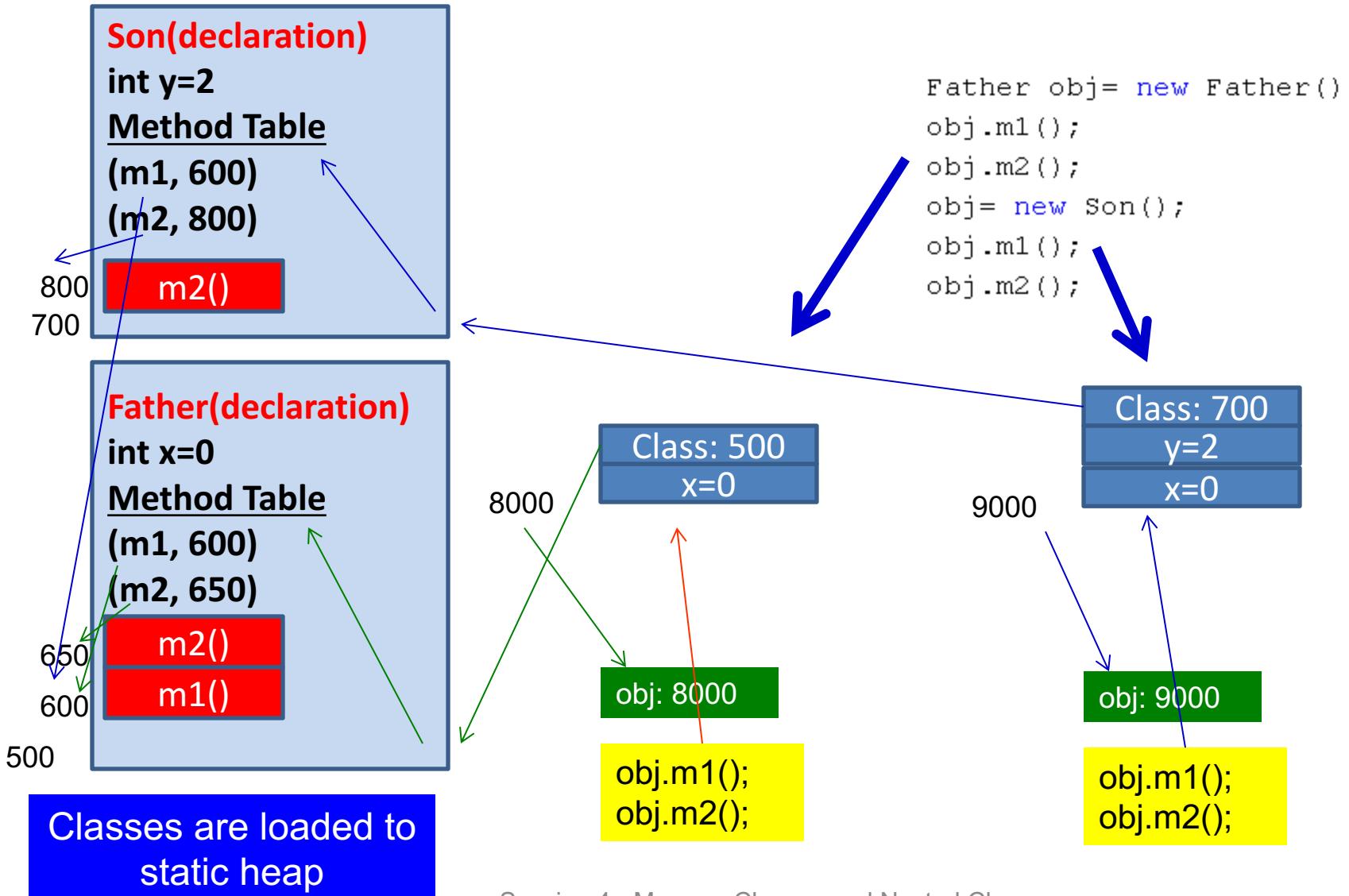
Overloaded methods: Methods have the same name but their parameters are different in a class

