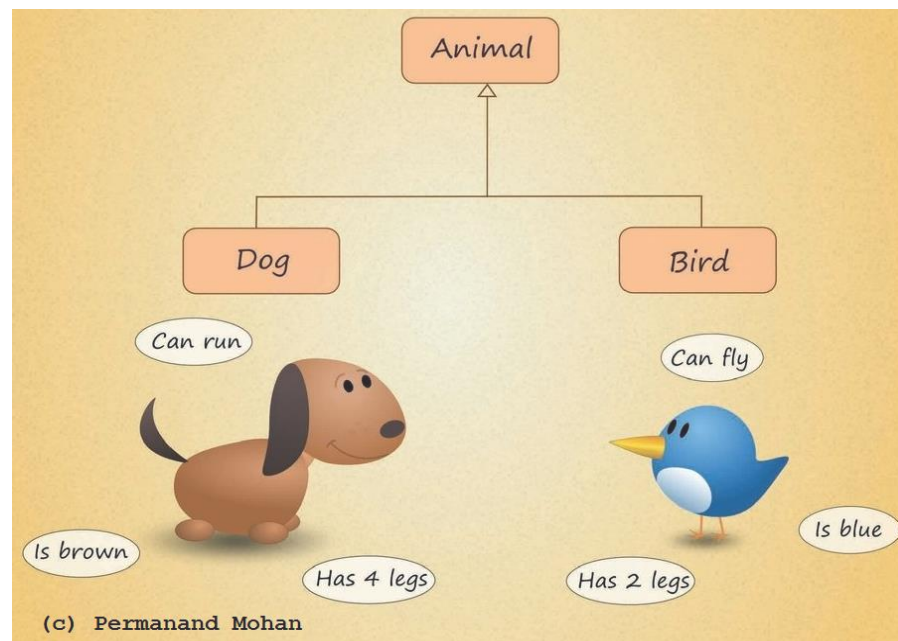




by Sinipull for codecall.net



Chapter 4

Thừa kế và đa hình

CT176 – LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Mục tiêu

Chương này nhằm giới thiệu
tính thừa kế và tính đa hình trong Java

Nội dung

- Thừa kế
 - Thừa kế là gì?
 - Thừa kế trong Java
 - Hàm xây dựng trong thừa kế
- Đa hình
 - Nạp đè phương thức
 - Đa hình
 - Ứng dụng của tính đa hình
- Lớp trừu tượng & Phương thức trừu tượng
- Đa thừa kế (multiple inheritance)
- Giao diện (interface)

Thừa kế là gì?

Khái quát hóa và chuyên biệt hóa

- Một đối tượng trong thực tế thường là một **phiên bản chuyên biệt** của một đối tượng khác khái quát hơn
- Khái niệm “côn trùng” mô tả một loài sinh vật rất chung chung với nhiều đặc tính (không xương sống, 3 cặp chân,...)
- Châu chấu và ong vò vẽ là côn trùng:
 - Chia sẻ chung các đặc điểm của côn trùng
 - Có một số đặc điểm riêng:
 - Châu chấu có khả năng nhảy
 - Ong vò vẽ có kim và khả năng chích

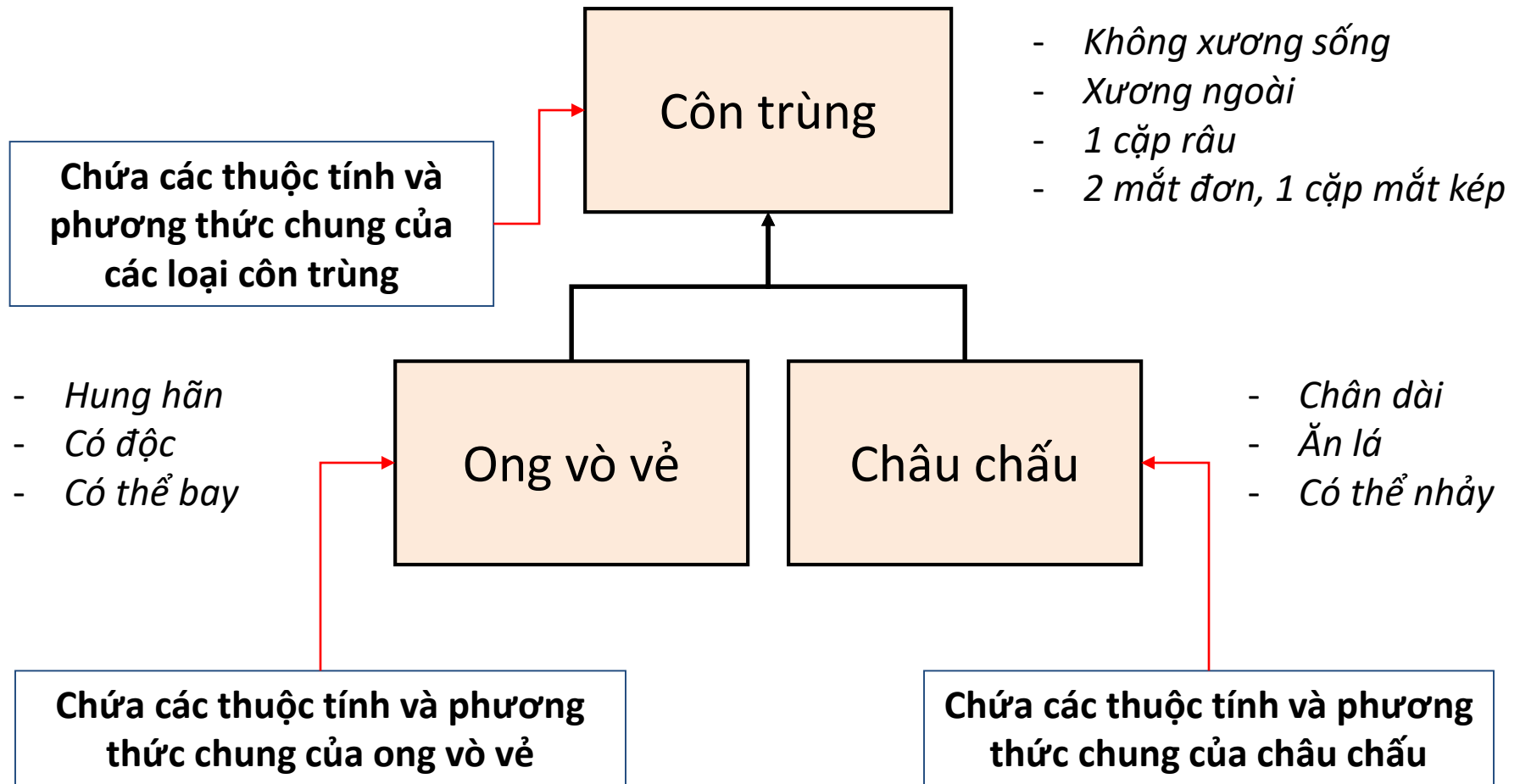
⇒ Châu chấu và ong vò vẽ **là** hai “phiên bản” đặc biệt của côn trùng

Thừa kế và quan hệ là (is-a)

