

Session 03

Classes and Objects

(<http://docs.oracle.com/javase/tutorial/java/javaOO/index.html>)

Objectives

- 1-Programming Paradigms
- 2-OOP basic concepts
- 3-How to identify classes
- 4-How to declare/use a class
- 5-Common modifiers (a way to hide some members in a class)
- 6-Controlling access to members of a class using modifiers
- 7-Overriding methods in sub-classes
- 8-Nested classes

9-Benefits of OO implementation:
inheritance, polymorphism

10-Working with interfaces

11-Working with Abstract methods and
classes

12-Enum type

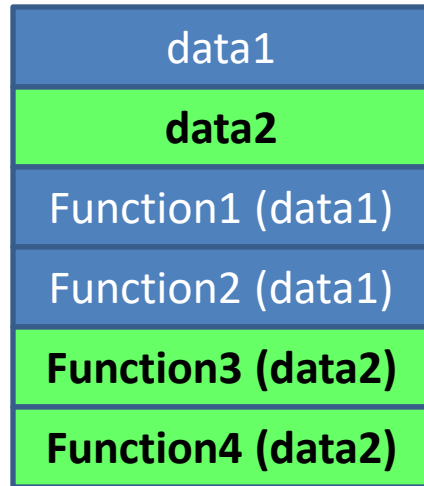
Programming Paradigms

- High-level programming languages (from 3rd generation languages) are divided into (Wikipedia):

Paradigm	Description
Procedural-oriented (imperative) paradigm-POP (3 rd generation language)	Program= data + algorithms. Each algorithm is implemented as a function (group of statements) and data are it's parameters (C-language)
Object-oriented paradigm (OOP) (3 rd generation language)	Programs = actions of some objects. Object = data + behaviors. Each behavior is implemented as a method (C++, Java, C#,...)
Functional paradigm (4 th generation language)	Domain-specific languages. Basic functions were implemented. Programs = a set of functions (SQL)
Declarative/Logic paradigm (5 th generation language)	Program = declarations + inference rules (Prolog, CLISP, ...)

Programming Paradigms: POP vs. OOP

Procedure-Oriented Program



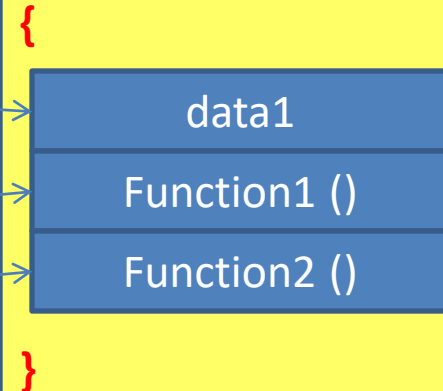
Object = Data + Methods

Basic Concepts

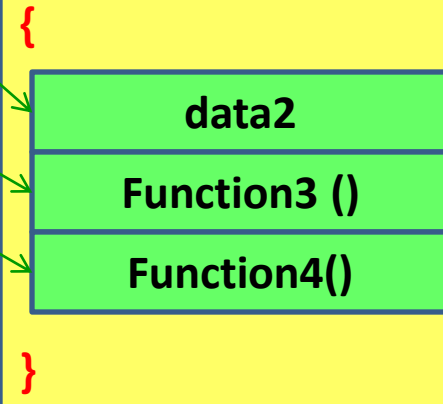
- Encapsulation
- Inheritance
- Polymorphism

Particular
methods:
Constructors

Class A



Class B



Common
methods
for
accessing a
data field:

Type getField()
void setField (Type newValue)

OOP Concepts

- Encapsulation
- Inheritance
- Polymorphism

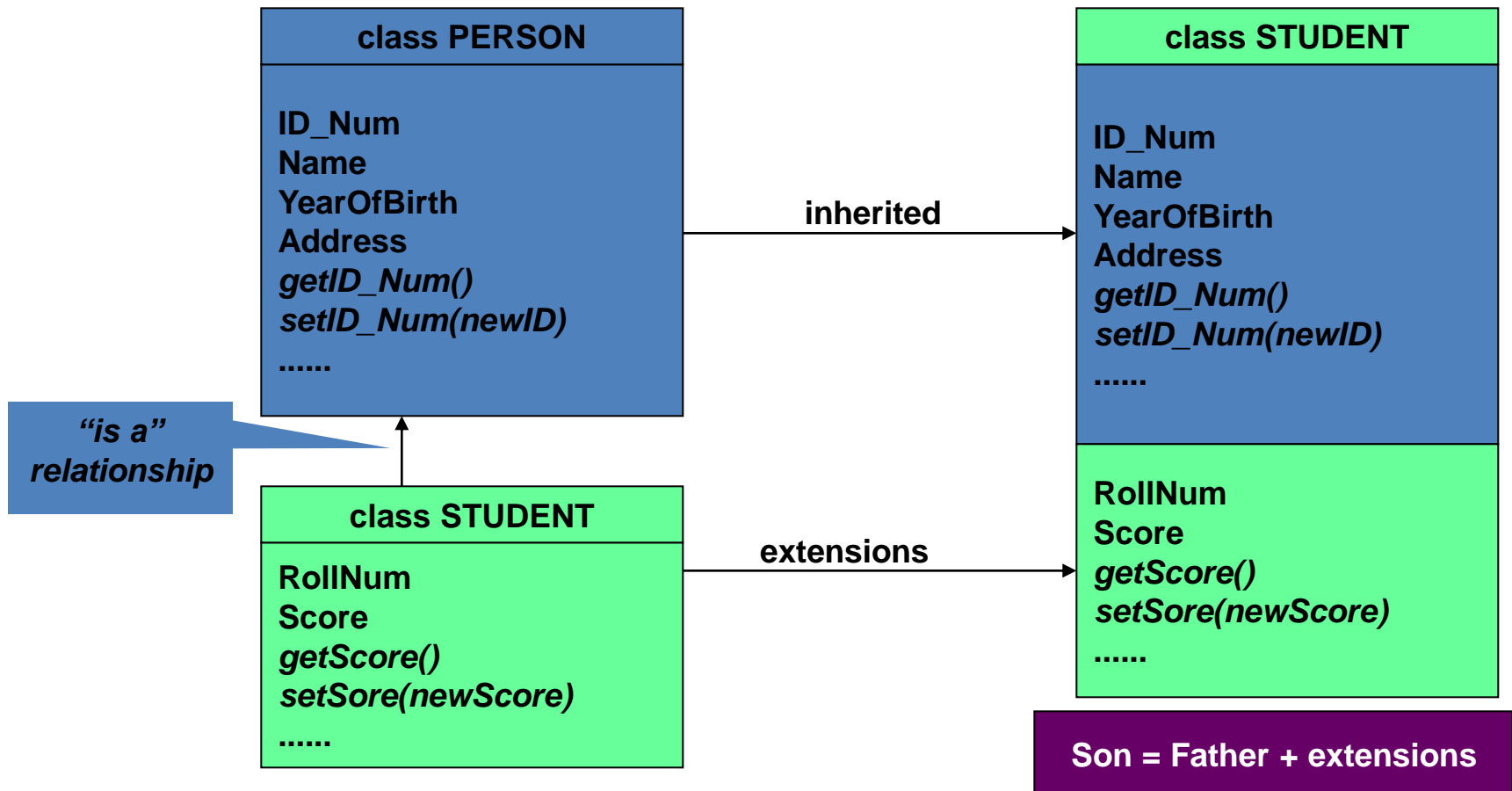
OOP Concepts: Encapsulation

Aggregation of data and behavior.

- Class = Data (fields/properties) + Methods
- Data of a class should be hidden from the outside.
- All behaviors should be accessed only via methods.
- A method should have a *boundary condition*: Parameters must be checked (use if statement) in order to assure that data of an object are always valid.
- **Constructor**: A special method it's code will execute when an object of this class is initialized.