# How Can Overridden Method be Determined?

```
class Father{  
    int x=0;  
    void m1() { System.out.println("m1"); }  
    void m2() { System.out.println("m2"); }  
}  
class Son extends Father {  
    int y=2;  
    void m2() { System.out.println("m2-overridden"); }  
}  
public class CallOverriddenMethod {  
    public static void main(String[] args){  
        Father obj= new Father();  
        obj.m1();  
        obj.m2();  
        obj= new Son();  
        obj.m1();  
        obj.m2();  
    }  
}
```



# How Can Overridden Methods be Determined?



# Final

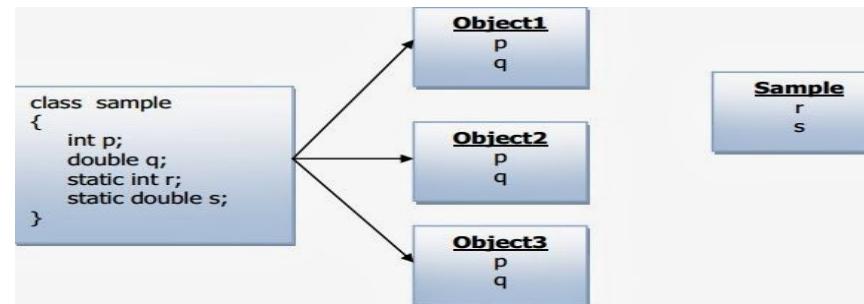
Final Variable  To create constant variables

Final Methods  Prevent Method Overriding

Final Classes  Prevent Inheritance

- If we don't use final class in java programs, it will still work. But that the class should not be inherited throughout the program. Or, maybe due to some reason we may want to restrict the extension of the class.

# Static



## Static Method

Static methods in Java belong to classes

## Static Class

nested class

# Java

member belongs to a type itself, rather than to an instance of that type.

## Static

## Static Block

It executes whenever the class is loaded in memory

## Static Variable

Memory allocation for a static variable happens only once in the class area when the class is loaded in the memory

# This

## Usage of java this keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

1

**this** can be used to refer current class instance variable.

2

**this** can be used to invoke current class method (implicitly)

3

**this()** can be used to invoke current class constructor.

4

**this** can be passed as an argument in the method call.

5

**this** can be passed as argument in the constructor call.

6

**this** can be used to return the current class instance from the method.

# Super

## Usage of Super Keyword

1

Super can be used to refer immediate parent class instance variable.