- **Thừa kế** được sử dụng để mô hình hóa mối quan hệ **là**: (hay thực hiện sự chuyên biệt hóa):
 - Lớp thừa kế: lớp con (subclass)
 - Lớp được thừa kế: lớp cha (superclass)
- Quan hệ giữa lớp cha và lớp con: “**là**”
 - Một con châu chấu “là” một côn trùng
 - Một con ong vò vẽ “là” một côn trùng
- Một lớp con là sự **chuyên biệt hóa** của lớp cha:
 - Mang tất cả các đặc điểm của lớp cha
 - Thêm một số đặc điểm đặc trưng riêng
- **Thừa kế** dùng để **mở rộng khả năng** của một lớp



Thừa kế và quan hệ là (is-a)



Thừa kế trong Java

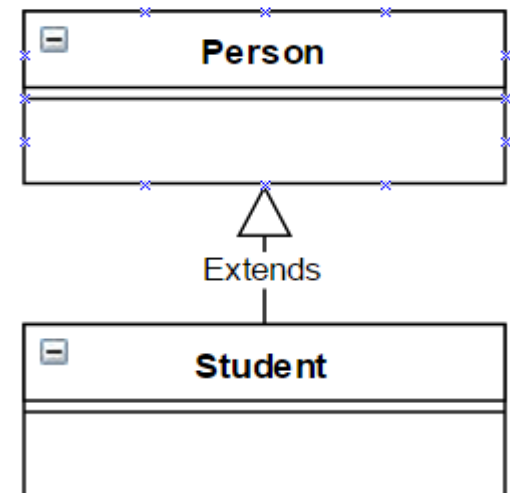
Tạo lớp thừa kế

- Khai báo:

```
modifier class <subclass name> extends <superclass name> {  
    //subclass members  
}
```

- Ví dụ: Tạo lớp Student thừa kế từ lớp Person

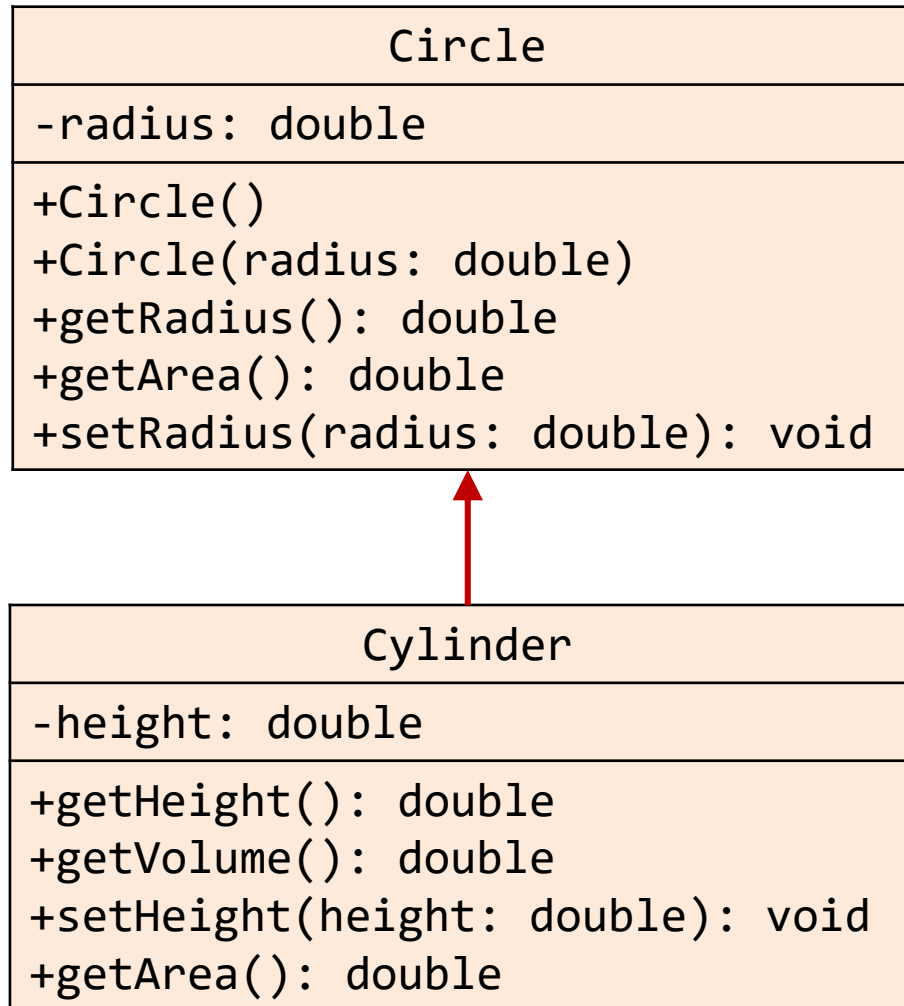
```
public class Student extends Person {  
    //Các thành phần của Lớp Student  
}
```



Qui tắc trong thừa kế

1. Lớp con **thừa kế** (có) **tất cả các thành phần** của lớp cha
2. Lớp con có thể **truy xuất** các thành phần **public** và **protected** của lớp cha
3. Lớp con có thể **có thêm** các thuộc tính, các phương thức mới
4. Lớp con có thể **nạp đè** (overriding) các phương thức của lớp cha

Ví dụ



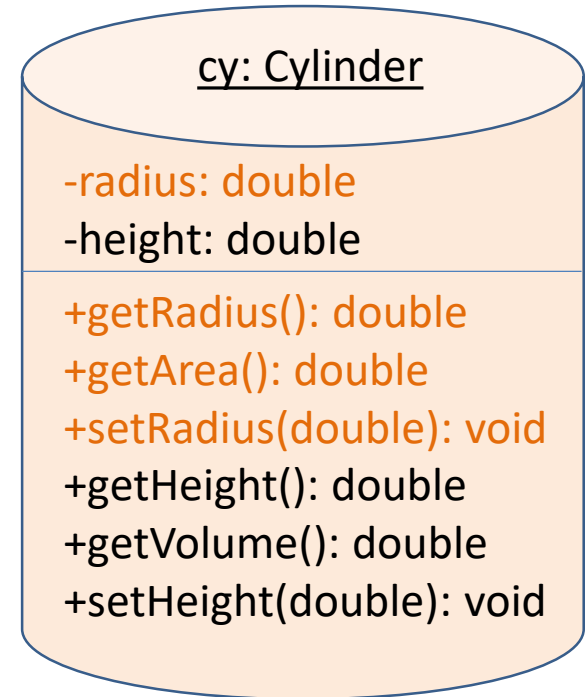
Ví dụ

```
class Circle {  
    private double radius;  
  
    public Circle() {  
        radius = 0;  
    }  
    public Circle(float r) {  
        radius = r;  
    }  
    public double getRadius() {  
        return radius;  
    }  
    public double getArea() {  
        return Math.PI*radius*radius;  
    }  
    public void setRadius(double r) {  
        radius = r;  
    }  
}
```

