OBJECT-ORIENTED LANGUAGE AND THEORY

6. AGGREGATION AND INHERITANCE

Nguyen Thi Thu Trang

trangntt@soict.hust.edu.vn

Tái sử dụng mã nguồn?

- Copy paste
- Viết hàm
- Thư viện, package…

How to re-use source code?

```
~~~~~~
~~x~~
~~~~~~
```

1. Copy and paste

```
~~~~~~
~~x~~
~~~~~~

~~~~~~
~~x~~
~~~~~~
```

```
~~~~~~
~~x~~
~~~~~~
```

2. Function (structural programming/procedural programming)

```
f1{~~~~~~
~~x~~
~~~~~~}
```

```
call f1
call f1
```

```
call f1
call f1
```

3. Class and Object (object-oriented programming)

Inheritance/Generalization   Association (**Aggregation**/Composition)

Lesson Goals

- Explaining concepts of source code re-usability
- Showing the nature, description of concepts relating to aggregation and inheritance
- Comparison of aggregation and inheritance
- Representing aggregation and inheritance in UML
- Explaining principles of inheritance and initialization order, object destruction in inheritance
- Applying techniques, principles of aggregation and inheritance in Java programming language

Outline

1. Source code re-usability
2. Aggregation
3. Inheritance

1. Re-usability

- Source code re-usability: re-use already existing source code
  - Structure programming: Re-use function/sub-program
  - OOP: When modeling real world, there exist many object types that have similar or related attributes and behaviors

  → *How to re-use already-written classes?*

1. Re-usability (2)

- How to use existing classes:
  - *Copying existing classes* → Redundant and difficult to manage if any changes
  - Creating new classes that re-use of **objects** of existing classes → **Aggregation**
  - Creating new classes based on the extension of existing **classes** → **Inheritance**

1. Re-usability (2)

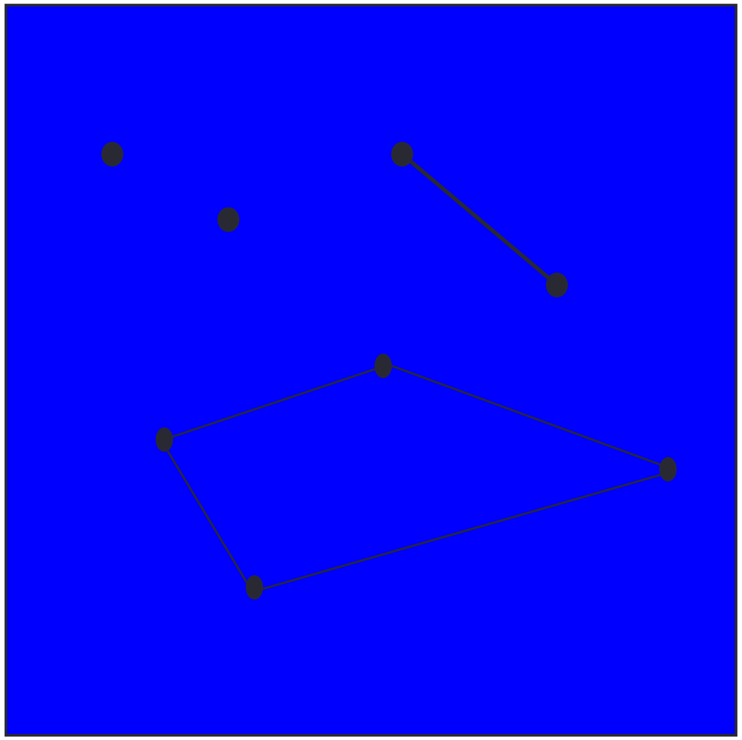
- Advantages
  - Reducing man-power, cost.
  - Improving software quality
  - Improving modeling capacity of the real world
  - Improving maintainability

Outline

1. Source code re-usability
2. Aggregation
3. Inheritance

2. Aggregation

- Example:
  - Point
    - A Quadrangle consists of 4 points
      → Aggregation
- Aggregation
  - Has-a or **is-a-part-of** relations

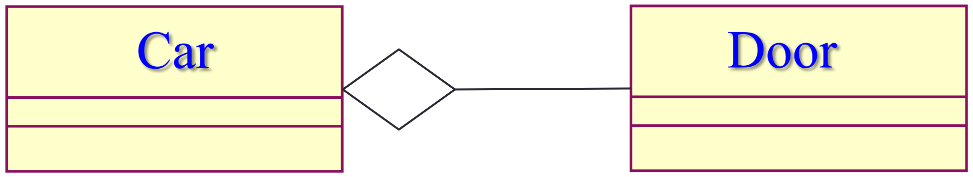


Main terms

- Aggregate
  - Members of a new class are objects of existing classes.
  - Aggregation re-uses via *objects*
- New class
  - Called Aggregate/Whole class
- Existing class
  - Member class (part)

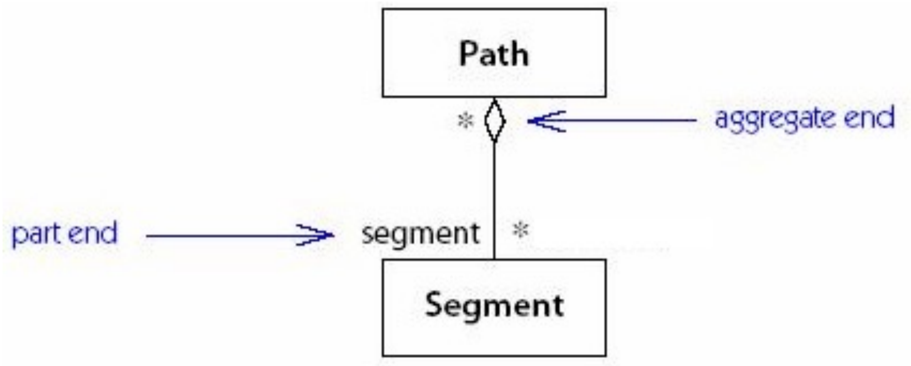
2.1. What is aggregation?

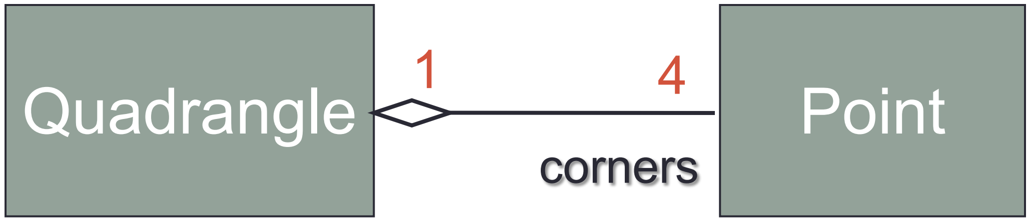
- The whole class contains objects of member classes
  - Is-a-part of the whole class
  - Re-use data and behavior of member classes via member objects

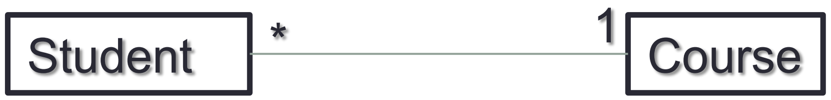


2.2. Representing aggregation in UML

- Using "diamond" at the head of whole class
- Using multiplicity at two heads:
  - A positive integer: 1, 2,...
  - A range (0..1, 2..4)
  - *: Any number
  - None: By default is 1



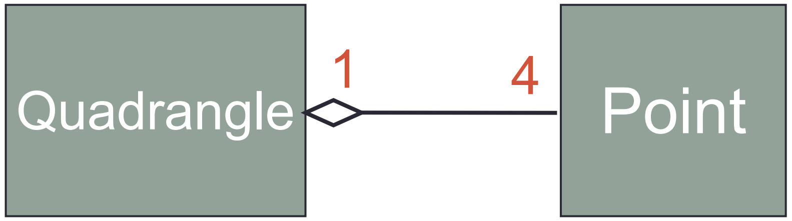



Student
*
1
Course

2.3. Example in Java

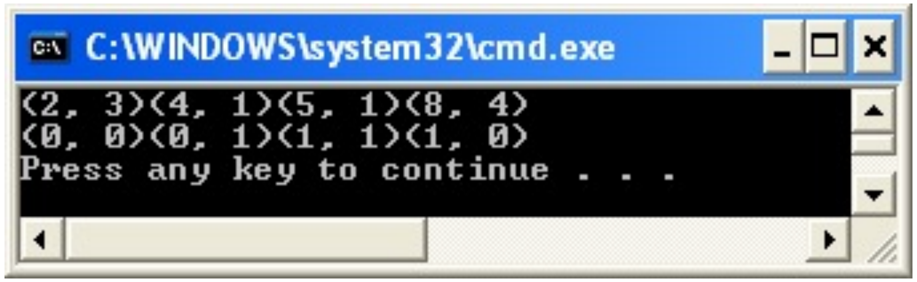
```java
class Point {
  private int x, y;
  public Point(){}
  public Point(int x, int y) {
     this.x = x; this.y = y;
  }
  public void setX(int x){ this.x = x; }
  public int getX() { return x; }
  public void print(){
      System.out.print("(" + x + ", "
                            + y + ")");
  }
}
```

```
class Quadrangle{
  private Point[] corners = new Point[4];
  public Quadrangle(Point p1,Point p2,Point p3,Point p4){
    corners[0] = p1; corners[1] = p2;
    corners[2] = p3; corners[3] = p4;
  }
  public Quadrangle(){
    corners[0]=new Point();    corners[1]=new Point(0,1);
    corners[2]=new Point(1,1); corners[3]=new Point(1,0);
  }
  public void print(){
    corners[0].print();  corners[1].print();
    corners[2].print();  corners[3].print();
    System.out.println();
  }
}
```



```
public class Test {
   public static void main(String arg[])
   {
      Point p1 = new Point(2,3);
      Point p2 = new Point(4,1);
      Point p3 = new Point(5,1);
      Point p4 = new Point(8,4);

      Quadrangle q1 = new Quadrangle(p1,p2,p3,p4);
      Quadrangle q2 = new Quadrangle();
      q1.print();
      q2.print();
   }
}
```

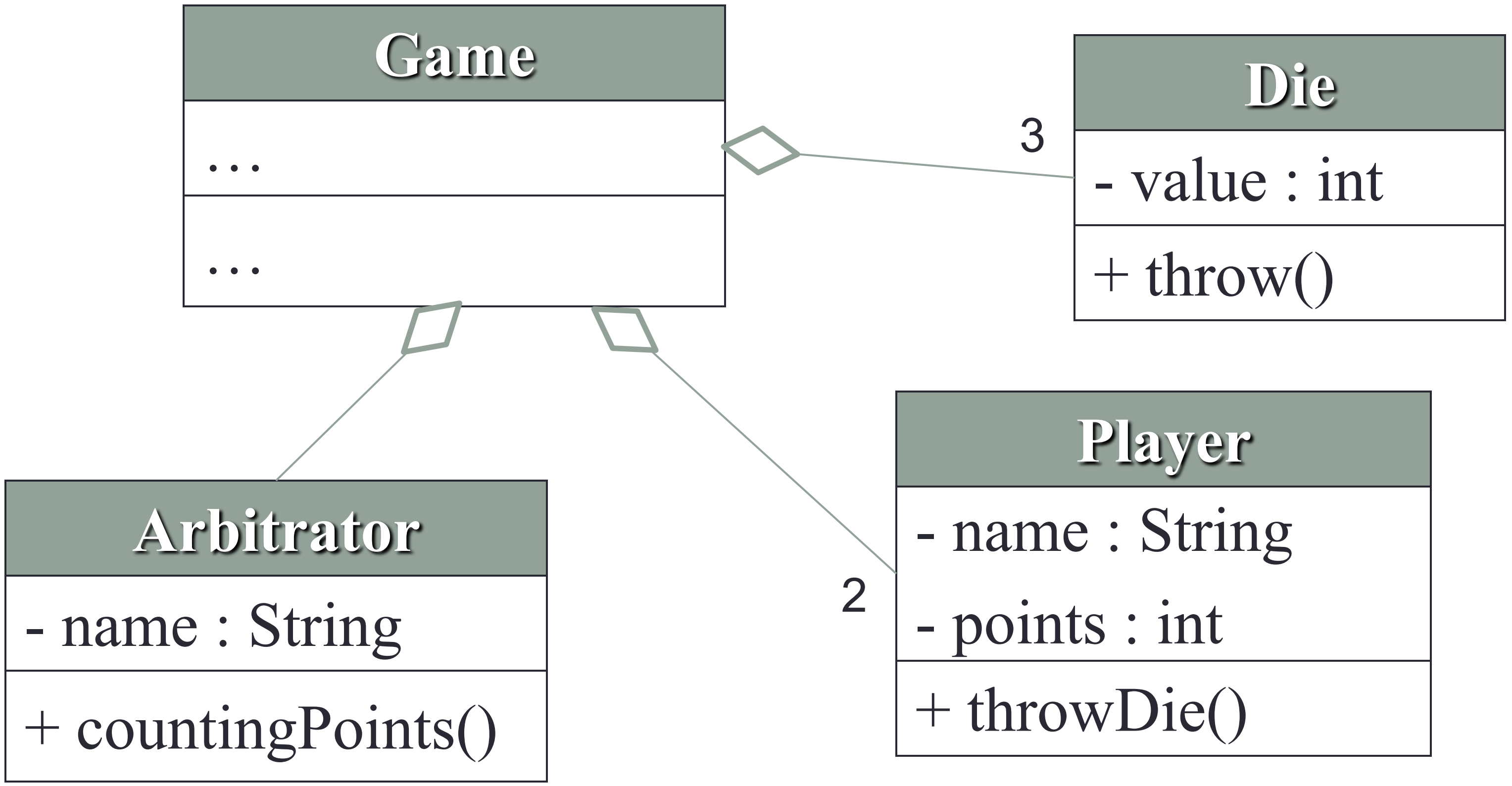


```
(2, 3)(4, 1)(5, 1)(8, 4)
(0, 0)(0, 1)(1, 1)(1, 0)
Press any key to continue . . .
```

Another example of Aggregation

- A game consisting of two players, 3 dies and an artitrator.
- Need 4 classes:
  - Player
  - Die
  - Arbitrator
  - Game

→ Game class is the aggregation of the 3 remaining classes



```
class Game
{
        Die die1, die2, die3;
        Player player1, player2;
        Arbitrator arbitrator1;
        ...
        Game(){//initialize for all members}
}
```

2.4. Initialization order in aggregation

- When an object is created, the attributes of that object must be initialized and assigned corresponding values.
- Member attributes must be initialized first

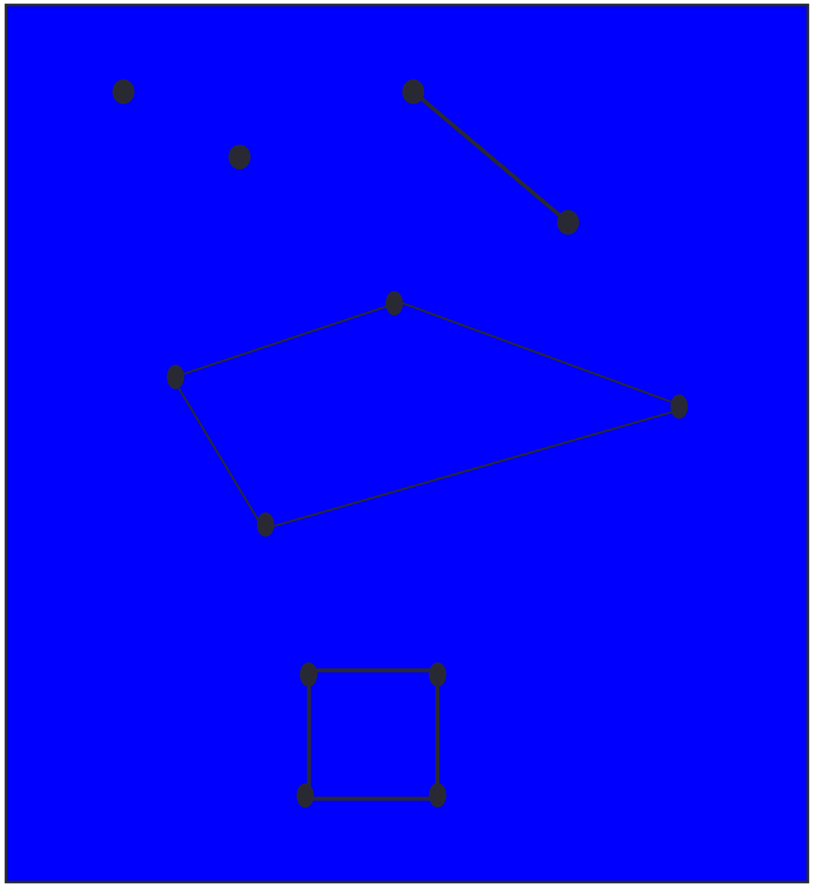
→ Constructor methods of member classes must be called first

Outline

1. Source code re-usability
2. Aggregation
3. Inheritance

3.1. What is Inheritance?

- Example:
  - Point
    - A quadrangle has 4 points
    - →Aggregation *(is a part of)*
  - →Quadrangle
    - Square
    - →Inheritance *(is a kind of)*
    - →Generalization



Main terms

- Inherit, Derive
  - Creating new class by extending existing classes.
  - New class inherits what are in existing classes and can have its own new features.
- Existing class:
  - Parent, superclass, base class
- New class:
  - Child, subclass, derived class

What is Inheritance?

- Principles to describe a class based on the extension of an existing class (single inheritance) or a set of existing classes (in case of multi-inheritance)
- Inheritance specifies a relationship between classes when a class shares it structure and/or behavior of a class or of other classes
- Inheritance is also called is-a-kind-of (or is-a) relationship
    - Child is a kind of parent

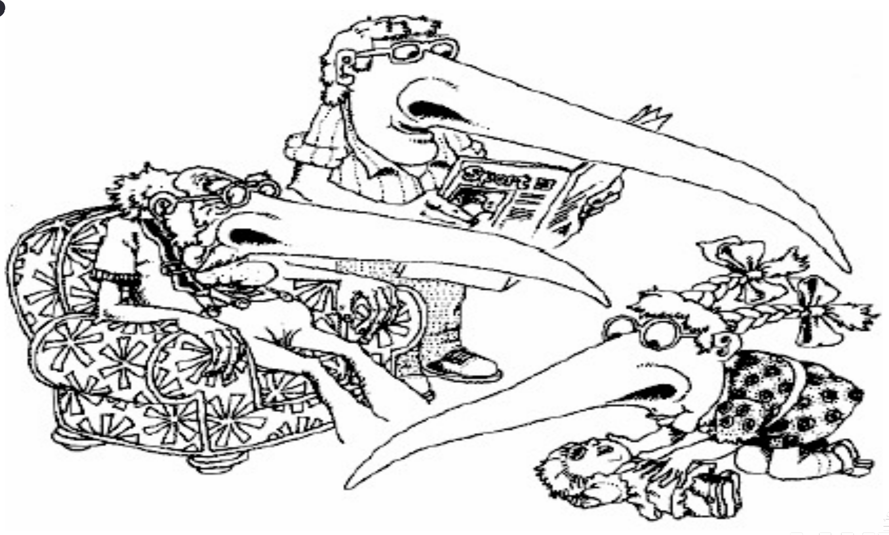
What is Inheritance?

- On "modularization" view: If B inherits A, all services of A will be available in B
- On "type" view: If B inherits A, at anywhere a representation of A is required, the representation of B might be a good replacement.

=> Polymorphism

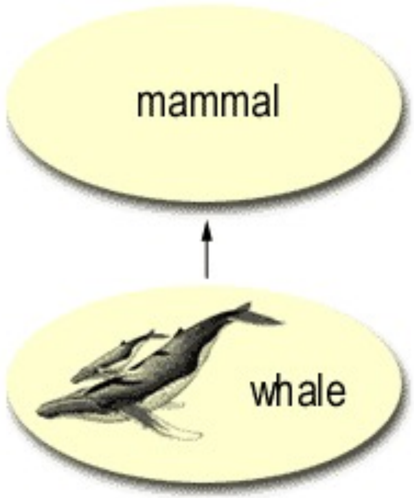
Child classes?

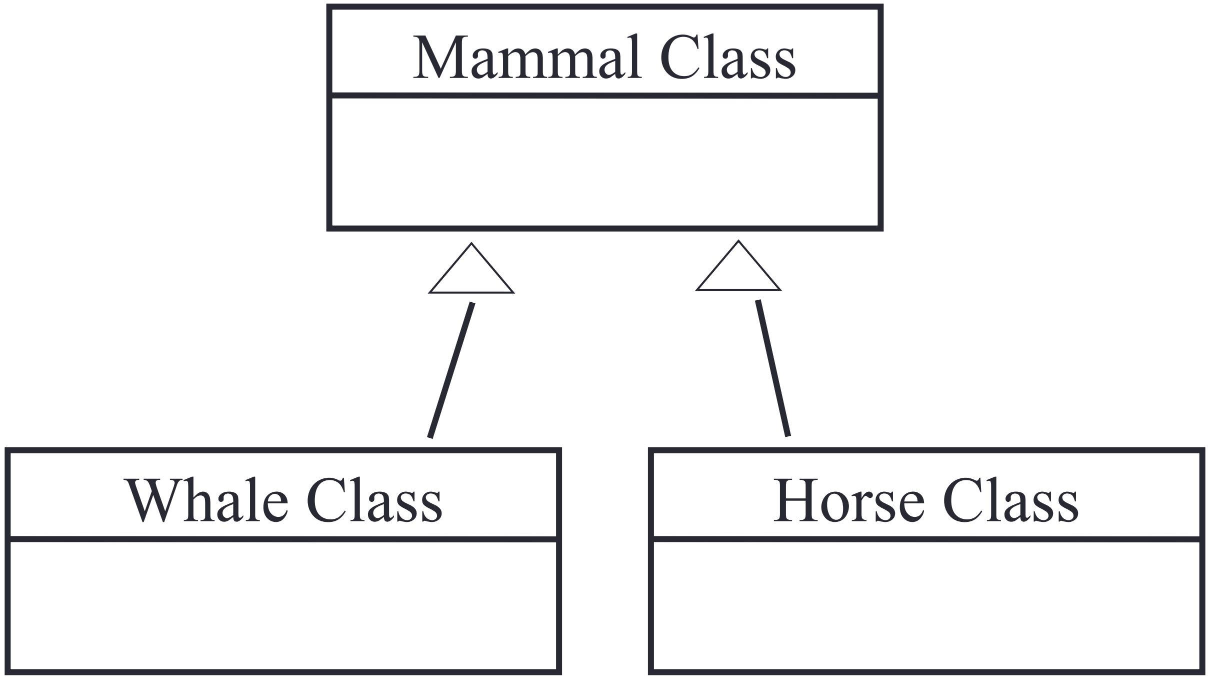
- Re-use by inheriting data and behavior of parent classes
- Can be customized in two ways (or both):
  - Extension: Add more new attributes/behaviors
  - Redefinition (Method Overriding): Modify the behavior inheriting from parent class



More example

- Whale class inherits from mammal class.
- A whale *is-a* mammal
- Whale class is *subclass*, mammal class is *superclass*





Similarity

- Both Whale and Horse have *is-a* relation with mammal class
- Both Whale and Horse have some common behaviors of Mammal
- Inheritance is a key to re-use source code – If a parent class is created, the child class can be created and can add some more information

3.2. Aggregation and Inheritance

- Comparing aggregation and inheritance?
  - Similarity
    - Both are techniques in OOP in order to re-use source code
  - Difference?

Difference between Aggregation and Inheritance

Inheritance

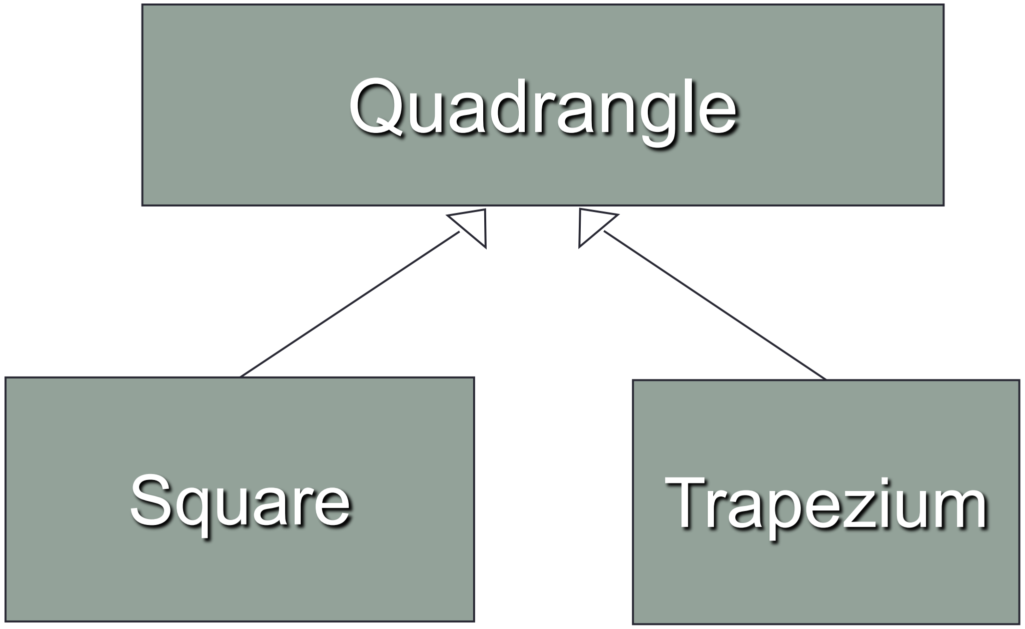
- Inheritance re-uses via class.
  - Creating new class by extending exisiting classes
- "is a kind of" relation
- Example: Car is a kind of transportation mean

Aggregation

- Aggregation re-uses via objects.
  - Create a reference to objects of existing classes in the new class
- "is a part of" relation
- Example: Car has 4 wheels

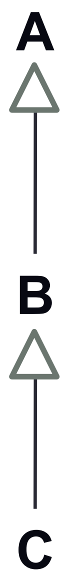
3.3. Representing Inheritance in UML

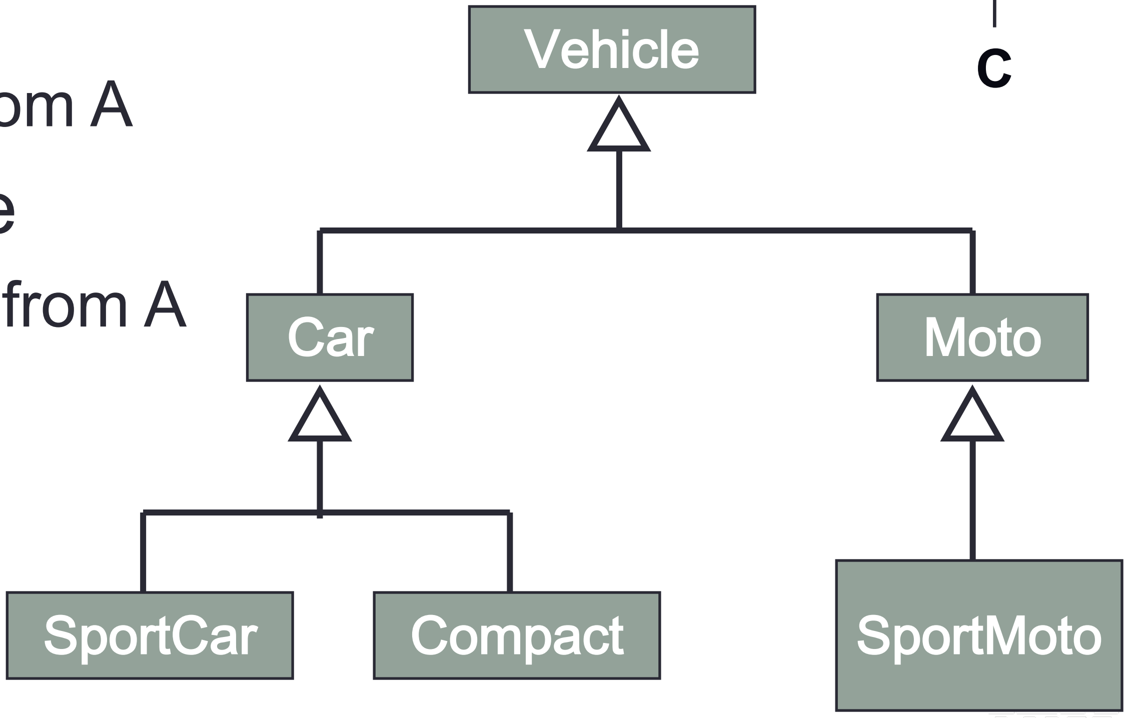
- Using "empty triangle" at parent class



3.4. Inheritance hierarchy

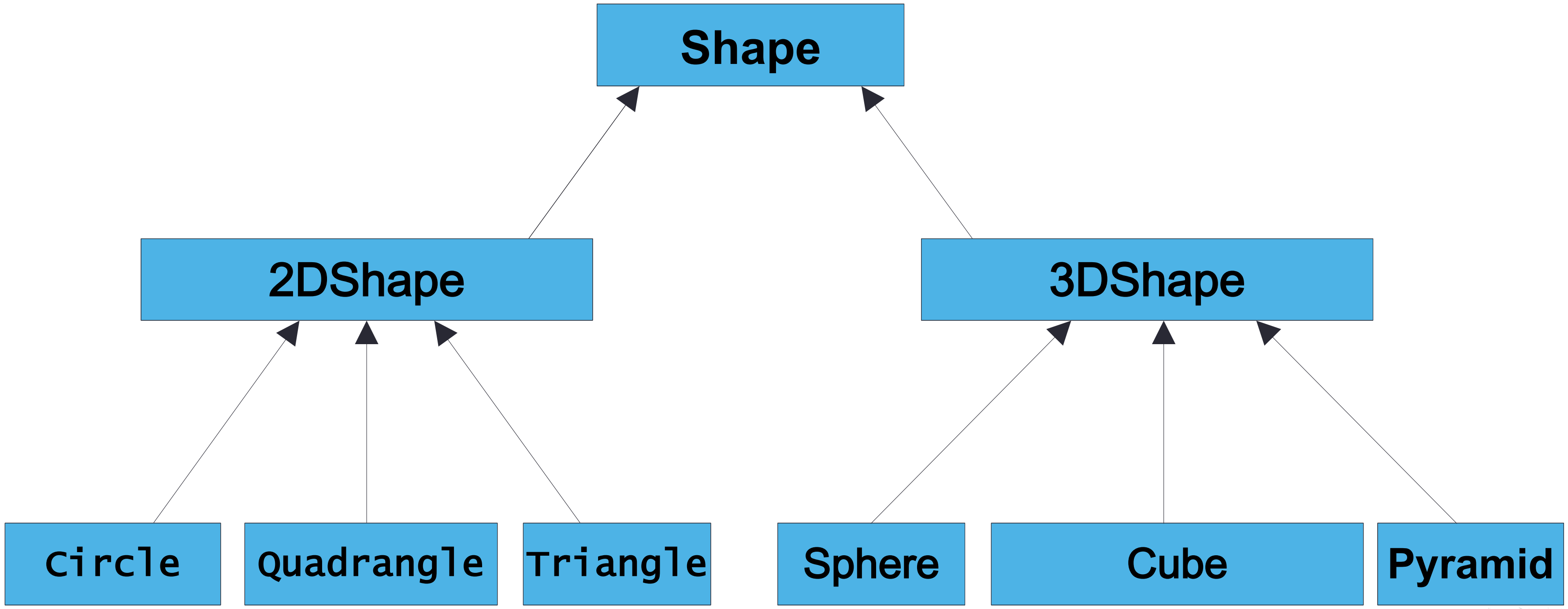
- Is hierarchy tree struture, representing inheritance relation between classes.
- Direct inheritance
  - B directly inherits from A
- Indirect inheritance
  - C indirectly inherits from A





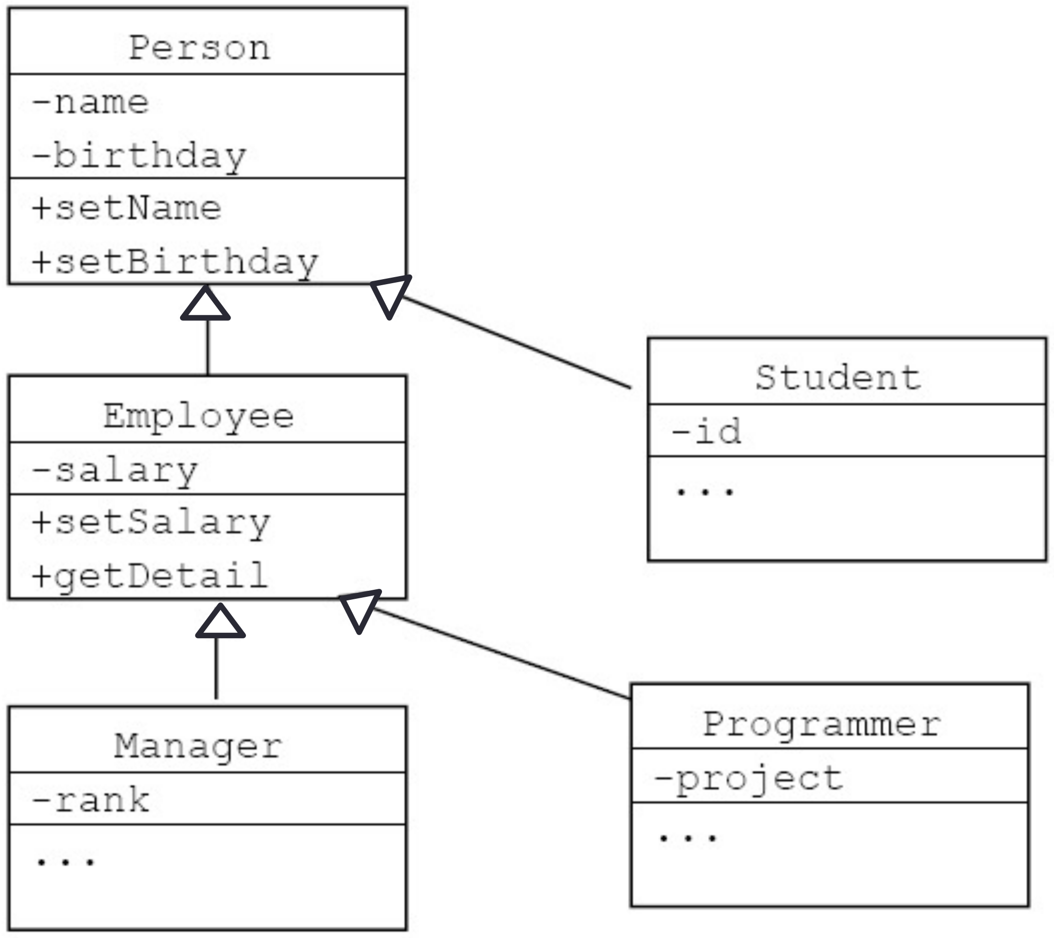
3.4. Inheritance hierarchy (2)

- Child classes having the same parent class are called **siblings**
- A child class inherits **all its ancestors**



3.4. Hierarchy tree (2)

All objects inherit from the basic class **Object**



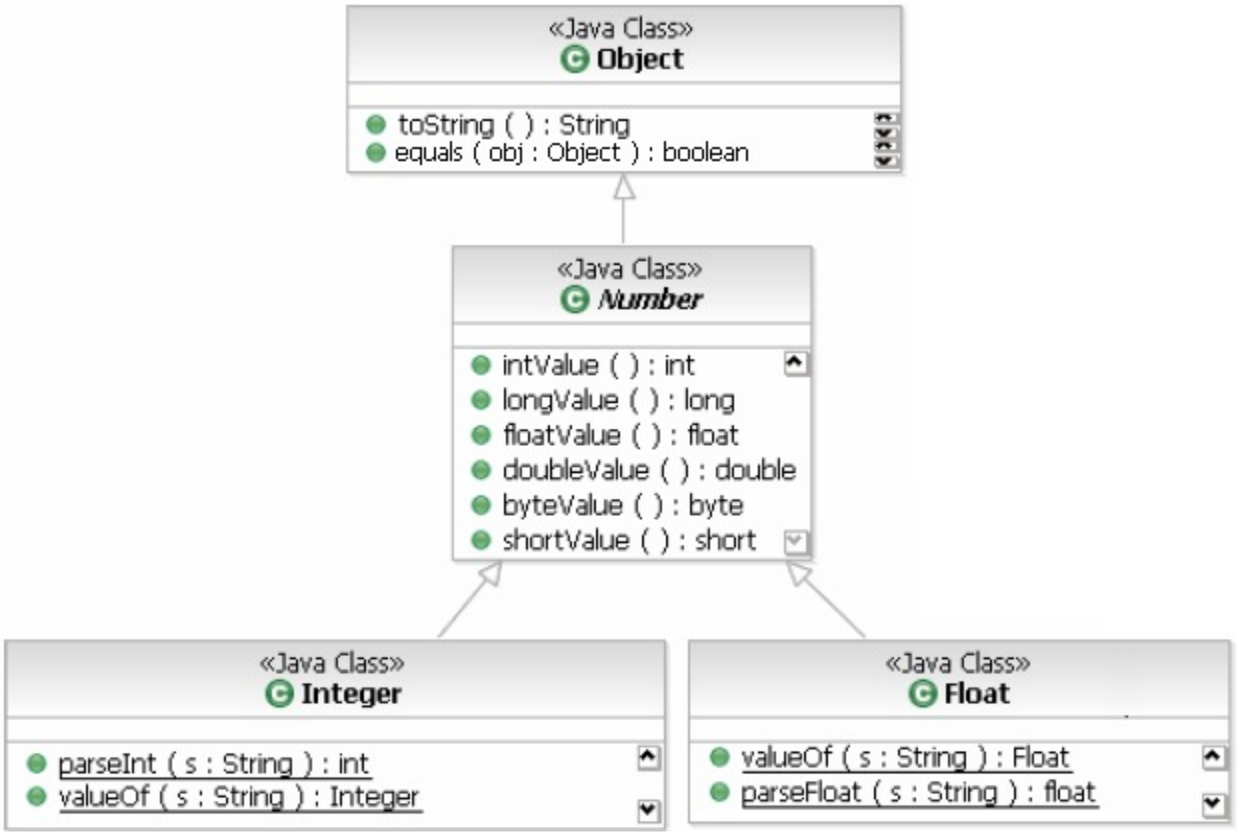
Class Object

- Class `Object` is defined in the standard package `java.lang`
- If a class is not defined as a child of another class, it is by default a direct child of class Object.

  → Class Object is the root class on the top level in the hierarchy tree

Class Object (2)

- Contains some useful methods that are inherited by all other classes, for example: toString(), equals()...



3.5. Inheritance rules

- Access attribute: `protected` (access modifier)
- Protected members in a parent class is accessed by:
  - Members of parent classes
  - Members of children classes
  - Members of classes in the same package as the parent class
- What does a child class inherit?
  - Inherit all the attributes/methods that are declared as public and protected in the parent class.
  - Does not inherit private attributes/methods.

3.5. Inheritance rules (2)

Visibility of members in parent class	`public`	**None (default)**	`protected`	`private`
Classes in the same package				
Child classes – same package				
Child classes – different package				
Different package, non-inher				

3.5. Inheritance rules (2)

	`public`	None	`protected`	`private`
Same package	Yes	Yes	Yes	No
Child classes – same package	Yes	Yes	Yes	No
Child classes – different package	Yes	No	Yes	No
Different package, non-inher	Yes	No	No	No

3.5. Inheritance rules (3)

- Methods that can not be inherited:
  - Construction and destruction methods
    - Methods that initialize and delete objects
    - These methods are only defined to work in a specific class
  - Assignment operation =
    - Performs the same task as construction method

3.6. Inheritance syntax in Java

- Inheritance syntax in Java:
  - `<SubClass> extends <SuperClass>`
- Example:

```
class Square extends Quandrangle {
  ...
}
class Bird extends Animal {
   ...
}
```

Example 1

```
public class Quadrangle {
   protected Point corners = new Point[4];
   public Quadrangle(){ ... }
   public void print(){...}
   …
}

public class Square extends Quadrangle {
  public Square(){
    corners[0]=new Point(0,0); corners[1]=new Point(0,1);
    corners[2]=new Point(1,0); corners[3]=new Point(1,1);
  }
}
public class Test{
  public static void main(String args[]){
        Square sq = new Square();
        sq.print();
  }
}
```

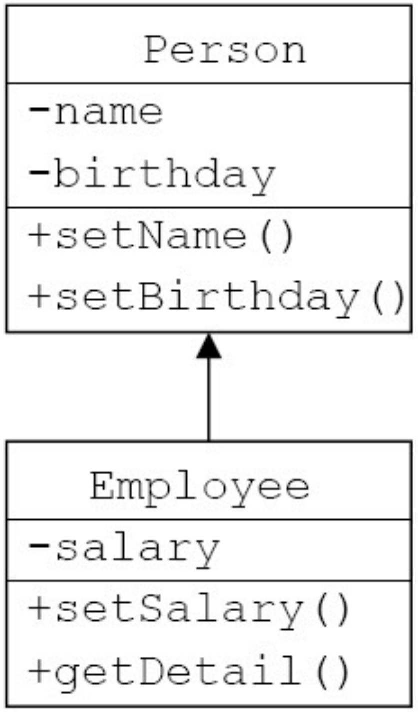
Using protected attributes of the parent class in the child class

Calling public method of parent class

Example 2

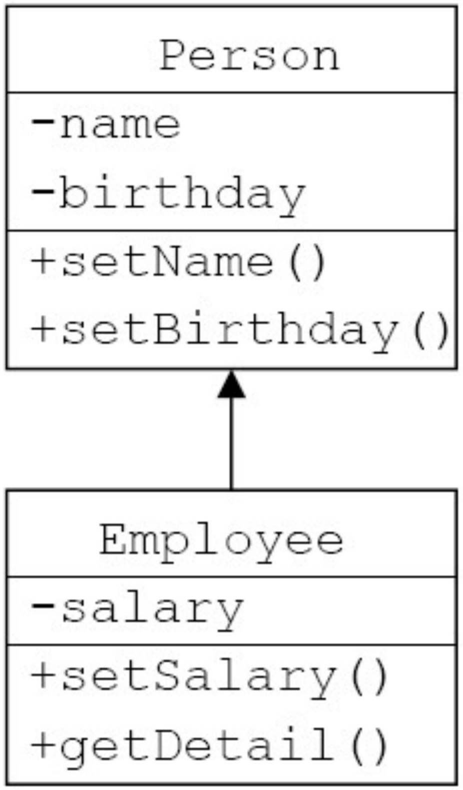
protected

```
class Person {
 private String name;
 private Date bithday;
 public String getName() {return name;}
 ...
}
class Employee extends Person {
 private double salary;
 public boolean setSalary(double sal){
  salary = sal;
  return true;
 }
  public String getDetail(){
  String s = name+", "+birthday+", "+salary;//Error
 }
}
```



Example 2 (cont.)

```
public class Test{
 public static void main(String args[]){
     Employee e = new Employee();
     e.setName("John");
     e.setSalary(3.0);
 }
}
```



Example 3 – Same package

```
public class Person {
 Date birthday;
 String name;
  ...
}
public class Employee extends Person {
  ...
 public String getDetail() {
     String s;
     String s = name + "," + birthday;
     s += ",“ + salary;
     return s;
 }
}
```

Example 3 – Different package

```
package abc;
public class Person {
 protected Date birthday;
 protected String name;
  ...
}

import abc.Person;
public class Employee extends Person {
  ...
 public String getDetail() {
      String s;
      s = name + "," + birthday + "," + salary;
      return s;
 }
}
```

Construction and destruction of objects in inheritance

- Object construction:
  - A parent class is initialized before its child classes.
  - Construction methods of a child class always call construction methods of its parent class at the very first command
    - Implicit call: whe the parent class has a **default constructor**
    - Explicit call (explicit)
- Object destruction:
  - Contrary to object initialization

3.4.1. Implicit call of constructor of parent class

```
public class Quadrangle {
  public Quadrangle(){
    System.out.println
    ("Parent Quadrangle()");
  }
  //. . .
}
public class Square
      extends Quadrangle {
  public Square(){
    //Implicit call "Quadrangle();"

     System.out.println
     ("Child Square()");
  }
}
```

```
public class Test {
  public static void
  main(String arg[])
  {
   HinhVuong hv =
      new HinhVuong();
   }
}
```

```
Parent Quadrangle()
Child Square()
```

Example

```
public class Quadrangle {
 protected Point[] corners=new Point[4];
  public Quadrangle(Point p1,Point p2,
                   Point p3,Point p4){
     corners[0] = p1; corners[1] = p2;
     corners[2] = p3; corners[3] = p4;
  }
}
public class Square extends
 Quadrangle {
  public Square(){
     System.out.println
        ("Child Square()");
  }
}
```

```
public class Test {
   public static void
   main(String arg[])
   {
     Square sq = new
            Square();
   }
}
```

Error

Cannot find symbol …

3.4.2. Implicit constructor call of parent class

- The first command in constructor of a child class can call the construtor of its parent class
  - `super(Danh_sach_tham_so);`
  - This is obliged if the parent class does not have any default constructor
    - Parent class already has a constructor with arguments
    - The constructor of child class must not have arguments.

```
public class Quadrangle {
   protected Point corners = new Point[4];
   public Quadrangle(){ ... }
   public Quadrangle(Point d1,Point d2,Point d3, Point d4)
   { ...}
  public void print(){...}
}
public class Square extends Quadrangle {
 public Square(){ super(); }
 public Square(Point p1,Point p2,Point p3,Point p4){
   super(d1, d2, d3, d4);
 }
}
public class Test{
 public static void main(String args[]){
      Square sq = new Square();
      sq.print();
 }
}
```

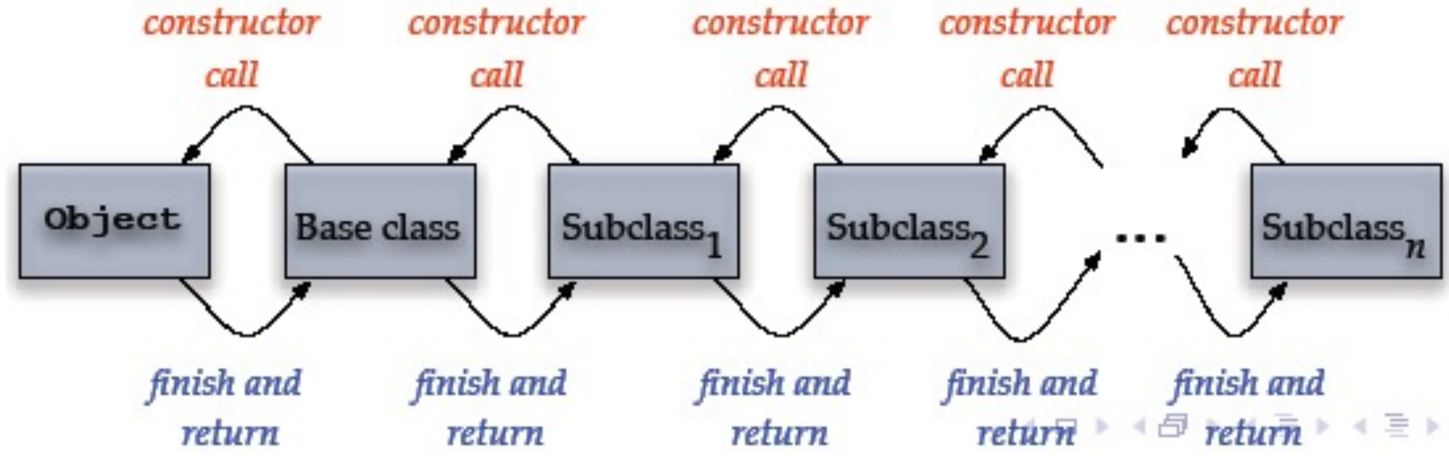
Example 1.1

Explicit constructor call of parent class
Constructor of child class has no arguments

```
public class Quadrangle {
 protected Point[] corners=new Point[4];
 public Quadrangle(Point p1,Point p2,
            Point p3,Point p4){
   System.out.println(”Parent Quadrangle()");
   corners[0] = p1; corners[1] = p2;
   corners[2] = p3; corners[3] = p4;
 }
}
public class Square extends Quadrangle {
  public Square(){
    super(new Point(0,0),new Point(0,1),new Point(1,1),
          new Point(1,0));
    System.out.println(”Child Square()");
  }
}
```

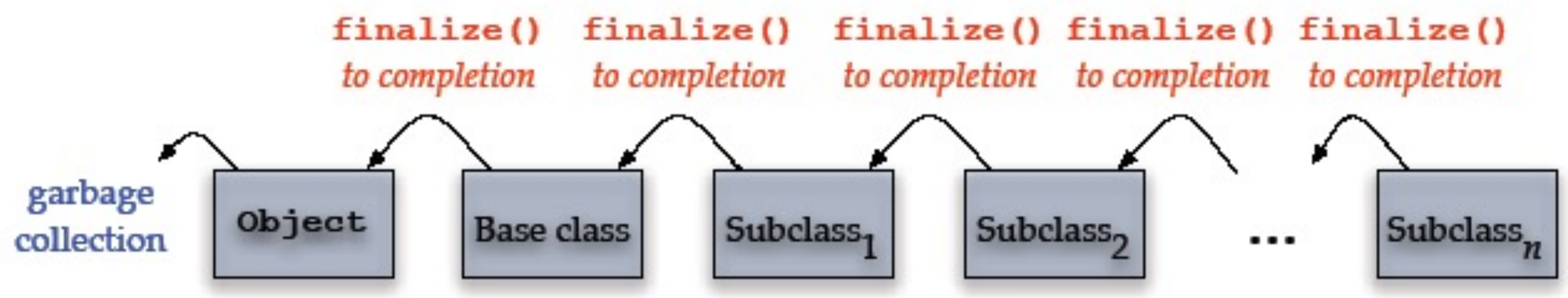
Implicit call of constructor

- When initializing an object, a serie of constructors will be called explicitly (via super() method call or implicitly)
- Constructor call of the most basic class in the hierarchy tree will be done last, but will finish first. The constructor of the derived class will finish at the last.



Implicit call of finalize()

- When an object is destroyed (by GC), a serie of finalize() methods will be called automatically.
- The order is inverse compared to the calls of constructors
  - Method finalize() of derived class is called first, then the ones of its parent class


finalize() to completion
finalize() to completion
finalize() to completion
finalize() to completion
finalize() to completion
garbage collection
Object
Base class
Subclass$_1$
Subclass$_2$
...
Subclass$_n$