2

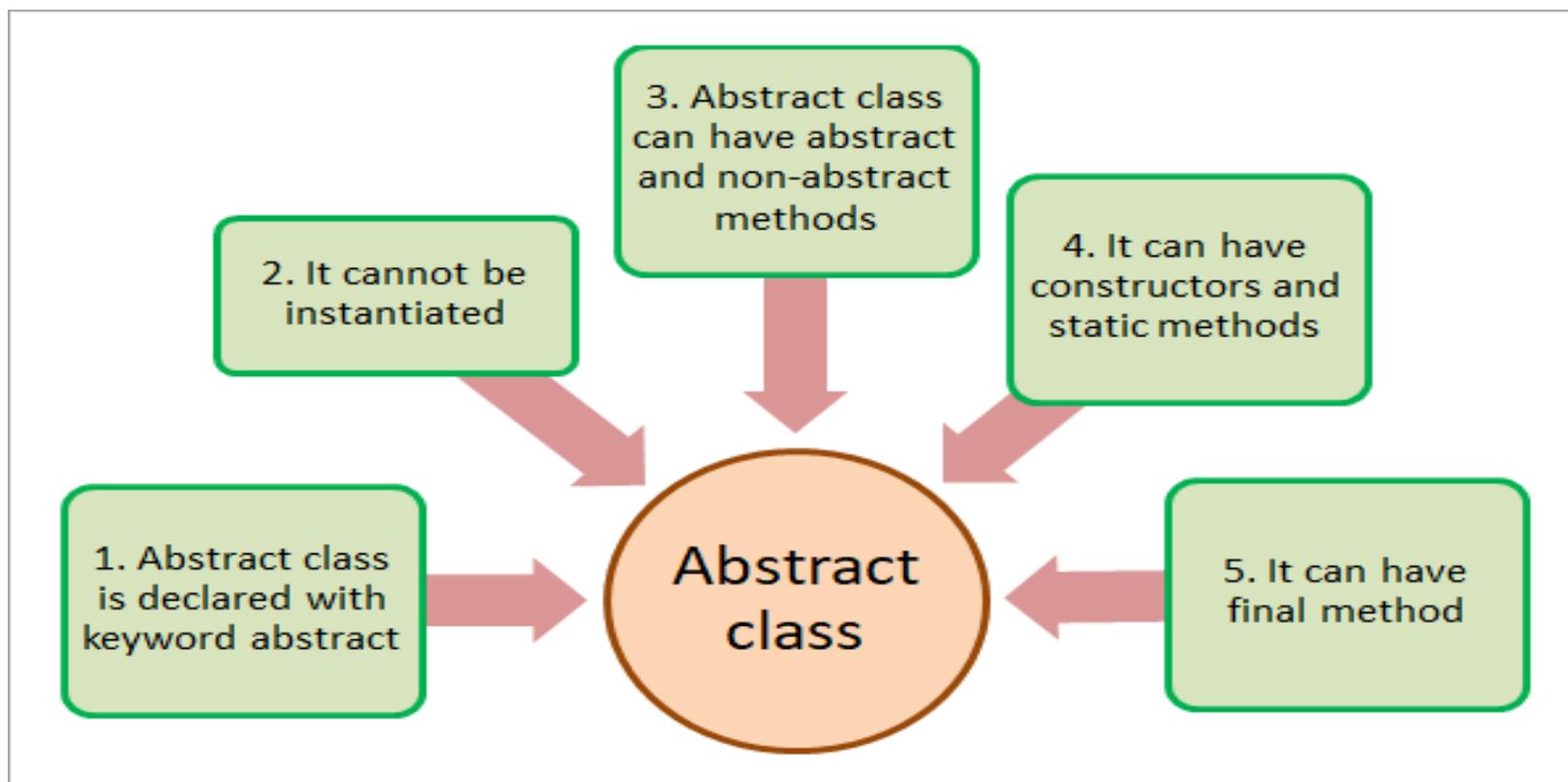
Super can be used to invoke immediate parent class method.

3

super() can be used to invoke immediate parent class constructor.

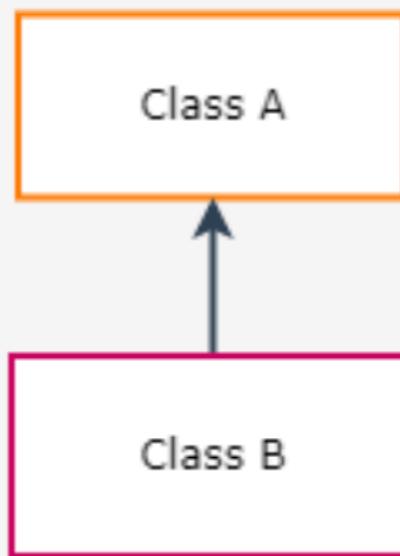
# Abstract

- Abstraction “displays” only the relevant attributes of objects and “hides” the unnecessary details.



# Extends

## Inheritance in Java



```
public class A {  
    ...  
}
```

```
public class B extends A {  
    ...  
}
```

# Interfaces

## Interface

- An interface **can** use **abstract** and **public** modifiers.
- The **abstract** modifier is **optional** for an Interface. All Interfaces are inherently abstract.
- Interfaces **without public access modifiers** are **visible within the package**.
- Interfaces **cannot** use **protected** and **private** access modifiers and **final**
- **Variables** declared in an Interface are **inherently public** and **final**. Using the **public** and **final** modifiers for the Interface variables is **optional**.
- Interfaces **cannot** declare **private** and **protected variables**.

## Abstract Class

- Abstract Class **can** use **abstract** and **public** modifiers.
- The **abstract** modifier **must** be used to declare a class as an Abstract Class.
- Abstract Class **without public access modifiers** remains **visible within the package**.
- Abstract Class **cannot** use **protected** and **private** access modifiers and **final** modifier.
- The Abstract Class variables must exclusively use **public** and **final** modifiers where required.
- The Abstract Class **can** declare **private** and **protected variables**.