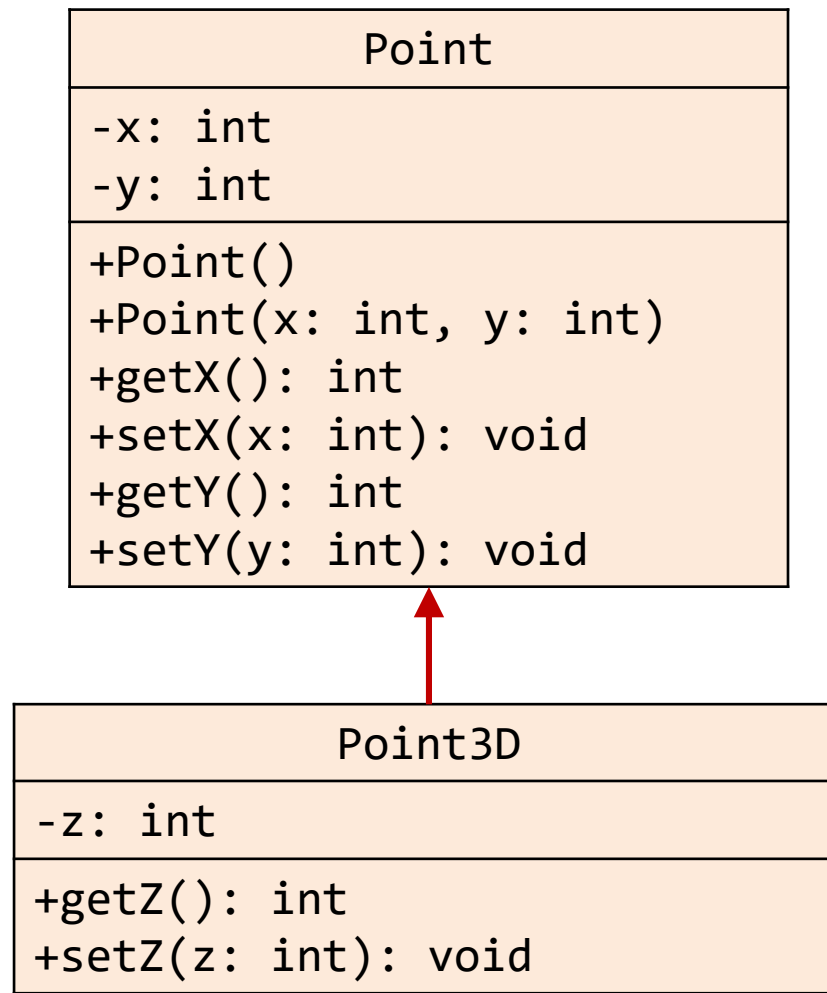
```
public class Cylinder extends Circle  
{  
    private double height;  
  
    public double getHeight() {  
        return height;  
    }  
    public void setHeight(double h) {  
        height = h;  
    }  
    public double getVolume() {  
        return getArea() * height;  
    }  
}
```

Ví dụ

```
Class UseCylinder {  
    public static void main(String []args) {  
        Cylinder cy = new Cylinder();  
  
        cy.setRadius(10.0);  
        cy.setHeight(5.0);  
  
        System.out.println("Cylinder radius: " +  
                           cy.getRadius());  
        System.out.println("Cylinder height: " +  
                           cy.getHeight());  
        System.out.printf("Cylinder volume: %.2f",  
                           cy.getVolume());  
    }  
}
```



Bài tập



Hàm xây dựng trong thừa kế

- Khi đối tượng thuộc lớp con được tạo ra:
 - Hàm xây dựng tương ứng của lớp con sẽ được gọi
 - Nếu hàm XD của lớp con không gọi đến hàm XD của lớp cha, **hàm XD mặc nhiên** của lớp cha sẽ tự động được gọi trước khi hàm XD lớp con được thực hiện

```
Cylinder cy = new Cylinder();
```

<u>cy: Cylinder</u>
-radius: 0
-height: ...

```
Circle() { radius=0; }  
Cylinder() {}
```

- Nếu muốn gọi hàm xây dựng của lớp cha, ta sử dụng từ khóa **super**:

```
super([các tham số cho hàm XD của lớp cha]);
```

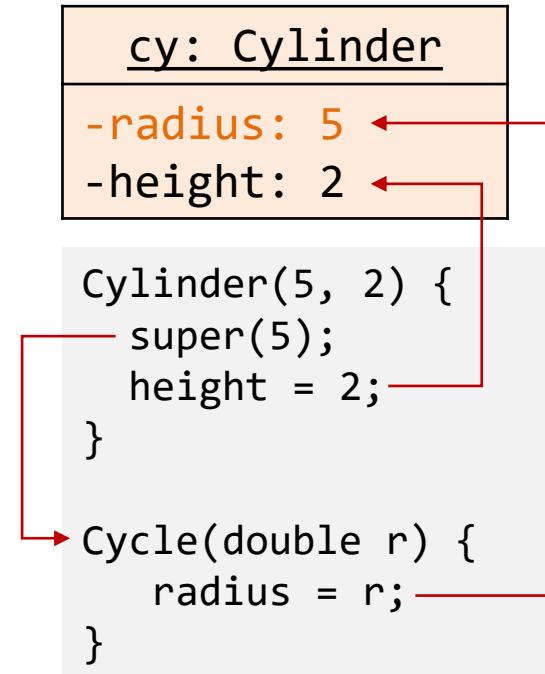
Hàm xây dựng trong thừa kế

```
public class Cylinder extends Circle {
    //các dữ liệu thành viên...

    public Cylinder() {
        super();
        height = 0;
    }

    public Cylinder(int r, int height) {
        super(r);
        this.height = height;
    }
    //các hàm thành viên khác...
}
```

```
public static void main(String []args) {
    Cylinder cy = new Cylinder(5, 2);
    //...
}
```



