OOP IN JAVA

Instructor:

Table of contents

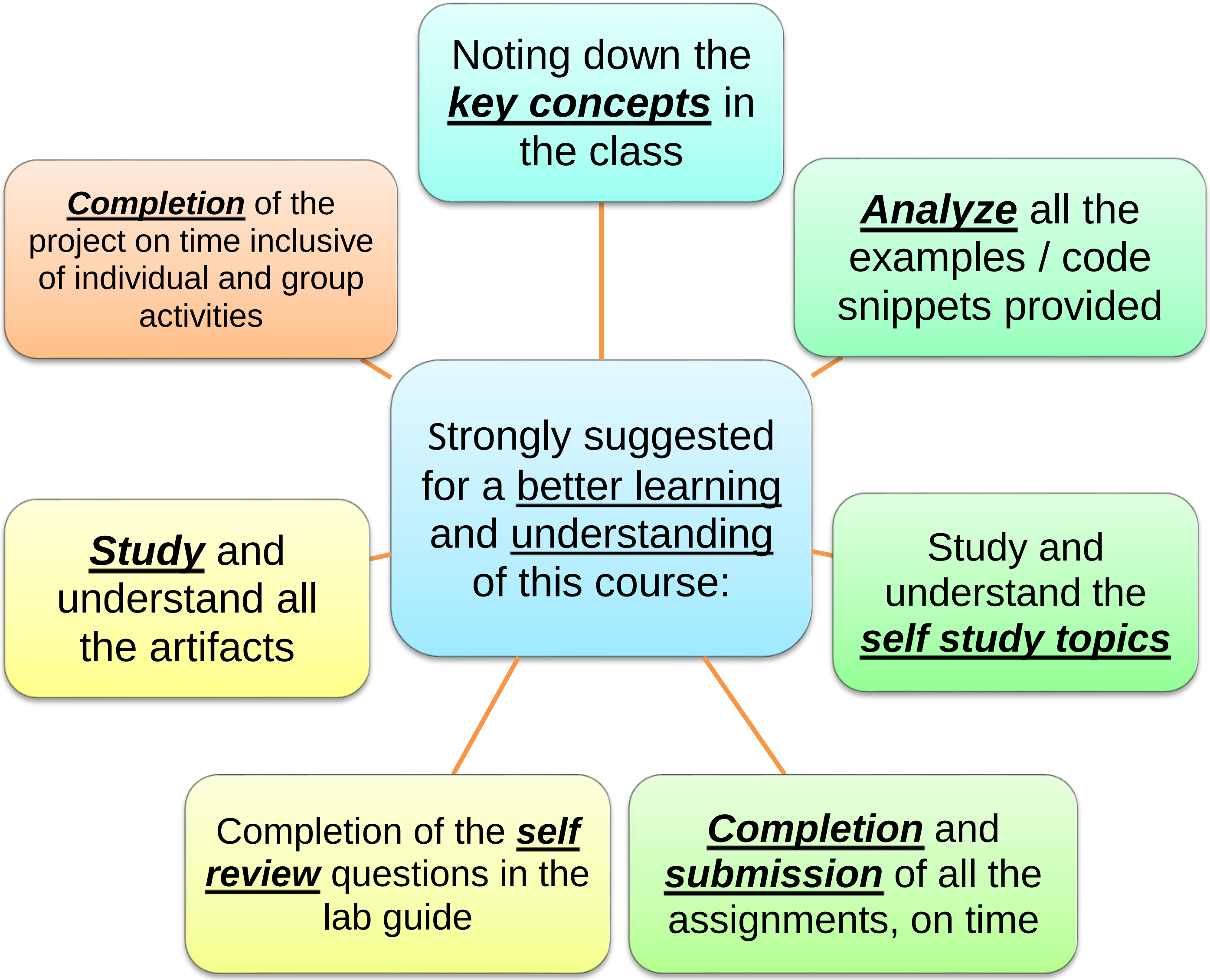
◊ OOPs Concepts

◊ Principles of OOP

◊ Encapsulation

◊ Inheritance

Learning Approach



Strongly suggested for a better learning and understanding of this course:

Noting down the ***key concepts*** in the class

Analyze all the examples / code snippets provided

Study and understand the ***self study topics***

Completion and ***submission*** of all the assignments, on time

Completion of the ***self review*** questions in the lab guide

Study and understand all the artifacts

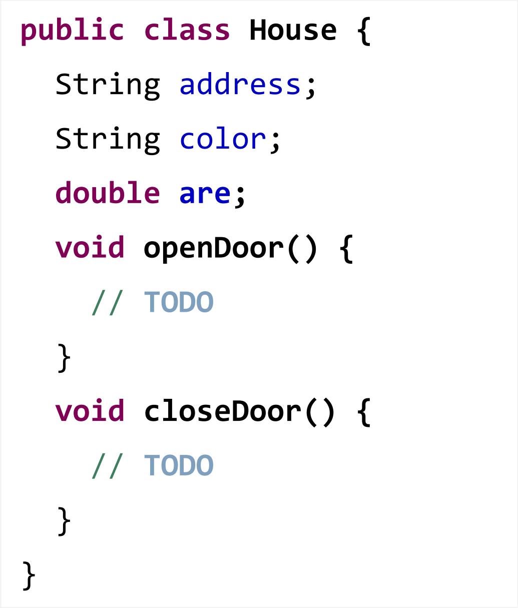
Completion of the project on time inclusive of individual and group activities

Section 1

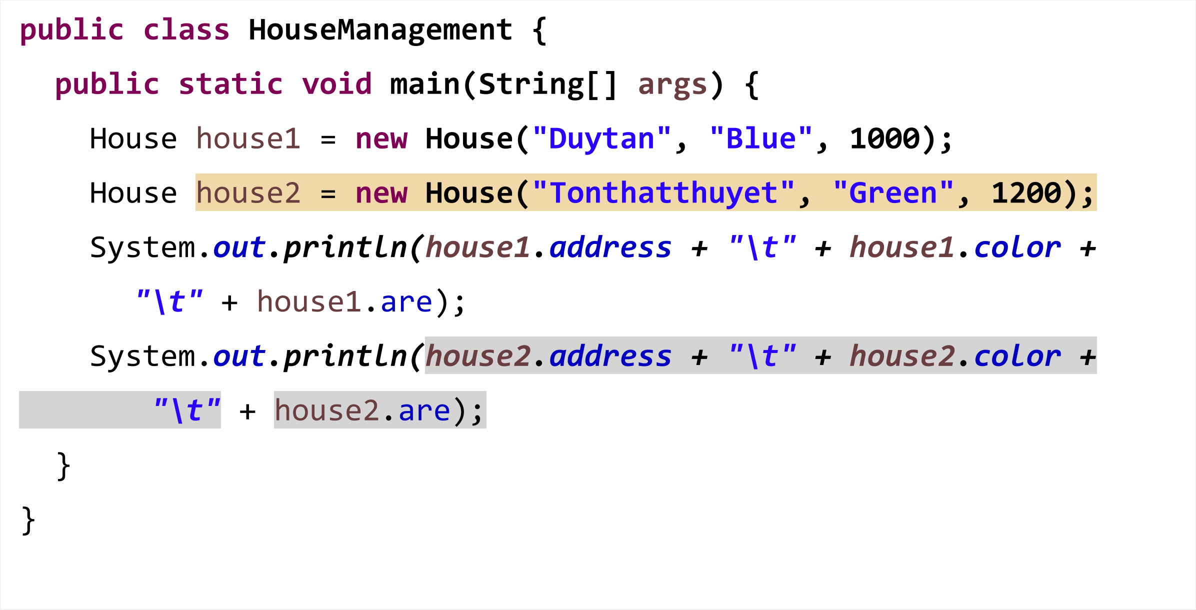
OOPs Concepts

What is a Class?

- A class can be considered as a **blueprint** using which you can create as many objects.
- For example, create a class **House** that has three instance variables:



```
public class House {
  String address;
  String color;
  double are;
  void openDoor() {
    // TODO
  }
  void closeDoor() {
    // TODO
  }
}
```



```
public class HouseManagement {
  public static void main(String[] args) {
    House house1 = new House("Duytan", "Blue", 1000);
    House house2 = new House("Tonthatthuyet", "Green", 1200);
    System.out.println(house1.address + "\t" + house1.color +
       "\t" + house1.are);
    System.out.println(house2.address + "\t" + house2.color +
         "\t" + house2.are);
  }
}
```

- This is just a *blueprint*, it does not represent any House
- We have created two objects, while creating objects we provided separate properties to the objects using constructor.

What is an Object

- **Object:** is a bundle of data and its behaviour (often known as methods).
- Objects have two characteristics: They have states and behaviors.
- **Example of states and behaviors**

Object: House

State: Address, Color, Area

Behavior: Open door, close door

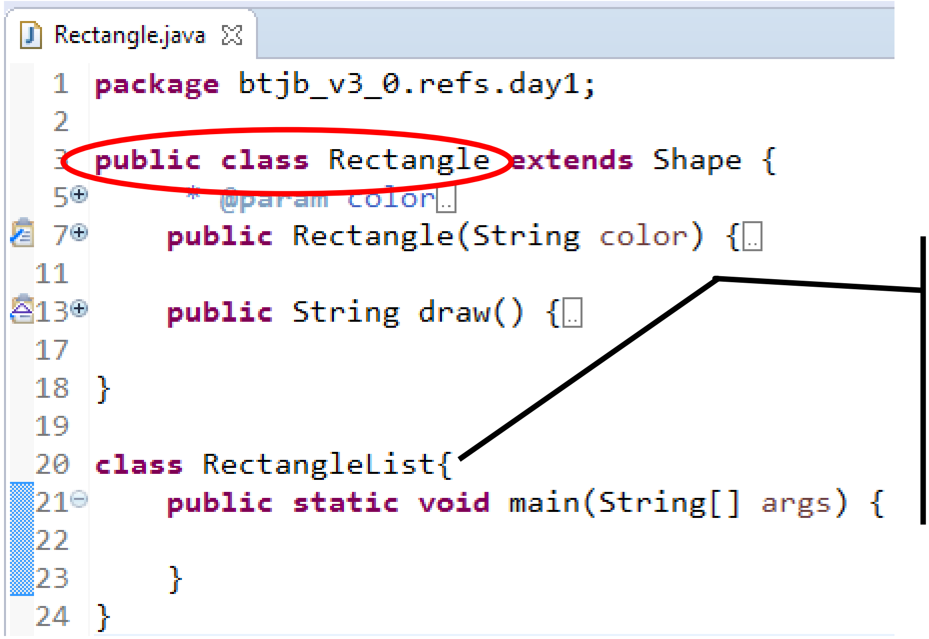
Class syntax

- Create new object type with `class` keyword.
- A class definition can contain:
  - ✓ instance variables (attribute/fields)
  - ✓ constructors
  - ✓ methods (instance method, static method)
- **Syntax:**

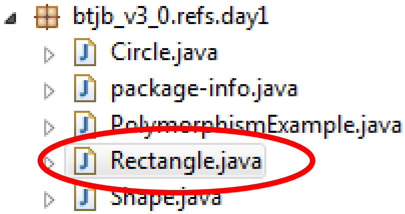
```
[public][<abstract><final>]class <ClassName>
[extends <SuperClass>] [implements <InterfaceName>]{
    <Attribute/Field declarations { initialization code }>
    <Constructors>
    <Methods>
}
```

Java Class: Modifiers

- `public`: that class is visible to all classes everywhere.
  - ✓ only one public class per file, must have same name as the file (this is how Java finds it!).



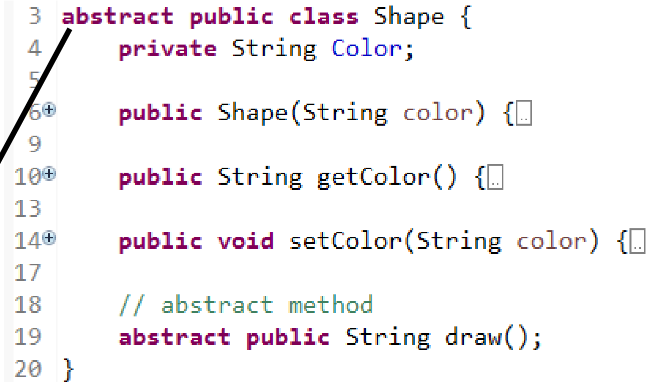
```java
package btjb_v3_0.refs.day1;

public class Rectangle extends Shape {
    * @param color...
    public Rectangle(String color) {...

    public String draw() {...

}

class RectangleList{
    public static void main(String[] args) {

    }
}
```



btjb_v3_0.refs.day1
- Circle.java
- package-info.java
- PolymorphismExample.java
- Rectangle.java
- Shape.java

- ✓ If a class has **no modifier** (the **default**, also known as **package-private**)
- ✓ It is visible only within its own package.

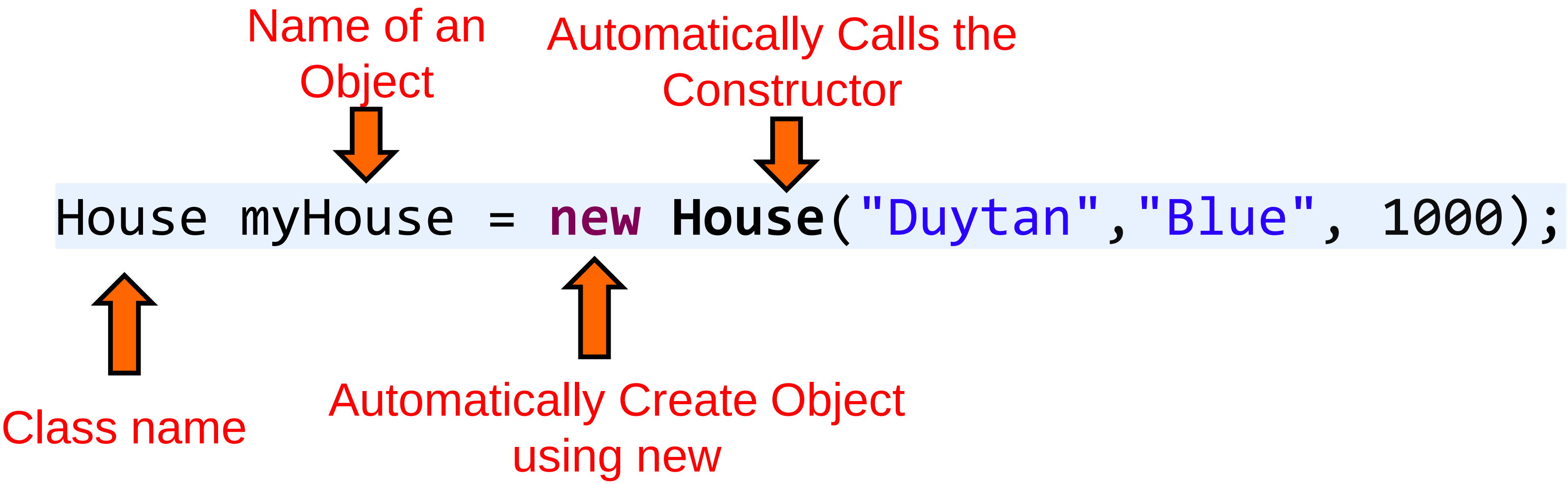
- `Abstract` modifier means that the class can be used as a superclass only.



```java
abstract public class Shape {
    private String Color;

    public Shape(String color) {...

    public String getColor() {...

    public void setColor(String color) {...

    // abstract method
    abstract public String draw();
}
```

Creating an Object

- Defining a class does not create an object of that class - this needs to happen explicitly[tường minh]:



Name of an Object

Automatically Calls the Constructor

```
House myHouse = new House("Duytan","Blue", 1000);
```

Class name

Automatically Create Object using new

- In general, an object must be created before any methods can be called.
  - ✓ the exceptions are *static* methods.

FooPrinter.java

```java
class FooPrinter {
    static final String UPPER = "FOO";
    static final String LOWER = "foo";

    // instance variable, do we print upper or lower?
    boolean printUpper = false;

    void upper() {
        printUpper = true;
    }

    void lower() {
        printUpper = false;
    }

    void print() {
        if (printUpper)
            System.out.println(UPPER);
        else
            System.out.println(LOWER);
    }
}
```

What does it mean to create an object?

```
public class SimpleClass {
    public static void main(String[] args) {
        FooPrinter foo = new FooPrinter();
        foo.print();
        foo.upper();
        foo.print();
    }
}
```

```
Output:
foo
FOO
```

- An object is a chunk of memory:
  - ✓ holds field values
  - ✓ holds an associated object type

- All objects of the same type share code
  - ✓ they all have same object type, but can have different field values.

Constructors

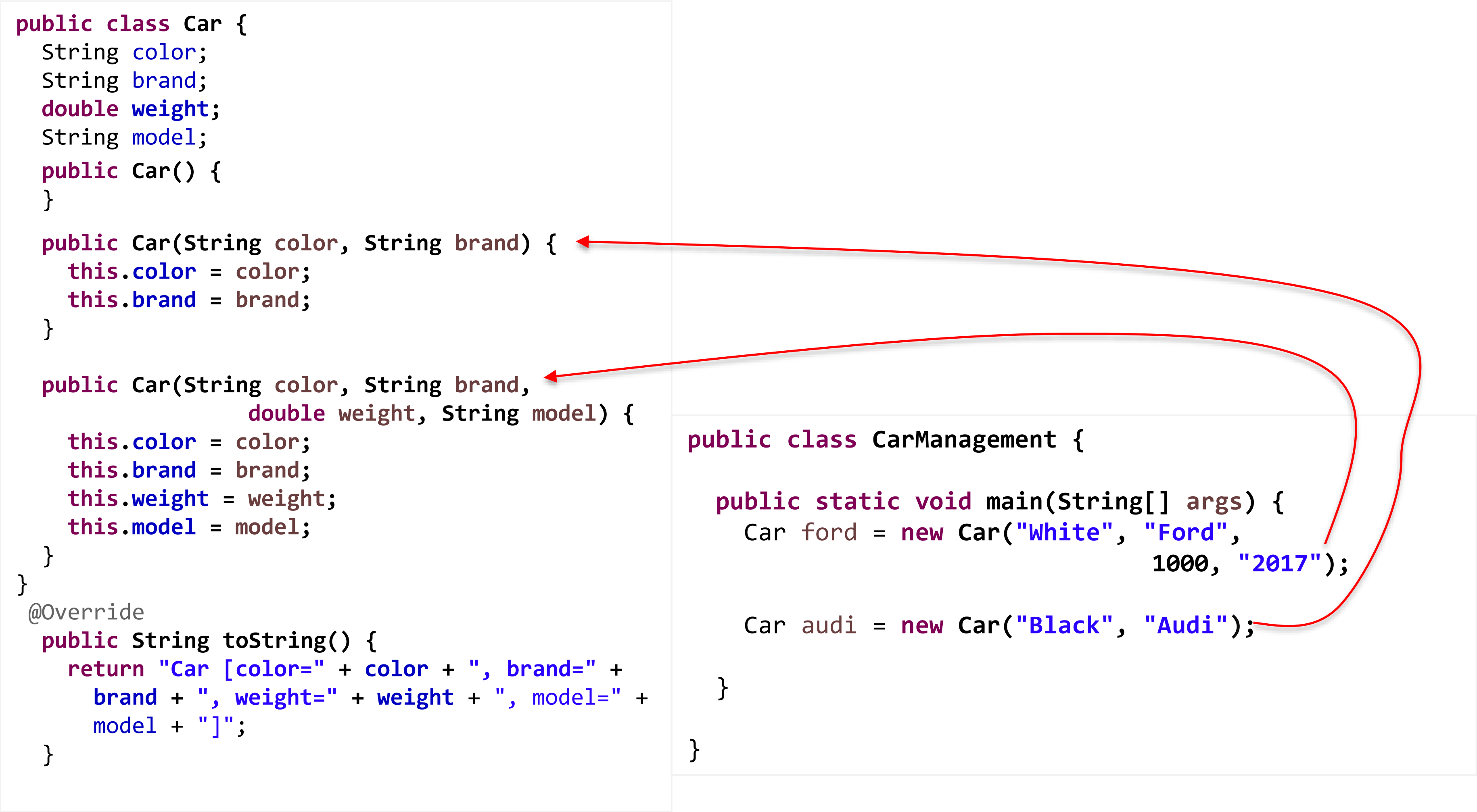
- Constructor is a block of code that initializes the newly created object.
  - ✓ Constructor has same name as the class
  - ✓ People often refer constructor as special type of method in Java. It **doesn't have a return type**
- You can create multiple constructors, each must accept different parameters.
- If you don't write any constructor, the compiler will (in effect) write one for you:

```
FooPrinter(){}
```

- If you include any constructors in a class, the compiler will not create a default constructor!

How does a constructor work

- When ***new keyword*** here creates the object of class Car and invokes the constructor to initialize this newly created object.



```java
public class Car {
  String color;
  String brand;
  double weight;
  String model;
  public Car() {
  }

  public Car(String color, String brand) {
    this.color = color;
    this.brand = brand;
  }

  public Car(String color, String brand,
                  double weight, String model) {
    this.color = color;
    this.brand = brand;
    this.weight = weight;
    this.model = model;
  }
}
 @Override
  public String toString() {
    return "Car [color=" + color + ", brand=" +
      brand + ", weight=" + weight + ", model=" +
      model + "]";
  }
```

```java
public class CarManagement {

  public static void main(String[] args) {
    Car ford = new Car("White", "Ford",
                              1000, "2017");

    Car audi = new Car("Black", "Audi");

  }

}
```

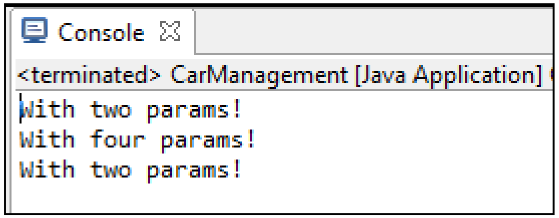
Multiple (overload) Constructors

- Must accept different parameters.
- One constructor can call another, use ***this***, not the classname:

```
public class Car {
  String color;
  String brand;
  double weight;
  String model;

  public Car() {
    System.out.println("No params!");
  }

  public Car(String color, String brand) {
    this.color = color;
    this.brand = brand;
    System.out.println("With two params!");
  }

  public Car(String color, String brand, double weight,
                                String model) {
    this(color, brand);
    this.weight = weight;
    this.model = model;
    System.out.println("With four params!");
  }
}
```

```
public class CarManagement {

  public static void main(String[] args) {
    Car ford = new Car("White", "Ford",
                            1000, "2017");

    Car audi = new Car("Black", "Audi");

  }

}
```

- **What will print out?**



```
<terminated> CarManagement [Java Application]
With two params!
With four params!
With two params!
```

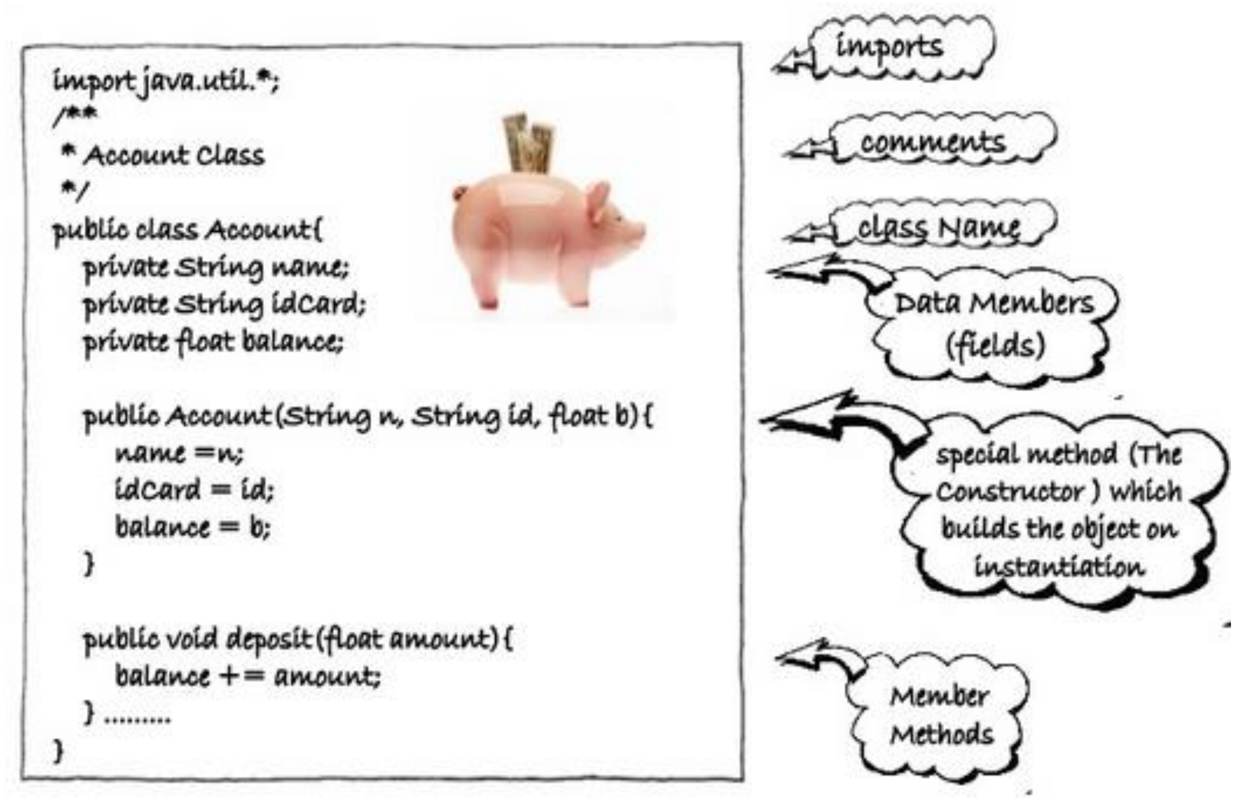
Destructors

- Nope!
- There is a **finalize()** method that is called when an object is destroyed:
  - ✓ You don't have control over when the object is destroyed (it might never be destroyed).
  - ✓ The JVM garbage collector takes care of destroying objects automatically (you have limited control over this process).

Instance variable (Field)

- Instance variable in java is used by objects to store their states
- **Fields** (data members) can be any primitive or reference type
- Syntax:

  [Access modifier] <Data type> <field_name>;



```
import java.util.*;
/**
 * Account Class
 */
public class Account{
   private String name;
   private String idCard;
   private float balance;

   public Account(String n, String id, float b){
      name =n;
      idCard = id;
      balance = b;
   }

   public void deposit(float amount){
      balance += amount;
   } ..........
}
```

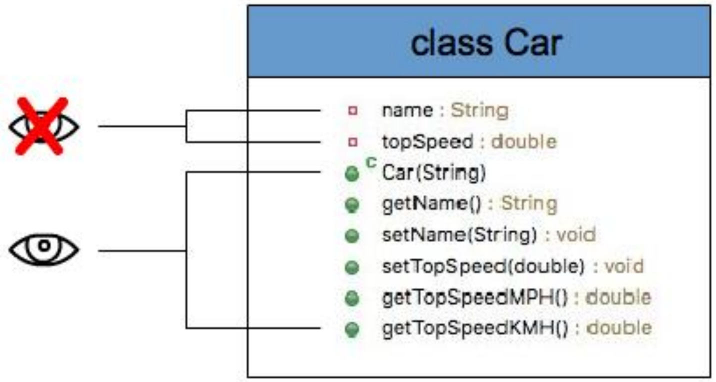
imports
comments
class Name
Data Members (fields)
special method (The Constructor) which builds the object on instantiation
Member Methods

Instance variable (Field)

- The following table shows the **access** to members permitted by each **modifier**:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- **Example:**



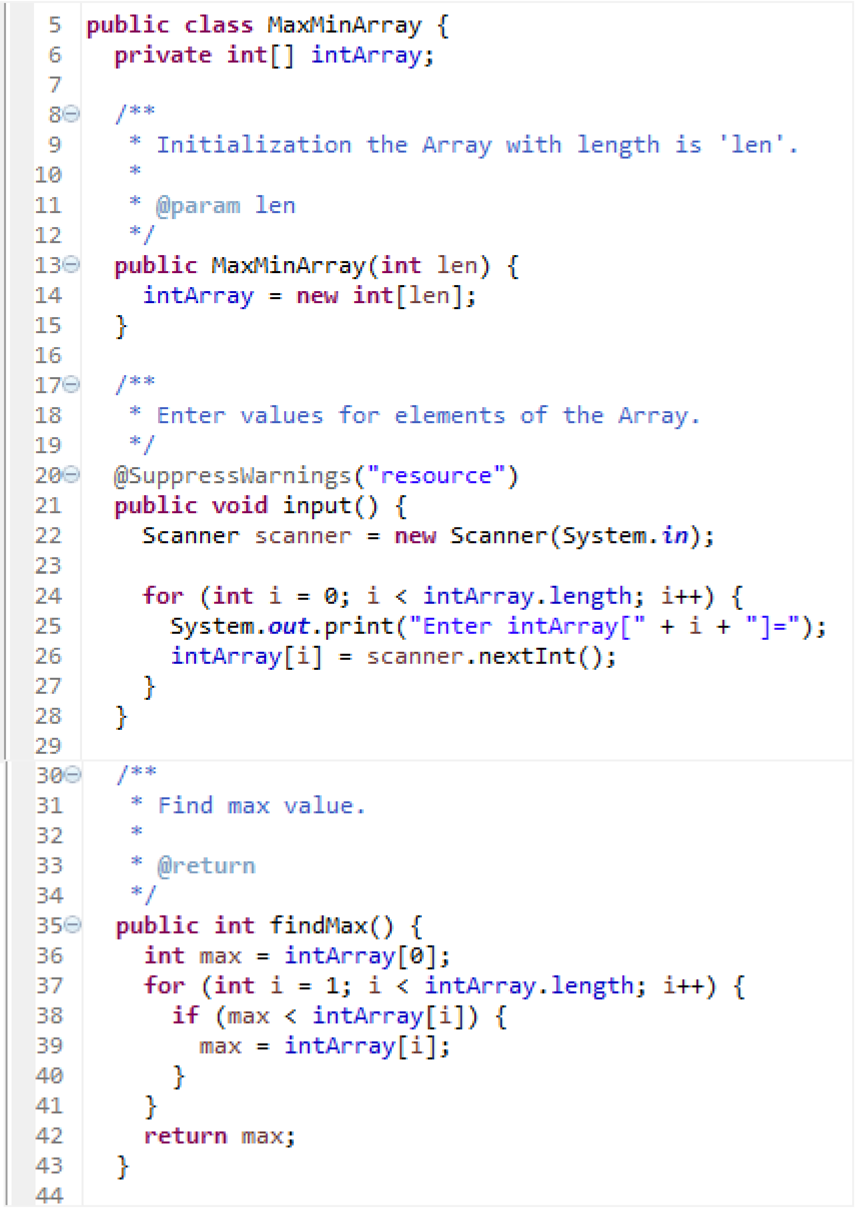
class Car

name : String
topSpeed : double
Car(String)
getName() : String
setName(String) : void
setTopSpeed(double) : void
getTopSpeedMPH() : double
getTopSpeedKMH() : double

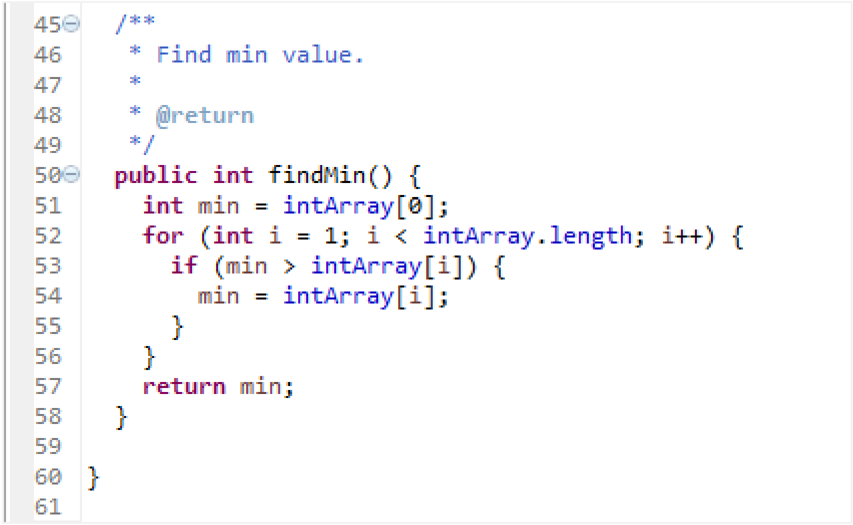
Instance method

- Instance method are methods which require an object of its class to be created before it can be called.
- Access modifiers: same idea as with fields.
  - ✓ `private`/`protected`/`public`/`no modifier`:
- No access modifier:
  - ✓ `abstract:` no implementation given, must be supplied by subclass.
  - ✓ `final`: the method cannot be changed by a subclass (no alternative implementation can be provided by a subclass).

Instance method

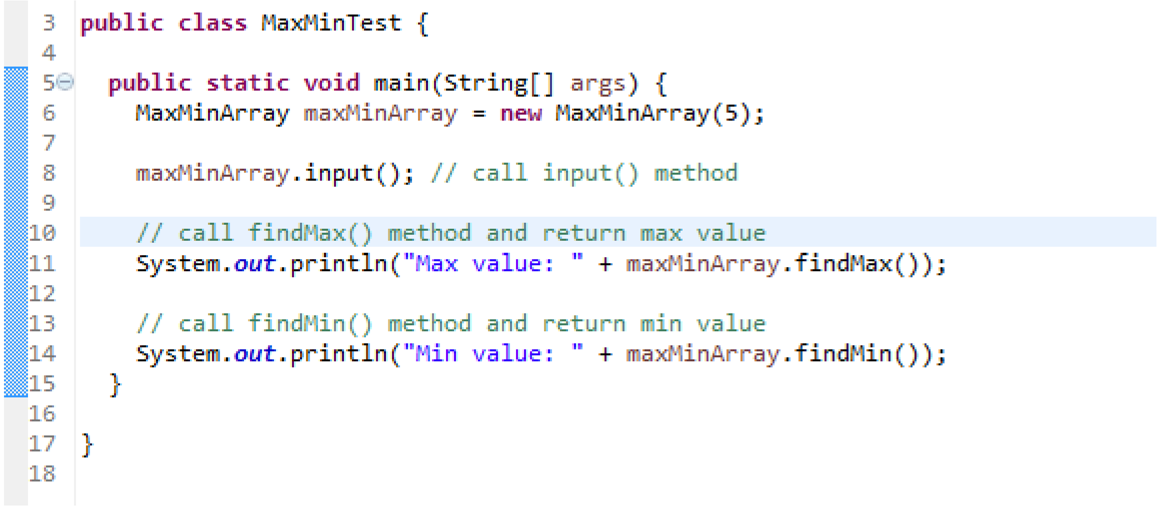


```java
public class MaxMinArray {
  private int[] intArray;

  /**
   * Initialization the Array with length is 'len'.
   *
   * @param len
   */
  public MaxMinArray(int len) {
    intArray = new int[len];
  }

  /**
   * Enter values for elements of the Array.
   */
  @SuppressWarnings("resource")
  public void input() {
    Scanner scanner = new Scanner(System.in);

    for (int i = 0; i < intArray.length; i++) {
      System.out.print("Enter intArray[" + i + "]=");
      intArray[i] = scanner.nextInt();
    }
  }

  /**
   * Find max value.
   *
   * @return
   */
  public int findMax() {
    int max = intArray[0];
    for (int i = 1; i < intArray.length; i++) {
      if (max < intArray[i]) {
        max = intArray[i];
      }
    }
    return max;
  }

```



```java
  /**
   * Find min value.
   *
   * @return
   */
  public int findMin() {
    int min = intArray[0];
    for (int i = 1; i < intArray.length; i++) {
      if (min > intArray[i]) {
        min = intArray[i];
      }
    }
    return min;
  }

}

```

Instance method



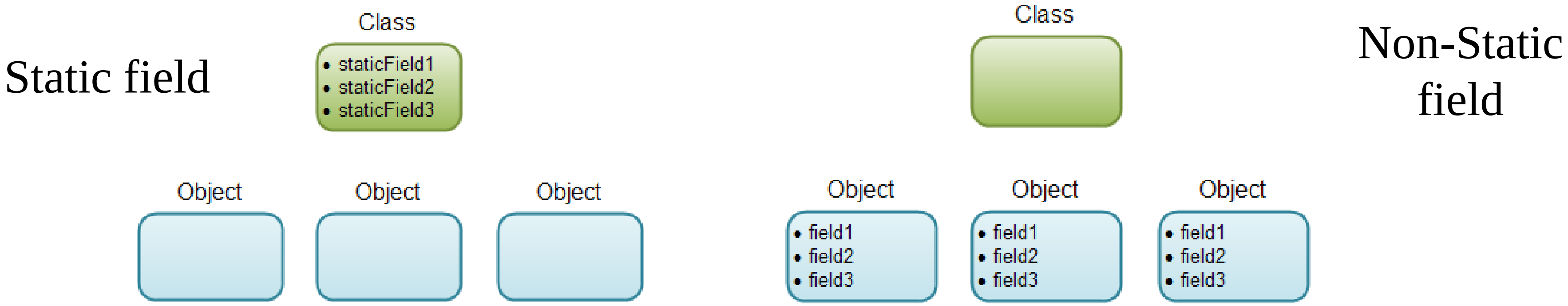
```java
public class MaxMinTest {

  public static void main(String[] args) {
    MaxMinArray maxMinArray = new MaxMinArray(5);

    maxMinArray.input(); // call input() method

    // call findMax() method and return max value
    System.out.println("Max value: " + maxMinArray.findMax());

    // call findMin() method and return min value
    System.out.println("Min value: " + maxMinArray.findMin());
  }

}
```

- Output:

```
Enter intArray[0]=4
Enter intArray[1]=2
Enter intArray[2]=-2
Enter intArray[3]=8
Enter intArray[4]=3
Max value: 8
Min value: -2
```

Static fields

- Fields declared static are called class fields (class variables).
  - ✓ others are called *instance fields*.
- There is only one copy of a static field, no matter how many objects are created.



Static fields Examples

```java
class Student {
    int rollno;
    String name;
    static String college;
    static {
        college = "ITS";
        System.out.println("Static block");
    }

    Student(int rollno, String name) {
        this.rollno = rollno;
        this.name = name;
        System.out.println("Constructor block");
    }

    void display() {
        System.out.println(rollno + " " + name + " " + college);
    }

    static void changeCollege() {
        college = "FU";
    }
}
```

```java
public static void main(String args[]) {
       // Student.changeCollege();
        Student s1 = new Student(111, "Karan");
        Student s2 = new Student(222, "Aryan");
 Student.changeCollege();
        s1.display();
        s2.display();
}
```

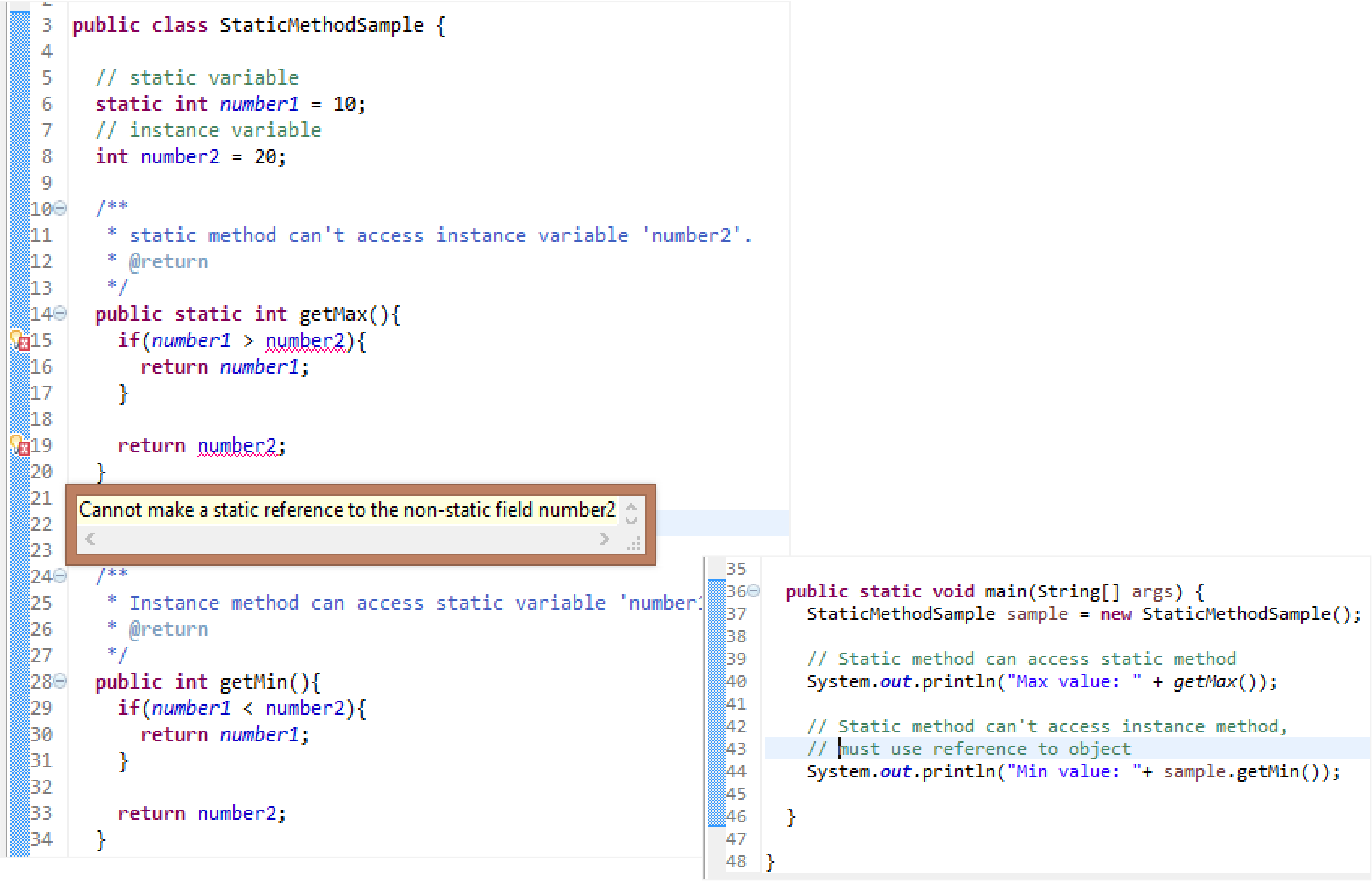
```
111 Karan FU
222 Aryan FU
```

Static methods

- Static methods are the methods in Java that can be called without creating an object of class.
  - ✓ Instance method can access the instance methods and instance variables directly.
  - ✓ Instance method can access static variables and static methods directly.
  - ✓ Static methods can access the static variables and static methods directly.
  - ✓ Static methods can't access instance methods and instance variables directly.
- **Syntax:**

```
static return_type method_name();
```

Static methods



```
public class StaticMethodSample {

  // static variable
  static int number1 = 10;
  // instance variable
  int number2 = 20;

  /**
   * static method can't access instance variable 'number2'.
   * @return
   */
  public static int getMax(){
    if(number1 > number2){
      return number1;
    }

    return number2;
  }
```

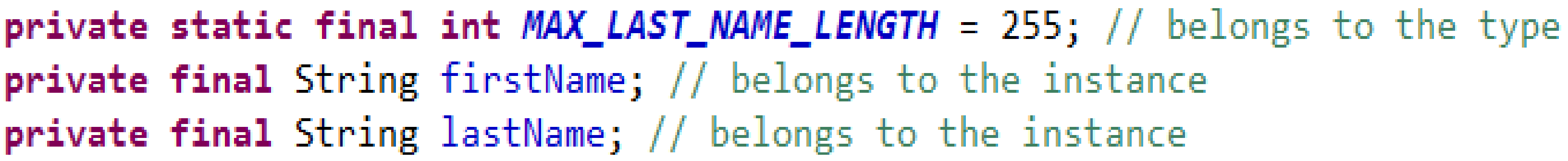
Cannot make a static reference to the non-static field number2

```
  /**
   * Instance method can access static variable 'number1
   * @return
   */
  public int getMin(){
    if(number1 < number2){
      return number1;
    }

    return number2;
  }
```

```

  public static void main(String[] args) {
    StaticMethodSample sample = new StaticMethodSample();

    // Static method can access static method
    System.out.println("Max value: " + getMax());

    // Static method can't access instance method,
    // must use reference to object
    System.out.println("Min value: "+ sample.getMin());

  }

}
```

Final Fields

- The keyword final means: once the value is set, it can never be changed.
  - ✓ They must be ***static*** if they belong to the ***class***.
  - ✓ ***Not be static*** if they belong to the ***instance*** of the class.
- Typically used for constants:

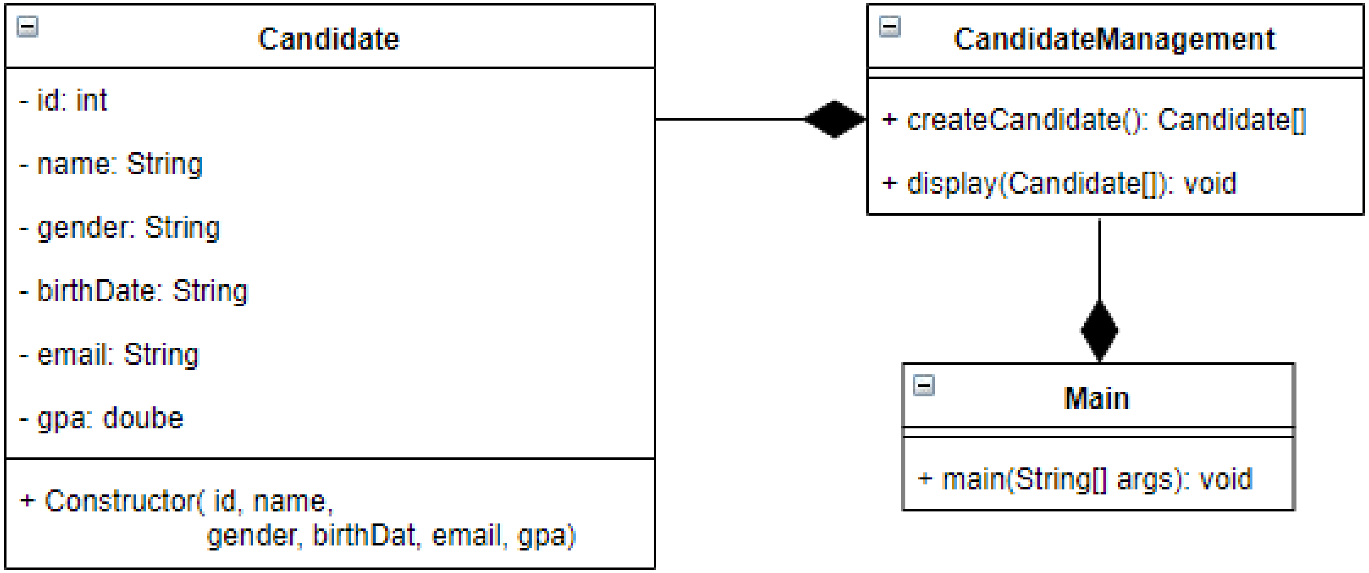


```
private static final int MAX_LAST_NAME_LENGTH = 255; // belongs to the type
private final String firstName; // belongs to the instance
private final String lastName; // belongs to the instance
```

- **Important Note:**
  - ✓ A **final variable** that is not initialized at the time of declaration is known as **blank final variable**.
    - We can **initialize** blank final variable **in constructor**.
  - ✓ A **static final variable** that is **not initialized** at the time of declaration is known as static blank final variable.
    - It can be **initialized** only in **static block**.

Practical time

- Implement the class diagram below by java:



Candidate
- id: int
- name: String
- gender: String
- birthDate: String
- email: String
- gpa: doube
+ Constructor(id, name, gender, birthDat, email, gpa)
CandidateManagement
+ createCandidate(): Candidate[]
+ display(Candidate[]): void
Main
+ main(String[] args): void

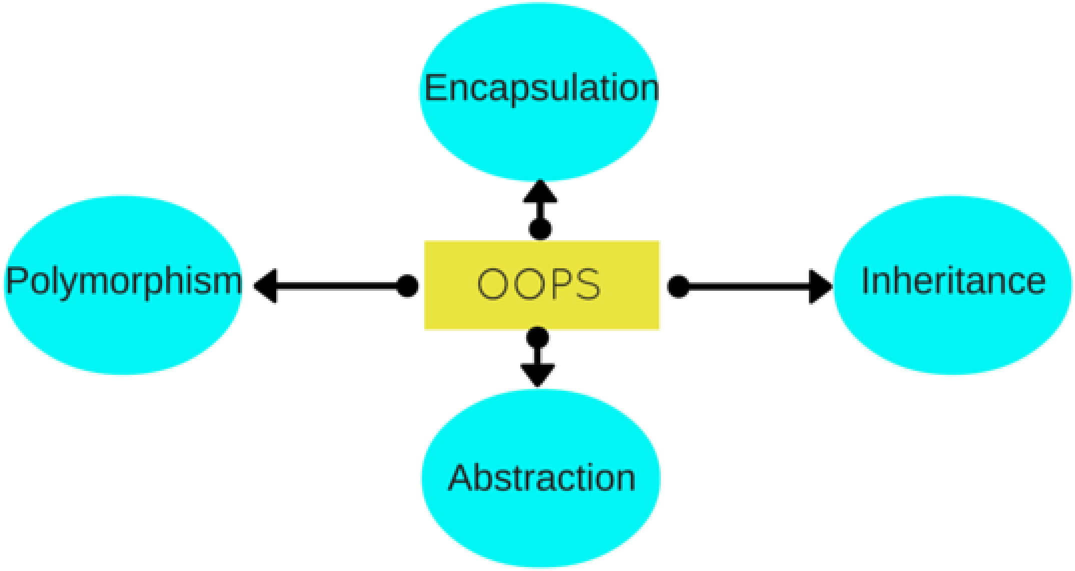
Section 2

Principles of OOP

Principles of OOP

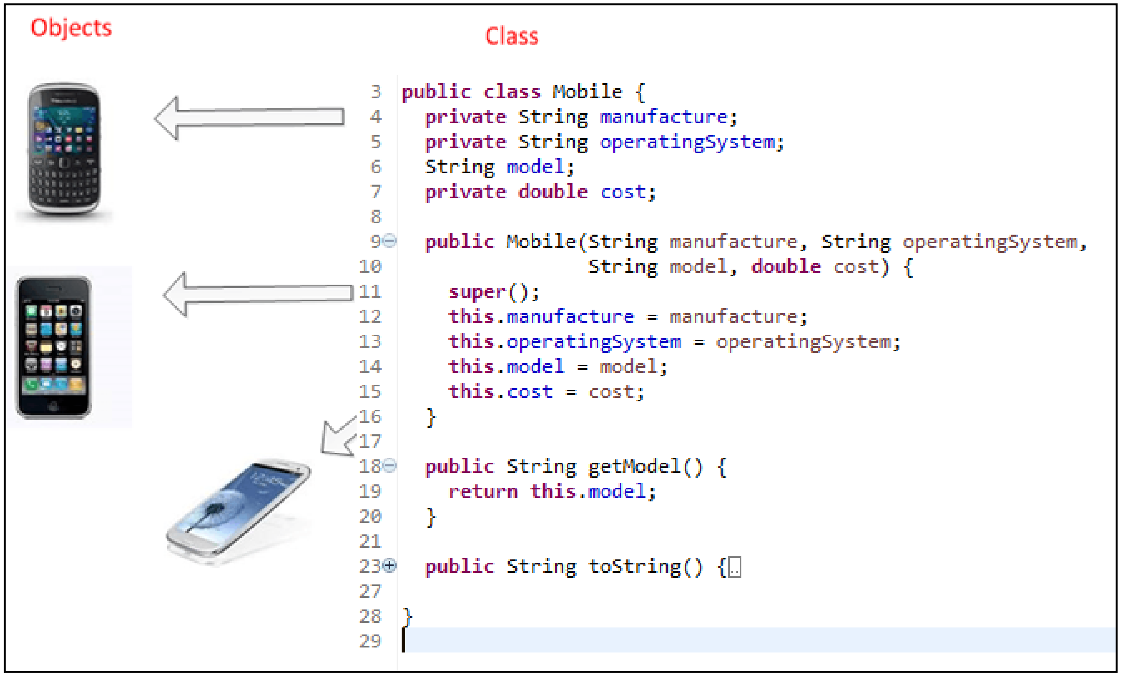
- Java is an **object oriented language** because it provides the features to implement an object oriented model.

- These features includes **Abstraction**, **encapsulation**, **inheritance** and **polymorphism**.



Inheritance

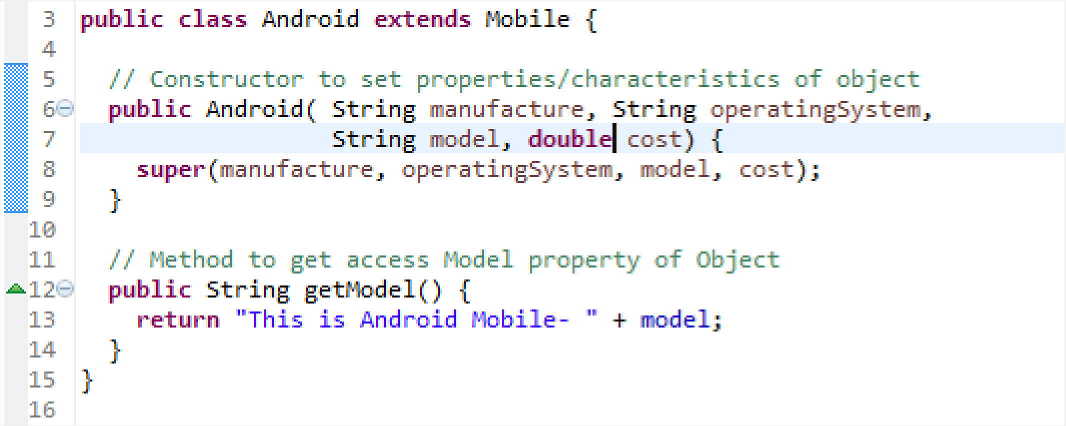
- *When one object acquires all the **properties** and **behaviors** of a **parent object***, it is known as inheritance. It provides code reusability.
- You can look into the following example for inheritance concept.
- **Mobile** class:



```java
public class Mobile {
  private String manufacture;
  private String operatingSystem;
  String model;
  private double cost;

  public Mobile(String manufacture, String operatingSystem,
                String model, double cost) {
    super();
    this.manufacture = manufacture;
    this.operatingSystem = operatingSystem;
    this.model = model;
    this.cost = cost;
  }

  public String getModel() {
    return this.model;
  }

  public String toString() {□

}

```

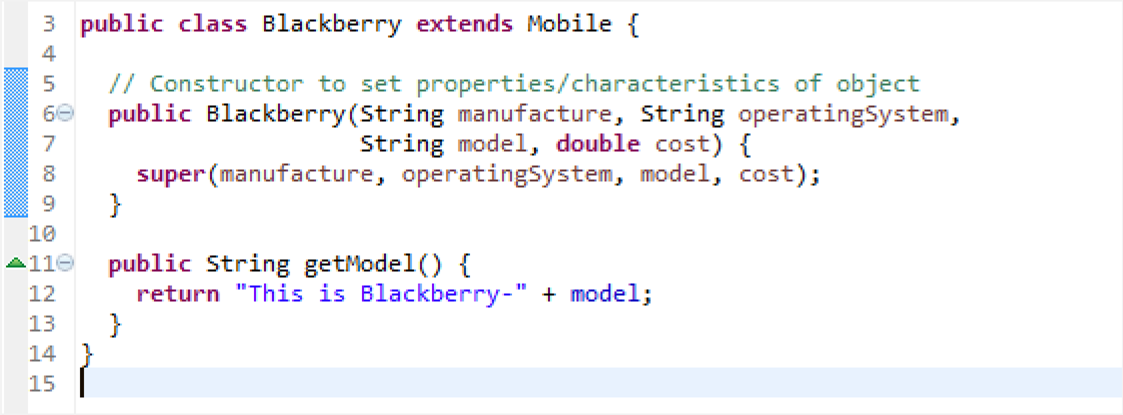
Inheritance

- The **Mobile** class extended by other specific class like **Android** and **Blackberry**.
- **Android** class:



```java
public class Android extends Mobile {

  // Constructor to set properties/characteristics of object
  public Android( String manufacture, String operatingSystem,
                  String model, double cost) {
    super(manufacture, operatingSystem, model, cost);
  }

  // Method to get access Model property of Object
  public String getModel() {
    return "This is Android Mobile- " + model;
  }
}

```

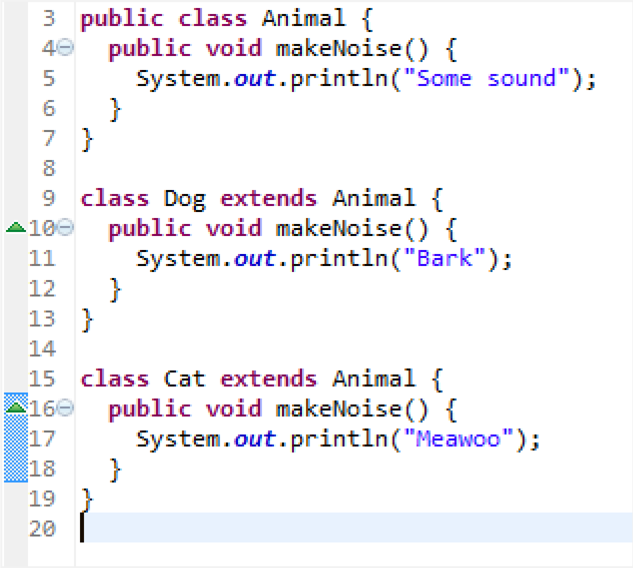
- **Blackberry** class



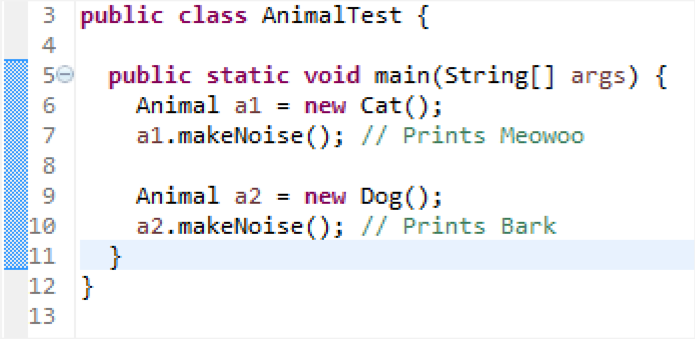
```java
public class Blackberry extends Mobile {

  // Constructor to set properties/characteristics of object
  public Blackberry(String manufacture, String operatingSystem,
                    String model, double cost) {
    super(manufacture, operatingSystem, model, cost);
  }

  public String getModel() {
    return "This is Blackberry-" + model;
  }
}

```

Polymorphism

- If *one task is performed by different ways*, it is known as polymorphism.
  - ✓ Use **method overloading** and **method overriding** to achieve polymorphism.



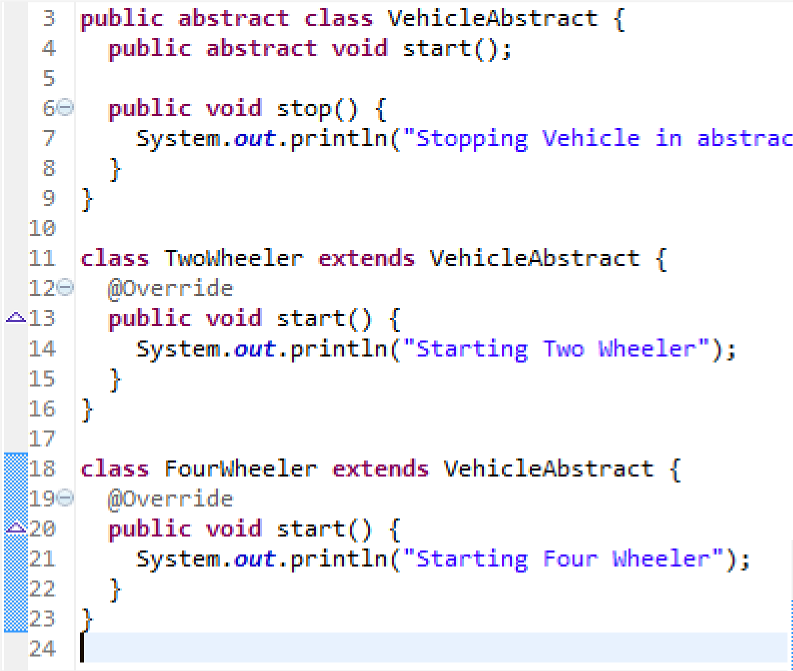
```java
public class Animal {
  public void makeNoise() {
    System.out.println("Some sound");
  }
}

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

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



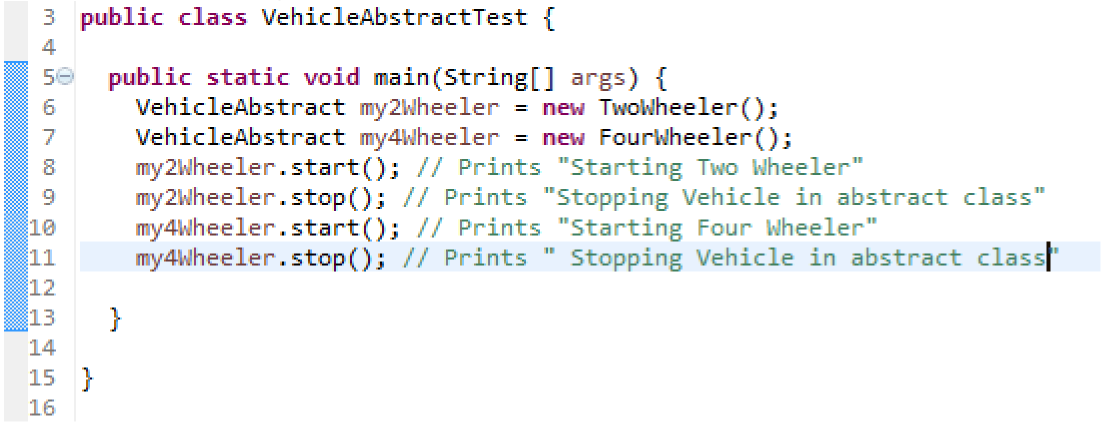
```java
public class AnimalTest {

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

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

Abstraction

- *Hiding internal details and showing functionality* is known as abstraction.
  - ✓ Use **abstract class** and **interface** to achieve abstraction.



```java
public abstract class VehicleAbstract {
  public abstract void start();

  public void stop() {
    System.out.println("Stopping Vehicle in abstract class");
  }
}

class TwoWheeler extends VehicleAbstract {
  @Override
  public void start() {
    System.out.println("Starting Two Wheeler");
  }
}

class FourWheeler extends VehicleAbstract {
  @Override
  public void start() {
    System.out.println("Starting Four Wheeler");
  }
}

```



```java
public class VehicleAbstractTest {

  public static void main(String[] args) {
    VehicleAbstract my2Wheeler = new TwoWheeler();
    VehicleAbstract my4Wheeler = new FourWheeler();
    my2Wheeler.start(); // Prints "Starting Two Wheeler"
    my2Wheeler.stop(); // Prints "Stopping Vehicle in abstract class"
    my4Wheeler.start(); // Prints "Starting Four Wheeler"
    my4Wheeler.stop(); // Prints " Stopping Vehicle in abstract class"

  }

}

```

Encapsulation

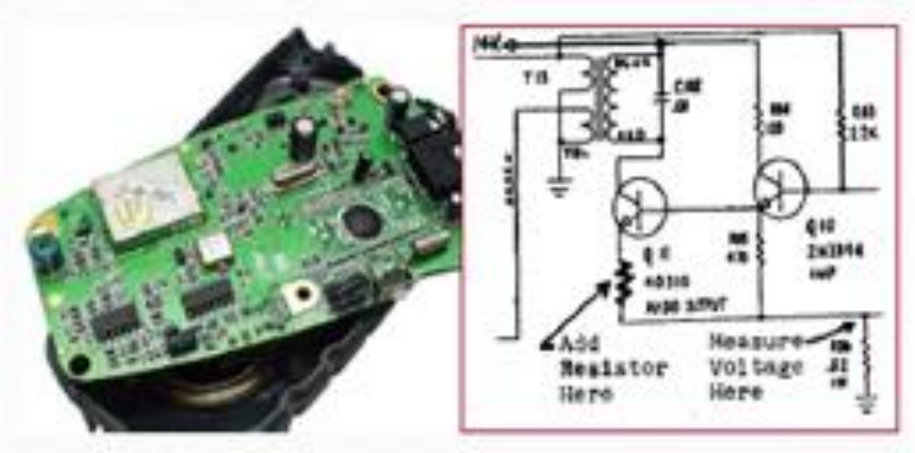
- Encapsulation means putting together all the **variables** (instance variables) and the **methods** into a single unit called **Class**.
- It also means hiding data and methods within an Object.
- A programmer can access and use the **methods** and **data** contained in the **black box but cannot change them**.
- Use access modifier: **private**, **protected**, **default**.

Section 3

Encapsulation

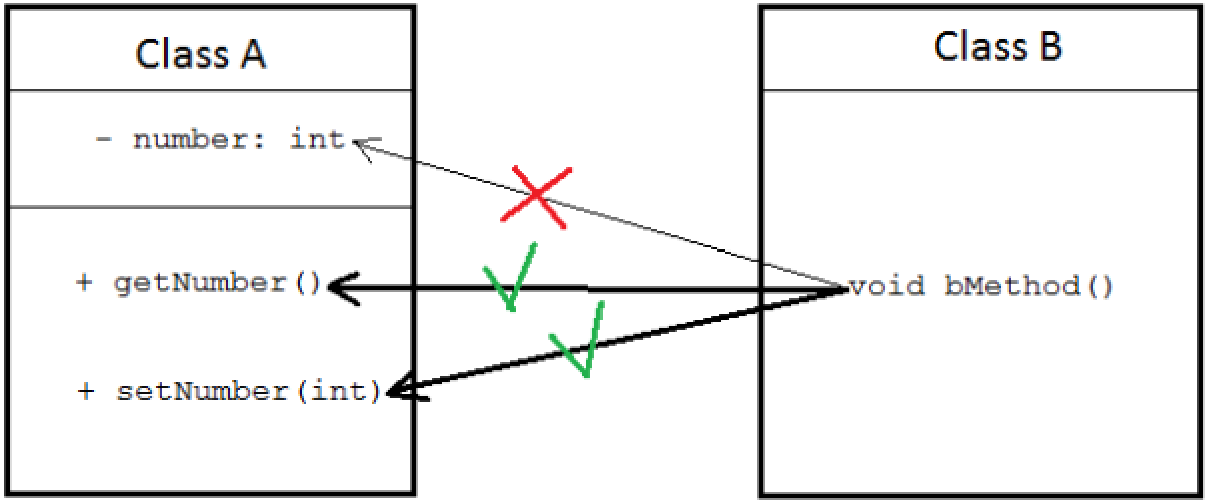
Encapsulation Overview

- **Encapsulation**: Hiding implementation details from clients
  - ✓ Is the technique of making the fields in a class private
  - ✓ Providing access to the fields via public methods.
    - Prevents the *code* and *data* being randomly accessed by other code defined outside the class.
    - The ability to *modify* our implemented code *without breaking* the code of others who use our code.



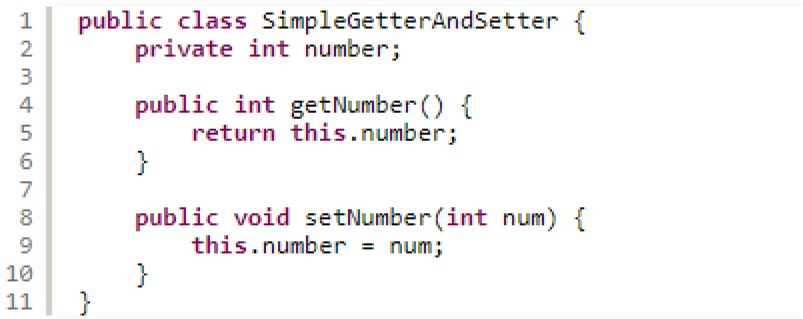
Getter and setter method

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



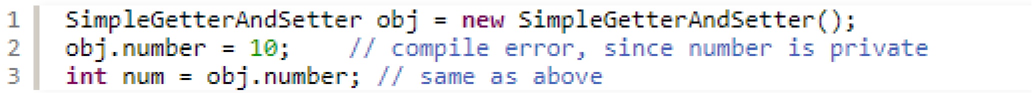
Getter and setter method

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



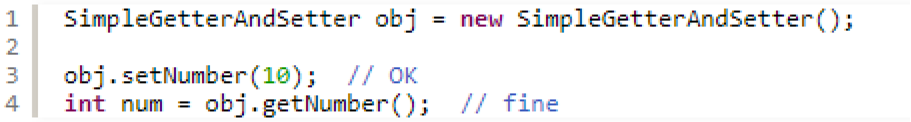
```java
public class SimpleGetterAndSetter {
    private int number;

    public int getNumber() {
        return this.number;
    }

    public void setNumber(int num) {
        this.number = num;
    }
}
```

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



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

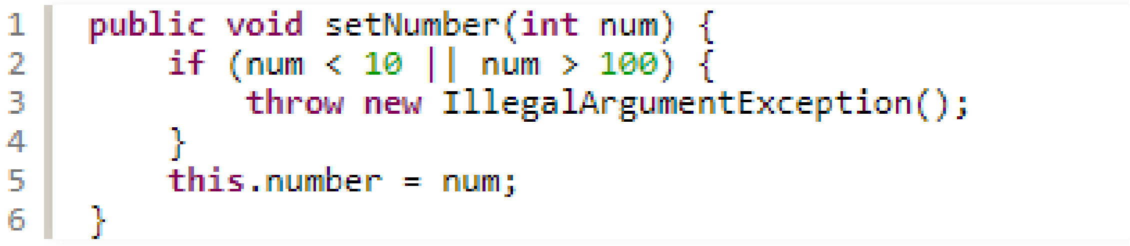
- Instead, the outside code have to invoke the getter, **getNumber**() and the setter, **setNumber**() in order to read or update the variable, for example:



```java
SimpleGetterAndSetter obj = new SimpleGetterAndSetter();

obj.setNumber(10);  // OK
int num = obj.getNumber();  // fine
```

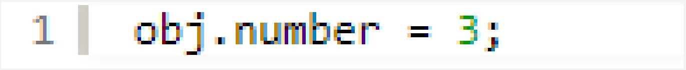
Why getter and setter?

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



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

- ✓ 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:



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

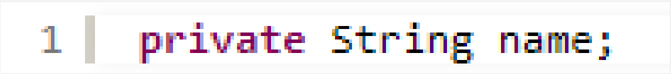
Naming convention for getter and setter

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

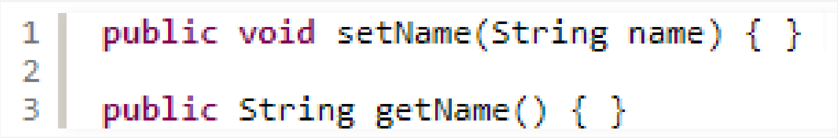
getXXX() and **setXXX()**

  - ✓ where XXX is name of the variable.

- For example with the following variable name:

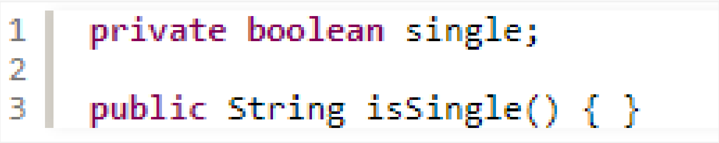


```
private String name;
```



```
public void setName(String name) { }

public String getName() { }
```

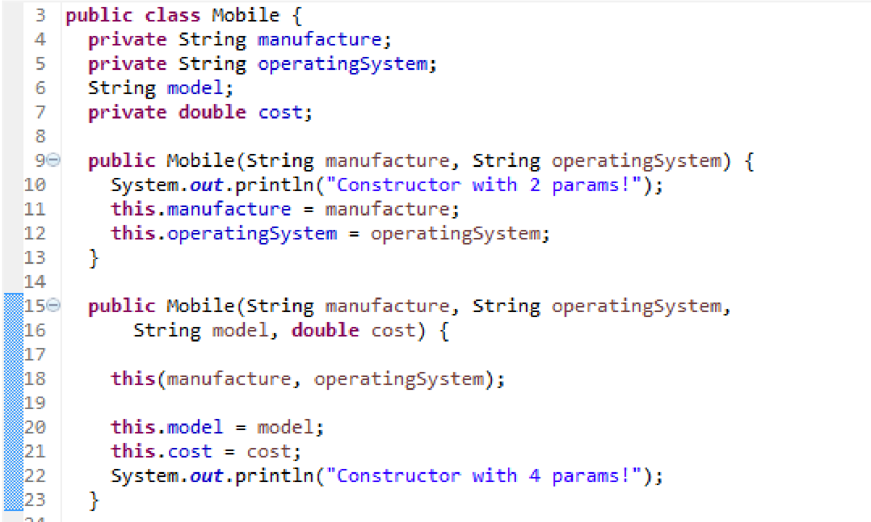
- If the variable is of type boolean, then the getter's name can be either **isXXX**() or **getXXX**(), but the former naming is preferred.



```
private boolean single;

public String isSingle() { }
```

this keyword

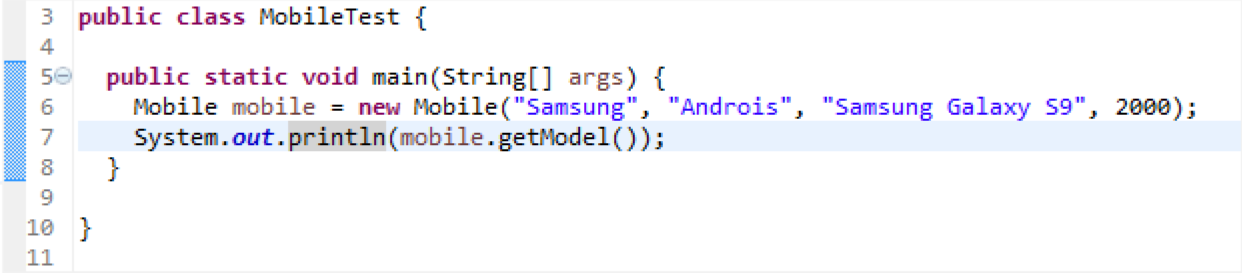
- "**this**" keyword in java can be used inside the *method* or *constructor* of Class.
- It ***(this)*** works as a reference to the current Object, whose Method or constructor is being invoked.
- **this** keyword with a **field** and **constructor:**



```java
public class Mobile {
  private String manufacture;
  private String operatingSystem;
  String model;
  private double cost;

  public Mobile(String manufacture, String operatingSystem) {
    System.out.println("Constructor with 2 params!");
    this.manufacture = manufacture;
    this.operatingSystem = operatingSystem;
  }

  public Mobile(String manufacture, String operatingSystem,
      String model, double cost) {

    this(manufacture, operatingSystem);

    this.model = model;
    this.cost = cost;
    System.out.println("Constructor with 4 params!");
  }

  public String getModel() {
    return this.model;
  }

  public String toString() {

}
```

Output:

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



```java
public class MobileTest {

  public static void main(String[] args) {
    Mobile mobile = new Mobile("Samsung", "Androis", "Samsung Galaxy S9", 2000);
    System.out.println(mobile.getModel());
  }

}
```

Section 4

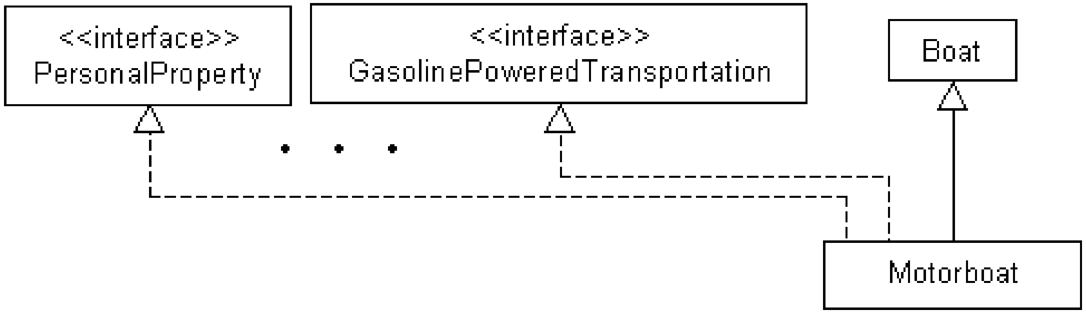
Inheritance

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 Overview (2/2)

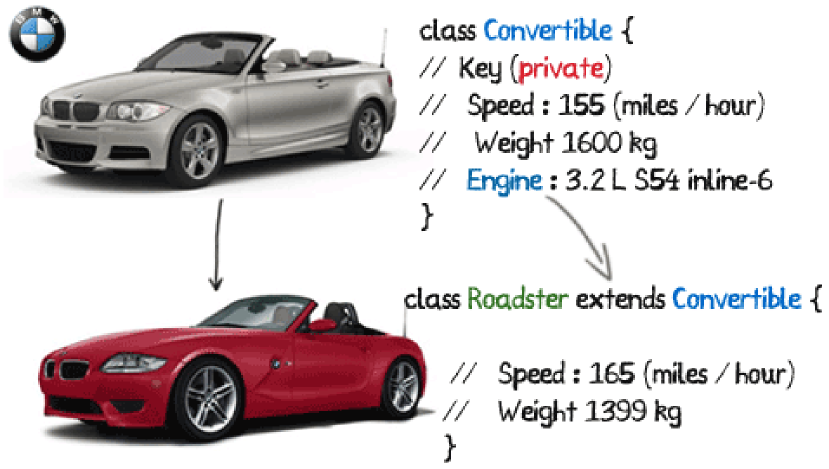
- **Two kinds:**
  - ✓ implementation: the code that defines methods.
  - ✓ interface: the method prototypes only.
- You can't extend more than one class!
  - ✓ the derived class can't have more than one base class.
- You can do multiple inheritance with *interface* inheritance.



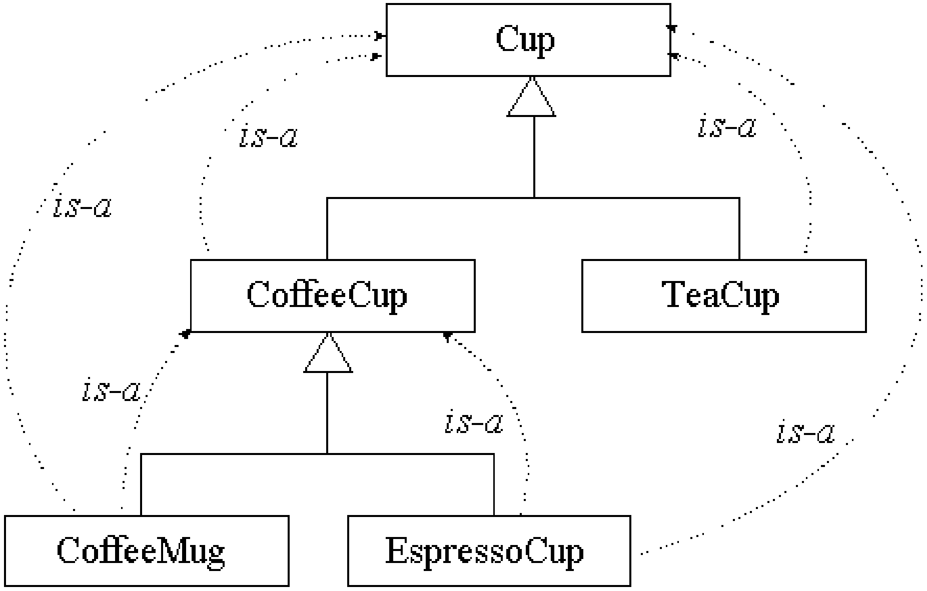
Inheritance Vocabulary (1/3)

- **Inheritance Vocabulary**:
  - ✓ Superclass/Subclass
  - ✓ OOP Hierarchy
  - ✓ Overriding
  - ✓ "isa“ - an instance of a subclass is-a instance of the superclass.

S



```
class Convertible {
// Key (private)
//  Speed : 155 (miles / hour)
//   Weight 1600 kg
//  Engine : 3.2 L S54 inline-6
}

class Roadster extends Convertible {
  //  Speed : 165 (miles / hour)
  //   Weight 1399 kg
  }
```



Inheritance Vocabulary (2/3)

- **"IS-A"**

✓ "IS-A" relationship – this thing **is a** type of that thing

  - Inheritance
  - Subclass object treated as superclass object

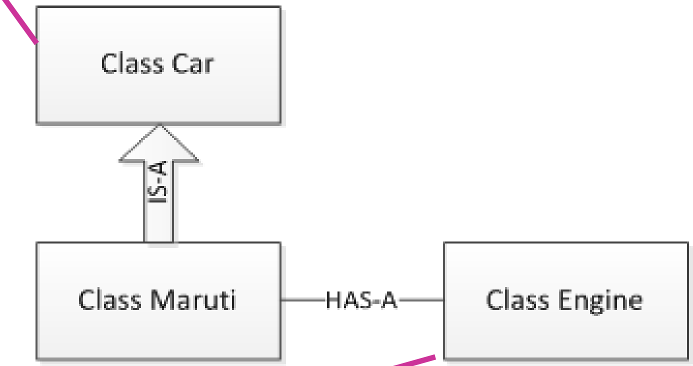
- **"HAS-A"**

✓ "HAS-A" relationship - class A **HAS-A** B if code in class A has a reference to an instance of class B.

  - Aggregation
  - Object contains one or more objects of other classes as members

Example: Maruti *is a* Car

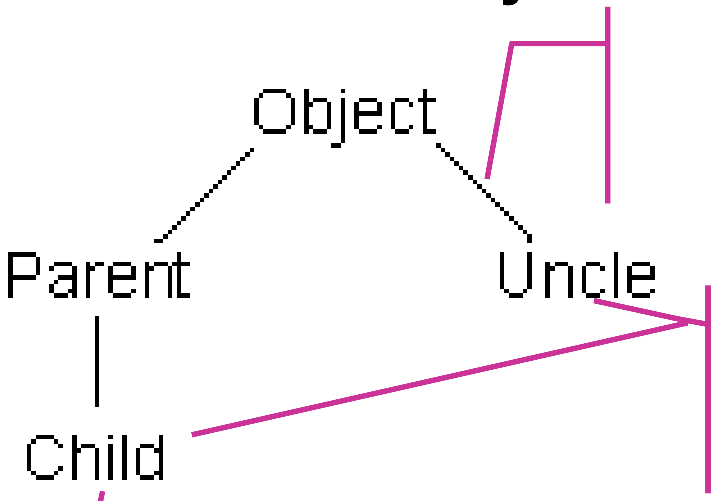
✓ Car properties/behaviors also Maruti properties/behaviors



Example: Maruti *has a* Engine

Inheritance Vocabulary (3/3)

- **Class hierarchy**



Direct superclass[kế *thừa trực tiếp]*:

✓ Inherited explicitly (one level up hierarchy)

Single inheritance

✓ Inherits from one superclass

Indirect (multilevel) superclass

✓ Inherited two or more levels up hierarchy

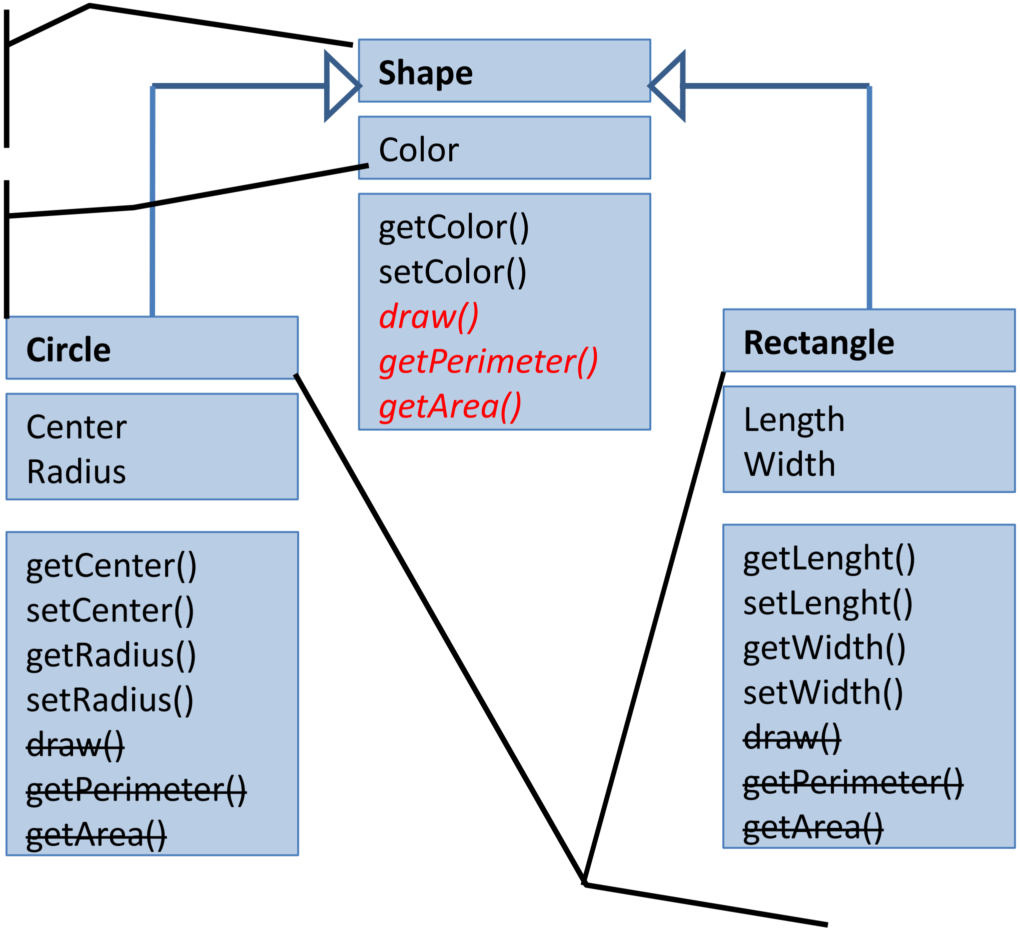
- **Multiple inheritance:**
  - ✓ Inherits from multiple superclasses
    - **Java does not support multiple inheritance in *classes***

Inheritance Example

Shape is superclass

Circle and Rectangle has Color property

- Circle isa Shape, but Shape is not a Circle.
- Method draw(), getPerimeter(), getArea() in Circle overriding method draw() , getPerimeter(), getArea() in Shape.
- If we add/remove property to/from Shape, then it's affected to Circle and Rectangle.



Shape

Color

getColor()
setColor()
draw()
getPerimeter()
getArea()

Circle

Center
Radius

getCenter()
setCenter()
getRadius()
setRadius()
~~draw()~~
~~getPerimeter()~~
~~getArea()~~

Rectangle

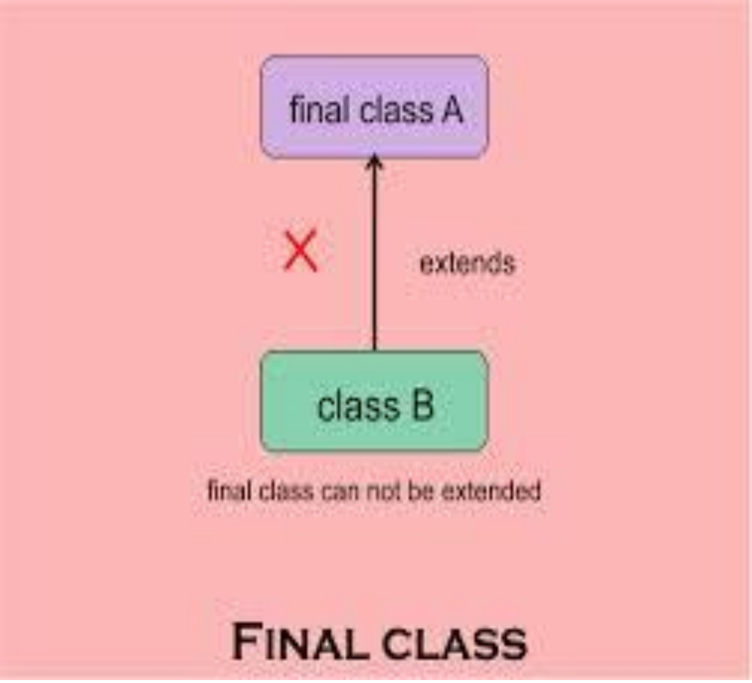
Length
Width

getLenght()
setLenght()
getWidth()
setWidth()
~~draw()~~
~~getPerimeter()~~
~~getArea()~~

Circle and Rectangle are subclasses

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.



Superclasses and Subclasses

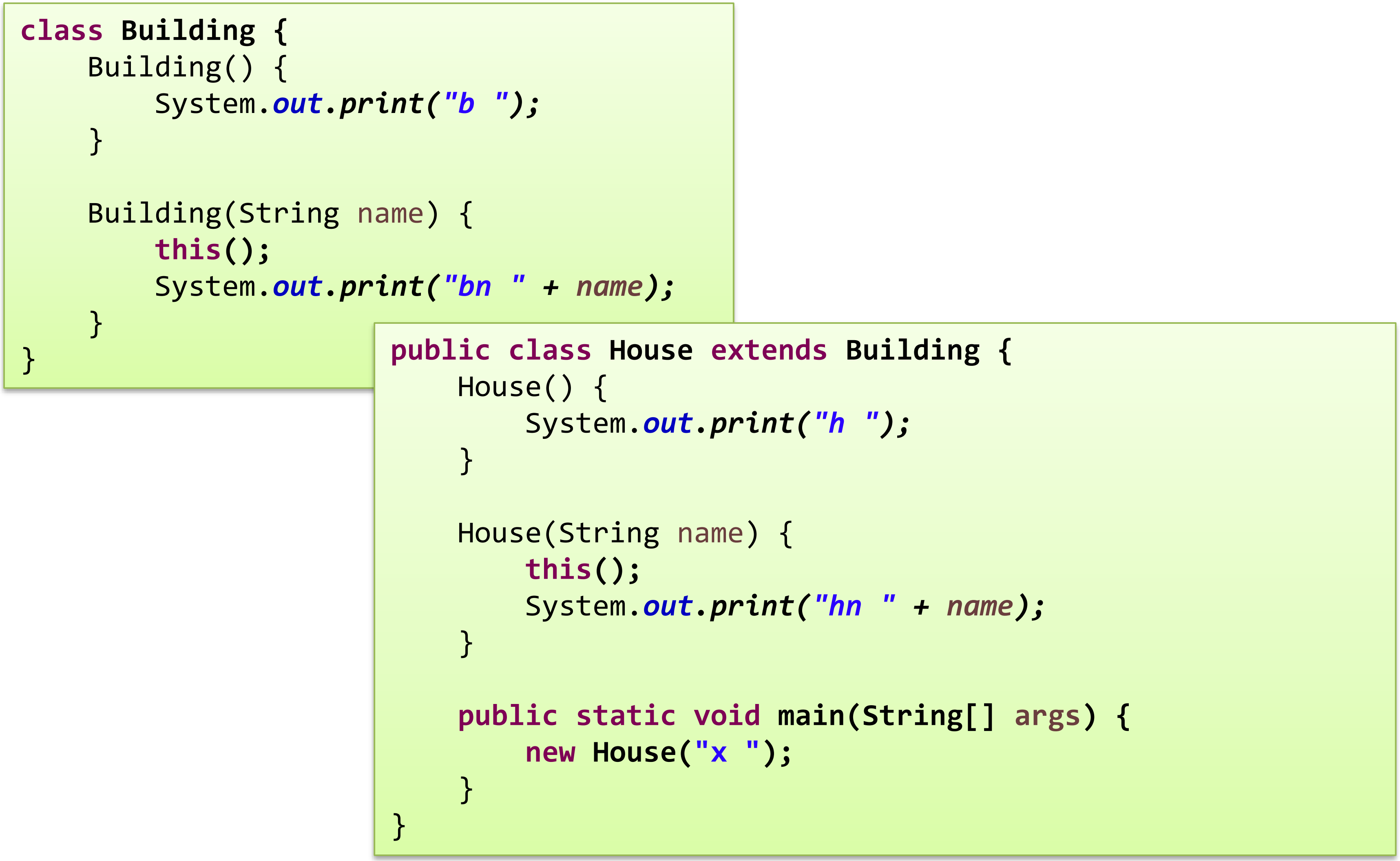
Contructor and Finalizers

- **Instantiating subclass object**
  - ✓ ***Chain of constructor calls***
    - Subclass constructor **invokes** superclass constructor
      - **Implicitly** or **explicitly**
    - Base of inheritance hierarchy
      - Last constructor called in chain is `Object`’s constructor
      - Original subclass constructor’s body finishes executing last.

Superclasses and Subclasses

Contructor and Finalizers

- **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 ");
    }
}
```

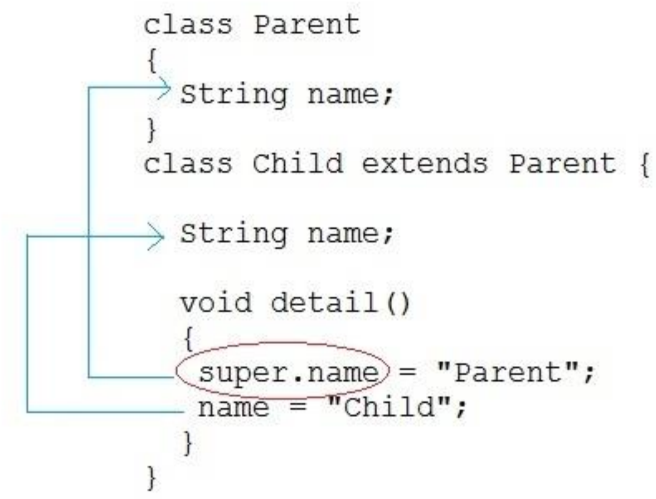
Superclasses and Subclasses

Contructor and Finalizers

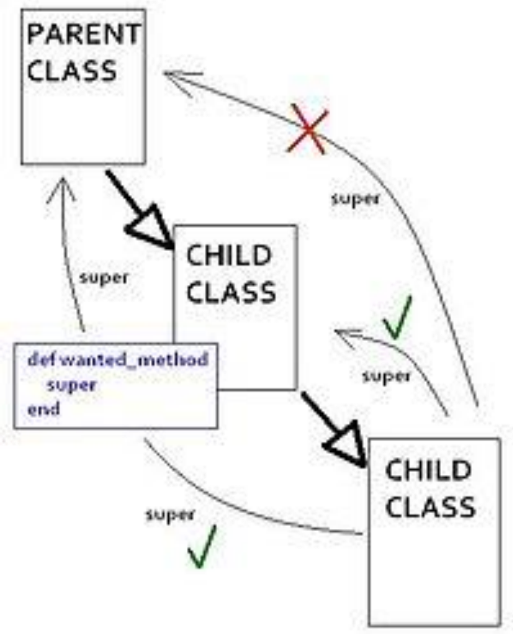
- Garbage collecting subclass object
  - ✓ Chain of `finalize` method calls
    - **Reverse** order of constructor chain
    - Finalizer of **subclass called first**
    - Finalizer of **next superclass** up hierarchy next
      - – Continue up hierarchy until final superreached
        - » After final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

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.



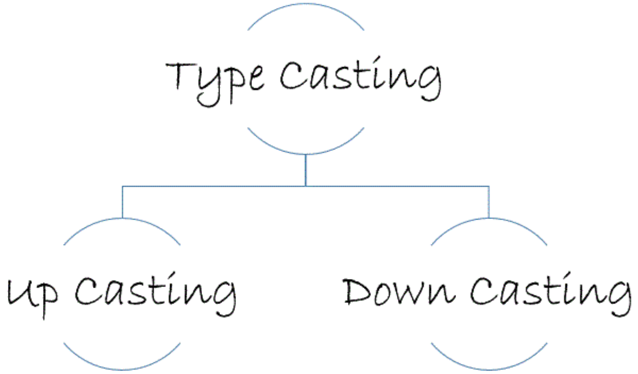
```
class Parent
{
  String name;
}
class Child extends Parent {

  String name;

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

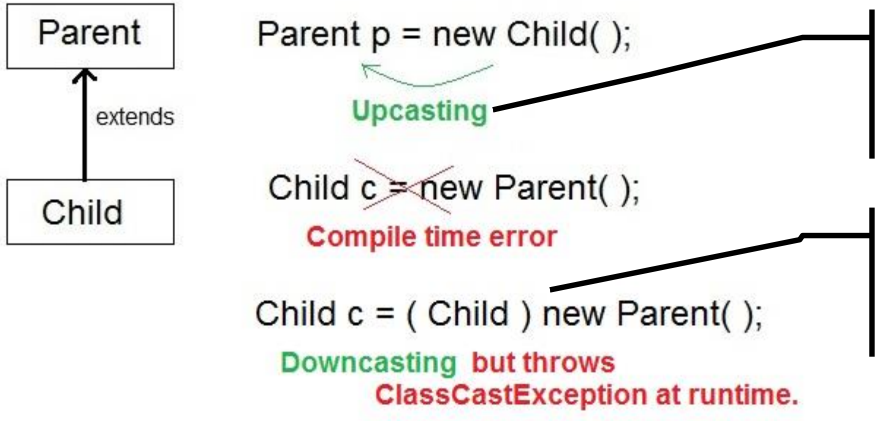


- Subclass methods are **not** superclass methods

Casting Objects

- Java permits[cho phép] an object of a subclass type to be treated as an object of any superclass type. *This is called upcasting.*
- Upcasting and downcasting are NOT like casting primitives from one to other.





```
Parent p = new Child( );
```

Upcasting

```
Child c ~~= new~~ Parent( );
```

Compile time error

```
Child c = ( Child ) new Parent( );
```

Downcasting but throws ClassCastException at runtime.

Upcasting is done automatically.

Downcasting must be manually done by the programmer

Casting Objects Examples

```java
class Animal {
    public void eat() {
        System.out.println("Generic Animal Eating Generically");
    }
}

class Horse extends Animal {
    public void eat() {
        System.out.println("Horse eating hay, oats, " + "and horse treats");
    }

    public void buck() {
        System.out.println("This is buck");
    }
}
```

```java
public class TestAnimals {
    public static void main(String[] args) {
        Animal a = new Animal();
        Animal b = new Horse(); // Animal ref, but a Horse object – upcasting
        Horse h = new Horse();
        a.eat(); // Runs what?
        b.eat(); // Runs what?

        // What is the result?
        Animal c = new Horse();
        c1.buck();           ➔ Cannot invoke subclass-only (Horse) methods on subclass
    }                        object through superclass (Animal) reference
}
```

protected Members in Inheritance (1/2)

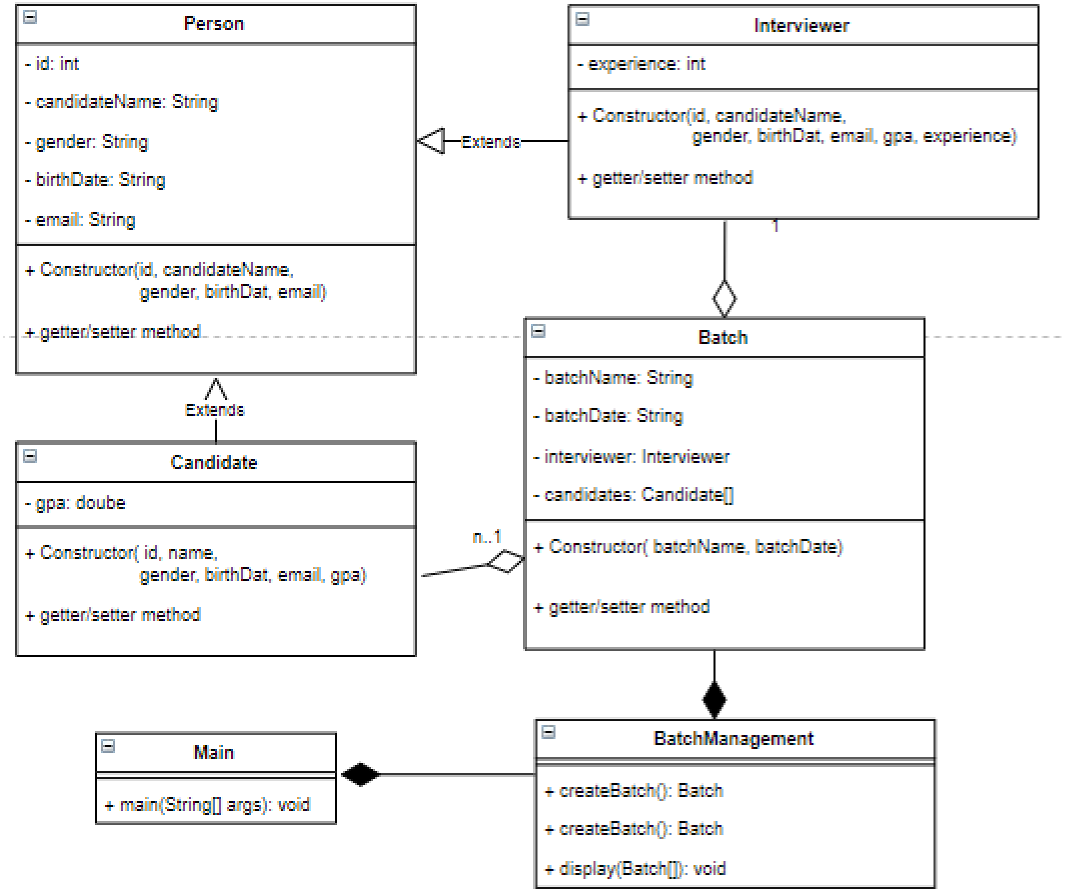
- **protected** access
  - ✓ Intermediate level of protection between **public** and **private;**
  - ✓ **protected** members accessible to:
    - superclass members
    - subclass members
    - Class members in the same package
  - ✓ Subclass access superclass member
    - Keyword **super** and a dot (.)
    - There is **no super.super**….

`protected` Members in Inheritance (2/2)

- Using `protected` instance variables
  - ✓ **Advantages**
    - subclasses can modify values directly
    - Slight increase in performance
      - – Avoid set/get function call overhead
  - ✓ **Disadvantages**
    - No validity checking
      - – subclass can assign illegal value
    - Implementation dependent
      - – subclass methods more likely dependent on superclass implementation
      - – superclass implementation changes may result in subclass modifications

Practice time

- In class diagrams, as shown in following Figure. Let’s implement it using Java:



Person
- id: int
- candidateName: String
- gender: String
- birthDate: String
- email: String

+ Constructor(id, candidateName, gender, birthDat, email)
+ getter/setter method

Interviewer
- experience: int

+ Constructor(id, candidateName, gender, birthDat, email, gpa, experience)
+ getter/setter method

Candidate
- gpa: doube

+ Constructor(id, name, gender, birthDat, email, gpa)
+ getter/setter method

Batch
- batchName: String
- batchDate: String
- interviewer: Interviewer
- candidates: Candidate[]

+ Constructor(batchName, batchDate)
+ getter/setter method

Main

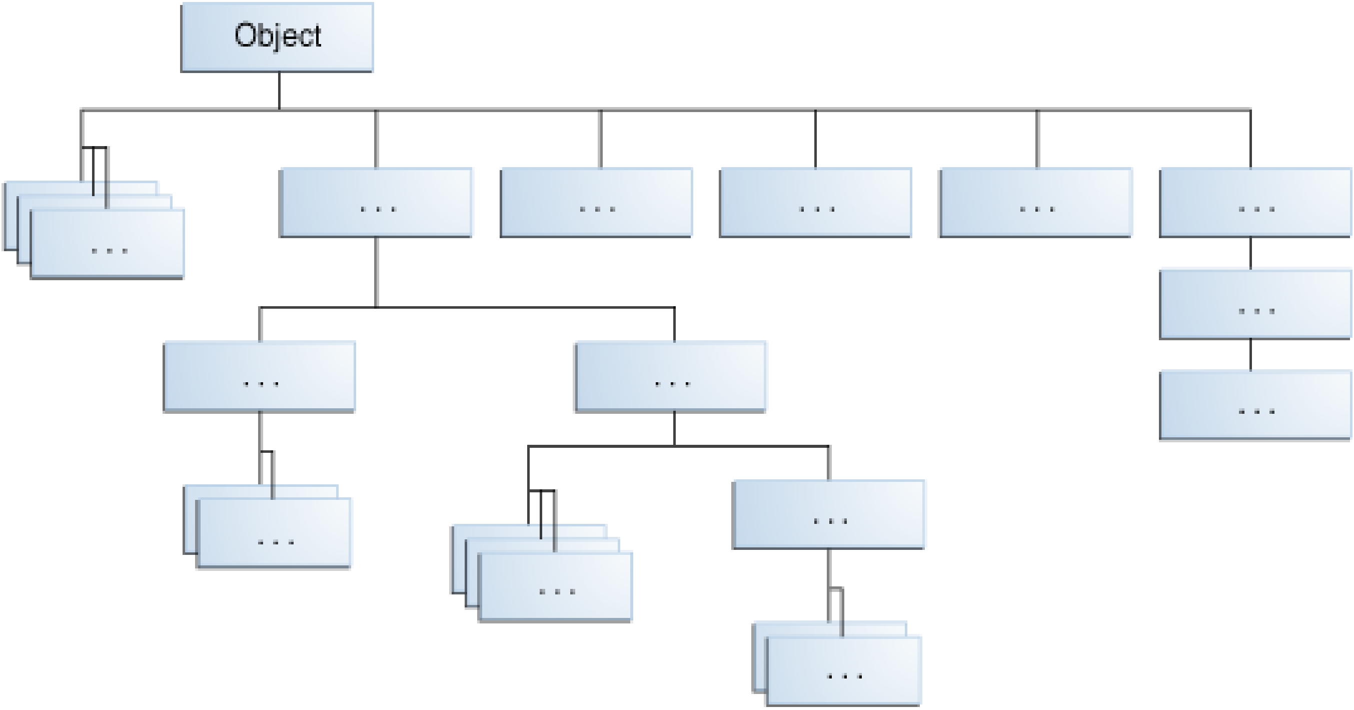
+ main(String[] args): void

BatchManagement

+ createBatch(): Batch
+ createBatch(): Batch
+ display(Batch[]): void

The class Object

- Granddaddy of all Java classes.
- All methods defined in the class Object are available in every class.
- Any object can be cast as an **Object**.



Summary

- Inheritance is a mechanism that allows one class to **reuse** the implementation provided by another.
- A class always **extends** exactly one superclass.
  - ✓ If a class does not explicitly extend another, it implicitly extends the class Object.
- A superclass method or field can be accessed using a **super**. keyword.
- Subclass objects can not access superclass’s private data unless they change into **protected** access level.
- If a constructor does not explicitly invoke another (this() or super()) constructor, it implicitly invokes the superclass's no-args constructor.
- Encapsulation:
  - ✓ Hiding implementation details from clients.

Thank you