# Interfaces

- Methods declared in an Interface are inherently public and abstract.
- Methods definition is not allowed in an Interface. The default and static methods are allowed in an Interface since Java 8.
- Interfaces cannot declare private and protected methods.
- An Interface can extend either single or multiple Interfaces using the keyword extends.
- An Interface cannot implement another Interface or extend Abstract Class.
- Interfaces are always 100% abstract with the exception of default and static methods since Java 8.
- Interfaces can be used when several classes are required to implement the same methods.
- Methods in an Abstract Class must be exclusively declared as public and abstract where required.
- Methods can be defined in an Abstract Class.
- Abstract Class can declare private and protected methods.
- An Abstract Class can extend only a single Abstract Class or Concrete Class.
- An Abstract Class can implement either single or multiple Interfaces using the keyword implements.
- An Abstract Class can be 100% abstract if all the methods declared within the class are abstract.
- Abstract Class can be used to partially implement the features of child classes.

# Interfaces

- Interfaces can be used for multiple inheritance.

**Example:**

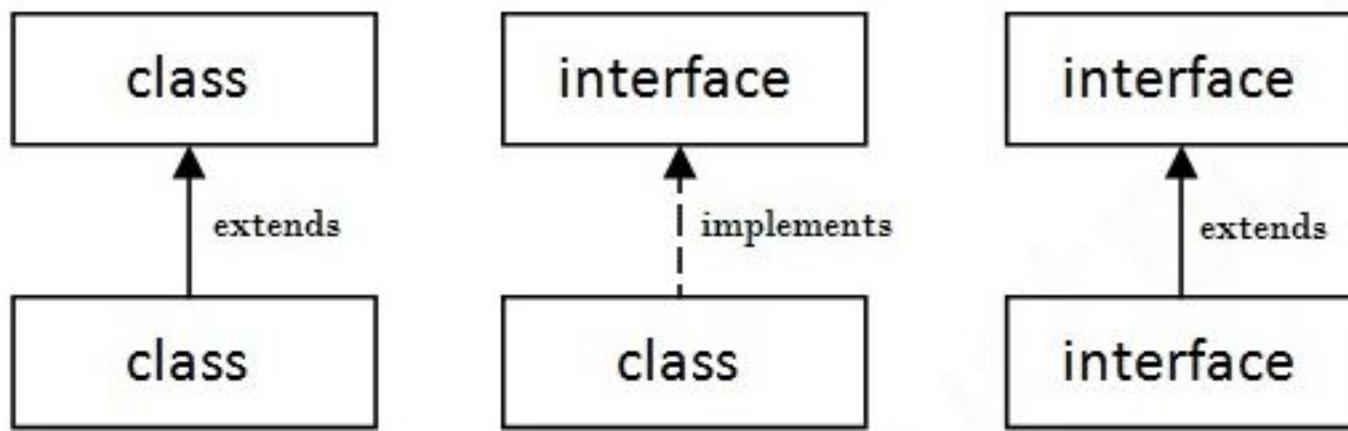
```
public interface A {  
  
    public final String DEFAULT_NAME = "Guest";  
  
    public abstract void printName(String name);  
}
```

- Abstract Class can extend another Abstract Class or Concrete Class and implement multiple Interfaces for multiple inheritance.

**Example:**

```
public abstract class CA {  
  
    public final String DEFAULT_NAME = "Guest";  
  
    public abstract void printName(String name);  
  
    public void printNumber(int number) {  
  
        System.out.println( number );  
    }  
}
```

# Implements



# Interfaces

- An *interface* is a reference type, similar to a class, that can contain *only* constants, initialized fields, static methods, prototypes (abstract methods, default methods), and nested types.
- It will be the **core** of some classes
- Interfaces cannot be instantiated because they have no-body methods.
- Interfaces can only be *implemented* by classes or *extended* by other interfaces.