Hàm xây dựng của lớp cha phải được gọi đầu tiên

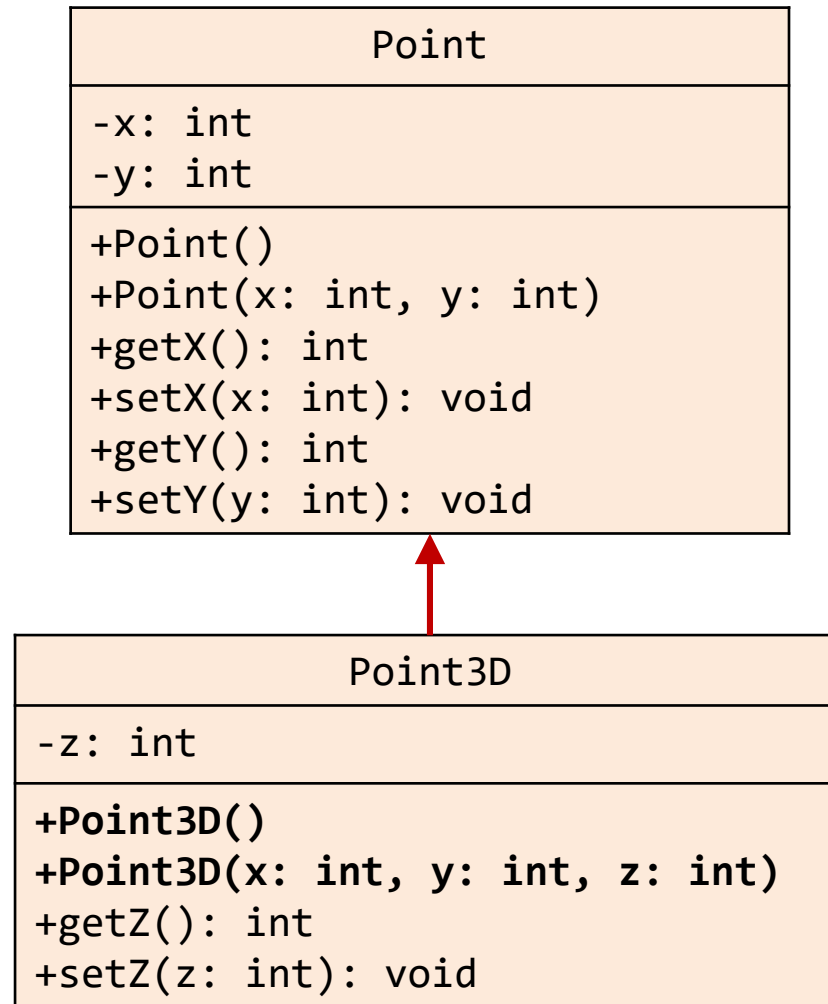
Từ khóa `super`

- Là một tham chiếu đến lớp cha hay các lớp tổ tiên (ancestor) của một lớp
- Cho phép các phương thức của lớp con truy xuất đến các thành phần lớp cha:
 - **`super`**([đối số]): truy xuất đến **hàm xây dựng** lớp cha
 - **`super`**.<pthức | ttính>: truy xuất đến **thành viên** lớp cha

```
class Cylinder {  
    public double getArea() {  
        return (2*Math.PI*radius*height) + (2*super.getArea());  
    }  
  
    //...  
}
```

gọi hàm `getArea()` của lớp `Circle`

Bài tập



Thành phần protected và final

- Thành phần `protected`:
 - Có thể được truy xuất bởi các phương thức trong lớp con và các phương thức trong cùng gói (package)
 - Đây là một hình thức giới hạn truy cập nằm giữa `public` và `private`
 - Về mặt `ngữ nghĩa`, đây là các thành phần dành cho các lớp con cháu
- Thành phần `final`:
 - Là các thành phần `không được phép nạp đè` trong lớp con
 - Được sử dụng để đảm bảo thành phần này chỉ được sử dụng bởi các lớp con hơn là thay đổi (nạp đè) chúng

Lớp `java.lang.Object`

- Java tổ chức các lớp dựa trên cách tiếp cận **gốc chung**:
 - Các lớp trong Java tạo thành cây phân cấp, trong đó lớp `Object` là gốc (root) của cây
 - Tất cả các lớp trong Java đều là con/cháu của lớp `Object`
 - Nếu một lớp không được khai báo thừa kế từ bất kỳ lớp nào, lớp đó mặc nhiên sẽ là lớp con của lớp `Object`
 - Một tham chiếu thuộc lớp `Object` có thể tham chiếu đến đối tượng thuộc bất kỳ lớp nào
 - Lớp này định nghĩa và cài đặt các phương thức và thuộc tính cơ bản mà một đối tượng bắt buộc phải có trong môi trường thực thi Java (JRE)

Lớp `java.lang.Object`

Phương thức	Mô tả
<code>public boolean equals(Object obj);</code>	So sánh hai đối tượng có “bằng” nhau hay không
<code>public String toString();</code>	Trả về chuỗi mô tả cho đối tượng
<code>public final Class getClass();</code>	Trả về kiểu (lớp) của đối tượng
<code>protected void finalize();</code>	Hàm hủy của đối tượng, sẽ được gọi bởi bộ thu hồi rác (garbage collector)

- Ngoài ra, còn các phương thức cần thiết để các đối tượng có thể thực thi trong môi trường đa luồng (multi-threading)

Nạp đề hàm & tính đa hình

Nạp đè hàm (method overriding)

- Lớp con có thể có thành viên (thuộc tính/phương thức) trùng với thành viên của lớp cha
- Định nghĩa một hàm thành viên lớp con có **chữ ký** trùng với hàm thành viên lớp cha gọi là **nạp đè hàm**

- Chữ ký hàm (method signature): bao gồm tên hàm + đối số

```
public void setHeight(double h) { ... }
```

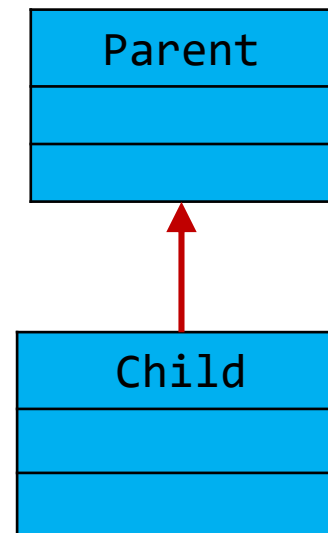
- Khi một phương thức của lớp cha bị đè, nó sẽ bị “che” đi bởi phương thức của lớp con
- Các thành viên **final** của lớp cha không thể bị nạp đè
- Muốn gọi hàm bị che đi ở lớp cha, ta dùng tham chiếu **super**

```
super.getArea()    (xem ví dụ trong phần từ khóa super)
```

Sự tương thích giữa tham chiếu & đối tượng

- Một **tham chiếu thuộc lớp cha** có thể tham chiếu đến:
 - Đối tượng thuộc lớp cha
 - Đối tượng thuộc lớp con
- Một **tham chiếu thuộc lớp con** chỉ có thể tham chiếu đến đối tượng thuộc lớp con

```
Parent p;  
p = new Parent(...); ✓  
p = new Child(...); ✓  
  
Child c;  
c = new Child(...); ✓  
c = new Parent(...); ✗
```



Tính đa hình (polymorphism)

- Cùng **1 thông điệp** nhưng sẽ được **xử lý khác nhau** tùy vào ngữ cảnh cụ thể
- Chỉ được thể hiện khi có sử dụng **thừa kế + nạp đè hàm**

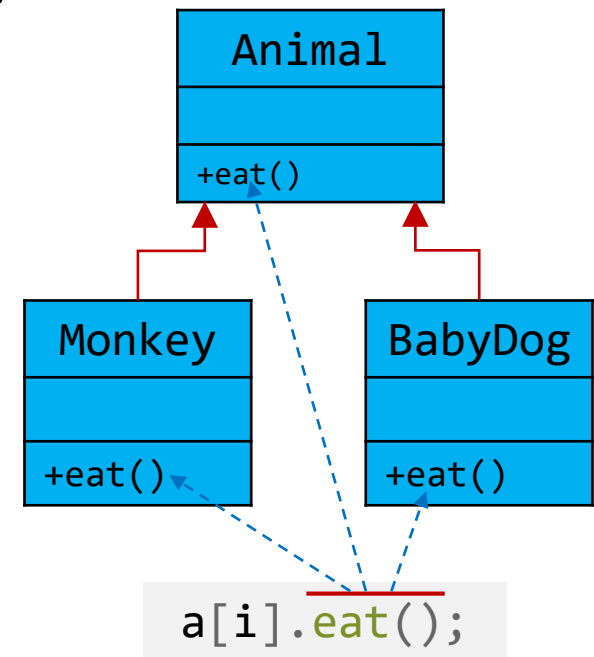
```
class Animal {  
    public void eat() {  
        System.out.println("Eating...");  
    }  
  
class Monkey extends Animal {  
    public void eat() {  
        System.out.println("Eating fruits...");  
    }  
  
class BabyDog extends Animal {  
    public void eat() {  
        System.out.println("Drinking milk...");  
    }  
}
```

```
Animal a[] = new Animal[3];  
  
a[0] = new Animal();  
a[1] = new Monkey();  
a[2] = new BabyDog();  
  
for (int i=0; i<3; i++)  
    a[i].eat();
```

```
Eating...  
Eating fruits...  
Drinking milk...
```

