OOP IN JAVA

Instructor:

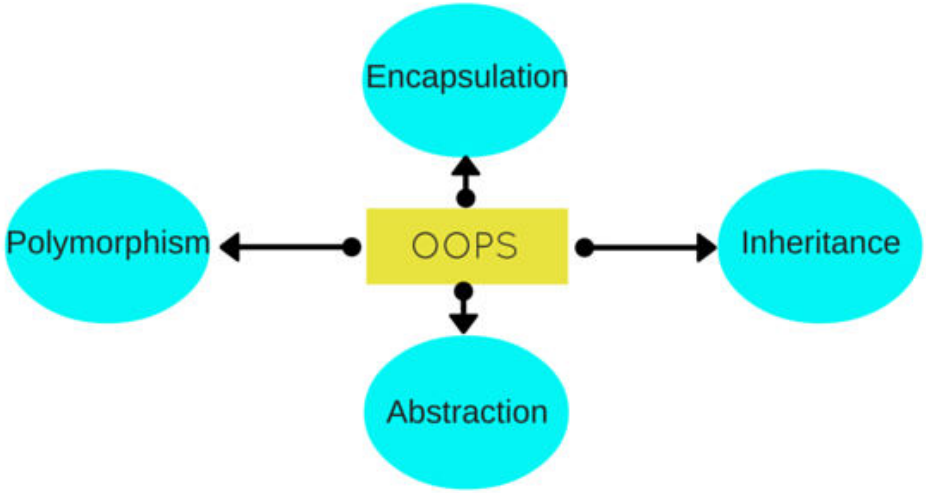
Table of contents

Section 1

Principles of OOP

Principles of OOP

- **Object - Oriented Programming** system (OOPs) is a programming paradigm based on the concept of "objects" that contain data and methods.
- The primary purpose of object-oriented programming is to **increase the flexibility and maintainability** of programs.
- Java is an **object oriented language** because it provides the features to implement an object oriented model.
- These features includes **Abstraction**, **Encapsulation**, **Inheritance** and **Polymorphism**.



Section 2

Encapsulation

Encapsulation in Java

- Two steps to implement encapsulation feature:
  - ✓ Make the **instance variables private** so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
  - ✓ Have **getter** and **setter methods** in the class to set and get the values of the fields.

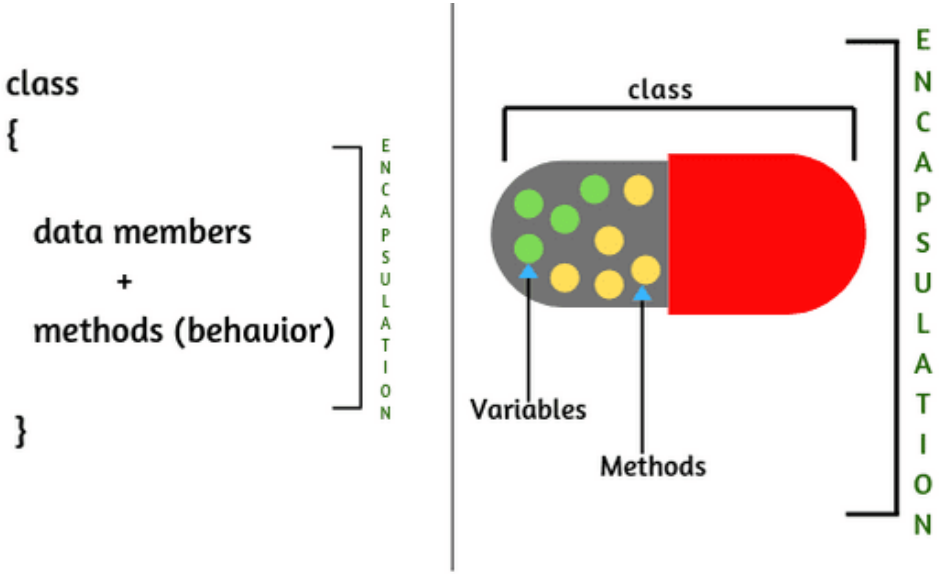
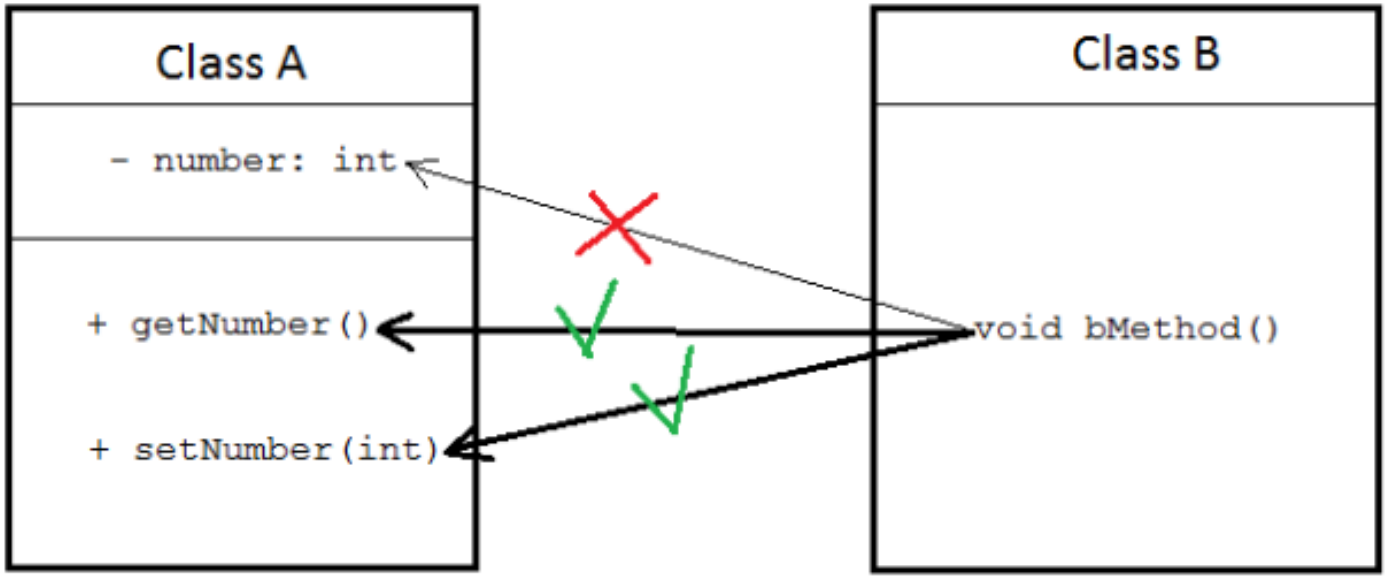


Fig: Encapsulation

Getter and setter method

- **Getter** and **setter** are two conventional methods that are used for **retrieving** and **updating** value of a variable.



Getter and setter method

- The following code is an example of simple class with a private variable and a couple of getter/setter methods:

```
1  public class SimpleGetterAndSetter {
2      private int number;
3
4      public int getNumber() {
5          return this.number;
6      }
7
8      public void setNumber(int num) {
9          this.number = num;
10     }
11 }
```

- "**number**" is private, code from outside this class cannot access the variable directly:

```
1  SimpleGetterAndSetter obj = new SimpleGetterAndSetter();
2  obj.number = 10;    // compile error, since number is private
3  int num = obj.number; // same as above
```

- Instead, the outside code have to invoke the getter, **getNumber**() and the setter, **setNumber**() in order to read or update the variable, for example:

```
1  SimpleGetterAndSetter obj = new SimpleGetterAndSetter();
2
3  obj.setNumber(10);  // OK
4  int num = obj.getNumber();  // fine
```

Why getter and setter?

- By using **getter** and **setter**, the programmer can control how to variables are accessed and updated in a **correct** manner.
- Example:

```
1  public void setNumber(int num) {
2      if (num < 10 || num > 100) {
3          throw new IllegalArgumentException();
4      }
5      this.number = num;
6  }
```

- ✓ That ensures the value of number is always set between 10 and 100.
- ✓ Suppose the variable number can be updated directly, the caller can set any arbitrary value to it:

```
1  obj.number = 3;
```

Naming convention for getter and setter

- The naming scheme of setter and getter should follow *Java bean naming convention* as follows:

getXXX() and **setXXX()**

  - ✓ where XXX is name of the variable.

- For example with the following variable name:

```
1   private String name;
```

```
1  public void setName(String name) { }
2
3  public String getName() { }
```

- If the variable is of type boolean, then the getter's name can be either **isXXX**() or **getXXX**(), but the former naming is preferred.

```
1  private boolean single;
2
3  public String isSingle() { }
```

this keyword

- "**this**" keyword in java can be used inside the *method* or *constructor* of Class.
- It (***this)*** works as a reference to the current Object, whose Method or constructor is being invoked.

this ko thể truy cập biến static và method static
vì static thuộc về Class chứ ko phải Object mà this lại tham chiếu tới Object

- **this** keyword with a **field** and **constructor:**

```
3  public class Mobile {
4    private String manufacture;
5    private String operatingSystem;
6    String model;
7    private double cost;
8
9    public Mobile(String manufacture, String operatingSystem) {
10     System.out.println("Constructor with 2 params!");
11     this.manufacture = manufacture;
12     this.operatingSystem = operatingSystem;
13   }
14
15   public Mobile(String manufacture, String operatingSystem,
16       String model, double cost) {
17
18     this(manufacture, operatingSystem);
19
20     this.model = model;
21     this.cost = cost;
22     System.out.println("Constructor with 4 params!");
23   }
24
25   public String getModel() {
26     return this.model;
27   }
28
30   public String toString() {
34
35 }
```

Output:

```
Constructor with 2 params!
Constructor with 4 params!
Samsung Galaxy S9
```

```
3  public class MobileTest {
4
5    public static void main(String[] args) {
6      Mobile mobile = new Mobile("Samsung", "Androis", "Samsung Galaxy S9", 2000);
7      System.out.println(mobile.getModel());
8    }
9
10 }
11
```

Access Modifiers

Access Modifier	Within class	Within package	Outside package by subclass only	Outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Section 3

Inheritance

Inheritance Overview (1/2)

- Inheritance allows you to define a new class by specifying only the ways in which it differs from an existing class.
- Inheritance promotes software reusability (tính tái sử dụng)
  - ✓ **Create new class from existing class**
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Terms used in Inheritance

❖ **Syntax**:

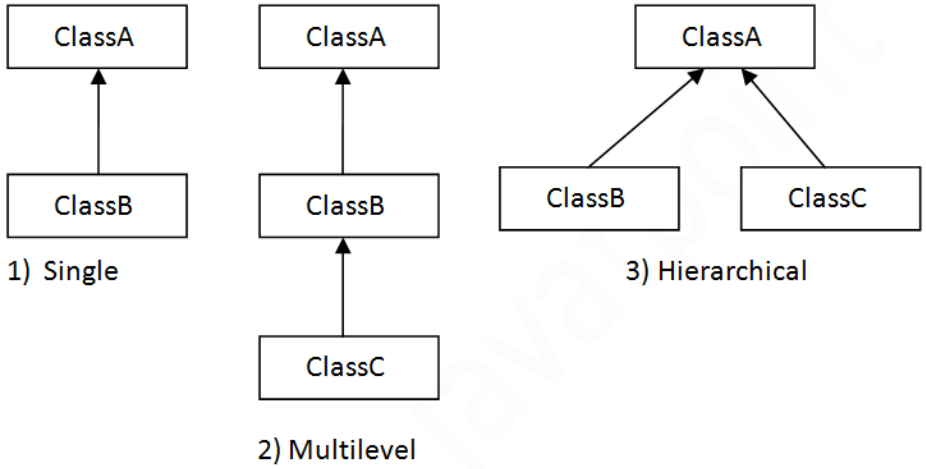
```
class SubclassName extends SuperclassName  {
   //methods and fields
}
```

❖ **Terms:**

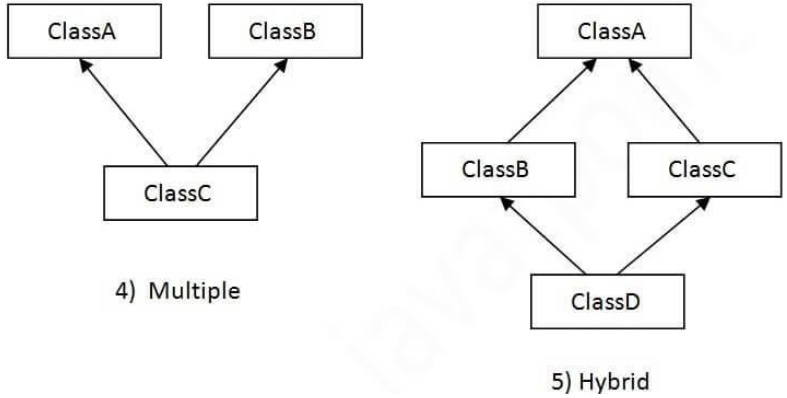
- ✓ **Class:** A class is a group of objects which have common properties.
- ✓ **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- ✓ **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- ✓ **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Types of inheritance in java

- Three types of inheritance in java: single, multilevel and hierarchical.

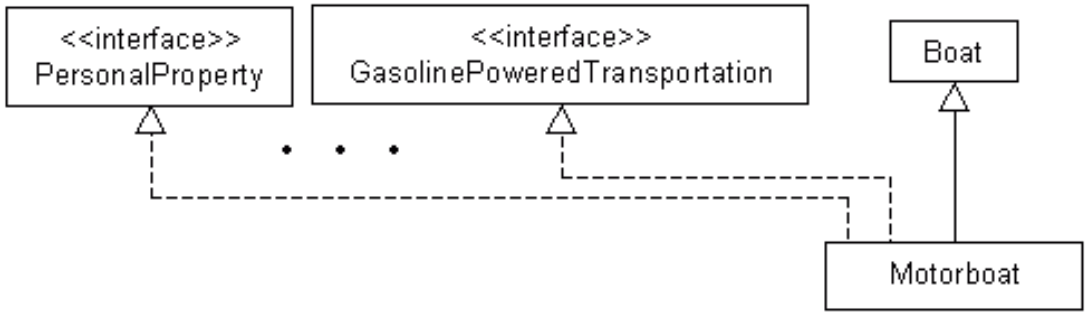


- *Note: Multiple inheritance is not supported in Java through class.*



Inheritance

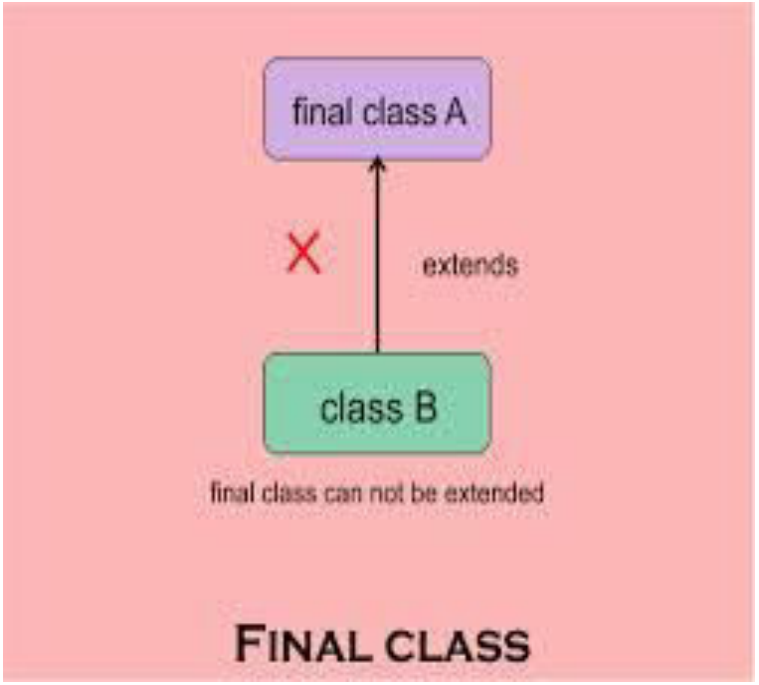
- **Two kinds:**
  - ✓ implementation: the code that defines methods.
  - ✓ interface: the method prototypes only.
- You can't extend more than one class!
  - ✓ the derived class can't have more than one base class.
- You can do multiple inheritance with *interface* inheritance.



Inheritance

❖ Final class:

- ✓ You can declare an class is `final` - this prevents the class from being subclassed.
- ✓ Of course, an `abstract` class cannot be a `final` class.



❖ Instantiating subclass object

✓ *Chain of constructor calls*

- Subclass constructor **invokes** superclass constructor luôn luôn
  - **Implicitly** or **explicitly**
- Base of inheritance hierarchy
  - Last constructor called in chain is `Object`'s constructor
  - Original subclass constructor's body finishes executing last.

tóm lại là constructor của thằng bé nhất dc gọi cuối cùng

Examples:

```
class Building {
    Building() {
        System.out.print("b ");
    }

    Building(String name) {
        this();
        System.out.print("bn " + name);
    }
}
```

```
public class House extends Building {
    House() {
        System.out.print("h ");
    }

    House(String name) {
        this();
        System.out.print("hn " + name);
    }

    public static void main(String[] args) {
        new House("x ");
    }
}
```

Building() dc gọi đầu tiên

final output:
b h hn x

- Garbage collecting subclass object
  - ✓ Chain of `finalize` method calls
    - **Reverse** order of constructor chain
    - Finalizer of **subclass called first**
    - Finalizer of **next superclass** up hierarchy next
      - Continue up hierarchy until final superreached
        - » After final superclass (`Object`) finalizer, object removed from memory

phương thức finalize() dc gọi từ lớp con đến cha , ông (ngược với thứ tự gọi của constructor)

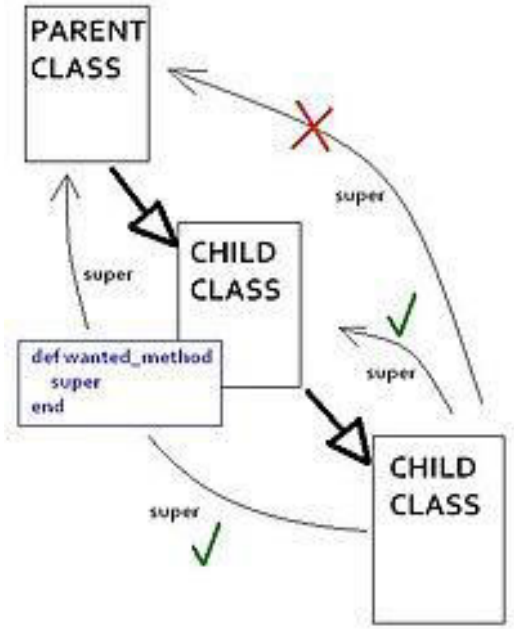
super keyword

- Can use **super** keyword to access all (non-private) superclass methods.
  - ✓ even those replaced with new versions in the derived class.
- Can use **super()** to call base class constructor.

```
class Parent
{
  String name;
}
class Child extends Parent {

  String name;

  void detail()
  {
    super.name = "Parent";
    name = "Child";
  }
}
```



- Subclass methods are **not** superclass methods

Thank you