# Interfaces...

```
1  public interface InterfaceDemo {
2      final int MAXN=100; // constant
3      int n=0; // Fields in interface must be initialized
4      static public int sqx(int x){ return x*x; }
5      public abstract void m1(); // abstract methods
6      abstract public void m2();
7      void m3(); // default methods
8      void m4();
9  }
10
11 class UseIt{
12     public static void main(String args[]){
13         InterfaceDemo obj= new InterfaceDemo();
14     }
15 }
```

# Interfaces...

```
public interface InterfaceDemo {  
    final int MAXN=100; // constant  
    int n=0; // Fields in interface must be initialized  
    static public int sqx(int x){ return x*x; }  
    public abstract void m1(); // abstract methods  
    abstract public void m2();  
    void m3(); // default methods  
    void m4();  
}  
  
class A implements InterfaceDemo{  
    // overriding methods  
    public void m1() { System.out.println("M1"); }  
    public void m2() { System.out.println("M2"); }  
    void m3() { System.out.println("M3"); }  
    void m4() { System.out.println("M4"); }  
}
```

m3(), m4() in A cannot implement m3(), m4() in InterfaceDemo, attempting to assign weaker access privileges, were public

Default methods of an interface must be overridden as public methods in concrete classes.

# Interfaces

...

```
public interface InterfaceDemo {  
    final int MAXN=100; // constant  
    int n=0; // Fields in interface must be initialized  
    static public int sqr(int x){ return x*x; }  
    public abstract void m1(); // abstract methods  
    abstract public void m2();  
    void m3(); // default methods  
    void m4();  
}
```

```
class A implements InterfaceDemo{  
    // overriding methods  
    public void m1() { System.out.println("M1"); }  
    public void m2() { System.out.println("M2"); }  
    public void m3() { System.out.println("M3"); }  
    public void m4() { System.out.println("M4"); }  
}
```

```
class UseIt{  
    public static void main(String args[]){  
        InterfaceDemo obj= new A();  
        obj.m1();  
        obj.m2();  
        obj.m3();  
        obj.m4();  
        int s= InterfaceDemo.sqr(5);  
        System.out.println("5x5=" + s);  
    }  
}
```

Output - FirstPrj (run) ×  
run:  
 M1  
 M2  
 M3  
 M4  
 5x5=25

# Interfaces

```
public interface BonusCalculator {  
    //variables are public static final  
    int BONUS = 2000;  
  
    //abstract method  
    void calcBonus(int x);  
  
    //default method  
    default void greetMessage(){  
        System.out.println("Have a good day");  
    }  
    //static method  
    static void check() {  
        System.out.println("Static method");  
    }  
}
```

# Abstract Classes

- Used to define ***what*** behaviors a class is required to perform without having to provide an explicit implementation.
- It is the result of so-high generalization
- Syntax to define a abstract class
  - *public abstract class className{ ... }*
- It isn't necessary for all of the methods in an abstract class to be abstract.
- An abstract class can also declare implemented methods.

# Abstract Classes...

```
1 package shapes;
2 public abstract class Shape {
3     abstract public double circumstance();
4     abstract public double area();
5 }
6 class Circle extends Shape {
7     double r;
8     public Circle (double rr) { r=rr; }
9     public double circumstance() { return 2*Math.PI*r; }
10    public double area() { return Math.PI*r*r; }
11 }
12 class Rect extends Shape {
13     double l,w;
14     public Rect(double ll, double ww) {
15         l = ll; w = ww;
16     }
17     public double circumstance() { return 2*(l+w); }
18     public double area() { return l*w; }
19 }
20 class Program {
21     public static void main(String[] args) {
22         Shape s = new Shape ();
23     }
24 }
```

```
20 class Program {
21     public static void main(String[] args) {
22         Shape s = new Circle(5);
23         System.out.println(s.area());
24     }
25 }
```

Output - Chapter06 (run)  
run:  
78.53981633974483

Modified

# Abstract Classes...

```
1  public abstract class AbstractDemo2 {  
2      void m1() // It is not abstract class  
3      { System.out.println("m1");  
4      }  
5      void m2() // It is not abstract class  
6      { // empty body  
7      }  
8      public static void main(String[] args)  
9      { AbstractDemo2 obj = new AbstractDemo2();  
10     }  
11 }
```

This class have no abstract method but it is declared as an abstract class. So, we can not initiate an object of this class.

# Abstract Classes...

Error.  
Why?

```
①  public abstract class AbstractDemo2 {  
②      void m1() // It is not abstract class  
3       { System.out.println("m1");  
4       }  
5       abstract void m2();  
6   }  
7   class Derived extends AbstractDemo2  
8   { public void m1() // override  
9       { System.out.println("m1");  
10      }  
11      public static void main(String[] args)  
12      { Derived obj = new Derived();  
13      }  
14  }
```

# Implementing Abstract Methods

- Derive a class from an abstract superclass, the subclass will inherit all of the superclass's features, all of **abstract methods** included.
- To replace an inherited abstract method with a concrete version, the subclass need merely override it.
- Abstract classes **cannot be instantiated**

# Class & Interface

Sr. No.	Key	Class	Interface
1	Supported Methods	A class can have both an abstract as well as concrete methods.	Interface can have only abstract methods. Java 8 onwards, it can have default as well as static methods.
	Multiple	Multiple Inheritance is not	Interface supports Multiple Inheritance

```
public class JavaTester {  
    public static void main(String args[ ]) {  
        Animal tiger = new Tiger();  
        tiger.eat();  
        Tiger tiger1 = new Tiger();  
        tiger1.eat();  
    }  
}  
interface Animal {  
    public void eat();  
}  
class Tiger implements Animal {  
    public void eat(){  
        System.out.println("Tiger eats");  
    }  
}
```

# Home Works

- Mô hình hóa yêu cầu sau vào máy tính.
  - Bộ giáo dục đã đề ra khung chương trình đào tạo cho ngành công nghệ thông tin như sau: Chương trình cần trải qua 3 quá trình:
    - Đào tạo cơ bản: dạy các môn lập trình cơ bản
    - Đào tạo nâng cao: dạy các môn về kiến trúc hệ thống, phân tích thiết kế hệ thống
    - Đào tạo chuyên ngành hẹp: đào tạo các chuyên ngành như: mobile, data, network
  - Có 3 trường đại học: FPT, Tôn Đức Thắng, Bách Khoa TPHCM: trường đại học sẽ mở ngành công nghệ thông tin, và phải triển khai theo đúng khung chương trình đào tạo ngành công nghệ thông tin của bộ giáo dục. Nhưng mỗi trường có 1 cách dạy khác nhau, dạy các môn học khác nhau.
  - Với vai trò là một cán bộ quản lý ở bộ giáo dục thì bạn hãy cho biết thông tin cụ thể của các môn học sẽ được giải dạy theo khung chương trình của các trường đại học : FPT, Tôn Đức Thắng, Bách Khoa - HCM. Hiện thực thông qua Interface và Abstract.

# Class & Abstract

- Why?
- What?
- Where?

# Interface & Abstract

- Why?
- What?
- Where?

# Interface & Abstract

- Why?
- What?
- Where?

# Anonymous Classes

**Anonymous classes** are classes which are not named but they are identified automatically by Java compiler.

**Where are they?** They are identified at initializations of interface/abstract class object but abstract methods are implemented as attachments.

**Why are they used?**

- Enable you to make your code more concise.
- Enable you to declare and instantiate a class at the same time.
- They are like local classes except that they do not have a name.
- Use them if you need to use a local class only once.

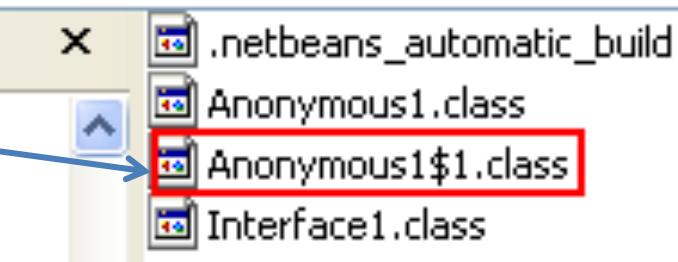
# Anonymous Class...

```
1 // New - Java Interface
2 public interface Interface1 {
3     void M1();
4     void M2();
5 }
6 class Anonymous1{
7     public static void main(String[] args) {
8         Interface1 obj = new Interface1() {
9             public void M1()
10            { System.out.println("M1"); }
11            public void M2()
12            { System.out.println("M2"); }
13        };
14        obj.M1();
15        obj.M2();
16    }
17 }
18 }
```

Class name is given by the compiler:  
**ContainerClass\$Number**

Anonymous class.

Chapter06\build\classes



Output - Chapter06 (run)

run:  
M1  
M2

BUILD SUCCESSFUL (total time: 0 seconds)

# Anonymous Class...

```
1 package adapters;
2 // abstract class contains all concrete methods
3 public abstract class MyAdapter {
4     public void M1() { System.out.println("M1"); }
5     public void M2() { System.out.println("M2"); }
6 }
7 class Program {
8     public static void main(String[] args) {
9         // Overriding one method
10        MyAdapter obj = new MyAdapter () {
11            public void M1()
12            {
13                System.out.println("M1 overridden");
14            }
15            obj.M2 ();
16            obj.M1 ();
17        }
18    }
}
```

Output - Chapter06 (run)

run:  
M2  
M1 overridden

Concrete methods but they can not be used because the class is declared as abstract one.

The abstract class can be used only when at least one of its methods is overridden

Anonymous class is a technique commonly used to support programmer when only some methods are overridden especially in event programming.

# Enum Types (1)

- An *enum type* is a special data type that enables for a variable to be a set of predefined constants.
- We use enum types any time you need to represent a fixed set of named-constants (uppercase).

New/ Java Enum

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY; // ; can be missed  
}
```

# Enum Types (2)

```
class DayInWeek{  
    Day d; // Using the enum Day  
    DayInWeek(Day d){  
        this.d= d;  
    }  
    public void tellItLikeItIs() {  
        switch (d) {  
            case MONDAY:  
                System.out.println("Mondays are bad.");  
                break;  
            case FRIDAY:  
                System.out.println("Fridays are better.");  
                break;  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println("Weekends are best.");  
                break;  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```

Output - FirstPrj (run) ×

```
run:  
Mondays are bad.  
Midweek days are so-so.
```

```
class UseDay {  
    public static void main(String[] args){  
        DayInWeek obj= new DayInWeek(Day.MONDAY);  
        obj.tellItLikeItIs();  
        obj= new DayInWeek(Day.WEDNESDAY);  
        obj.tellItLikeItIs();  
    }  
}
```

# Enum Types (3)

```
public enum Planet { // Enum for some planets (mass, radius)
    MERCURY (3.303e+23, 2.4397e6), VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6), MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7), SATURN (5.688e+26, 6.0268e7),
    URANUS (8.686e+25, 2.5559e7), NEPTUNE (1.024e+26, 2.4746e7);

    // Fields of a Planet object
    private final double mass; // in kilograms
    private final double radius; // in meters
    // Create a planet, onstructor, public can not be used
    Planet(double mass, double radius) {
        this.mass = mass; // initialize mass
        this.radius = radius; // initialize radius
    }
    private double mass() { return mass; }
    private double radius() { return radius; }
    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

# Enum Types (4)

```
26 class PlanetUse{  
27     public static void main(String[] args) {  
28         double yourWeightOnEarth = 60; // kg  
29         double mass = yourWeightOnEarth/Planet.EARTH.surfaceGravity();  
30         for (Planet p : Planet.values())  
31             System.out.printf("Your weight(kg) on %s is %f%n",  
32                             p, p.surfaceWeight(mass));  
33     }  
34 }
```

The screenshot shows a Java code editor and an output window. The code in the editor demonstrates how to use an enum to calculate a person's weight on various planets based on their weight on Earth. The output window shows the results of running the program, displaying the calculated weight for each planet from Mercury to Neptune.

**Output - FirstPrj (run) ×**

```
run:  
Your weight(kg) on MERCURY is 22.665457  
Your weight(kg) on VENUS is 54.299946  
Your weight(kg) on EARTH is 60.000000  
Your weight(kg) on MARS is 22.724231  
Your weight(kg) on JUPITER is 151.833452  
Your weight(kg) on SATURN is 63.960932  
Your weight(kg) on URANUS is 54.307632  
Your weight(kg) on NEPTUNE is 68.299684
```

# Test

```
1  /* What is the output of the following program */
2  class Study_1A{
3      void M() { System.out.println("A");}
4  }
5  class Study_1B extends Study_1A{
6      void M() { System.out.println("B"); }
7  }
8  class Study_1C{
9      void M() { System.out.println("C"); }
10 }
11 public class Study_1 {
12     public static void main(String[] args) {
13         Study_1A obj= new Study_1A();
14         obj.M();
15         obj=new Study_1B();
16         obj.M();
17         obj= new Study_1C();
18         obj.M();
19     }
20 }
```

a) ABC

b) AAC

c) ABA

d) Compile-time  
error

Study\_1A and Study\_1C are  
inconvertible

# Test