Liên kết tĩnh và liên kết động

- Tính đa hình được thực hiện bởi **liên kết động** (dynamic binding):
 - Liên kết giữa **lời gọi hàm** và **định nghĩa hàm** sẽ được thực hiện **lúc thực thi** chương trình (runtime)
 - Liên kết động chỉ được áp dụng cho các phương thức và thuộc tính **bị nạp đề**
- Liên kết giữa lời gọi hàm và định nghĩa hàm không bị nạp đề:
 - Được thực hiện **lúc biên dịch**
 - Được gọi là **liên kết tĩnh** (static binding)
 - Áp dụng cho cả thuộc tính



Liên kết tĩnh và liên kết động

```
class Animal {  
    public static int count = 0 ;  
    public Animal() {  
        count++;  
    }  
}  
  
class Monkey extends Animal {  
    public static int count = 0 ;  
    public Monkey() {  
        count++;  
    }  
}  
  
class BabyDog extends Animal {  
    public static int count = 0 ;  
    public BabyDog() {  
        count++;  
    }  
}
```

```
Animal a[] = new Animal[3];
```

```
a[0] = new Animal();
```

```
a[1] = new Monkey();
```

```
a[2] = new BabyDog();
```

```
for (int i=0; i<3; i++)  
    System.out.println(a[i].count);
```

```
System.out.println(Animal.count);
```

```
System.out.println(Monkey.count);
```

```
System.out.println(BabyDog.count);
```

Kết quả: 3

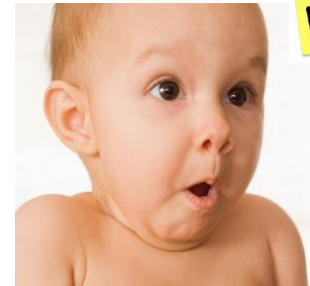
3

3

3

1

1



WHY?

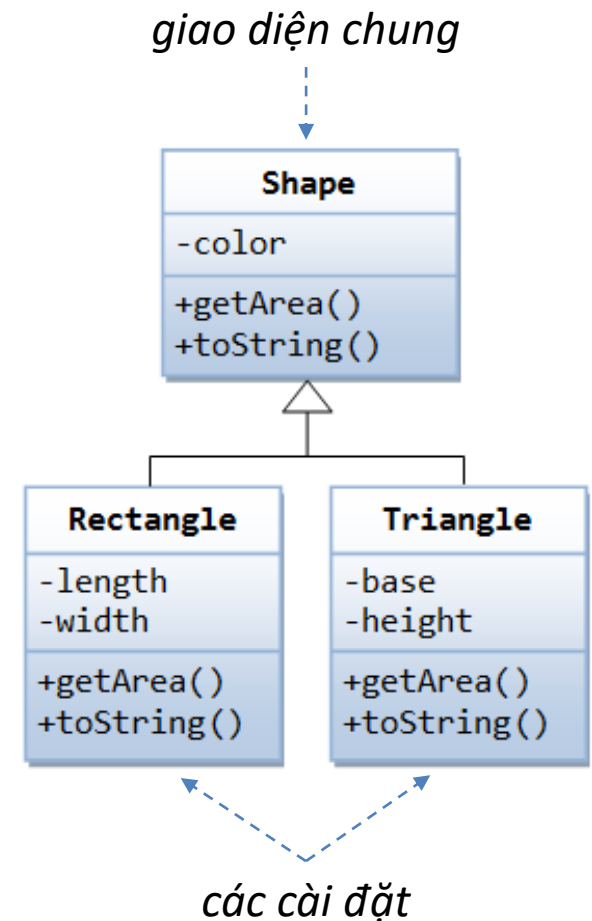
Ghi nhớ về tính đa hình

- Một **tham chiếu kiểu lớp cha**:
 - Có thể **tham chiếu** đến đối tượng của lớp cha và đối tượng của lớp con
 - Chỉ có thể **truy xuất** các thành phần của lớp cha
- Liên kết động chỉ được áp dụng cho các phương thức **bị ghi đè** (overriding)
- Không thể nạp đè các thành phần **final** của lớp cha
- Các phương thức **bị chồng** (overloading) không được áp dụng liên kết động

Đa hình \approx Thừa kế + Nạp đè hàm

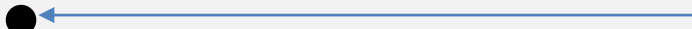
Ứng dụng của tính đa hình

- Tách rời giữa “**giao diện**” (interface) và “**cài đặt**” (implementation), cho phép nhiều người lập trình cùng tham gia vào giải quyết một vấn đề phức tạp dựa trên một “giao diện” đã định nghĩa sẵn
- Cho phép **quản lý các đối tượng** trong chương trình một cách hiệu quả hơn



Ứng dụng của tính đa hình

```
public class Shape {  
    private String color;  
  
    public Shape (String color) {  
        this.color = color;  
    }  
  
    public Shape () {  
        this.color = "Unknown";  
    }  
  
    public String toString() {  
        return "color=\"" + color + "\"";  
    }  
  
    public double getArea() {  
        System.out.println("Shape unknown! Cannot compute area!");  
        return -1;    // error  
    }  
}
```



```
public void inputValue() {  
    Scanner s = new Scanner(System.in);  
    System.out.println("Choose color: ");  
    this.color = s.nextLine();  
}
```

Ứng dụng của tính đa hình

```
public class Rectangle extends Shape {  
    private int len, width;  
  
    public Rectangle() {  
        super();    this.len = 0;    this.width = 0;  
    }  
  
    public Rectangle(String color, int len, int width) {  
        super(color);    this.len = len;    this.width = width;  
    }  
  
    public String toString() {  
        return "Rectangle (" + len + ", " + width + "), " + super.toString();  
    }  
  
    public double getArea() {  
        return len * width;  
    }  
    ● ←  
}
```

```
    public void inputValue() {  
        Scanner s = new Scanner(System.in);  
        super.inputValue();  
        System.out.println("Enter length: ");  
        this.length = s.nextInt();  
        System.out.println("Enter width: ");  
        this.width = s.nextInt();  
    }
```

Ứng dụng của tính đa hình

```
public class Triangle extends Shape {  
    private int base, height;  
  
    public Triangle(String color, int base, int height) {  
        super();    this.base = 0;    this.height = 0;  
    }  
  
    public Triangle(String color, int base, int height) {  
        super(color);    this.base = base;    this.height = height;  
    }  
  
    public String toString() {  
        return "Triangle (" + base + ", " + height + "), " + super.toString();  
    }  
  
    public double getArea() {  
        return 0.5*base*height;  
    }  
    ● ←  
}
```

```
    public void inputValue() {  
        Scanner s = new Scanner(System.in);  
        super.inputValue();  
        System.out.println("Enter base: ");  
        this.base = s.nextInt();  
        System.out.println("Enter height: ");  
        this.height = s.nextInt();  
    }
```