OOP Concepts: Inheritance - 1

Ability allows a class having members of an existed class → Re-used code, save time

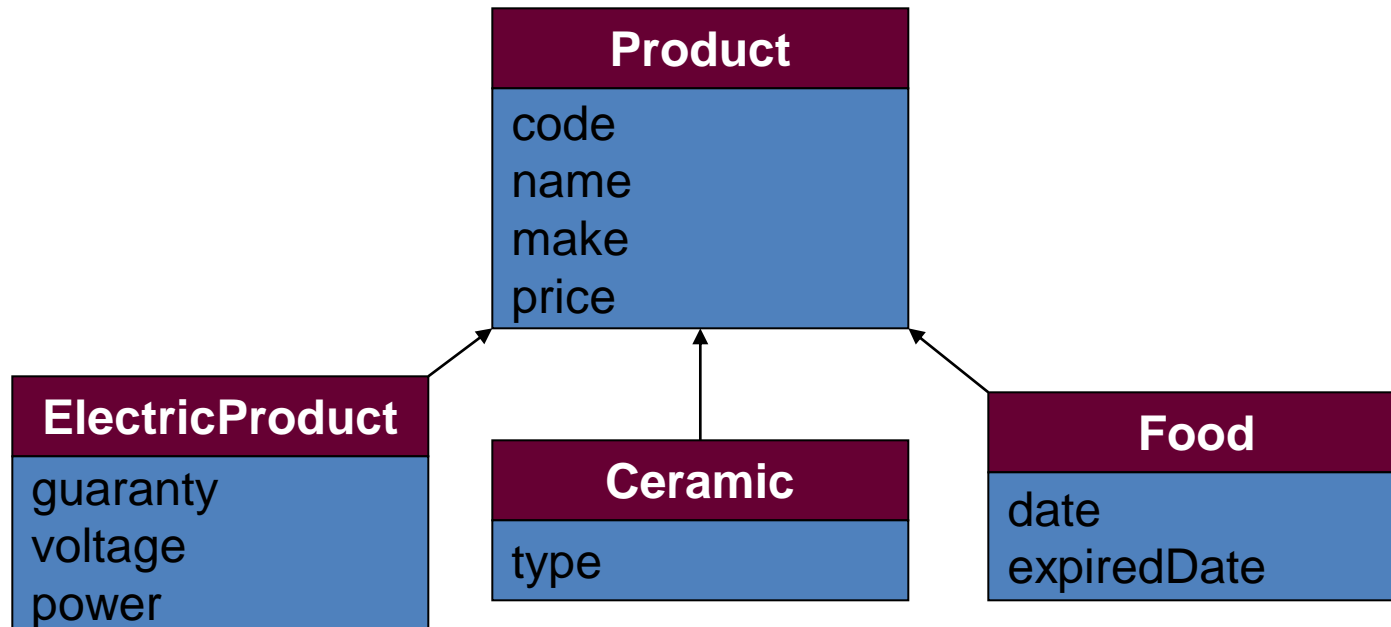


OOP Concepts: Inheritance - 2

How to detect father class?

Finding the intersection of concerned classes.

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



OOP Concepts: **Polymorphism**

Ability allows many versions of a method based on overloading and overriding methods techniques.

Overloading: A class can have some methods which have the same name but their parameter types are different.

Overriding: A method in father class can be overridden in it's derived classes (body of a method can be replaced in derived classes).

How to Identity a Class

- Main noun: Class
- Nouns as modifiers of main noun: Fields
- Verbs related to main noun: Methods

Vehicle has attributes of manufacturer, manufacture year, cost, color

Write a Java program that will allow **input** a **Vehicle**, **output** its.

```
class Vehicle {  
    String manufacturer;  
    int manufacture year;  
    double cost;  
    String color;  
    Vehicle input() {  
        <code>  
    }  
    void output() {  
        <code>  
    }  
}
```

Declaring/Using a Java Class

```
[public] class ClassName [extends FatherClass] {  
    [modifier] Type field1 [= value];  
    [modifier] Type field2 [= value];  
    // constructor  
    [modifier] ClassName (Type var1,...) {  
        <code>  
    }  
    [modifier] methodName (Type var1,...) {  
        <code>  
    }  
    .....  
}
```

The diagram illustrates the components of a Java class declaration. A red arrow points from the opening curly brace of the class to a red box. Another red arrow points from the closing curly brace to a red box. A blue arrow points from the class name 'ClassName' to a blue box. A green arrow points from the constructor's opening curly brace to a blue box. A green arrow points from the constructor's body '<code>' to a blue box. A green arrow points from the method's opening curly brace to a blue box. A green arrow points from the method's body '<code>' to a blue box. A green arrow points from the method's closing curly brace to a blue box. A green arrow points from the ellipsis '.....' to a blue box.

Modifiers will be introduced later.

How many constructors should be implemented? → Number of needed ways to initialize an object.

What should we will write in constructor's body? → They usually are codes for initializing values to descriptive variables

Defining Constructors

- Constructors that are invoked to create objects from the class **blueprint**.
- Constructor declarations look like method declarations—except that they use the **name of the class** and have **no return type**.
- The compiler automatically provides a no-argument, default constructor for any class **without** constructors.

Defining Methods

- Typical method declaration:

```
[modifier] ReturnType methodName (params) {  
    <code>  
}
```

- Signature: data help identifying something
- Method Signature:
 name + order of parameter types

Passing Arguments a Constructor/Method

- Java uses the mechanism passing by value. Arguments can be:
 - Primitive Data Type Arguments
 - Reference Data Type Arguments (objects)

Vehicle Management System

- Code a class named **Car**, **Motor**, **Truck** are directly derived from base class **Vehicle** object.


```
public class Vehicle { //super class
    private String manufacturer;
    private int year;
    private double cost;
    private String color;
    public Vehicle() {}
    public Vehicle(String manufacturer, int
year,double cost, String color){
        this.manufacturer=manufacturer;
        this.year=year;
        this.cost=cost;
        this.color=color;
    }
}
```

```
public String getManufacturer() {  
    return manufacturer;}  
    public void setManufacturer(String  
manufacturer) {  
        this.manufacturer = manufacturer;}  
    public int getYear() {  
        return year;}  
    public void setYear(int year) {  
        this.year = year;}  
    public double getCost() {  
        return cost;}  
    public void setCost(double cost) {  
        this.cost = cost;}
```

```
public String getColor() {  
    return color;  
}  
public void setColor(String color) {  
    this.color = color;  
}  
public String toString() {  
    return  
manufacturer+"\t"+year+"\t"+cost+"\t"+color;  
}  
}
```

Creating Objects

- Class provides the blueprint for objects; you create an object from a class.
 - `Point p = new Point(23, 94);`
- Statement has three parts:
 - **Declaration**: are all variable declarations that associate a variable name with an object type.
 - **Instantiation**: The new keyword is a Java operator that creates the object (memory is allocated).
 - **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object (values are assigned to fields).

Type of Constructors

Create/Use an object of a class

- ***Default constructor***: Constructor with no parameter.
- ***Parametric constructor***: Constructor with at least one parameter.

- **Create an object**

ClassName obj1=new ClassName();

ClassName obj2=new ClassName(params);

- **Accessing a field of the object**

object.field

- **Calling a method of an object**

object.method(params)

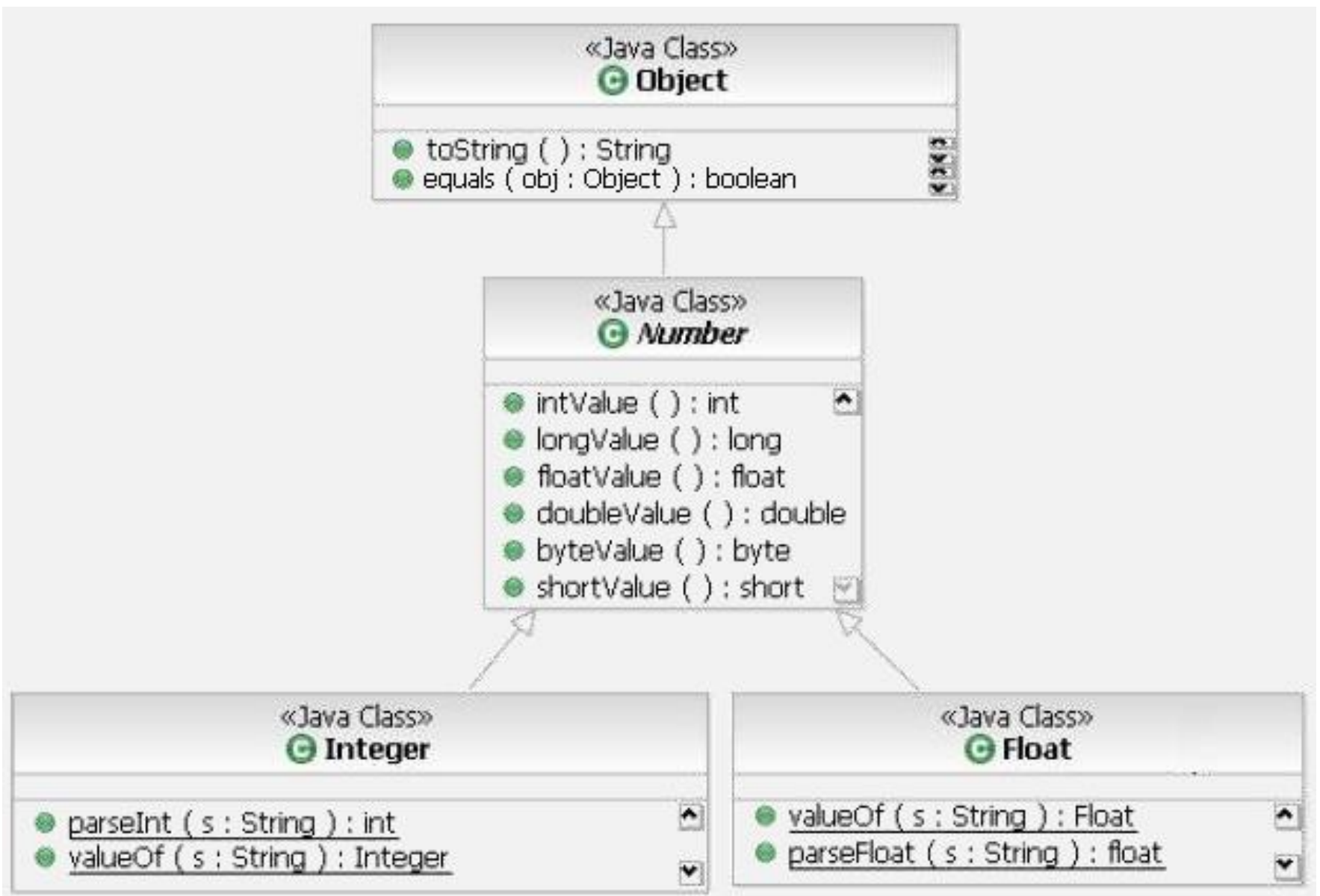
Notes about Constructors

- If we do not implement any constructor, compiler will insert to the class a system default constructor.
- If we implement **a** constructor, compiler does **not** insert default constructor.

Object class

- The **java.lang.Object** class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
- **Class constructors: Object()** This is the Single Constructor.

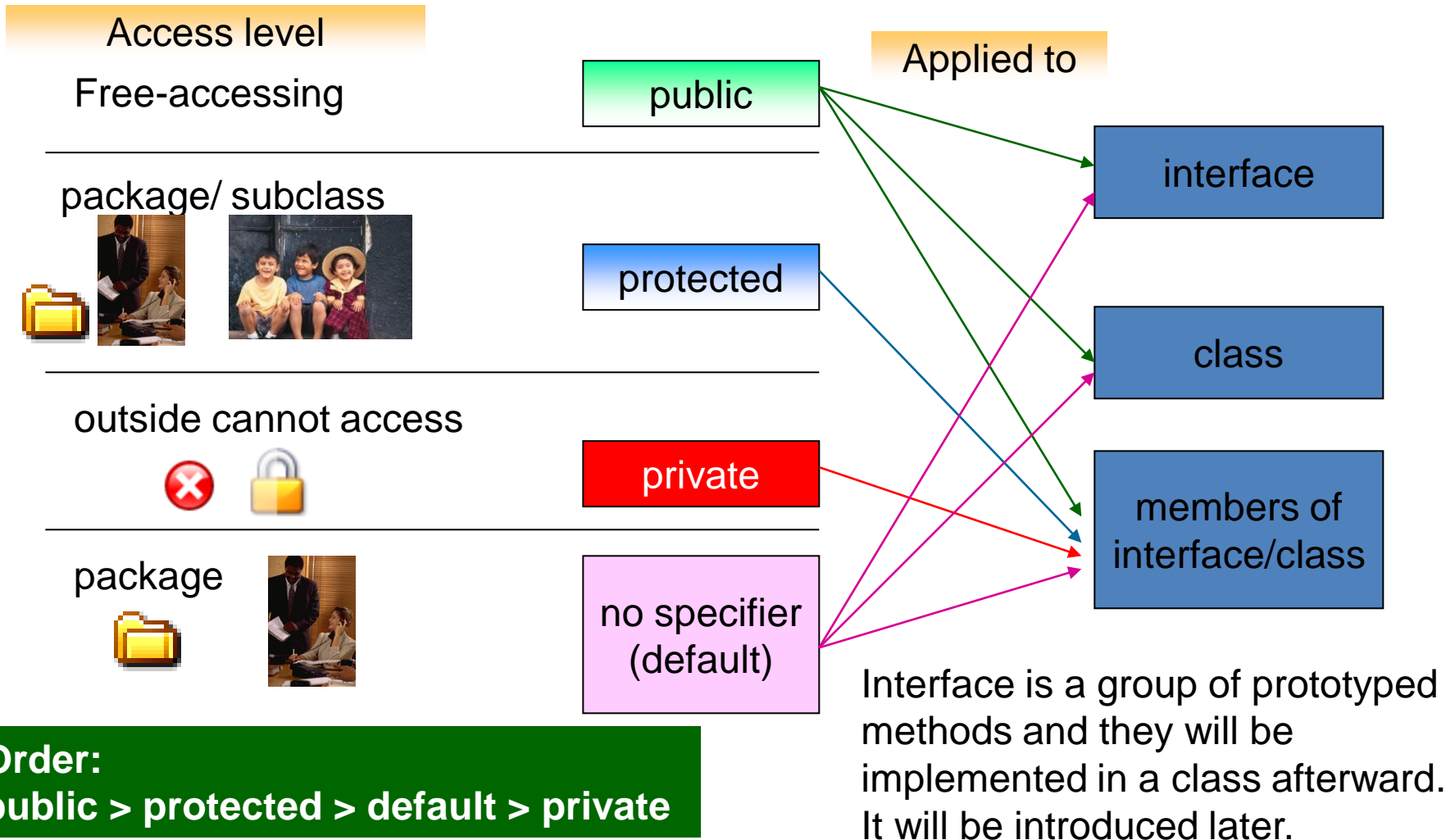
- String **toString()** This method returns a string representation of the object.
- boolean **equals(Object obj)** This method indicates whether some other object is "equal to" this one.
- int **hashCode()** This method returns a hash code value for the object.
- Class **getClass()** This method returns the runtime class of this Object.



Common Modifiers - 1

- Modifier (linguistics) is a word which can bring out the meaning of other word (adjective → noun, adverb → verb)
- Modifiers (OOP) are keywords that give the compiler information about the nature of code (methods), data, classes.
- Java supports some modifiers in which some of them are common and they are called as access modifiers (`public`, `protected`, `default`, `private`).
- Common modifiers will impose level of accessing on
 - class (where it can be used?)
 - methods (whether they can be called or not)
 - fields (whether they may be read/written or not)

Common Modifiers - 2



Access Level

Modifier	Class	Same Package	Subclass-Outside package	World
private	Y	N	N	N
No (default)	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Tips on Choosing an Access Level

- Use the **most restrictive** access level that makes **sense for a particular member**.
Use private unless you have a good reason not to.
- **Avoid public fields** except for constants.
Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Overloading Method

```
/* Overloading methods Demo. */
```

```
public class Box {
```

```
    int length=0;
```

```
    int width=0;
```

```
    int depth=0;
```

```
    // Overloading constructors
```

```
    public Box(){
```

```
    }
```

```
    public Box(int l){
```

```
        length = l>0? l: 0; // safe state
```

```
    }
```

```
    public Box(int l, int w){
```

```
        length = l>0? l: 0; // safe state
```

```
        width = w>0? w: 0;
```

```
    }
```

```
    public Box(int l, int w, int d){
```

```
        length = l>0? l: 0; // safe state
```

```
        width = w>0? w: 0;
```

```
        depth = d>0? d: 0;
```

```
    }
```

```
    // Overloading methods
```

```
    public void setEdge (int l,int w){
```

```
        length = l>0? l: 0; // safe state
```

```
        width = w>0? w: 0;
```

```
    }
```

```
    public void setEdge (int l,int w,int d){
```

```
        length = l>0? l: 0; // safe state
```

```
        width = w>0? w: 0;
```

```
        depth = d>0? d: 0;
```

```
    }
```

```
    public void output(){
```

```
        String S= "[" + length + "," + width
```

```
                + "," + depth + "];
```

```
        System.out.println(S);
```

```
    }
```

```
    /* Use the class Box */
```

```
    public class BoxUse {
```

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

```
            Box b= new Box();
```

```
            b.output();
```

```
            b.setEdge(7,3);
```

```
            b.output();
```

```
            b.setEdge(90,100,75);
```

```
            b.output();
```

```
        }
```

```
    }
```

Output - FirstPrj (run) x



run:



[0,0,0]



[7,3,0]

[90,100,75]

Access Modifier Overridden

Projects Services

Chapter02

- Source Packages
 - boxPkg
 - Box.java
 - overridenDemo
 - ClassA.java
 - ClassB.java
 - rectPkg
 - Rectangle.java

ClassA.java

```
1 package overridenDemo;
2 public class ClassA {
3     public void M1() { }
4     protected void M2() { }
5     void M3() { }
6     private void M4() { }
7 }
8
```

ClassB.java

```
1 package overridenDemo;
2 public class ClassB extends ClassA {
3     protected void M1() { }
4     void M2() { }
5     private void M3() { }
6     void M4() { }
7 }
```

Legal

private

default

protected

public

Illegal

public

protected

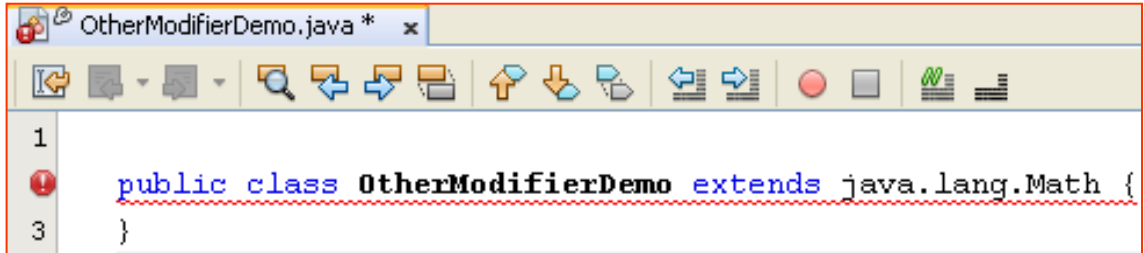
default

private

The sub-class must be more opened than it's father

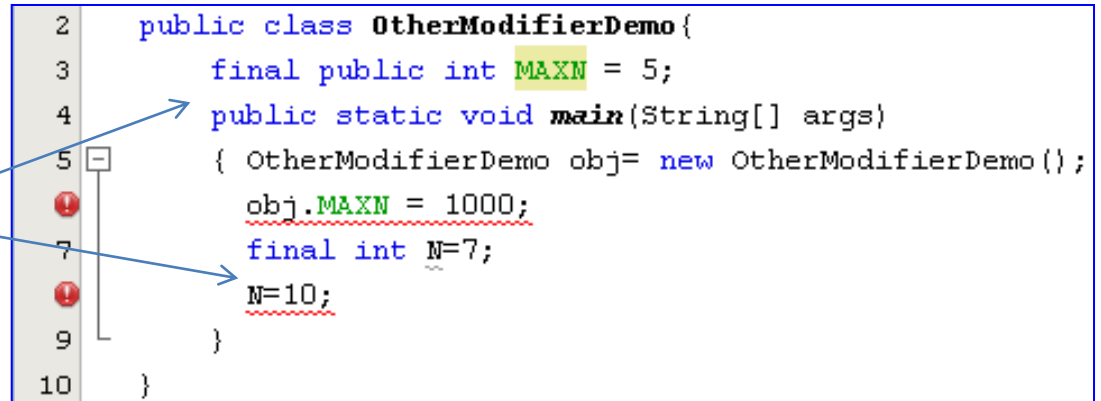
Modifier *final*

- **Final class:**
Class can not have sub-class
- **Final data** is a constant.
- **Final method:**
a method can not be overridden.



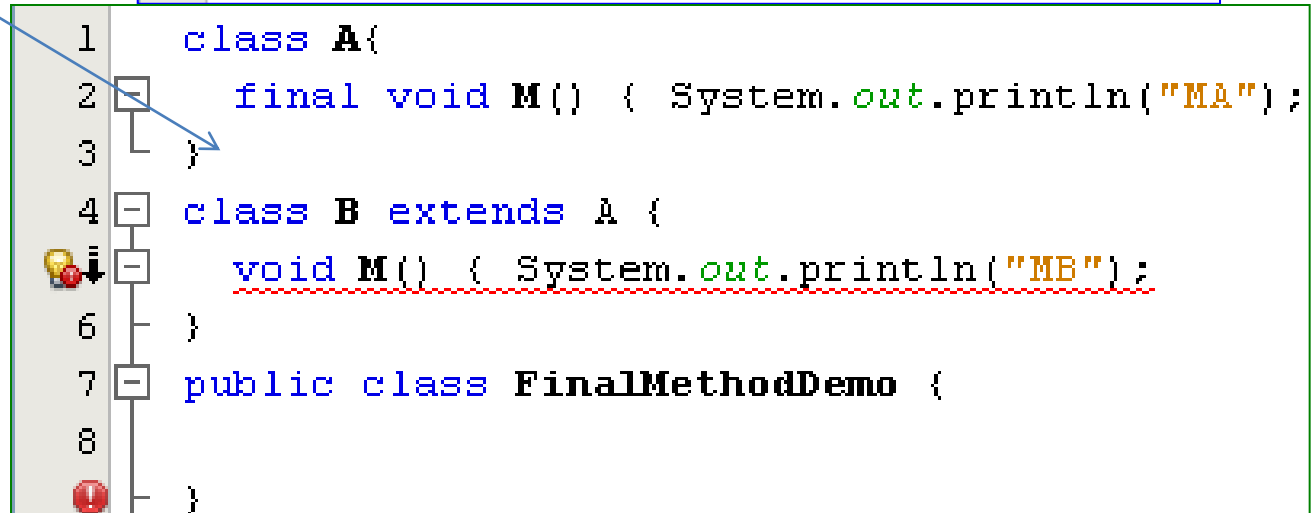
```
1 public class OtherModifierDemo extends java.lang.Math {
3 }
```

This screenshot shows the first two lines of a Java file named 'OtherModifierDemo.java'. Line 1 defines the class 'OtherModifierDemo' as 'public' and 'extends java.lang.Math'. Line 3 is an opening curly brace. A red arrow points from the 'Final class' bullet point to the 'public class' line.



```
2 public class OtherModifierDemo{
3     final public int MAXN = 5;
4     public static void main(String[] args)
5     { OtherModifierDemo obj= new OtherModifierDemo();
6       obj.MAXN = 1000;
7       final int N=7;
8       N=10;
9     }
10 }
```

This screenshot shows the 'main' method of the 'OtherModifierDemo' class. Line 3 declares a 'final public int MAXN = 5;'. Line 6 shows 'obj.MAXN = 1000;'. Line 7 declares a 'final int N=7;'. Line 8 shows 'N=10;'. Blue arrows point from the 'Final data' bullet point to the 'final public int MAXN' and 'final int N' declarations.



```
1 class A{
2     final void M() { System.out.println("MA");
3 }
4 class B extends A {
5     void M() { System.out.println("MB");
6 }
7 public class FinalMethodDemo {
8 }
9 }
```

This screenshot shows three classes. Class 'A' has a 'final void M()' method. Class 'B' extends 'A' and has its own 'void M()' method. Class 'FinalMethodDemo' is a public class. A green arrow points from the 'Final method' bullet point to the 'final void M()' method in class 'A'.

Modifier *static*

Class variable/ Object variable

- Object variable: Variable of each object
- Class Variable: A variable is shared in all objects of class. It is stored separately. It is declared with the modifier ***static***
- Class variable is stored separately. So, it can be accessed as:

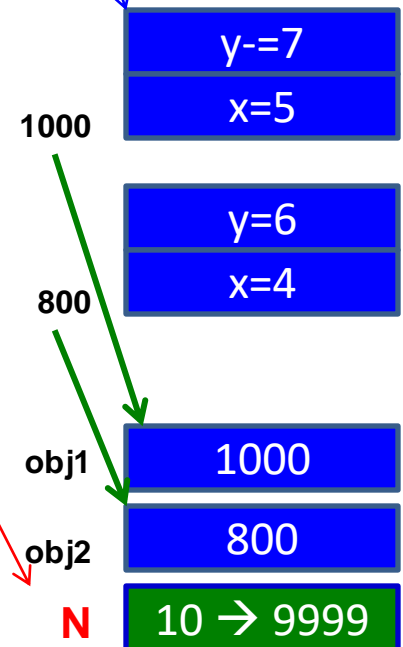
object.staticVar

ClassName.staticVar

Modifier *static*: Class variable/ Object variable

```
public class StaticVarDemo {  
    static int N=10; // class variable  
    int x, y; // object variable  
    public StaticVarDemo(int xx, int yy){  
        x= xx; y=yy;  
    }  
    public void setN( int nn){  
        N= nn;  
    }  
    public void output(){  
        System.out.println(N + "," + x + "," + y);  
    }  
}
```

```
public class StaticVarDemoUse {  
    public static void main(String args[]){  
        StaticVarDemo obj1= new StaticVarDemo(5,7);  
        StaticVarDemo obj2= new StaticVarDemo(4,6);  
        obj1.output();  
        obj2.output();  
        obj1.setN(9999);  
        obj1.output();  
        obj2.output();  
        System.out.println(StaticVarDemo.N);  
    }  
}
```



Output - FirstPrj (run) x

```
run:  
10,5,7  
10,4,6  
9999,5,7  
9999,4,6  
9999
```

Modifier *static*:

static code – Free Floating Block

```
public class StaticCodeDemo {  
    public static int N=10;  
    int x=5, y=7;  
    static {  
        System.out.println("Static code:" + N);  
    }  
    int sum(){  
        return x+y;  
    }  
    static {  
        System.out.println("Static code: Hello");  
    }  
}
```

All static code run only one time when the first time the class containing them is accesses

```
public class StaticCodeDemoUse {  
    public static void main(String args[]){  
        System.out.println(StaticCodeDemo.N);  
        StaticCodeDemo obj= new StaticCodeDemo();  
        System.out.println(obj.sum());  
    }  
}
```

The second access

```
Output - FirstPrj (run) x  
run:  
Static code:10  
Static code: Hello  
10  
12
```

Modifier *static*: Static method

- It is called as class method/global method and it is called although no object of this class is created.
- Entry point of Java program is a static method
- Syntax for calling it: **ClassName.staticMethod(args)**
- ***Static methods***:
 - can access class variables and class methods directly **only**.
 - **cannot** access instance variables or instance methods directly—they must use an object reference.
 - **cannot** use the `this` keyword as there is no instance for this to refer to

What should be static in a class?

- Constants:
 - The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

*static final double PI =
3.141592653589793;*

Methods with Arbitrary Number of Arguments

A group is treated as an array
group.length → number of elements
group[i]: The element at the position i

```
1 public class ArbitraryDemo {
2     public double sum(double... group){
3         double S=0;
4         for (double x: group) S+=x;
5         return S;
6     }
7     public String concat(String... group){
8         String S="";
9         for (String x: group) S+=x + " ";
10        return S;
11    }
12    public static void main(String[] args){
13        ArbitraryDemo obj= new ArbitraryDemo();
14        double total= obj.sum(5.4, 3.2, 9.08, 4);
15        System.out.println(total);
16        String line = obj.concat("I", "love", "you", "!");
17        System.out.println(line);
18    }
19 }
```

Output - FirstPrj (run) x



run:
21.68
I love you !

Example: a vehicle

```
public class SingleVehicle {  
    public Vehicle input() {  
        Vehicle v;  
        String manufacturer;  
        int year;  
        double cost;  
        String color;  
        Scanner in=new Scanner(System.in);  
        System.out.print("Manufacturer:");  
        manufacturer=in.nextLine();  
        System.out.print("Manufacture  
year:");
```

```
year=Integer.parseInt(in.nextLine());  
System.out.print("Cost:");  
cost=Double.parseDouble(in.nextLine());  
System.out.print("Color:");  
color=in.nextLine();  
v=new Vehicle(manufacturer, year, cost,  
    color);  
return v;  
    }  
public void output(Vehicle v) {  
    System.out.println(v.toString());  
    }  
}
```


Example: Array of vehicles

```
public class ArrayVehicle {  
    public Vehicle[] input(int n) {  
        Vehicle[] v=new Vehicle[n];  
        String manufacturer;  
        int year;  
        double cost;  
        String color;  
        Scanner in=new Scanner(System.in);  
        for(int i=0;i<n;i++) {  
            System.out.print("Manufacturer:");  
            manufacturer=in.nextLine();  
        }  
    }  
}
```

```
System.out.print("Manufacture year:");
year=Integer.parseInt(in.nextLine());
System.out.print("Cost:");
cost=Double.parseDouble(in.nextLine());
System.out.print("Color:");
color=in.nextLine();
v[i]=new Vehicle(manufacturer, year, cost,
    color);}
return v;    }
public void output(Vehicle[] v){
    System.out.println("Manufacturer    Year    Cost
    Color");
        for(int i=0;i<v.length;i++)
            System.out.println(v[i].toString());
    }}
```

Simple Uses of Inner Classes

- **Inner classes** are classes defined within other classes
 - The class that includes the inner class is called the **outer class**
 - There is no particular location where the definition of the inner class (or classes) must be place within the outer class
 - Placing it first or last, however, will guarantee that it is easy to find

Simple Uses of Inner Classes

- An inner class definition is a member of the outer class in the same way that the instance variables and methods of the outer class are members
 - An inner class is local to the outer class definition
 - The name of an inner class may be reused for something else outside the outer class definition
 - If the inner class is private, then the inner class cannot be accessed by name outside the definition of the outer class

Inner/Outer Classes

```
public class Outer
{
    private class Inner
    {
        // inner class instance variables
        // inner class methods

    } // end of inner class definition

    // outer class instance variables
    // outer class methods
}
```

Simple Uses of Inner Classes

- There are two main advantages to inner classes
 - They can make the outer class more self-contained since they are defined inside a class
 - Both of their methods have access to each other's private methods and instance variables
- Using an inner class as a helping class is one of the most useful applications of inner classes
 - If used as a helping class, an inner class should be marked private

Shadowing (1)

- Declaration of a type in a particular scope has the same name as another declaration in the enclosing scope, then the declaration ***shadows*** the declaration of the enclosing scope.

Shadowing Variable (2)

```
public class ShadowTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1; // shadowing  
  
        void methodInFirstLevel(int x) {  
            System.out.println("x = " + x);  
            System.out.println("this.x = " + this.x);  
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);  
        }  
    }  
  
    public static void main(String... args) {  
        ShadowTest st = new ShadowTest();  
        ShadowTest.FirstLevel fl = st.new FirstLevel();  
        fl.methodInFirstLevel(23);  
    }  
}
```

Output - FirstPrj (run) x

run:
x = 23
this.x = 1
ShadowTest.this.x = 0

Mechanism: Local data is treated first

Implementing Object-Oriented Relationships

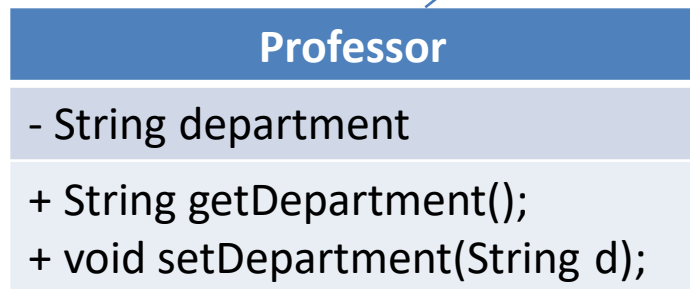
- 3 common relations in classes:
 - “is a/ a kind of”
 - “has a”
 - association
- Examples:
 - Student **is a** person
 - “A home is a house that **has a** family and a pet.”
 - A Professor has some students and a professor can be contained in one Student

Implementing Object-Oriented Relationships...

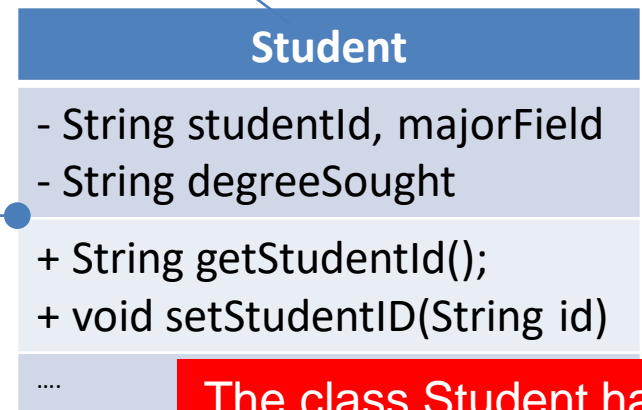
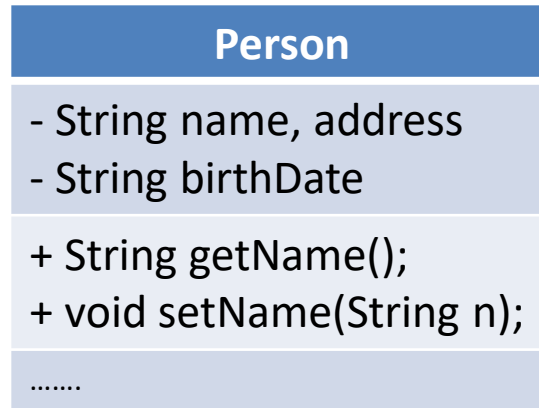
The relation “is a” is implemented as a sub-class

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

The relation “has a” is implemented as reference



The class Professor has the field Student[] students



is a

teach

The class Student has the field Professor pr

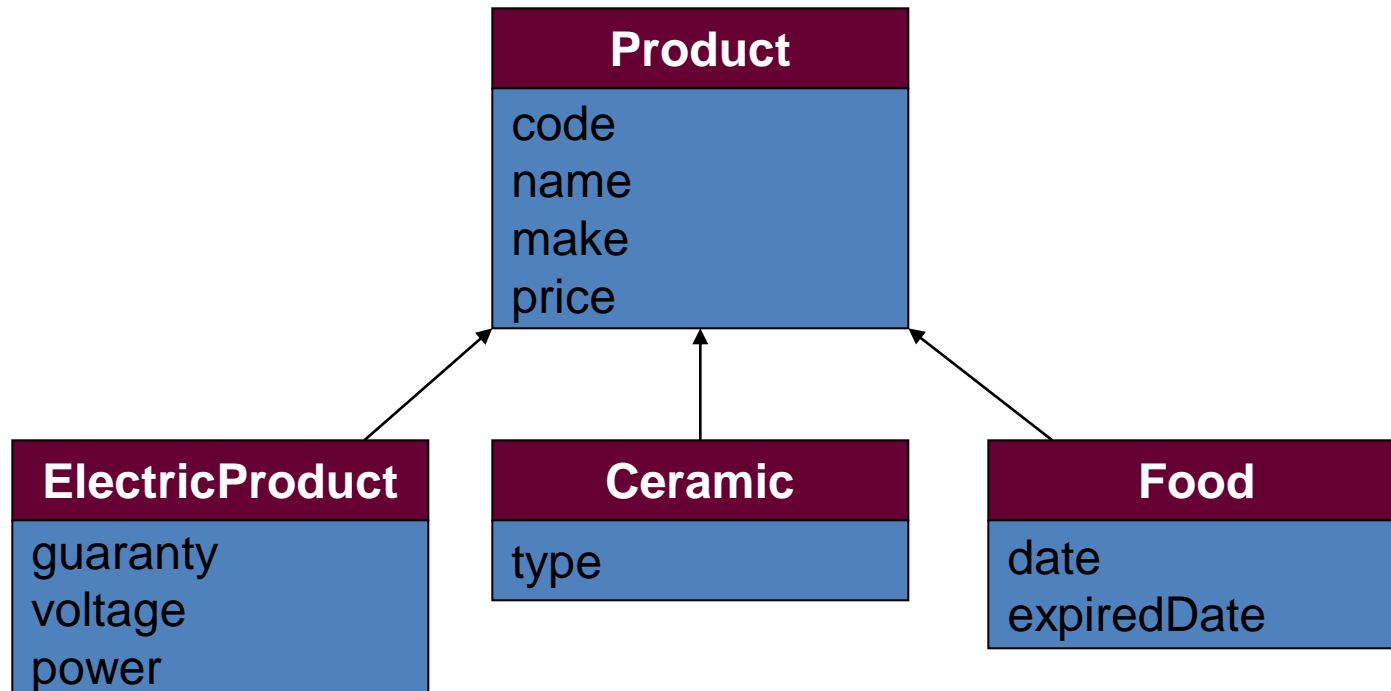
Inheritance

- 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 super class, 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 super class 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     }
7 public Rectangle(int l, int w)
8     {
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    {
12    return "[" + getLength() + "," + getWidth() + "]\n";
13    }
14    // Getters, Setters
15    public int getLength() { return length; }
16    public void setLength(int length) { this.length = length; }
17    public int getWidth() { return width; }
18    public void setWidth(int width) { this.width = width; }
19    public int area() { return length*width; }
20 }
```

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 volumn() {
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)
```



```

public class Point {
    private int x,y;
    public Point(int x,int y){
        this.x=x;  this.y=y;}
    ....//getter & setter }
public class Polygon {
    protected Point d1, d2, d3, d4;
    public void setD1(Point d1) {this.d1=d1;}
    public Point getD1(){return d1;}
    @Override
    public String toString() {
        return
d1.getX()+"\t"+d1.getY()+"\t"+d2.getX()+"\t"+d2.ge
tY()+
"\t"+d3.getX()+"\t"+d3.getY()+"\t"+d4.getX()+"\t"+
d4.getY();  }}

```

```
public class Square extends Polygon{
    public Square() {
        d1 = new Point(0,0); d2 = new Point(0,1);
        d3 = new Point (1,0); d4 = new Point(1,1);
    }
}

public class Main {
    public static void main(String args[]) {
        Square sq = new Square();
        System.out.println(sq.toString());
    }
}
```

```

public class Person {
    private String name;
    private String bithday;
    public Person() {
        } // getter & setter }
public class Employee extends Person {
    private double salary;
    public double getSalary() { return salary; }
    public void setSalary(double salary){
        this.salary = salary;    }
    @Override
    public String toString() {
        //return name + "\t" + birthday + "\t" + salary;
        return super.getName() + "\t" +
super.getBithday() + "\t" + salary;
    }}

```

```
public class Main {  
    public static void main(String  
        args[]) {  
        Employee e = new Employee();  
        e.setName("To Ngoc Van");  
        e.setBirthday("3/4/1994");  
        e.setSalary(4.4);  
        System.out.println(e.toString() );  
    }  
}
```

Output???

```
1. public class Polygon {  
    protected Point d1, d2, d3, d4;  
    public Polygon() {  
        System.out.println("Polygon class");  
    }.....}  
  
2. public class Square extends Polygon{  
    public Square() {  
        System.out.println("Square class");  
    }  
}  
  
3. public class Main {  
    public static void main(String args[]){  
        Square hv = new Square();  
    }  
}
```

```
1. public class Polygon {  
    protected Point d1, d2, d3, d4;  
    //a constructor is not default constructor  
    .....}  
  
2. public class Square extends Polygon{  
    public Square() {  
        System.out.println("Square class");  
    }  
}  
  
3. public class Main {  
    public static void main(String args[]){  
        Square hv = new Square();  
    }  
}
```

Why Error???

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 (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");  
    }  
}
```

Hiding

```
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();  
    }  
}
```

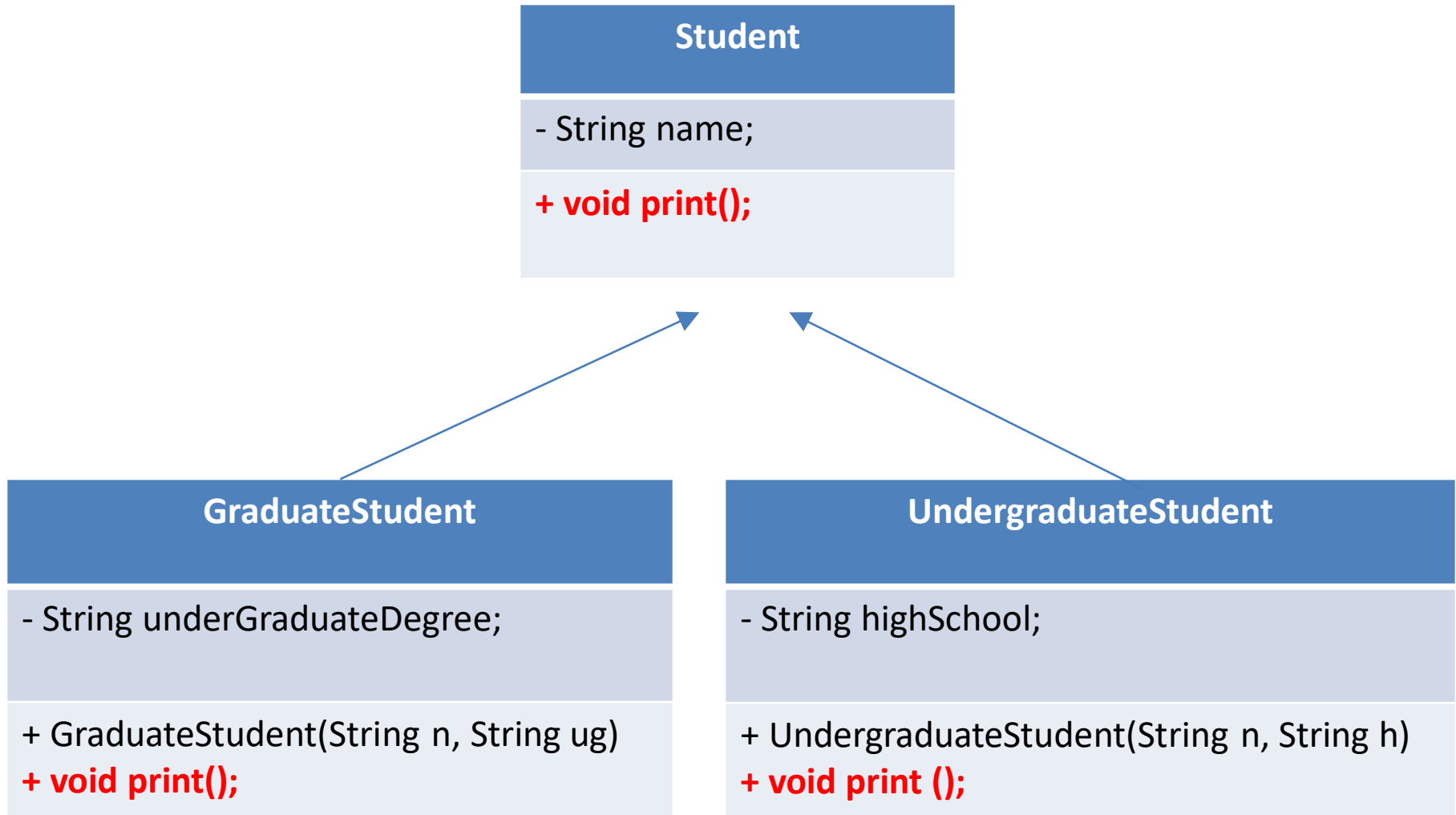
Output - FirstPrj (run) x

```
run:  
I am a father  
I am a father  
I am a son
```


Polymorphism

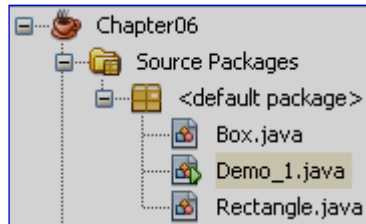
- The ability of two or more objects belonging to ***different*** classes to respond to exactly the ***same*** message (method call) in different class-specific ways.
- ***Inheritance combined with overriding facilitates polymorphism.***

Polymorphism...



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 }
15 }
```

```
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 // Overriding the toString method
8 // of the Rectangle class
9 public String toString()
10 { return "[" + length + "," + width +
11    "," + height + "]"; }
12 }
13 }
```

Output - Chapter06 (run)

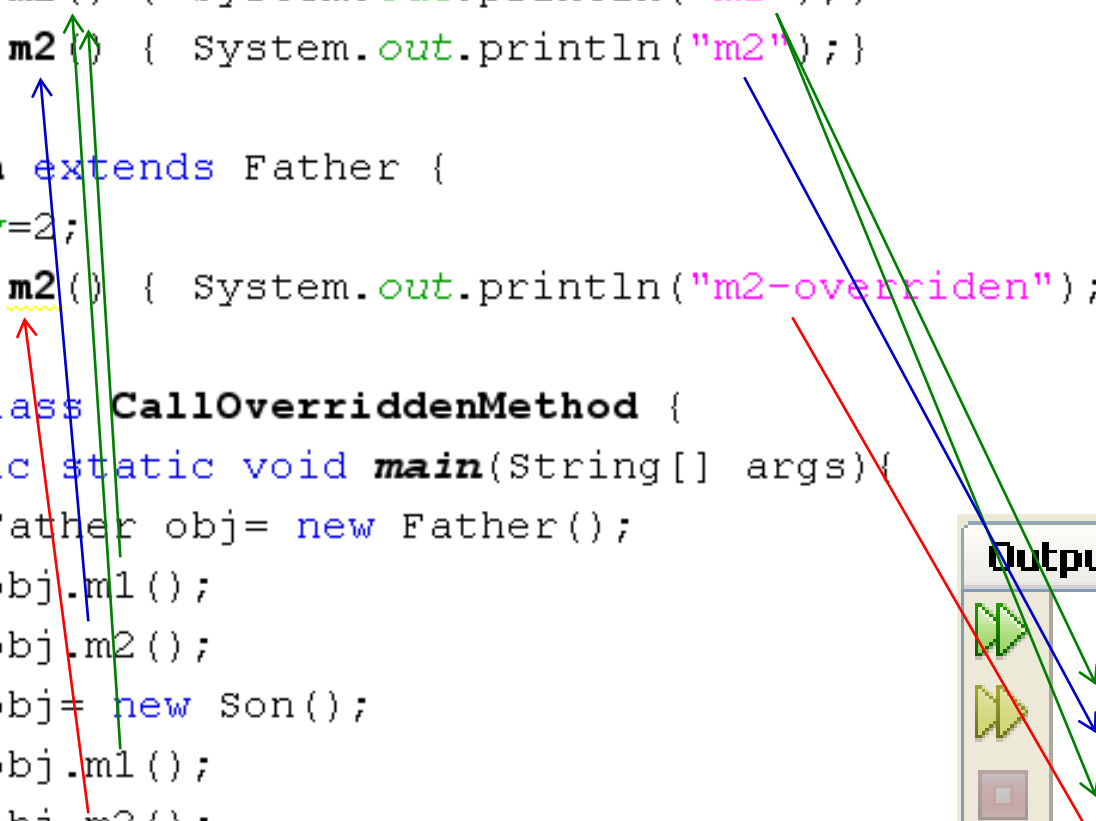
```
run:
[5,0]
[10,20]
[5,10,15]
```

```
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();
    }
}
```



Output - FirstPrj (run) x

```
run:
m1
m2
m1
m2-overridden
```

Example

- Code a class named **Car**, **Motor**, **Truck** are directly derived from base class **Vehicle** object.

```
public class Car extends Vehicle {  
    private String typeofEngine;  
    private int seats;  
    public Car() {  
        super() ;  
    }  
}
```

```
public Car(String manufacturer, int
    year, double cost, String color, String
    typeofEngine, int seats){
    super(manufacturer, year, cost, color);
    this.typeofEngine=typeofEngine;
    this.seats=seats;
}

public String gettypeofEngine() {
    return typeofEngine;
}

public void settypeofEngine(String
    typeofEngine) {
    this.typeofEngine = typeofEngine;
}
```

```
public int getSeats() {  
    return seats;  
}  
  
public void setSeats(int seats) {  
    this.seats = seats;  
}  
  
public String toString() {  
    return  
        super.toString()+"\t"+typeofEngine+"  
        \t"+seats;  
    }  
}
```

Example: Array of Car, Motor, Truck

```
public class ArrayVehicles {  
    private Vehicle[] list;  
    private int n;  
    public ArrayVehicles () {  
        n=0;  
        list=new Vehicle[50];  
    }  
    public int getNumberOfVehicle () {  
        return n;  
    }  
}
```



```
private Vehicle input() {  
    Vehicle v;  
    String manufacturer;  
    int year;    double cost;  
    String color;  
    Scanner in=new Scanner(System.in);  
    System.out.print("Manufacturer:");  
    manufacturer=in.nextLine();  
    System.out.print("Manufacture year:");  
    year=Integer.parseInt(in.nextLine());  
    System.out.print("Cost:");  
    cost=Double.parseDouble(in.nextLine());  
    System.out.print("Color:");  
    color=in.nextLine();  
    v=new Vehicle(manufacturer, year, cost,  
color);  
    return v;  
}
```

```
public void inputMotor() {  
    double power;  
    Vehicle v=input();  
    Scanner in=new Scanner(System.in);  
    System.out.print("Power:");  
    power=in.nextDouble();  
    list[n++]=new  
    Motor(v.getManufacturer(),v.getYear(  
    ),v.getCost(),v.getColor(), power);  
}
```

```
public void inputCar() {  
    String typeofEngine;  
    int seats;  
    Vehicle v=input();  
    Scanner in=new Scanner(System.in);  
    System.out.print("Type of engine:");  
    typeofEngine=in.nextLine();  
    System.out.print("Number of Seats:");  
    seats=in.nextInt();  
    list[n++]=new  
    Car(v.getManufacturer(),v.getYear(),  
    v.getCost(),v.getColor(), typeofEngine,  
    seats);  
}
```

```
public int getNumberOfCar() {  
    int count=0;  
    for(int i=0;i<n;i++)  
        if(list[i] instanceof Car)  
            count++;  
    return count; }  
  
public int getNumberOfMotor() {  
    int count=0;  
    for(int i=0;i<n;i++)  
        if(list[i] instanceof Motor)  
            count++;  
    return count;  
}
```

```

public void output() {
    if (getNumberOfCar() > 0) {
        System.out.println("List of Cars");
        System.out.println("Manufacturer   Year
Cost       Color TypeofEngine
NumberOfSeats");
        for (int i = 0; i < n; i++) {
            if (list[i] instanceof Car)
                System.out.println(list[i].toString());
        }
        System.out.println("-----");
        System.out.println("Number of
cars: " + getNumberOfCar());
    }
}

```

```

if (getNumberOfMotor() > 0) {
    System.out.println("List of Motors");
    System.out.println("Manufacturer   Year
Cost       Color Power");
    for (int i = 0; i < n; i++) {
        if (list[i] instanceof Motor)
            System.out.println(list[i].toString());
    }
    System.out.println("-----");
    System.out.println("Number of
motors: " + getNumberOfMotor());
}

```

```

if (getNumberOfTruck() > 0) {
    System.out.println("List of Trucks");
    System.out.println("Manufacturer   Year
Cost       Color Load");
    for (int i = 0; i < n; i++) {
        if (list[i] instanceof Truck)
            System.out.println(list[i].toString());
    }
    System.out.println("-----");
    System.out.println("Number of
trucks: " + getNumberOfTruck());
}
}

```

Menu

```
public class Main {  
    public static void main(String[]  
args) {  
        ArrayVehicles a=new  
ArrayVehicles();  
        Scanner in=new  
Scanner(System.in);
```



```
while (true) {  
    System.out.print("\n 1. input a Car");  
    System.out.print("\n 2. input a Motor");  
    System.out.print("\n 3. input a Truck");  
    System.out.print("\n 4. output");  
    System.out.print("\n 0. Exit");  
    System.out.print("\n Your choice (0->4) :  
");  
    int choice;  
    choice=in.nextInt();  
    switch(choice) {  
        case 1:a.inputCar();  
            break;
```

```
case 2:a.inputMotor();
    break;
case 3:a.inputTruck();
    break;
case 4:a.output();
    break;
case 0:
    System.out.println("Bye!!!!");
    System.exit(0);
    break;
default:System.out.println("only to
choose (0->4)");
    } } } }
```

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), static 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 sqr(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 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");}  
    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.

```
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) x



run:



M1



M2



M3

M4

5x5=25

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 circumference();
4     abstract public double area();
5 }
6 class Circle extends Shape {
7     double r;
8     public Circle (double rr) { r=rr; }
9     public double circumference() { 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 circumference() { 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 }
```

Modified

Output - Chapter06 (run)

run:
78.53981633974483

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  
8 class Derived extends AbstractDemo2  
9 { public void m1() // override  
10 { System.out.println("m1");  
11 }  
12     public static void main(String[] args)  
13 { Derived obj = new Derived();  
14 }  
15 }
```

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***

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
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 provide the implementation of interface .	Interface can't provide the implementation of abstract class .
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword <code>?extends?</code> .	An interface class can be implemented using keyword <code>?implements?</code> .
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

Type Casting in Inheritance

- You can cast an instance of a child class to its parent class. Casting an object of child class to a parent class is called **upcasting**.
- Casting an object of a parent class to its child class is called **downcasting**.
- **class** Vehicle { **String** manufacturer; }
- **class** Car **extends** Vehicle {String typeofEngine; }
- **public class** TypeCastExample {
public static void main(**String**[] args) {
 Vehicle v1 = **new** Vehicle();
 Vehicle v2 = **new** Car();//upcasting
 Car v3 = (Car) **new** Vehicle();//downcasting
 Car v4 = **new** Car(); } }

Enum Types

- 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 Type simple example

```
enum Season { SPRING, SUMMER, AUTUMN, WINTER }
```

```
class Main {  
    static void fun(Season x)  
    {switch(x)  
        {case SPRING: System.out.println("It is spring"); break;  
          case SUMMER: System.out.println("It is summer"); break;  
          case AUTUMN: System.out.println("It is autumn"); break;  
          case WINTER: System.out.println("It is winter");  
        }  
    }  
    public static void main(String[] args) {  
        Season x = Season.WINTER; fun(x);
```

```
        for(Season y: Season.values()) {  
            System.out.print(y + ": "); fun(y);  
        }  
        System.out.println();  
    }  
}
```

```
It is winter  
SPRING: It is spring  
SUMMER: It is summer  
AUTUMN: It is autumn  
WINTER: It is winter
```

Enum Type with parameter constructor

```
enum Season {  
    SPRING(25, 11), SUMMER(32, 13), AUTUMN(23, 10), WINTER(10, 9);  
    private final int avgTemp, dayLength;  
    Season(int x, int y) {  
        avgTemp = x; dayLength = y;  
    }  
    public void display() {  
        System.out.println(this + " average temperature is " + avgTemp);  
        System.out.println(this + " average day's length is " + dayLength);  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Season x = Season.WINTER;  
        x.display();  
        System.out.println();  
    }  
}
```

WINTER average temperature is 10
WINTER average day's length is 9

Summary

- The class, how to declare fields, methods, and constructors.
- Hints for class design:
 - Main noun → Class
 - Descriptive nouns → Fields
 - Methods: Constructors, Getters, Setters, Normal methods
- Creating and using objects.
- Use the dot operator to access the object's instance variables and methods.

- Overriding methods in sub-classes
- Controlling Access to Members of a Class using modifiers
- Nested Classes
- Benefits of OO implementation: Inheritance and Polymorphism
- Working with Interfaces.
- Working with Abstract Methods and Classes.
- Enum Type