```
/* What is the output of the following program */  
  
class Study_1A{  
    void M() { System.out.println("A");}  
}  
  
class Study_1B extends Study_1A{  
    void M() { System.out.println("B"); }  
}  
  
class Study_1C{  
    void M() { System.out.println("C"); }  
}  
  
public class Study_1 {  
    public static void main(String[] args) {  
        Object obj= new Study_1A();  
        obj.M();  
        obj=new Study_1B();  
        obj.M();  
        obj= new Study_1C();  
        obj.M();  
    }  
}
```

a) ABC

b) AAC

c) ABA

d) Compile-time  
error

The java.lang.Object class  
does not have the M()  
method

# Test

```
/* What is the output of the following program */
class Study_1A{
    void M() { System.out.print("A"); }
}
class Study_1B extends Study_1A{
    void M() { System.out.print("B"); }
}
class Study_1C{
    void M() { System.out.print("C"); }
}
public class Study_1 {
    public static void main(String[] args)  {
        Study_1A obj= new Study_1A();
        obj.M();
        obj=new Study_1B();
        obj.M();
        Object obj2= new Study_1C();
        ((Study_1A)obj2).M();
    }
}
```

a) ABC

b) AAA

c) ABA

d) None of the others

AB and a ClassCastException

# Test

```
/* What is the output of the following program */
class Study_1A{
    void M() { System.out.print("A"); }
}
class Study_1B extends Study_1A{
    void M() { System.out.print("B"); }
}
class Study_1C extends Study_1B {
    void M() { System.out.print("C"); }
}
public class Study_1 {
    public static void main(String[] args) {
        Study_1A obj= new Study_1A();
        obj.M();
        obj=new Study_1B();
        obj.M();
        obj= new Study_1C();
        obj.M();
    }
}
```

a) AAA

b) ACB

c) None of the others

d) ABC

# Test

```
/* What is the output of the following program */
class Study_1A{
    void M() { System.out.print("A"); }
}
class Study_1B extends Study_1A{
    void M() { System.out.print("B"); }
}
class Study_1C extends Study_1B {
    void M() { System.out.print("C"); }
}
public class Study_1 {
    public static void main(String[] args)  {
        Study_1C obj= new Study_1C();
        obj.M();
        obj=new Study_1B();
        obj.M();
        obj= new Study_1A();
        obj.M();
    }
}
```

a) ABC

b) AAA

c) ABA

d) None of the others

Compile-time error  
( Type conformity violation)

# Test

```
public class Study_2 {  
    static int N = 10;  
    int x = 120;  
    static{  
        N = 50;  
        System.out.print("A");  
    }  
    public void M(){  
        System.out.print(x);  
    }  
    public static void main(String [] args){  
        Study_2 obj = new Study_2();  
        obj.M();  
    }  
}
```

a) 120

b) 120A

c) None of the  
others

d) A120

# Test

```
public class Study_2 {  
    static int N = 10;  
    int x = 120;  
    static {  
        N = 7;  
        System.out.print("A" + N);  
        x = 500;  
    }  
    public void M() {  
        System.out.print( x );  
    }  
    public static void main(String [] args) {  
        Study_2 obj = new Study_2();  
        obj.M();  
    }  
}
```

a) A7500

b) 500A7

c) 500

d) None of the others

Compile-time error  
( static code can not access  
instance variables)

# Test

```
public class Study_2 {  
    static int N = 2;  
    int x = 10;  
    static{  
        N = 5;  
        int y = 7;  
        System.out.print("A" + (N + y) );  
    }  
    public void M(){  
        System.out.print( x + y );  
    }  
    public static void main(String [] args){  
        Study_2 obj = new Study_2();  
        obj.M();  
    }  
}
```

a) A1210

b) 10A12

c) 17

d) None of the others

Compile-time error  
( The y variable is out of scope)

# Summary

- Benefits of OO implementation:  
Inheritance and Polymorphism
- Working with Interfaces.
- Working with Abstract Methods and Classes.
- Anonymous Classes
- Enum Type