Ứng dụng của tính đa hình

```
public class TestShape2 {  
    public static void main(String[] args) {  
        Shape []sList = new Shape[10];  
        int opt, count = 0;  
        do {  
            Scanner kb = new Scanner(System.in);  
            System.out.print("Choose shape (0: exit, 1: Rect, 2: Triangle): ");  
            opt = kb.nextInt();  
  
            if (opt == 1)           //rectangle  
                sList[count] = new Rectangle();  
            else if (opt == 2)     //triangle  
                sList[count] = new Triangle();  
  
            if (opt == 1 || opt == 2)  
                sList[count++].inputValue();  
        } while (opt != 0);  
  
        for (int i=0; i< count; i++)  
            System.out.println(sList[i]);  
    }  
}
```

```
Choose shape (0: exit, 1: Rectangle, 2: Triangle): 1  
Choose color: white  
Enter length: 1  
Enter width: 2  
Choose shape (0: exit, 1: Rectangle, 2: Triangle): 2  
Choose color: blue  
Enter base: 2  
Enter height: 3  
Choose shape (0: exit, 1: Rectangle, 2: Triangle): 1  
Choose color: green  
Enter length: 4  
Enter width: 5  
Choose shape (0: exit, 1: Rectangle, 2: Triangle): 0  
Rectangle (1, 2), color="white"  
Triangle (2, 3), color="blue"  
Rectangle (4, 5), color="green"
```

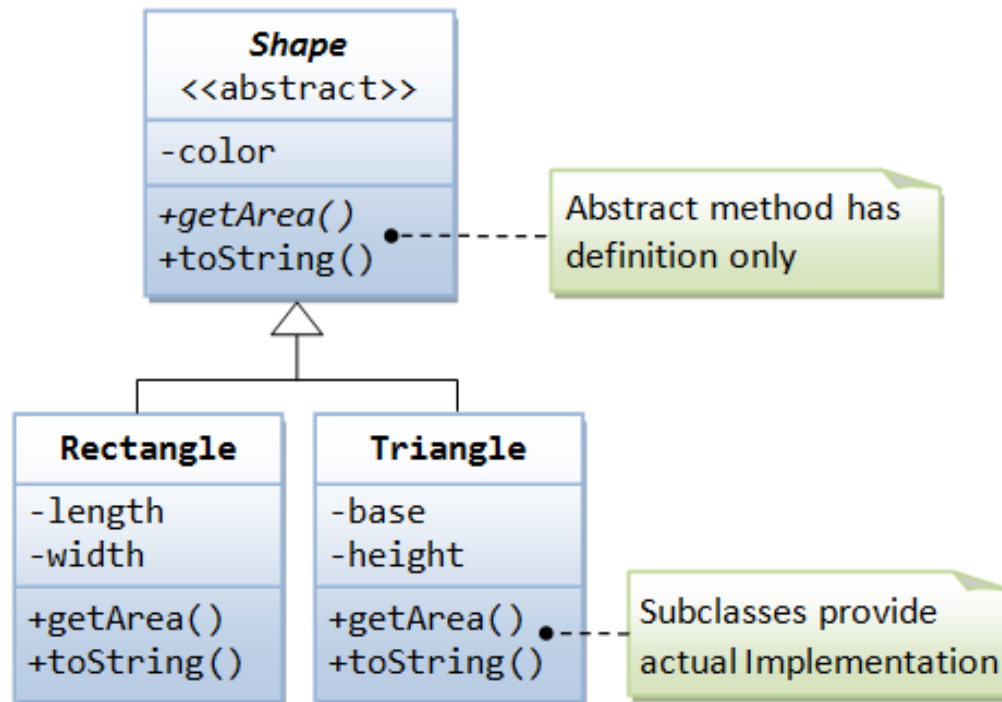
Phương thức trừu tượng & Lớp trừu tượng

Phương thức trừu tượng

- Là phương thức **chỉ có khai báo**, không có cài đặt
- Dùng từ khóa **abstract** được để khai báo một phương thức là trừu tượng
- Các lớp con **phải cài đặt** các phương thức trừu tượng của lớp cha
- Dùng cho các phương thức chưa có định nghĩa cụ thể trong ngữ cảnh của lớp đó
- Là một phương pháp để bắt buộc các lớp con phải cài đặt các phương thức theo yêu cầu

Phương thức trừu tượng

```
public class Shape {  
    //các thành viên khác ...  
  
    public abstract double getArea();    //diện tích các loại hình khác  
                                         //nhau thì khác nhau về cách tính  
}
```



Lớp trừu tượng

- Là lớp không thể dùng để tạo đối tượng
- Thường được sử dụng để thừa kế (là lớp cha cho các lớp khác)
- Lớp trừu tượng thể hiện một dạng “chung chung” hoặc trừu tượng của các lớp dẫn xuất từ đó.
- Để khai báo một lớp là trừu tượng:
 - Lớp chứa phương thức trừu tượng mặc nhiên là lớp trừu tượng
 - Nếu lớp không có phương thức trừu tượng: thêm từ khóa **abstract** trước từ khóa **class** trong khai báo lớp

Lớp trừu tượng

```
public class TestShape {  
    public static void main(String[] args) {  
        Shape s1 = new Rectangle("red", 4, 5);  
        System.out.println(s1);  
        System.out.println("Area is " + s1.getArea());  
  
        Shape s2 = new Triangle("blue", 4, 5);  
        System.out.println(s2);  
        System.out.println("Area is " + s2.getArea());  
  
        // Cannot create instance of an abstract class  
        Shape s3 = new Shape("green"); //Compilation Error!!  
    }  
}
```

