Software Engineering Department
School of Information & Communication Technology
Hanoi University of Science and Technology

Object Oriented Programming

Chapter 05. Aggregation and Inheritance

Cao Tuấn Dũng
dungct@soict.hust.edu.vn

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 are aggregation or re-usability of objects of existing classes → Aggregation
  - Creating new classes based on the extension of existing classes → Inheritance

1. Re-usability (2)

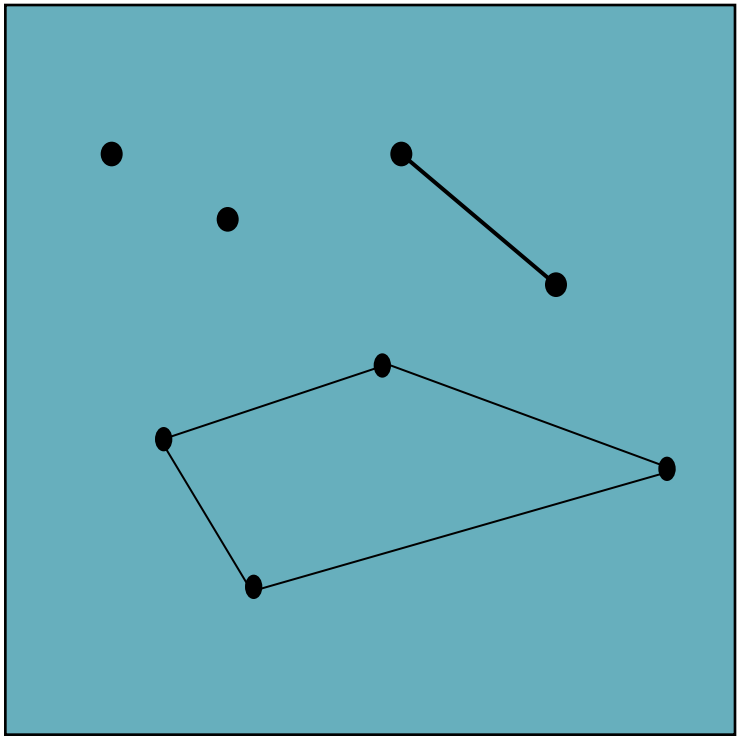
- Advantages
  - Reducing man-power, cost.
  - Improving software quality
  - Improving modeling capacity of the real world
  - Improving maintainability

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



2.1. Nature of aggregation

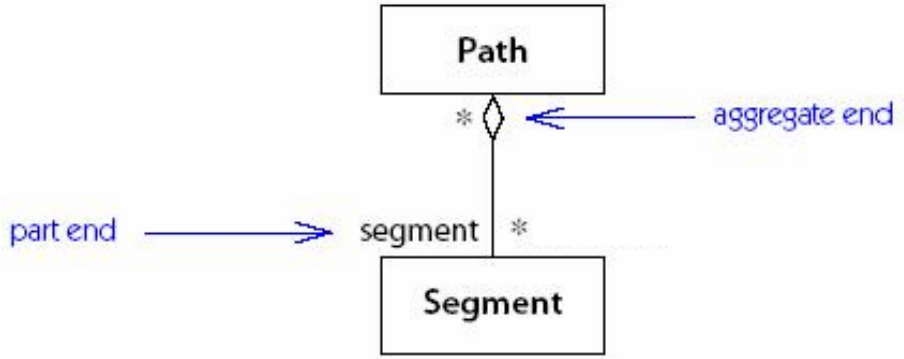
- Aggregate
  - Members of a new class are objects of existing classes.
  - Aggregation re-uses via *objects*
- New class
  - Aggregate/Whole class,
- Existing class
  - Member class (Part).

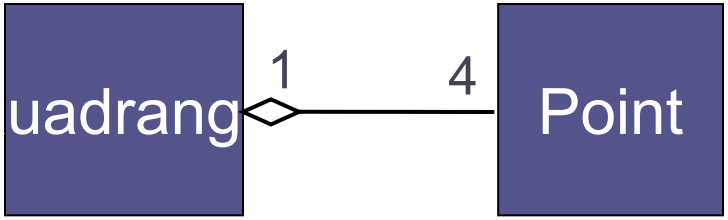
2.1. Nature of aggregation (2)

- 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

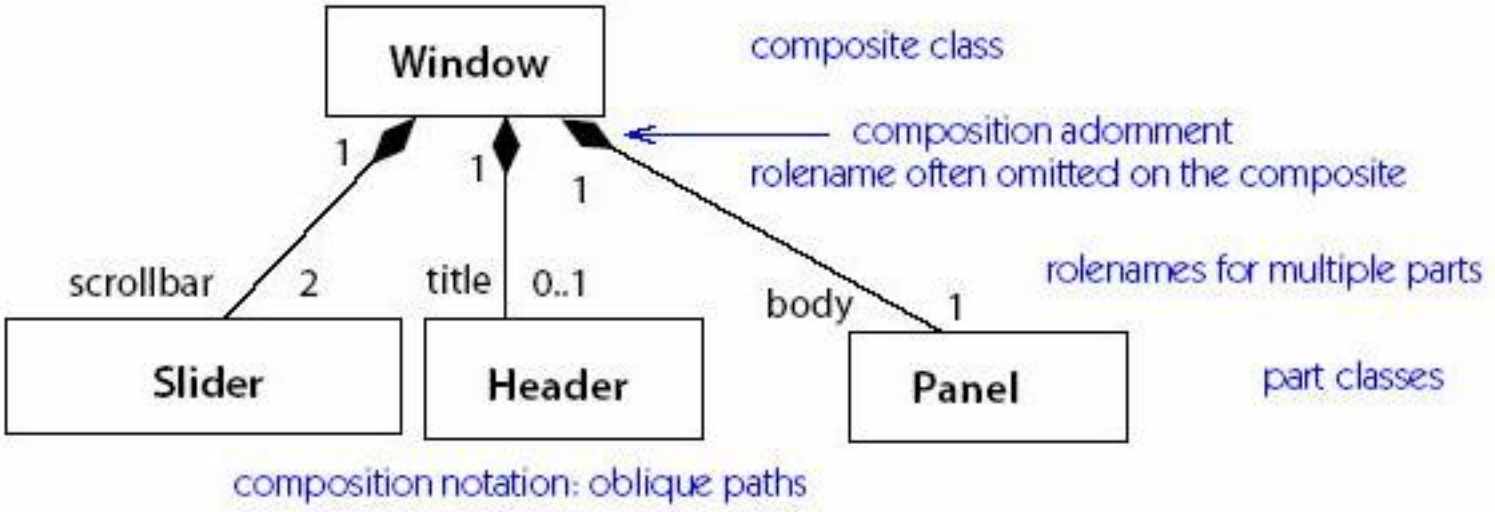
2.2. Representing aggregation in UML

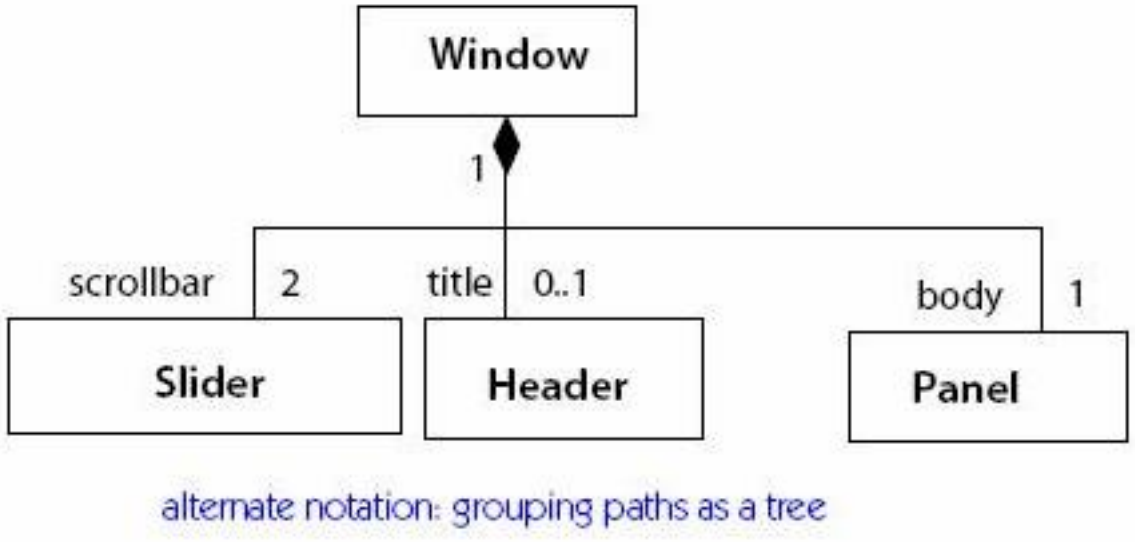
- Using "diamond" at the head of whole class
- Using multiplicity at two heads:
  - A positive integer: 1, 2,...
  - A range (0..1, 2..4)
  - *: Any number
  - None: By default is 1





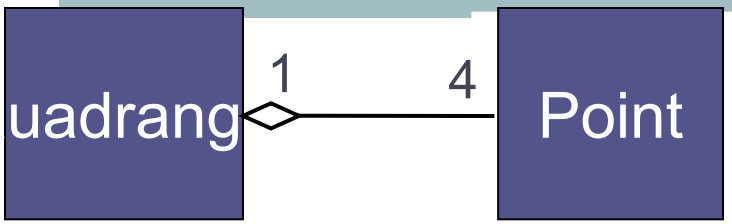
Example


Window
composite class
composition adornment
rolename often omitted on the composite
1
1
1
scrollbar
2
title
0..1
body
1
rolenames for multiple parts
Slider
Header
Panel
part classes
composition notation: oblique paths


Window
1
scrollbar
2
title
0..1
body
1
Slider
Header
Panel
alternate notation: grouping paths as a tree

2.3. Illustration in Java

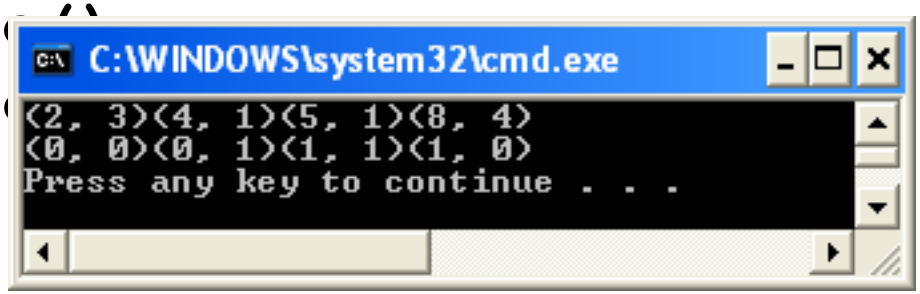
```
class Point {
  private int x, y;
  public Point(){}
  public Point(int x, int y) {
     this.x = x; this.y = y;
  }
  public void setX(int x){ this.x = x; }
  public int getX() { return x; }
  public void hienThiPoint(){
      System.out.print("(" + x + ", "
                              + y + ")");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




```
class Quadrangle {
  private Point d1, d2;
  private Point d3, d4;
  public Quadrangle(Point p1, Point p2,
                    Point p3, Point p4){
    d1 = p1; d2 = p2; d3 = p3; d4 = p4;
  }
  public Quadrangle(){
    d1 = new Point();      d2 = new Point(0,1);
    d3 = new Point (1,1); d4 = new Point (1,0);
  }
  public void printQuadrangle(){
    d1.printPoint();  d2.printPoint();
    d3.printPoint();  d4.printPoint();
    System.out.println();
  }
}
```

```
public class Test {
   public static void main(String arg[])
   {
      Point d1 = new Point(2,3);
      Point d2 = new Point(4,1);
      Point d3 = new Point (5,1);
      Point d4 = new Point (8,4);

      Quadrangle tg1 = new Quadrangle(d1, d2,
  d3, d4);
      Quadrangle tg2 = new Quadrangle();
      tg1.printQuadrangl
      tg2.printQuadrangl
   }
}
```



Example 2

```
class Person {
  private String name;
  private Date bithday;
  public String getName() { return name; }
  ...
}

class Employee {
  private Person me;
  private double salary;
  public String getName() {
     return me.getName();
  }
  ...
}
```

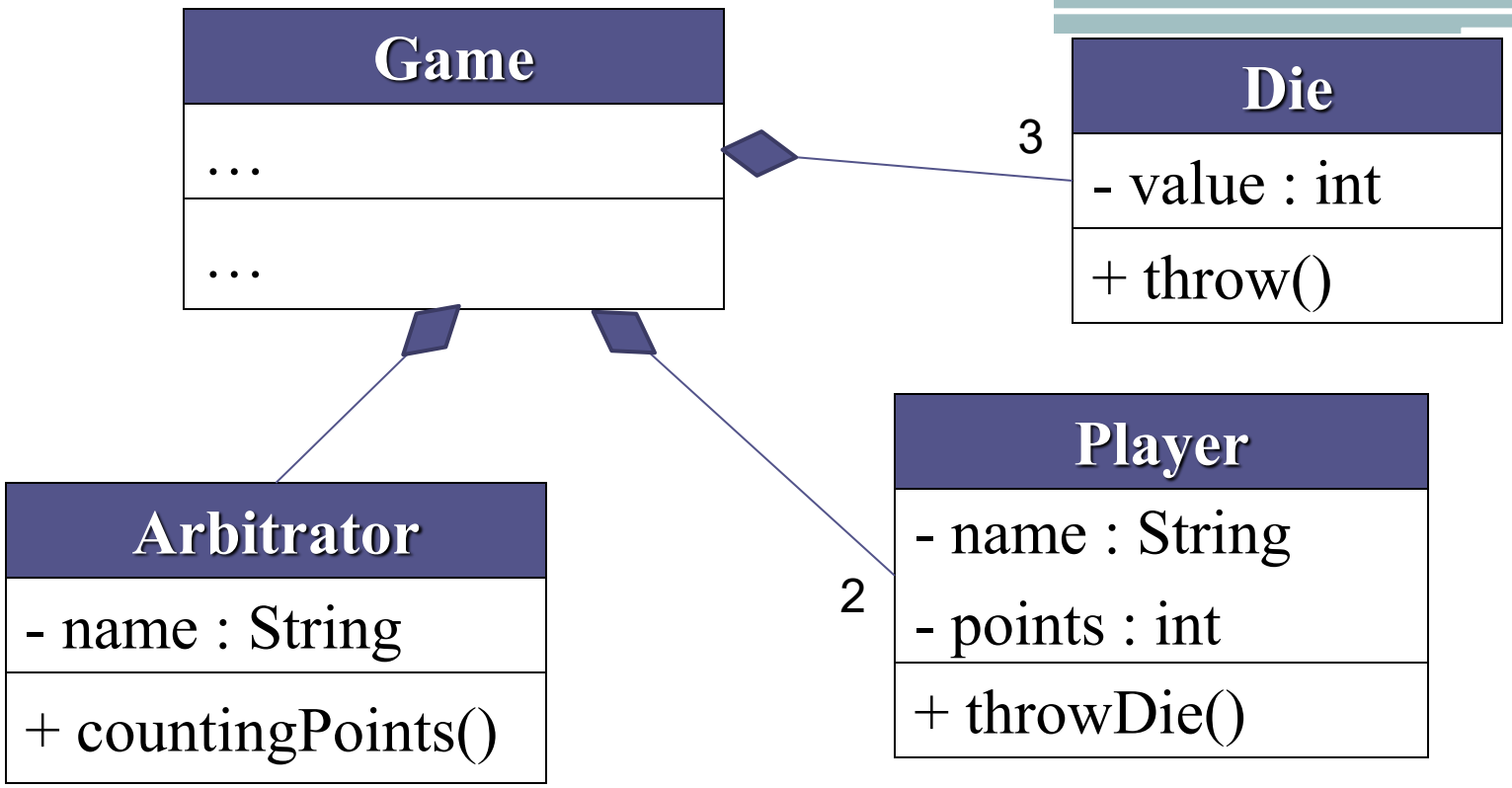
Example 2

```
class Manager {
  private Employee me;
  private Employee assistant;
  public setAssistant(Employee e) {...}
  ...
}
...
Manager junior = new Manager();
Manager senior = new Manager();
senior.setAssistant(junior); //error
```

An 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

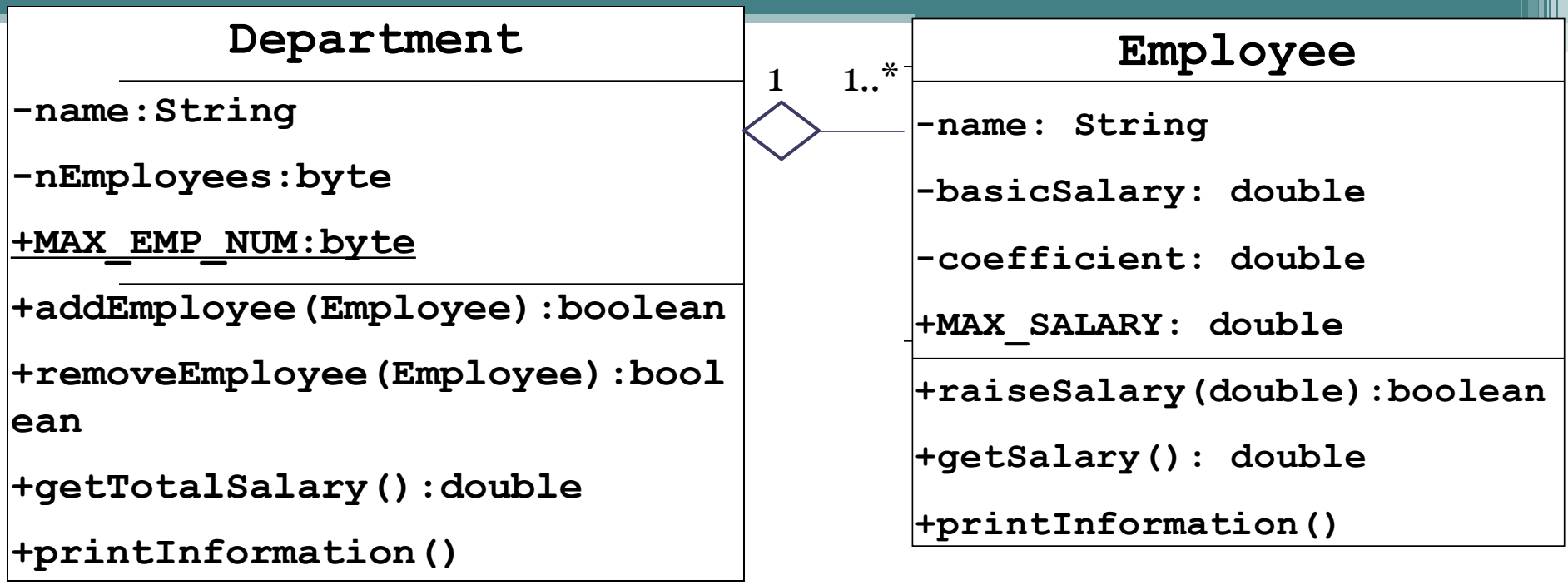



```
class Game
{
        Die die1, die2, die3;
        Player player1, player2;
        Arbitrator arbitrator1;
        ...
}
```

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

→ Construction methods of member classes must be called first



- Write class **`Department`** described in above UML diagram with parameterized constructor, knowing that:
  - Value of **`MAX_EMP_NUM is 100`**
  - Employee adding and removing is perfomed using stack
  - **`getTotalSalary()`** calculate total salary of all employees of this department.
  - **`printInformation()`** display information about department and all employees.

```
public class Department {
  private String name; private byte nEmployees;
  public static final MAX_EMP_NUM = 100;
  private Employee[] employeeList;
  public boolean addEmployee(Employee e){
       if (nEmployees < MAX_EMP_NUM) {
             employeeList[nEmployees] = e; nEmployees++;
            return true;
       } else return false;
  }
  public boolean removeEmployee(Employee e){
      if (nEmployees > 0) {
            Employee tmp = employeeList[nEmployees -1];
            employeeList[soNhanVien-1] = null;
            nEmployees --;
            return true;  ///employeeList[nEmployees];
      } else return false;
  }
```

```
// (cont.)
public Department(String name){
    employeeList = new Employee[MAX_EMP_NUM];
    this.name = name; nEmployees = 0;
}
public double getTotalSalary(){
    double total = 0.0;
    for (int i=0;i<nEmployees;i++)
          total += employeeList[i].getSalary();
    return total;
}
public void printInformation(){
    System.out.println("Name of Dept: "+name);
    System.out.println("Number of employees: " +
nEmployees);
    System.out.println("Employees info:");
    for (int i=0;i<soNhanVien;i++)
          employeeList[i].printInformation();
}
```

Discussion

In above example

- What is Existing class? New class?
  - Existing class: Employee
  - New class: Department
- New class re-uses existing classes via?
  - Array of objects of class Employee
- What does new class re-use from existing classes?
  - getSalary () in method getTotalSalary()
  - printInformation() in method printInformation()

Outline

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

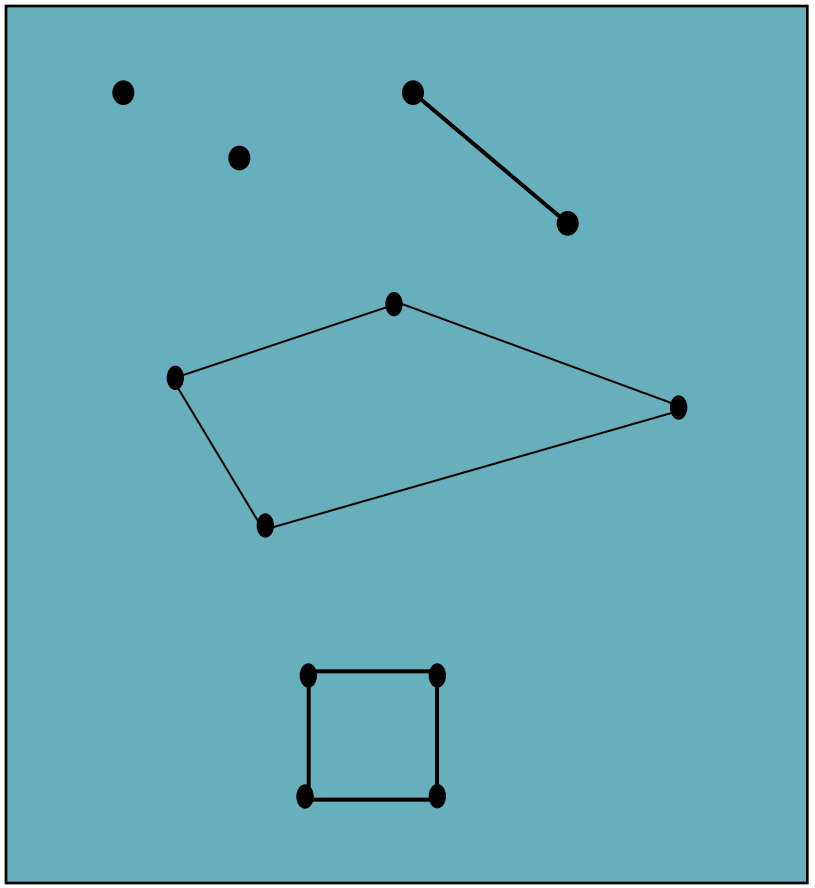
3. Overview of Inheritance

- Example:
  - Point
    - A quadrangle has 4 points

    →Aggregation

  →Quadrangle
    - Square

    → Inheritance



3.1.1. The Nature of Inheritance

- 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

Inheritance

- Principles to describe a class based on the extension/concretization of an existing class or a set of existing classes (in case of multi-inheritance)
- On modularity view: If B inherits A, all services of A will be available in B
- On "typing" view: If B inherits A, at anytime if a instance of A is required, the instance of B might be a response.

Inheritance

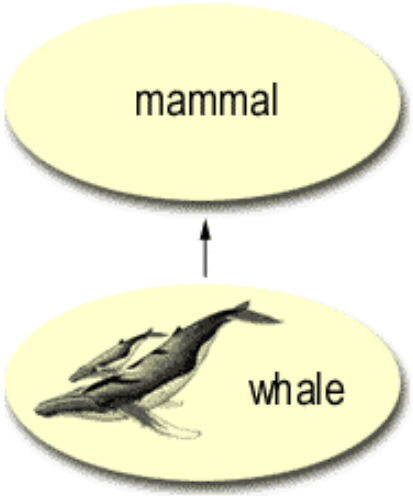
- *Inheritance specifies a* relationship between classes when a class shares It structure and/or behavior of a class or of other classes.
- 1 genealogy tree of the classes is created in which a *subclass* inherits from one or many *superclasses*
- Inheritance is also called *is-a* relation.

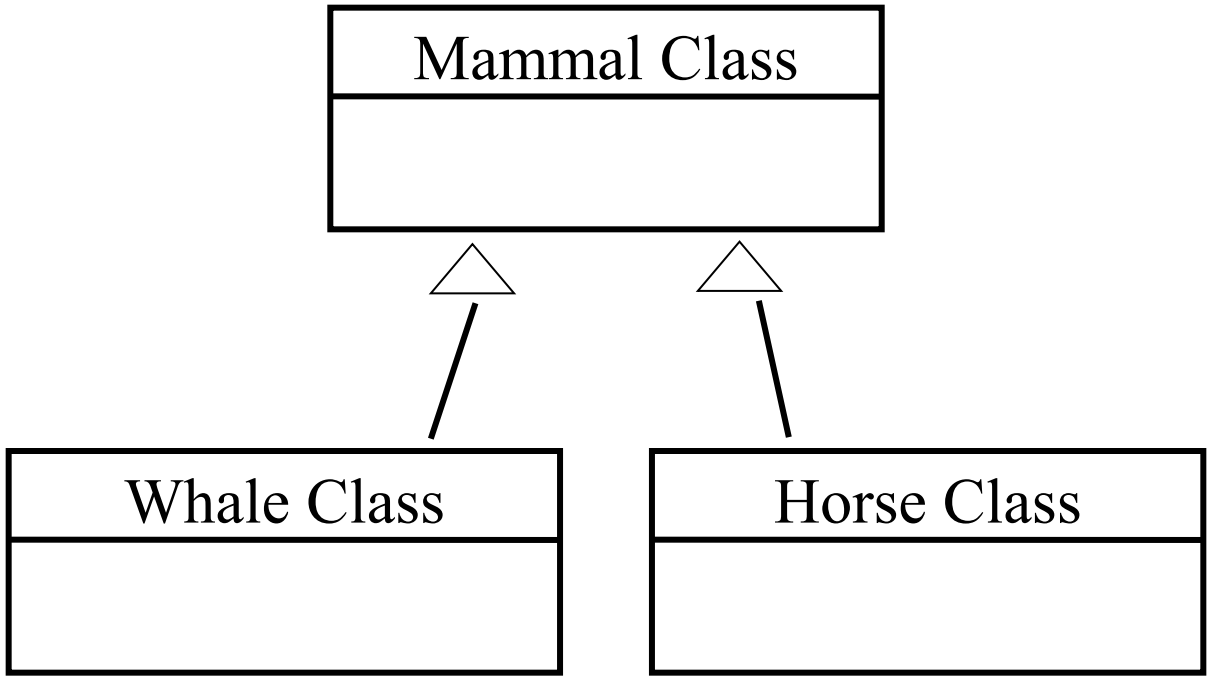
3.1.1. The Nature of Inheritance (2)

- Child class
  - Is is-a-kind-of) of parent class
  - Re-uses by inheriting data and behavior of parent classes
  - Concretizes in order to adapt to new usage targets
    - Extension: Add more new attributes/behavior
    - Redefinition (Method Overriding): Modify the behavior inheriting from parent class

Similarity

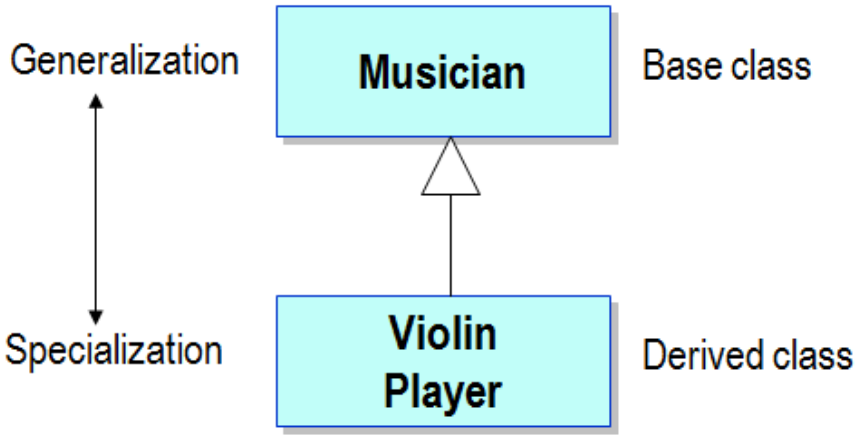
- Whale class inherits from mammal class.
- A whale *is-a* mammal)
- Whale class is *subclass*, mammal class is *superclass*





Inheritance

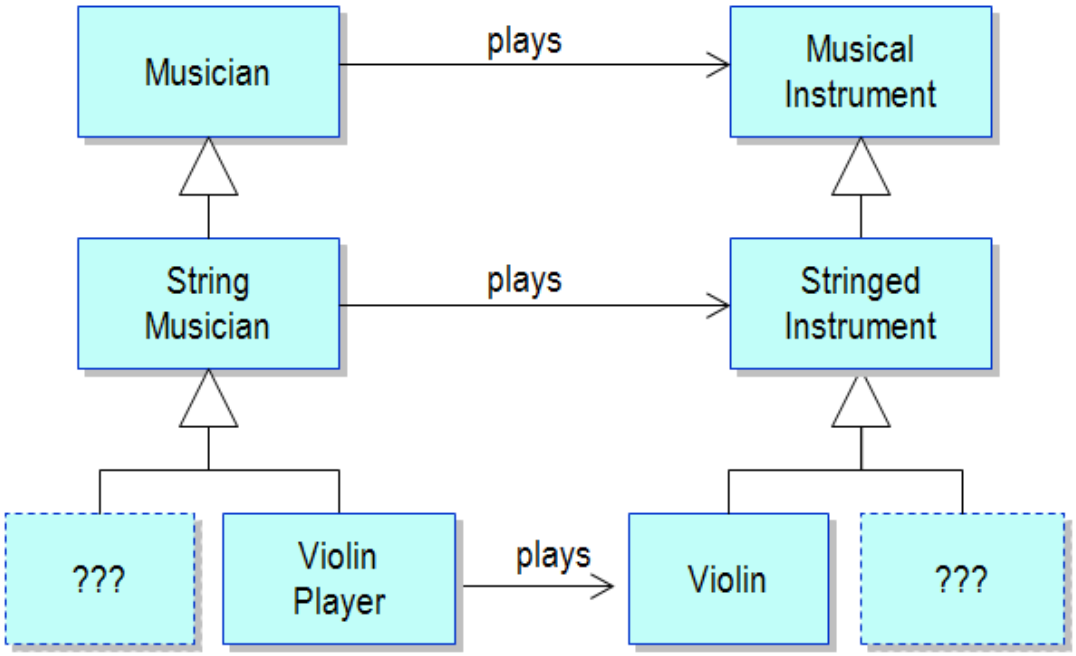
- Inheritance specifies an “is a kind of" relationship
  - Inheritance is a class relationship
  - New classes specialize existing classes



Is this a good example of inheritance ?

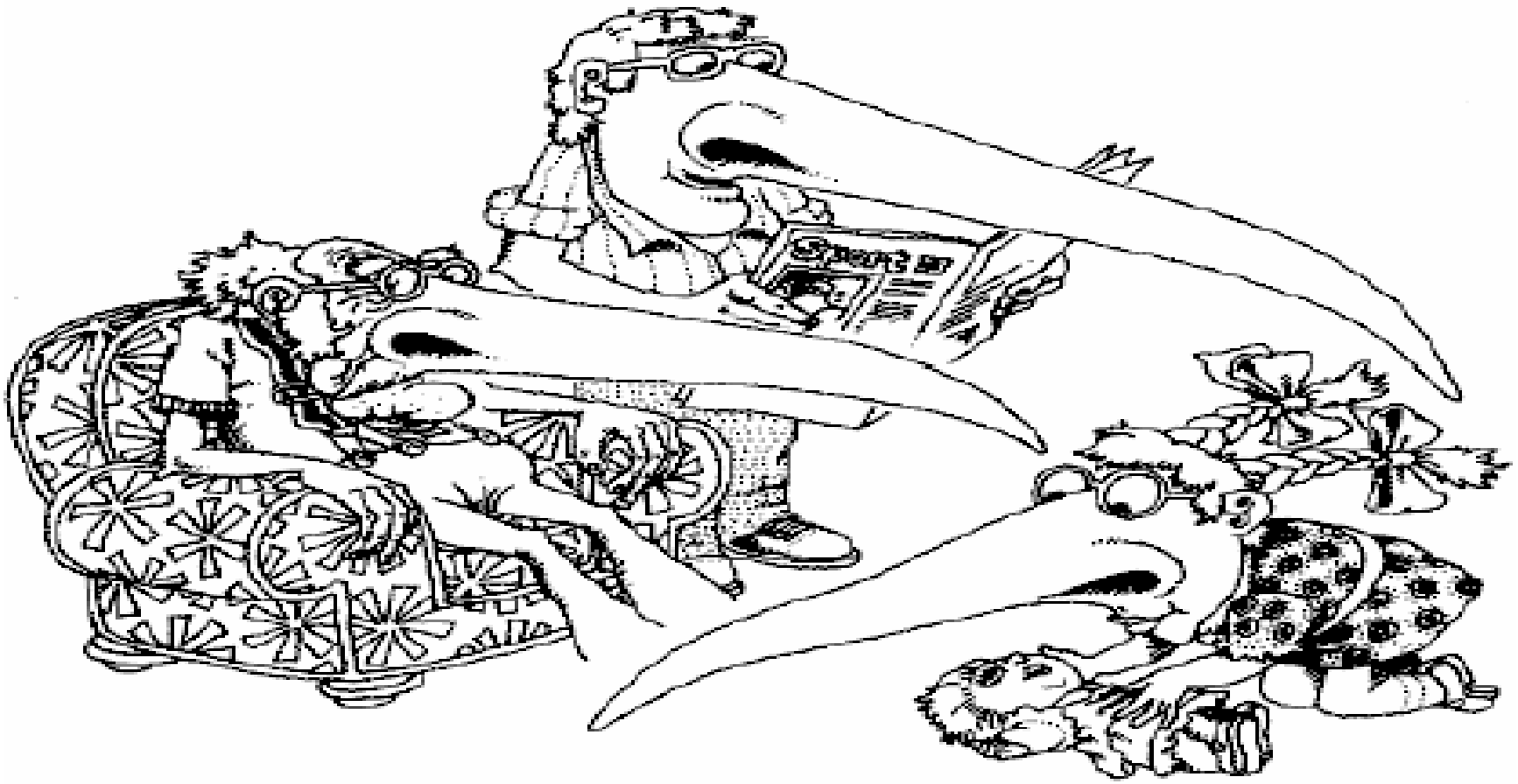
Class Hierarchies

- Classes related by inheritance form class hierarchies



Inheritance

- Both Whale and Horse has *is-a* relation with mammal class
- Both Whale 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



3.1.2. Aggregation and Inheritance

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

Distinguish aggregation and inheritance

Inheritance

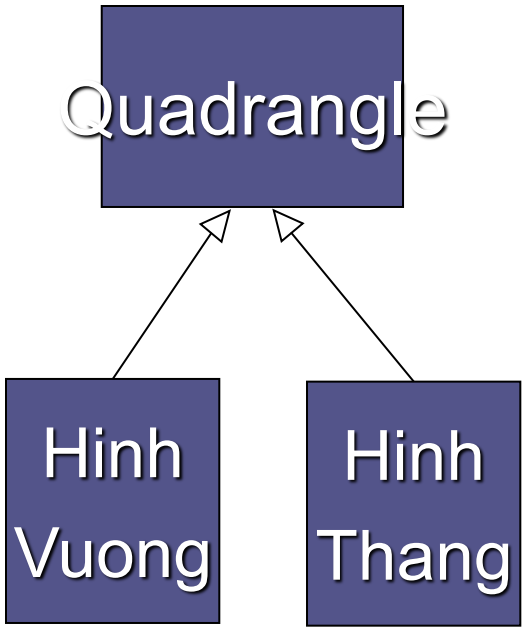
- Inheritance re-uses via class.
  - Creating new class by extending exisiting classes
- "is a kind of" relation
- Example: Car is a kind of transportation mean

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

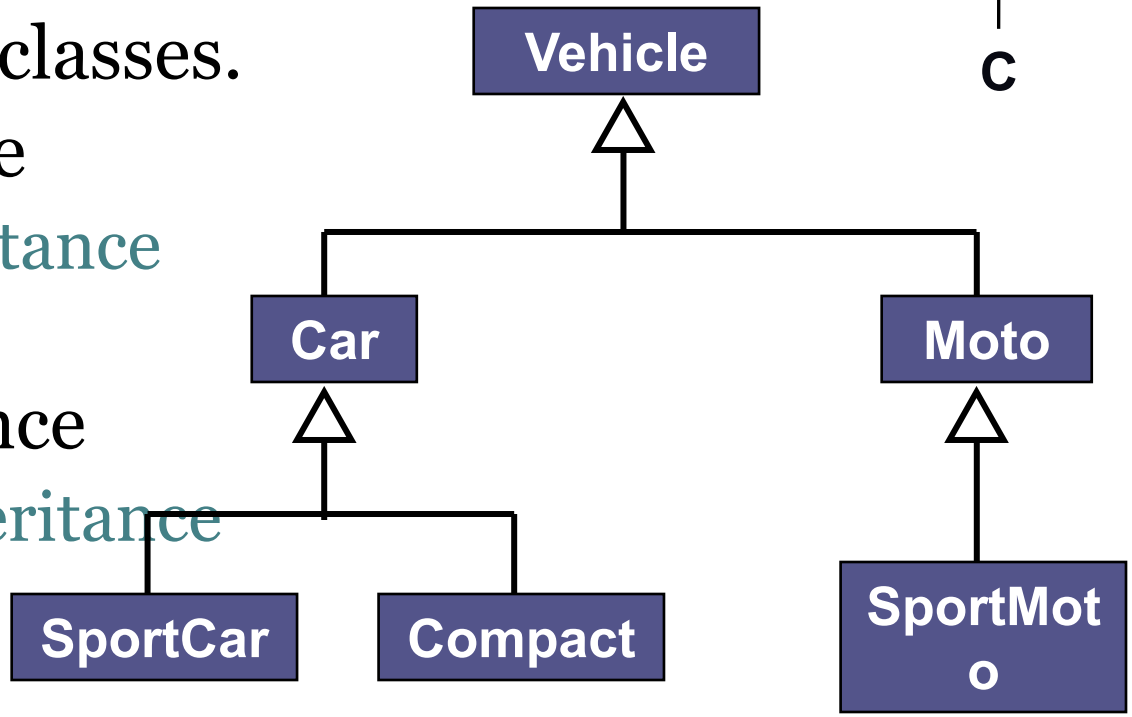
3.1.3. Representing Inheritance in UML

- Using “empty triangle” at parent class



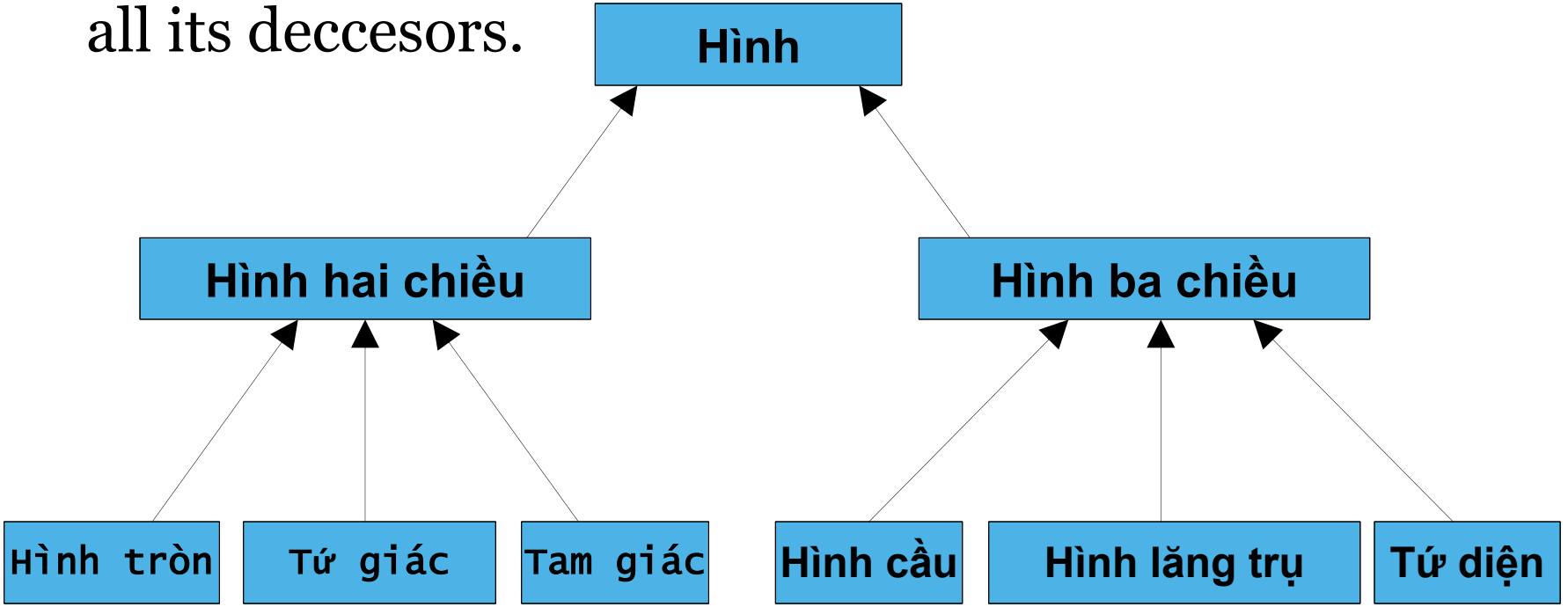
3.1.4. Inheritance hierarchy

- Is hierarchy tree struture, representing inheritance relation between classes.
- Direct inheritance
  - B is direct inheritance from A
- Indirect inheritance
  - C is indirect inheritance from A



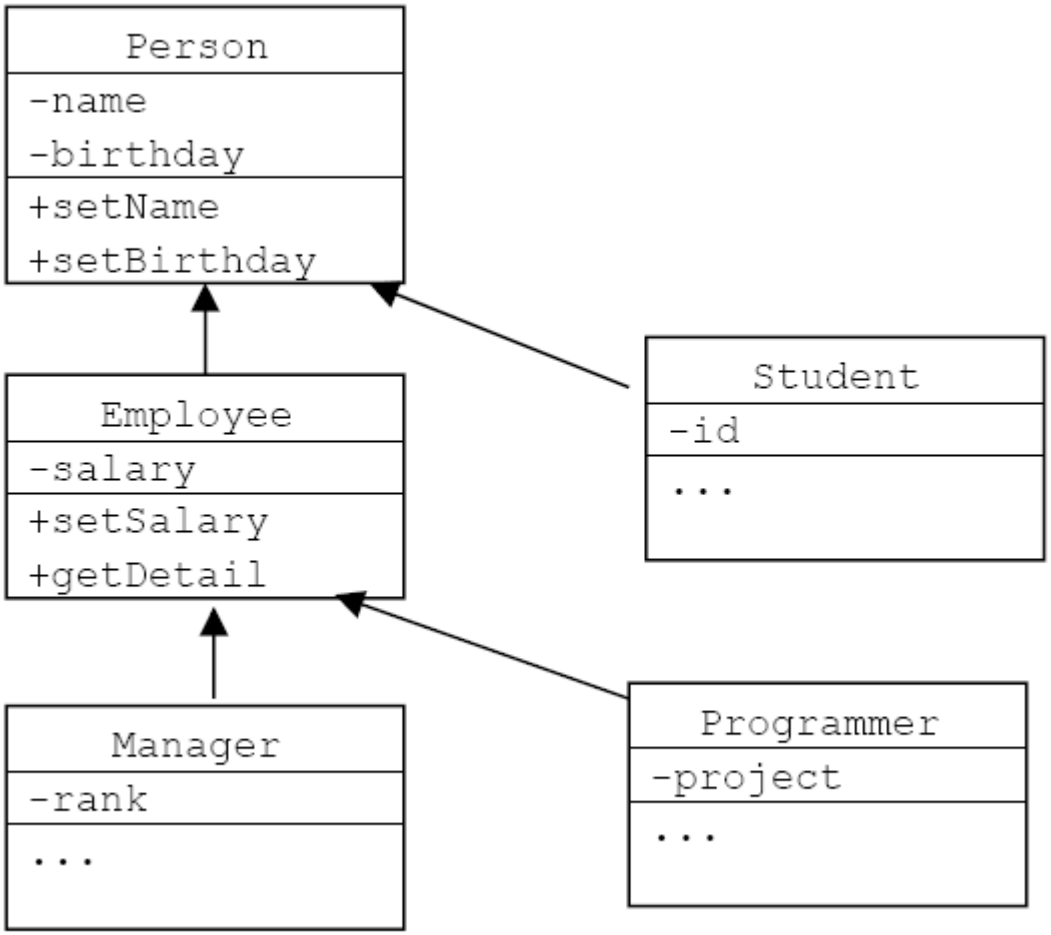
3.1.4. Inheritance hierarchy (2)

- Child classes having the same parent class are called siblings
- Classes that inherit will be inherited by the classes in lower level in the hierachy tree → A child class inherits all its deccesors.



3.1.4. Hierarchy tree (2)

All objects inherit from the basic class Object

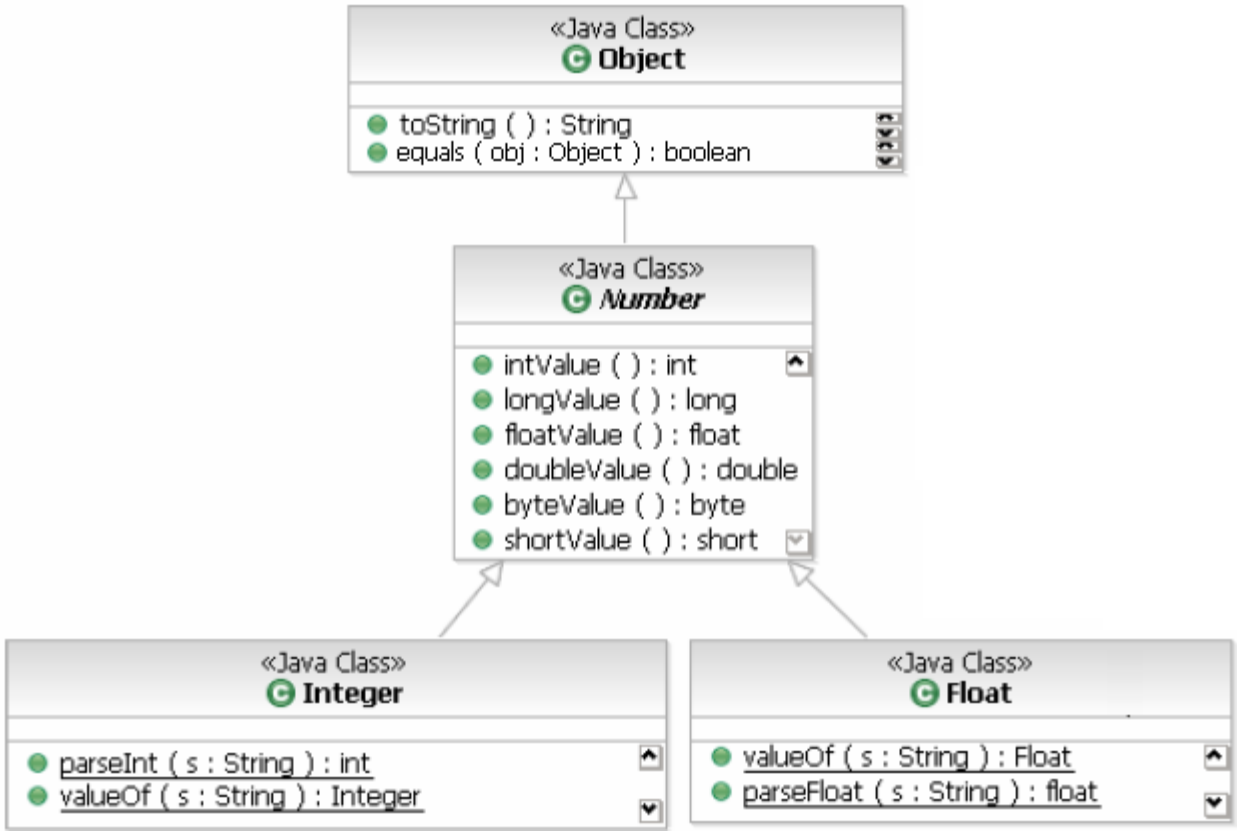


Class Object

- 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()...



3.2. Inheritance rules

- Access attribute: protected
- Protected member 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.

3.2. Inheritance rules (2)

	`public`	None	`protected`	`private`
Same class				
Child classes – same package				
Child classes – different package				
Different package, non-inher				

3.2. Inheritance rules (2)

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

3.2. 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

3.3. Inheritance syntax in Java

- Inheritance syntax in Java:
  - `<Child Class> extends <Parent Class>`
- Example:

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

Example 1.1

```
public class Quadrangle {
   protected Point d1, d2, d3, d4;
   public void setD1(Point _d1) {d1=_d1;}
   public Point getD1(){return d1;}
   public void printQuadrangle(){...}
   …
}

public class Square extends Quadrangle {
  public Square(){
      d1 = new Point(0,0); d2 = new Point(0,1);
      d3 = new Point(1,0); d4 = new Point(1,1);
  }
}
public class Test{
  public static void main(String args[]){
      Square hv = new Square();
      hv.printQuadrangle();
  }
}
```

Using protected attributes of the parent class in the child class

Calling public method of parent class

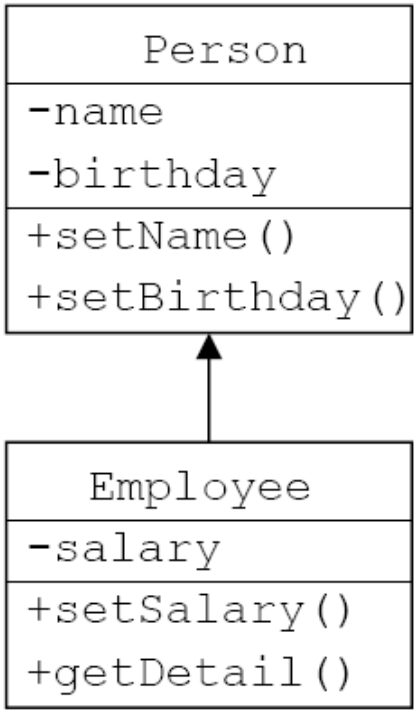
Example 1.2

```
public class Quadrangle {
  protected Point d1, d2, d3, d4;
  public void printQuadrangle(){...}
  public Quadrangle(){...}
  public Quadrangle(Point d1, Point d2,
                    Point d3, Point d4) { ...}
}
public class Square extends Quadrangle {
  public Square(){ super(); }
  public Square(Point d1, Point d2,
                      Point d3, Point d4){
      super(d1, d2, d3, d4);
  }
}
public class Test{
  public static void main(String args[]){
      Square hv = new Square();
      hv.printQuadrangle();
  }
}
```

Example 2

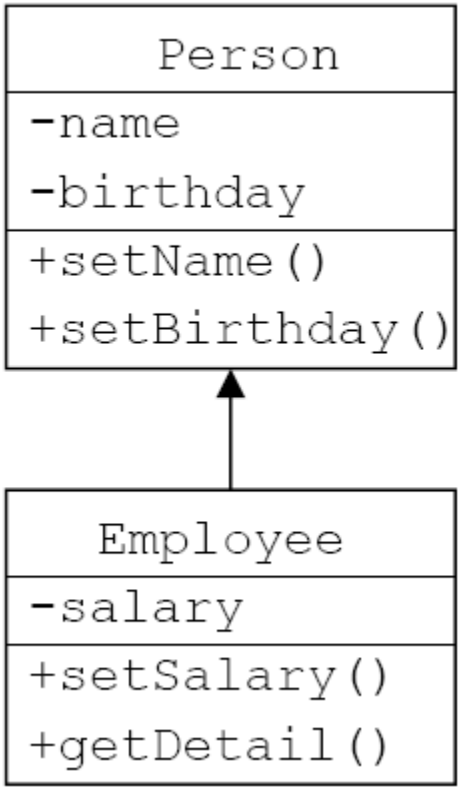
protected

```
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;//Loi
  }
}
```



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;
  }
}
```

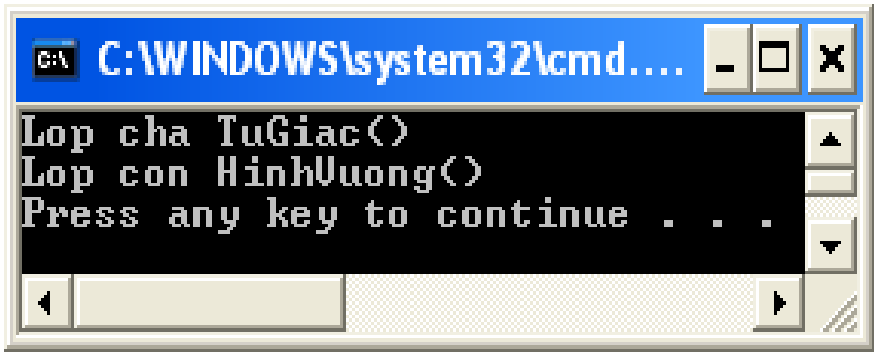
Construction and destruction of objects in inheritance

- 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 {
   protected Point d1, d2;
   protected Point d3, d4;
   public Quadrangle(){
    System.out.println
        ("Lop cha
  Quadrangle()");
   }
   //…
}
public class Square
      extends Quadrangle {
   public Square(){
    //Tu dong goi
  Quadrangle()
    System.out.println
    ("Lop con Square()");
   }
}
```

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



3.4.2. Implicit constructor call of parent class

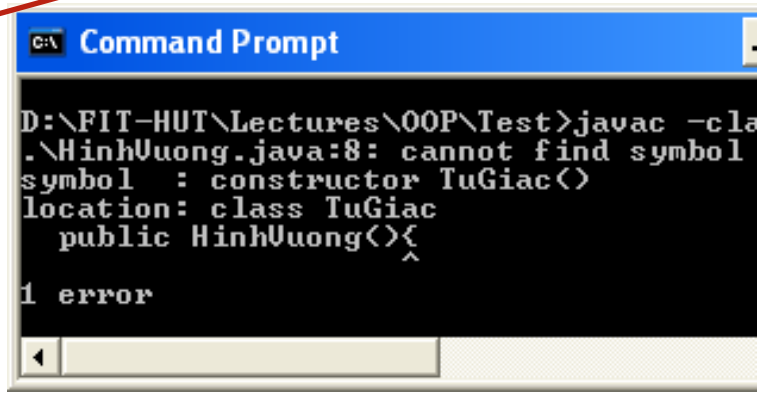
- The first command in constructor of a child class can call the construtor of its parent class
  - `super(arguments_list);`
  - This is obliged if the parent class does not have any default constructor
    - Parent class already has a constructor with arguments
    - The constructor of child class must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

Example

```
public class Quadrangle {
  protected Point d1, d2;
  protected Point d3, d4;
  public Quadrangle(Point d1,
  Point d2, Point d3, Point
  d4){
    System.out.println("Lop cha
     Quadrangle(d1, d2, d3,
  d4)");
    this.d1 = d1; this.d2 = d2;  }
    this.d3 = d3; this.d4 = d4;
   }
}
public class Square extends
  Quadrangle {
   public Square(){
     System.out.println
    ("Lop con Square()");
   }
```

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

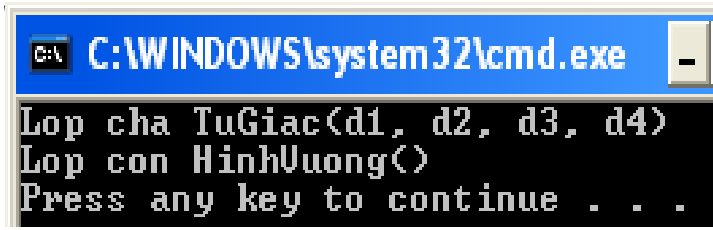
Lỗi



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

```
public class Quadrangle {
  protected Point d1,d2,d3,d4;
public Quadrangle(Point d1, Point d2,
          Point d3, Point d4){
    System.out.println("Lop cha
       Quadrangle(d1, d2, d3, d4)
     this.d1 = d1; this.d2 = d2;
     this.d3 = d3; this.d4 = d4;
  }
}
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("Lop con Square()");
  }
}
```

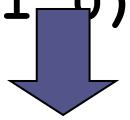
```
. . .
Square hv = new
  Square();
```



Explicit constructor call of parent class Constructor of child class has arguments

```
public class Quadrangle {
  protected Point d1,d2,d3,d4;
  public Quadrangle(Point d1,
  Point d2, Point d3, Point d4){
   System.out.println
    ("Lop cha Quadrangle(d1,d2,d3,d4)");
   this.d1 = d1; this.d2 = d2;
   this.d3 = d3; this.d4 = d4;
  }
}
public class Square extends Quadrangle {
 public Square(Point d1, Point d2,
   Point d3, Point d4){
   super(d1, d2, d3, d4);
    System.out.println("Lop con
          Square(d1,d2,d3,d4)");
  }
}
```

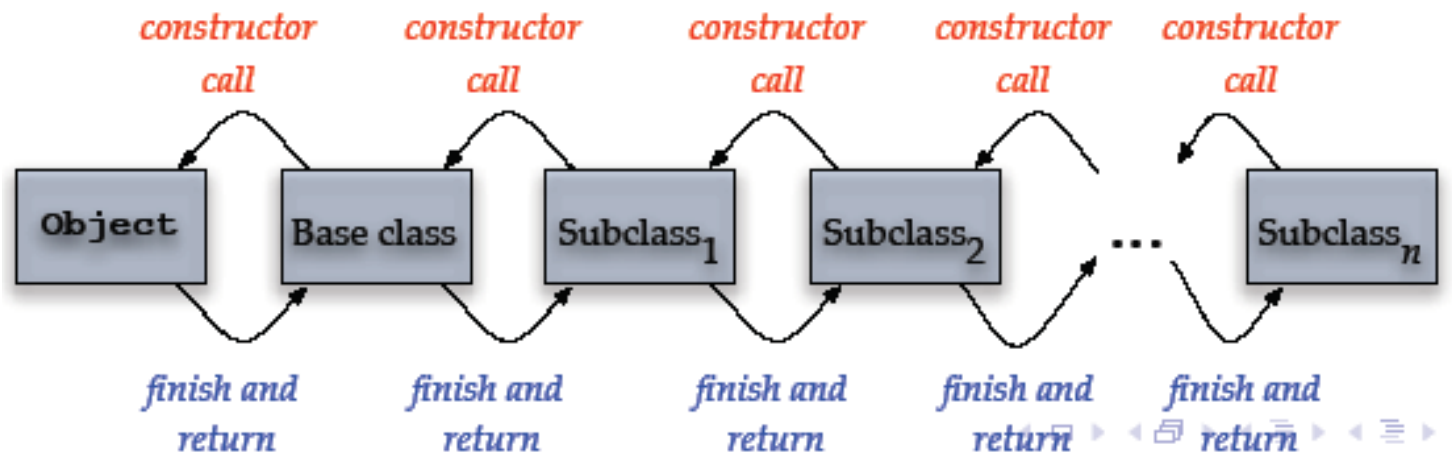
```
. . .
Square hv =
  new Square(
    new Point(0,0),
    new Point(0,1),
    new Point(1,1),
    new
Point(1,0));
```



```
Lop cha TuGiac(d1, d2, d3, d4)
Lop con HinhVuong(d1, d2, d3, d4)
```

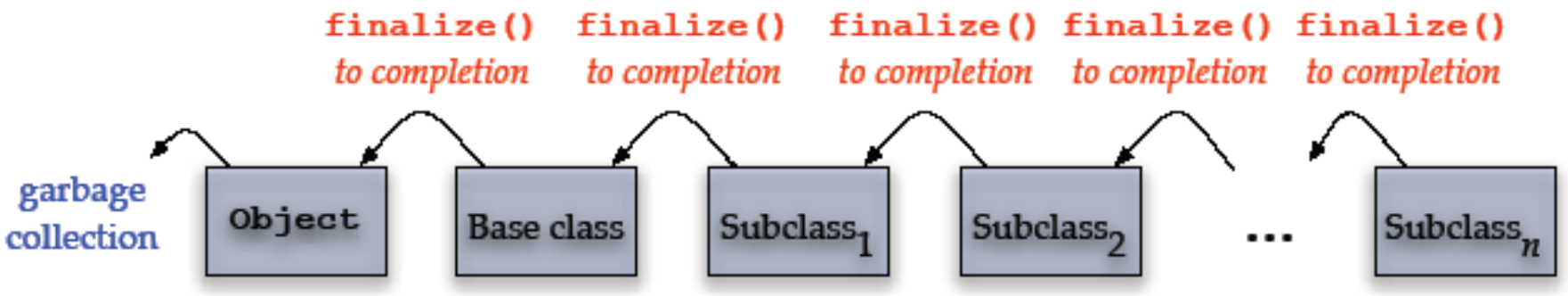
Implicit call of constructor

- 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



Inheritance characteristics in C++

Inheritance declaration

```
class DerivedClass : access-specifier BaseClass
{
  // Body of the derived class
};
```

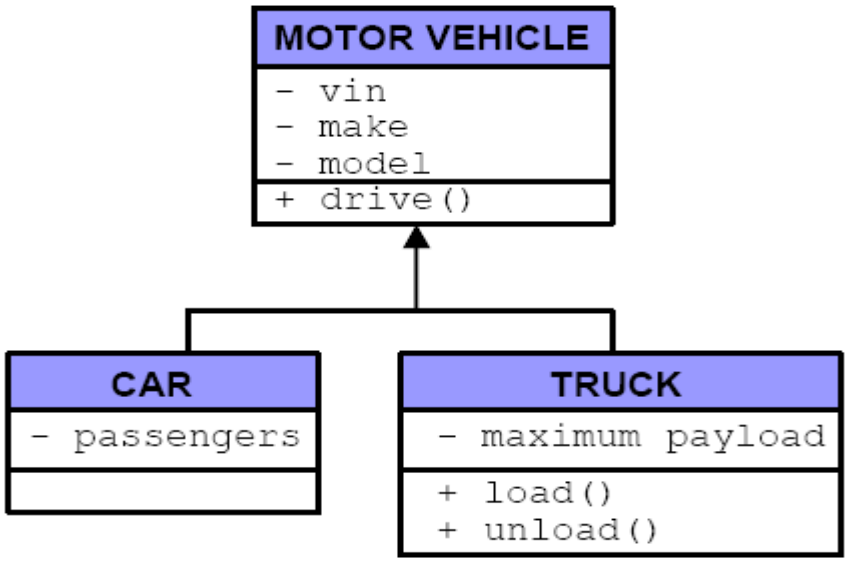
DerivedClass
BaseClass
Access-specifier: public, protected, private

Inheritance declaration

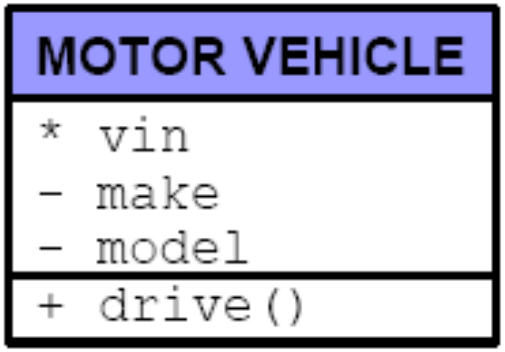
Suppose that there are two classes A and B; to build class C derived from A and B, we do as follows:

```
class  C : public A, public B
    {
    private:
          // khai báo các thuộc tính
    public:
          // các phương thức
};
```

Example: MotorVehicle



Example: MotorVehicle



- Starting by defining a basic class, MotorVehicle

```
class MotorVehicle {
public:
      MotorVehicle(int vin, string make, string model);
      ~MotorVehicle();
      void drive(int speed, int distance);
private:
      int vin;
      string make;
      string model;
};
```

Example: MotorVehicle

- Then, define constructor, destructor, and function drive() (here, we temporarily define drive())

```
MotorVehicle::MotorVehicle(int vin, string make, string model)
{
      this->vin = vin;
      this->make = make;
      this->model = model;
}
MotorVehicle::~MotorVehicle() {}
void MotorVehicle::drive(int speed, int distance)
{
            cout << "Dummy drive() of MotorVehicle." << endl;
}
```

Declaration of child classes

- The declaration of a child class is similar to its representation in object relational diagram, we focus only on the difference compared to its parent classes
- Advantages
  - Simplify class declaration,
  - Support encapsulation principles of OOP
  - Support code re-usability (re-use the definitions of attributes and functions)
  - Hiding information can play an important role in creating inheritance tree

Definition of child class Car

Showing the relation between the child class **Car** and its parent class **MotorVehicle**

```
class Car : public MotorVehicle
{
    public:
        Car (int passengers);
        ~Car();
    private:
            int passengers;
};
```

Class Car

```
class Car : public MotorVehicle
{
public:
        Car (int vin, string make, string model, int passengers);
        ~Car();
private:
                int passengers;
};
```

Convention: all arguments of the parent class are located at the head of the list.

```
Car::Car(int vin, string make, string model, int passengers)
{
        this->vin = vin;
        this->make = make;
        this->model = model;
        this->passengers = passengers;
}
Car::~Car() {}
```

Definition of child class

- Disadvantages: access directly data of basis classes
  - Missing encapsulation: have to know well about basic classes
  - Do not re-use constructor code of basic classes
  - Can not initialize private members of basic classes because of missing access permission
- Rule: an object of a child class consists of an object of its parent class and complement features of the child class
  - A representation of the basic class will be created first, then it will "add" the complement features of the derived class
- Hence, we will use constructor of the basic class.

Definition of child class

- To use constructor of a basic class, we use initialization list of constructor (similarly to the initialization of constant members)
  - The only way to create the part belonging to the representation of the parent class that is created first

```
Car::Car(int vin, string make, string model, int passengers)
: MotorVehicle(vin, make, model)
{
    this->passengers = passengers;
}
```

Calling constructor of **MotorVehicle** with arguments **vin**, **make**, **model**

We do not need to initialize **vin**, **make**, **model** from inside constructor of **Car** anymore

Definition of child class

- To ensure that a representation of the basis class is always created first, if we skip constructor call of the basic class at the initialization of the derived class, the compiler will automatically insert the default constructor of the basic class
- Although we need to call explicitly constructor of the basic class, at the destructor of the derived class, the similar call for destructor of the basic class is not neccessary
  - this is done automatically

Definition of child class

ck như

```
class Truck : public MotorVehicle {
public:
      Truck (int vin, string make, string model,
      int maxPayload);
      ~Truck();
      void Load();
      void Unload();
private:
            int maxPayload;
};
```

```
Truck::Truck(int vin, string make, string model,
int maxPayload)
: MotorVehicle(vin, make, model)
{
this->maxPayload = maxPayload;
}
Truck::~Truck() {}
void Truck::Load() {…}
void Truck::Unload() {…}
```

Access to inheritance members

- The access to inheritance members depends not only on how they are declared (with keywords public, protected or private) in base class.

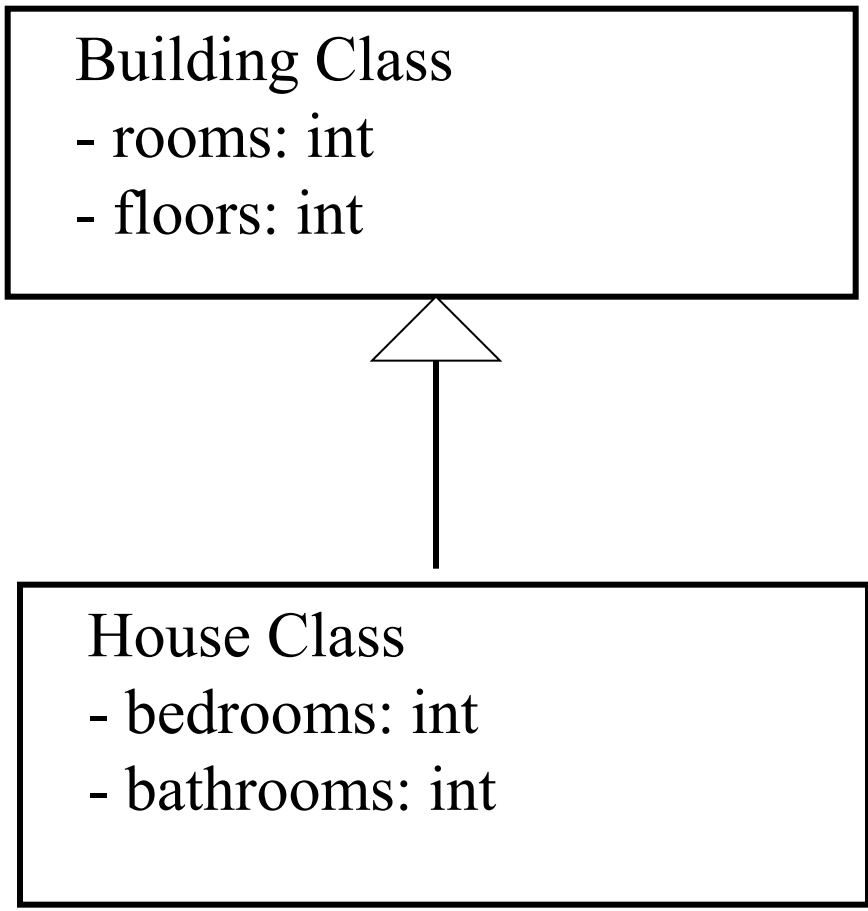
- Access also depends on the inheritance type - public, protected or private – Inheritance type defined in the **definition of derived class.**

Public inheritance

- Public inheritance means:
  - public members of base class become public members của derived class;
  - protected members of base class become protected members of derived class;
  - private members of base class can not be accessed in derived class.

Example

- Suppuse there are two following inheritance classes



```
class Building
{
public:
    void setRooms(int numRooms);
    int getRooms() const;
    void setFloors(int numFloors);
    int getFloors() const;
private:          // ??
    int rooms;   // Number of rooms
    int floors;    // Number of floors
};
class House : public Building
{
public:
    void setBedrooms(int numBedrooms);
    int getBedrooms() const;
    void setBathrooms(int numBathrooms);
    int getBathrooms() const;
private:
     int bedrooms;   // Number of bedrooms
     int bathrooms;   // Number of bathrooms
};
```

Can not access from derived class => chuyển thành : protected

Class MotorVehicle and Truck

- Return to the inheritance tree with MotorVehicle that is the basic class
  - All data members are declared as **private**, hence they *are only accessed* from instances of MotorVehicle
  - All the derived classes have no permission to access private members of MotorVehicle
- Hence, the following code

is not correct

```
class MotorVehicle {
...
private:
        int vin;
        string make;
        string model;
};
```

```
void Truck::Load() {
            if (this->make == "Ford") {
            ...
            }
}
```

Class MotorVehicle and Truck

- Suppose we want child classes of MotorVehicle to be able to access its data
- Replace the keyword **private** by **protected**,
- The following code will not give error anymore

```
class MotorVehicle {
...
protected:
        int vin;
        string make;
        string model;
};
```

```
void Truck::Load() {
            if (this->make == "Ford") {
            ...
            }
}
```

Protected inheritance

- public members of base class become **protected** members of derived class;
- protected members of base class become protected members of derived class;
- private members of base class can not be accessed in derived class.
- Inheritance relation will be seen from
  - All methods in Car,
  - All methods of child classes of Car
- However, all other objects of C++ will not see this relation

Private inheritance

- public members of base class become **private** members of derived class;
- protected members of base class become **private** of derived class;
- private members of base class can not be accessed in derived class.
- only its representation knows that it is inherited from its parent classes
- External objects can not interact with the child class as with the parent class, because all the inheritance members become **private** in the child class
- We can use **private** inheritance to create a child class that have all the features of the parent class without being known from outside.