Phương thức & lớp trừu tượng

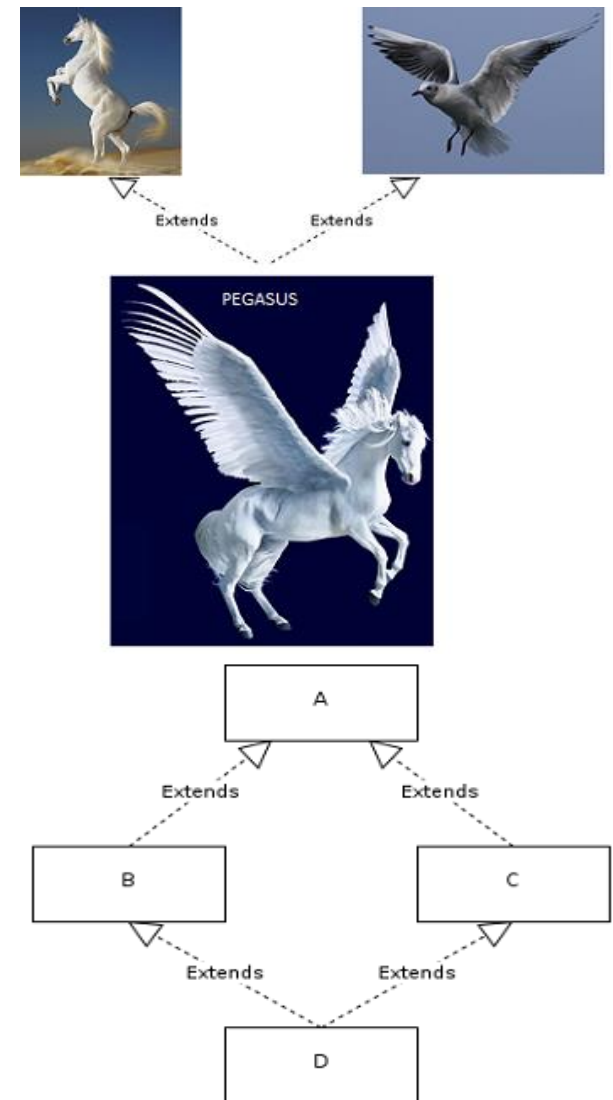
- Cung cấp một chuẩn (standard) hay giao diện (interface) cho việc phát triển ứng dụng
- Một phương thức trừu tượng không thể được khai báo `final` hay `private`
 - Final: không thể được nạp đè
 - Private: không thấy được bởi lớp con nên không thể nạp đè
- Nên lập trình **dựa vào giao diện**, không dựa vào cài đặt
 - Tạo tham chiếu thuộc lớp cha
 - Tham chiếu đến thể hiện cụ thể của lớp con
 - Gọi đến các phương thức được định nghĩa ở lớp cha

Đa thừa kế (multiple inheritance)

Đa thừa kế

- Đa thừa kế:
 - Một lớp con thừa kế từ **nhiều lớp cha**
 - Còn được gọi là **đa thừa kế cài đặt** (multiple inheritance of implementation)
- Java không hỗ trợ đa thừa kế:
 - Tránh **xung đột** các thuộc tính của các lớp cha (diamond problem)
 - Đảm bảo tính **đơn giản** của ngôn ngữ

JAVA: A *simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, dynamic language.*



Đa thừa kế

- Java hỗ trợ **đa thừa kế kiểu** (multiple inheritance of types): một lớp có thể cài đặt nhiều giao diện

```
//One class implements multiple interfaces  
public class <classname> implements <interface names> {  
    ...  
}
```

- Một lớp có thể vừa thừa kế, vừa cài đặt interface

```
public class <classname> extends <superclass> implements <interfaces> {  
    ...  
}
```

Giải lập đa thừa kế

- Dùng hàm mặc nhiên (default method) của giao diện:
 - Chỉ thừa kế được phương thức
 - Chỉ được hỗ trợ từ Java 8

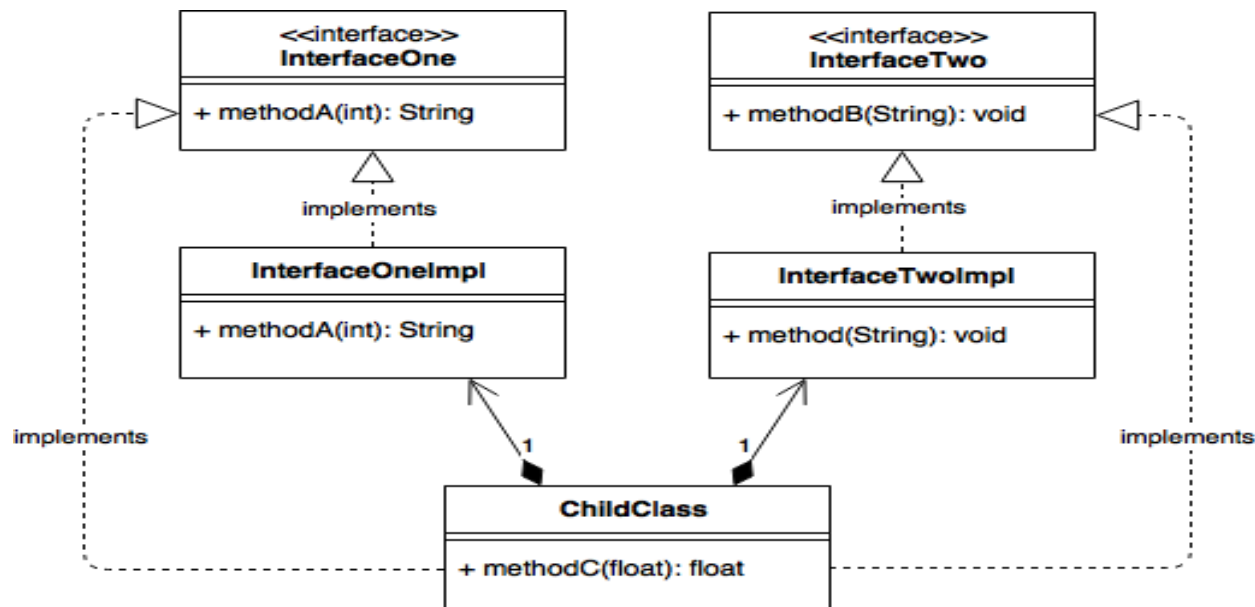
```
public class Button implements Clickable, Accessible {  
    public static void main(String[] args) {  
        Button button = new Button();  
        button.click();  
        button.access();  
    }  
}
```

```
interface Clickable{  
    default void click(){  
        System.out.println("click");  
    }  
}  
  
interface Accessible{  
    default void access(){  
        System.out.println("access");  
    }  
}
```

- Lưu ý: phương pháp này chỉ thừa kế p/thức, không thừa kế được thuộc tính

Giải lập đa thừa kế

- Dùng quan hệ composition:
 - “Không thật” chính xác về mặt ngữ nghĩa: quan hệ thừa kế là quan hệ **là**, trong khi composition là quan hệ **bao gồm**
 - Đây là phương pháp giải lập đa thừa kế được sử dụng rộng rãi
 - Cho phép thừa kế cả thuộc tính và phương thức



Giải lập đa thừa kế

```
class InterfaceOneImpl implements InterfaceOne {  
    //class properties ...  
  
    @Override  
    public String methodA(int a) {  
        //...  
    }  
}  
  
class InterfaceTwoImpl implements InterfaceTwo {  
    //class properties ...  
  
    @Override  
    public void methodB(String s) {  
        //...  
    }  
}
```

```
interface InterfaceOne {  
    String methodA(int a);  
}  
  
interface InterfaceTwo {  
    void methodB(String s);  
}
```

Giải lập đa thừa kế

```
public class ChildClass implements InterfaceOne, InterfaceTwo {  
    private InterfaceOne one;  
    private InterfaceTwo two;  
  
    ChildClass(InterfaceOne one, InterfaceTwo two) {  
        this.one = one;  
        this.two = two;  
    }  
  
    @Override  
    public String methodA(int a) {  
        return one.methodA(a);  
    }  
  
    @Override  
    public void methodB(String s) {  
        two.methodB(s);  
    }  
}
```

Giao diện (interface)

Giao diện (interface)

- Một giao diện có thể xem là một lớp **hoàn toàn ảo**: tất cả các phương thức đều không được cài đặt
- Một giao diện:
 - Chỉ chứa các **khai báo** của các phương thức với thuộc tính truy cập public
 - Hoặc các **hằng số tĩnh** public (public static final...)
 - Được khai báo bằng từ khóa **interface**

```
[public] interface <interfaceName> [extends <superInterface>] {  
    //khai báo của các hằng số (static final ...)  
  
    //khai báo các phương thức  
}
```

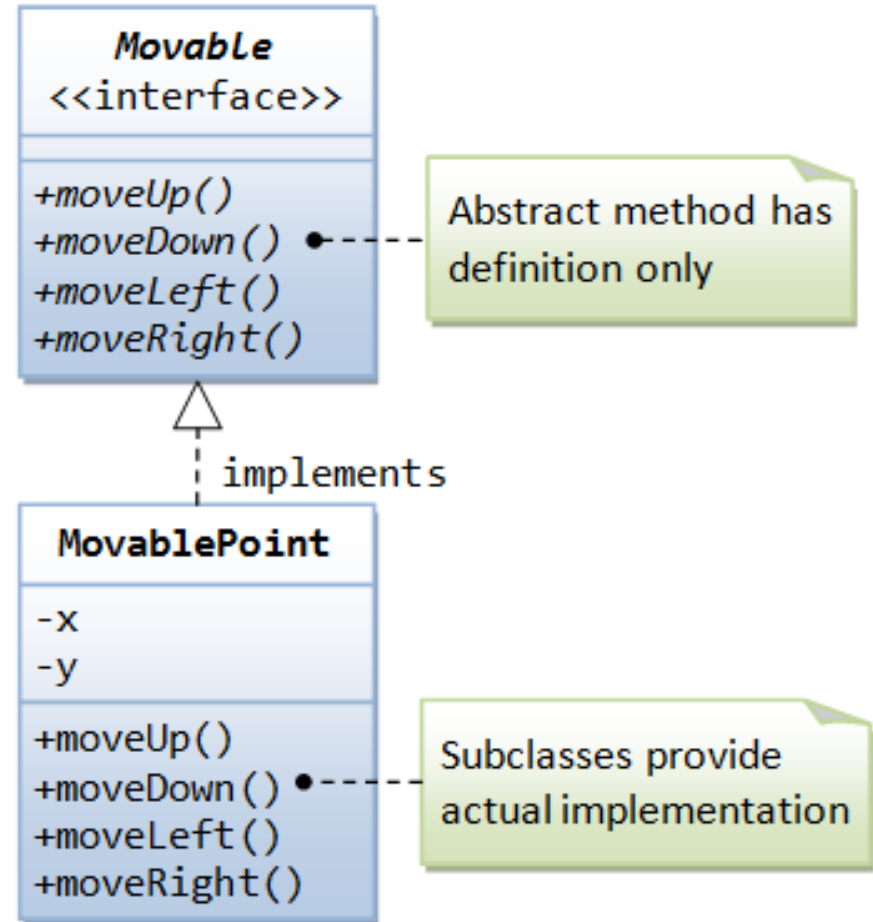

Giao diện (interface)

- Giao diện đóng vai trò như một “**cam kết**” (contract):
 - Giao diện **có thể làm được gì** (nhưng không chỉ định làm như thế nào)
 - Qui ước đặt tên: tiếp vị ngữ **-able** (có khả năng/có thể)
- Một lớp có thể cài đặt (implement) **các** giao diện:
 - Cài đặt tất cả các thuộc tính của các giao diện
 - ⇒ Xác nhận **khả năng** của lớp có thể làm được gì
 - Sử dụng từ khóa **implements**
- Không thể tạo đối tượng thuộc một giao diện, nhưng có thể **tạo tham chiếu** thuộc kiểu giao diện

Ví dụ

```
public interface Movable {

    //abstract methods to be implemented
    //by the subclasses
    public void moveUp();
    public void moveDown();
    public void moveLeft();
    public void moveRight();
}
```



Ví dụ

```
public class MovablePoint implements Movable {
    private int x, y;    //coordinates of the point

    public MovablePoint(int x, int y) {
        this.x = x;    this.y = y;
    }
    public String toString() {
        return "(" + x + "," + y + ")";
    }
    public void moveUp() { y--; }
    public void moveDown() {
        y++;
    }
    public void moveLeft() {
        x--;
    }
    public void moveRight() {
        x++;
    }
}
```

Kết quả:

```
(5,5)
(5,6)
(6,6)
```

```
public class TestMovable {
    public static void main(String[] args) {
        Movable m1 = new MovablePoint(5, 5);

        System.out.println(m1);
        m1.moveDown();
        System.out.println(m1);
        m1.moveRight();
        System.out.println(m1);
    }
}
```

Phương thức mặc định (default method)

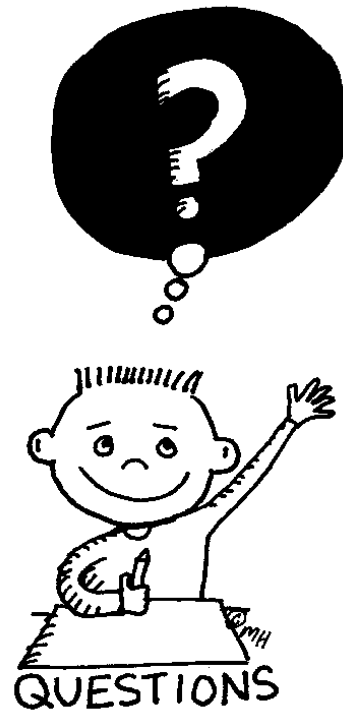
- Từ Java 8, các giao diện có thể có các **phương thức mặc định**:
 - Là phương thức được cài đặt (có thân hàm)
 - Cho phép thêm vào giao diện các phương thức mà không làm các lớp đã cài đặt giao diện bị lỗi
- Các lớp cài đặt phương thức có thể cài đặt hay không cài đặt các phương thức mặc định:
 - Không cài đặt: thừa kế phương thức mặc định
 - Cài đặt: nạp đè phương thức mặc định của giao diện
 - Không cài đặt, chỉ khai báo: phương thức ảo.
- Cú pháp: thêm từ khóa **default** trước khai báo

Tổng kết

- Tính thừa kế cho phép sử dụng lại mã (reuse code)
- Lớp thừa kế được gọi là lớp con, lớp được thừa kế được gọi là lớp cha
- Lớp con có tất cả các thành phần của lớp cha
 - Định nghĩa thêm thuộc tính hoặc phương thức mới
 - Nạp đè hàm của lớp cha
- Quan hệ giữa lớp con và lớp cha là quan hệ là
- Tính đa hình cho phép các loại đối tượng khác nhau ứng xử khác nhau với cùng 1 thông điệp
- Đa hình: thừa kế + nạp đè hàm + liên kết động

Tổng kết

- Java không hỗ trợ đa thừa kế (lớp con có hơn 1 lớp cha)
- Các kỹ thuật mô phỏng đa thừa kế:
 - Hàm mặc nhiên của giao diện
 - Quan hệ composition (delegation)
- Giao diện:
 - Đóng vai trò như một “cam kết” về tính năng của một kiểu
 - Như là một lớp hoàn toàn ảo: chỉ có khai báo phương thức, không có định nghĩa phương thức và các thuộc tính



Question?

CT176 – LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG