

Chapter 3

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

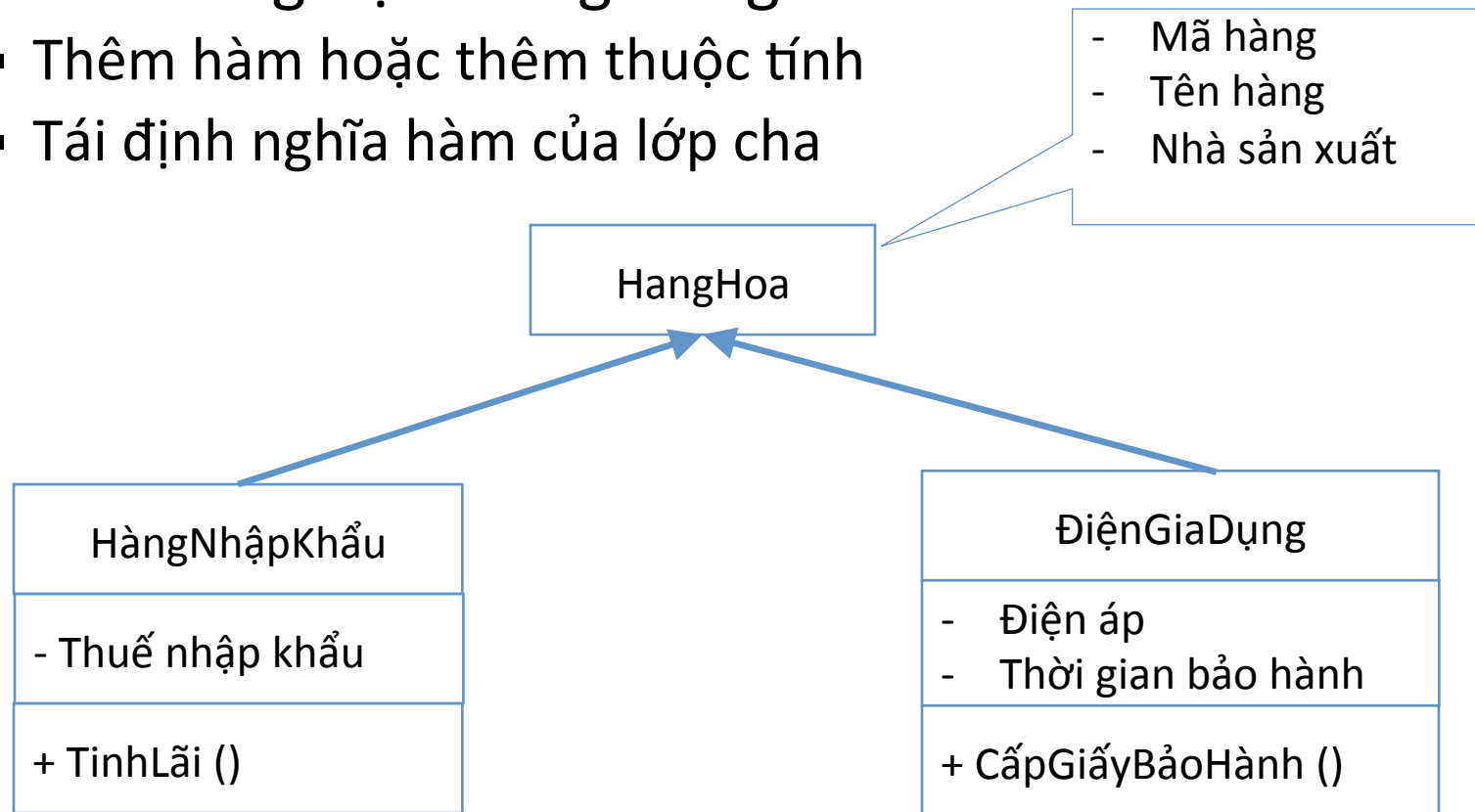
Thừa kế và quan hệ là (is-a)

- **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

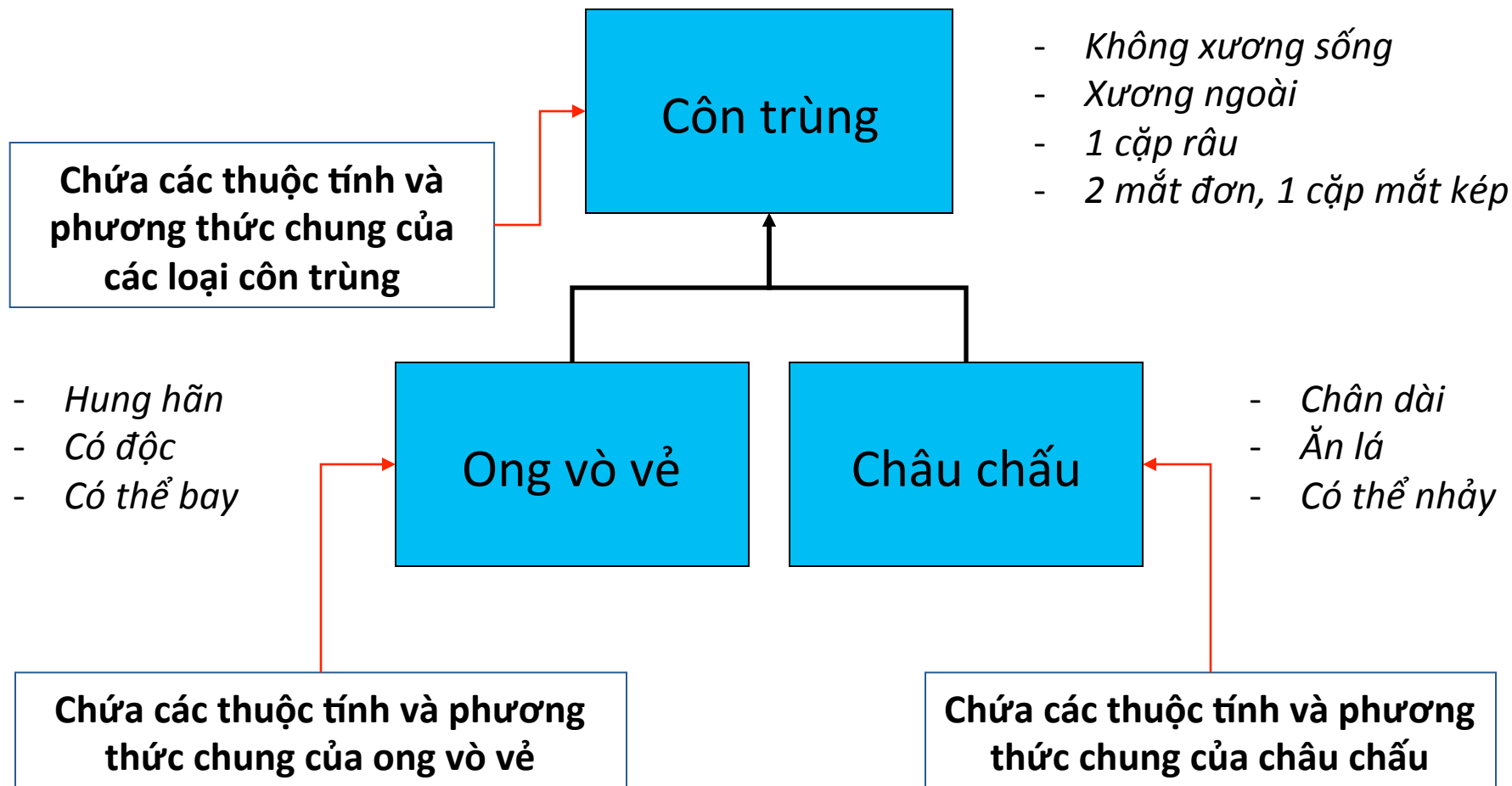


Thừa kế và quan hệ là (is-a)

- **Thừa kế (inheritance):**
- Lớp con phải chính là lớp cha ngoài ra nó còn có thêm những đặc trưng riêng của nó:
 - Thêm hàm hoặc thêm thuộc tính
 - Tái định nghĩa hàm của lớp cha



Thừa kế và quan hệ là (is-a)



Inheritance vs Composition

```
class Engine {} // The engine class.
```

```
class Automobile{} // Automobile class  
which is parent to Car class.
```

```
// Car is an Automobile, so Car class  
extends Automobile class.
```

```
class Car extends Automobile{
```

```
    // Car has a Engine so, Car class has an  
instance of Engine class as its member.
```

```
    private Engine engine;
```

```
}
```

Thừa kế trong Java

Qui tắc trong thừa kế

1. Lớp con thừa kế (có) tất cả các thành phần của lớp cha
2. Lớp con chỉ 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

Qui tắc trong thừa kế

- Lớp cha **không thể truy xuất** được thành phần của lớp con.
- Các thành phần của lớp con sẽ che **đi các thành phần trùng tên** trong lớp cha
 - Khi truy xuất thành phần trùng tên đó sẽ truy xuất thành phần của lớp con.
 - Muốn truy xuất thành phần của lớp cha thì phải chỉ rõ:
super.<tên hàm>
- Lớp kế thừa từ một lớp khác gọi là lớp dẫn xuất (lớp con), lớp dùng để xây dựng lớp dẫn xuất được gọi là lớp cơ sở (lớp cha)

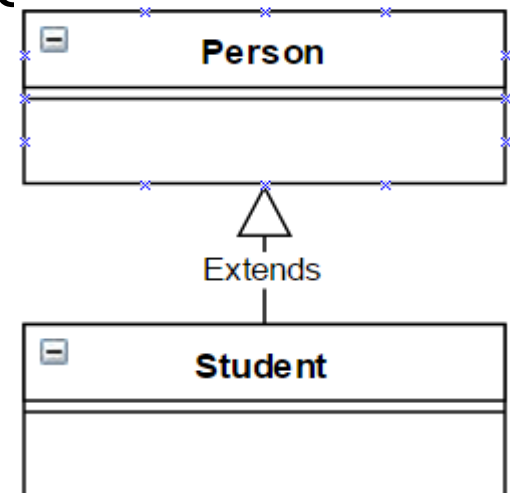
Tạo lớp thừa kế

- Khai báo:

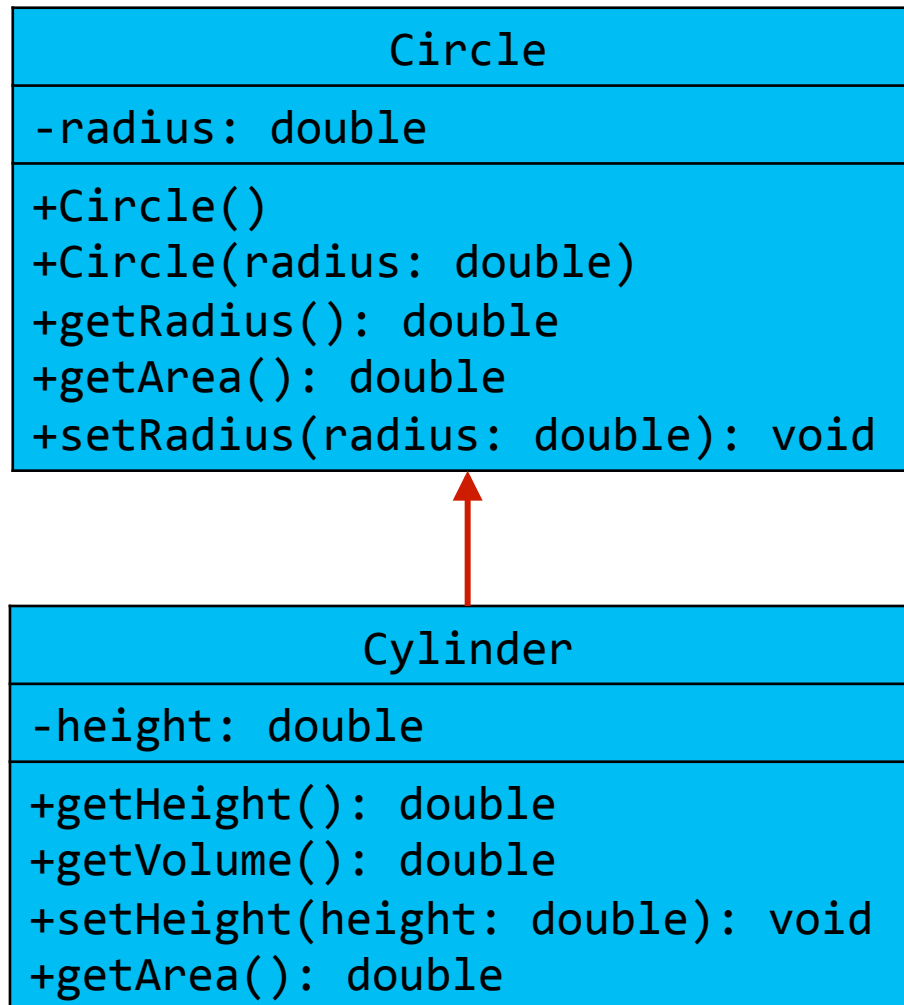
```
modifier class <subclass name> extends <superclass name> {  
    //subclass members  
}
```

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

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



Ví dụ



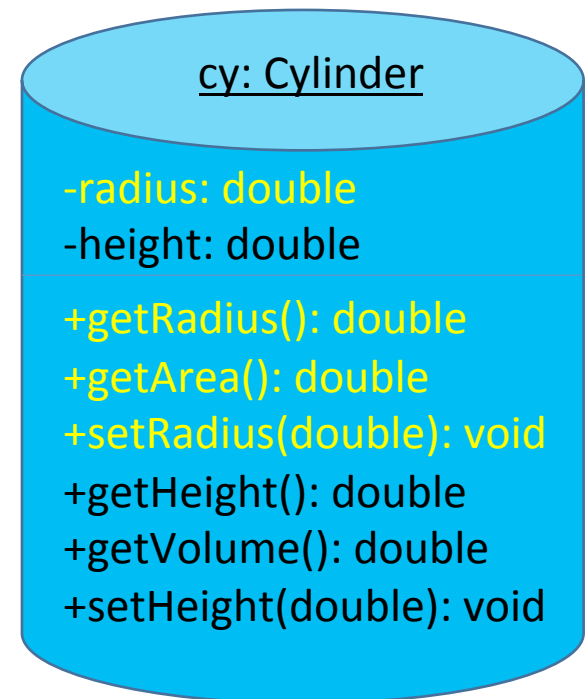
Ví dụ

```
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;  
    }  
}
```

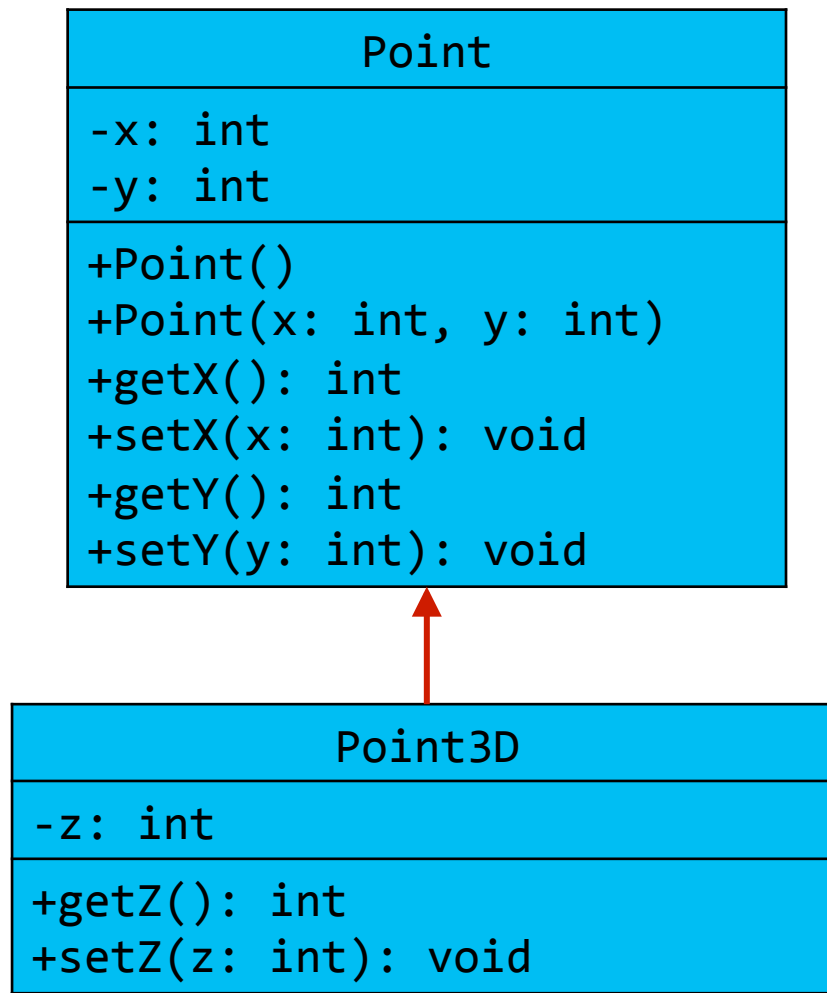
```
public class Cylinder extends Circle  
{  
    private double height;  
  
    public double getHeight() {  
        return height;  
    }  
    public void setHeight(double h) {  
        height = h;  
    }  
    public double getVolume() {  
        return getArea() * height;  
    }  
}
```

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

- Lớp con **Không** kế thừa các phương thức khởi tạo lớp cha.
- Có hai giải pháp gọi phương thức xây dựng của lớp cơ sở:
 - Sử dụng phương thức khởi tạo mặc định
 - Gọi hàm xây dựng một cách tường minh
- Việc khởi tạo thuộc tính của lớp cơ sở nên giao phó cho constructor của lớp cơ sở
- Sử dụng từ khóa **super** để gọi constructor của lớp cơ sở
 - Constructor của lớp cơ sở bắt buộc phải được thực hiện đầu tiên
 - Nếu lớp cơ sở không có constructor tường minh thì java sẽ tự động gọi constructor mặc định

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]);
```

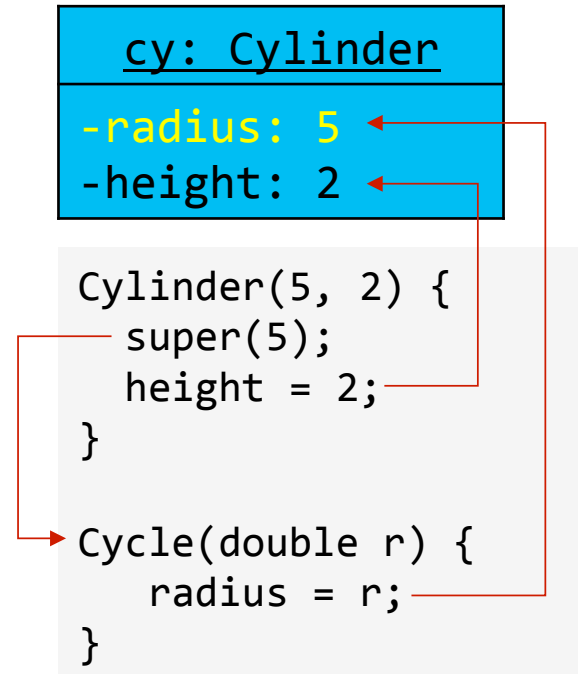
Hàm xây dựng trong thừa kế

```
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

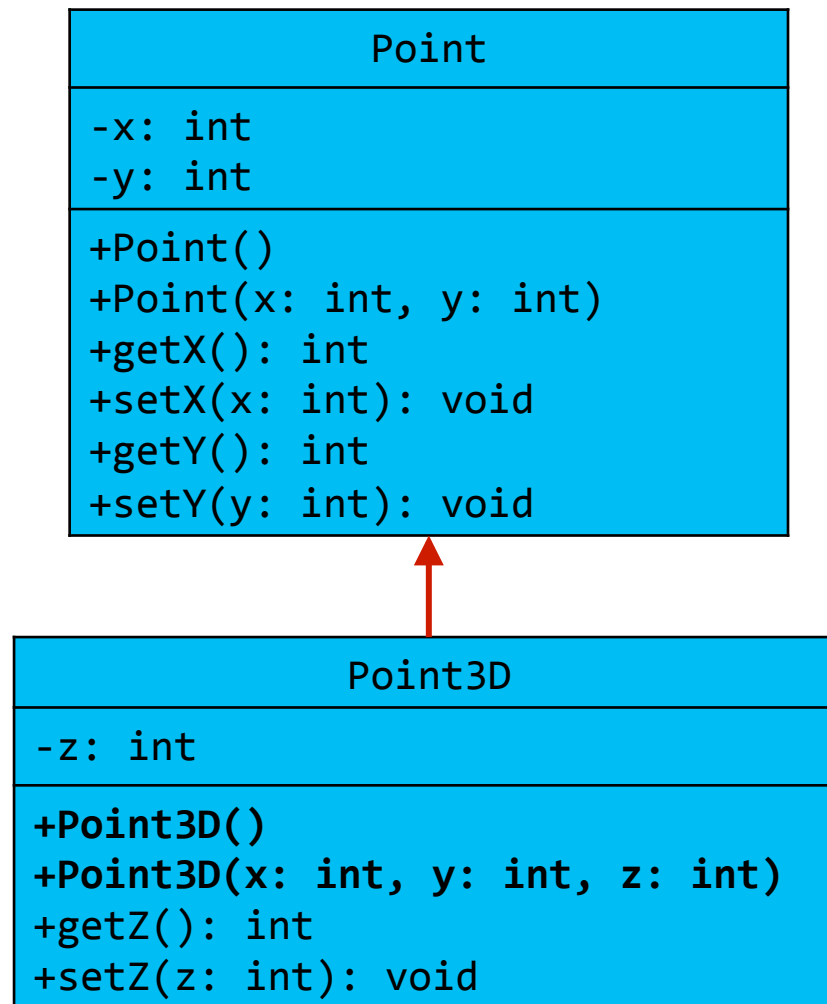
Từ khóa `super`

- Là một tham chiếu đến lớp cha hay cơ sở 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

Thành phần protected và final

- Thuộc tính final

- Là hằng số, chỉ được gán giá trị khởi tạo một lần, không thay đổi được giá trị

- Phương thức final

- Không cho phép định nghĩa lại ở lớp dẫn xuất

- Tham số final

- Không thay đổi được giá trị của tham chiếu

- Lớp final

Không định nghĩa được lớp dẫn xuất

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)
```

Nạp đề hàm (method overriding)

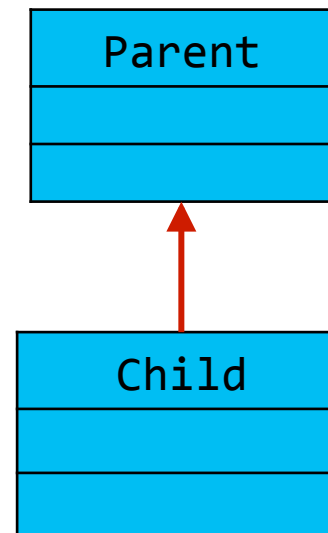
```
public class Test {  
    public static void main  
(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p (double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    public void p (double i) {  
        System.out.println(i);  
    }  
}
```

```
class B {  
    public void p (double i) {  
        System.out.println(i * 2);  
    }  
  
    public void p (int i) {  
        System.out.println(i * 3);  
    }  
}  
  
class A extends B {  
    public void p ( ) {  
        System.out.println(i);  
    }  
}
```

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
- Thể hiện khi có sử dụng với **nạp đề hàm**

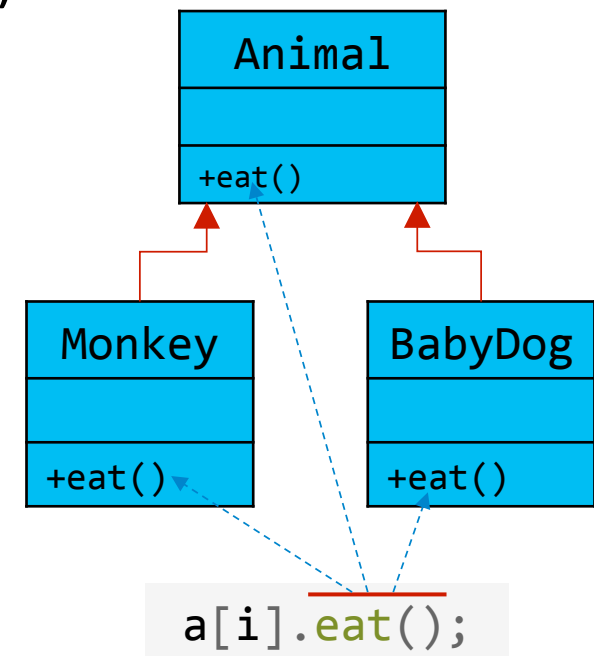
```
class Animal {  
    public void eat() {  
        System.out.println("Eating..."); }  
}  
  
class Monkey extends Animal {  
    public void eat() {  
        System.out.println("Eating fruits..."); }  
}  
  
class BabyDog extends Animal {  
    public void eat() {  
        System.out.println("Drinking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

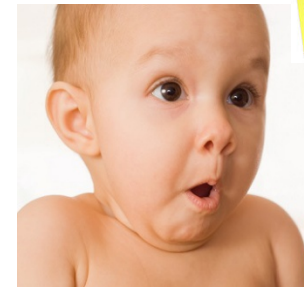
```
class Animal {  
    public static int count = 0 ;  
    public Animal() {  
        count++;  
    }  
}  
  
class Monkey extends Animal {  
    public static int count = 0 ;  
    public Monkey() {  
        count++;  
    }  
}  
  
class BabyDog extends Animal {  
    public static int count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

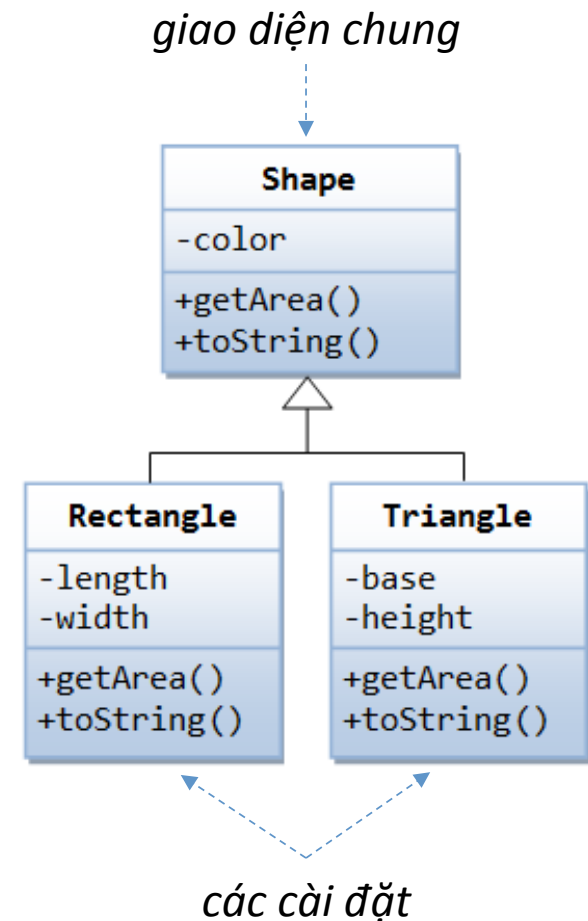
Ghi nhớ về tính đa hình

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

● ←

```
public String toString() {  
    return "color=\"" + color + "\"";  
}
```

```
public double getArea() {  
    System.out.println("Shape unknown! Cannot compute area!");  
    return -1;    // error  
}  
}
```

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

Lớp trừu tượng

- Lớp trừu tượng là một dạng lớp đặc biệt, trong đó các phương thức chỉ được khai báo ở dạng khuôn mẫu (template) mà không được cài đặt chi tiết.
- Là lớp **không thể** dùng để tạo đối tượng
- Thường được sử dụng trong thừa kế (là lớp cha cho các lớp khác)
- Lớp trừu tượng được khai báo như các lớp thông thường, ngoại trừ có thêm từ khoá **abstract**:

```
[public] abstract class <tên lớp>{ ... }
```

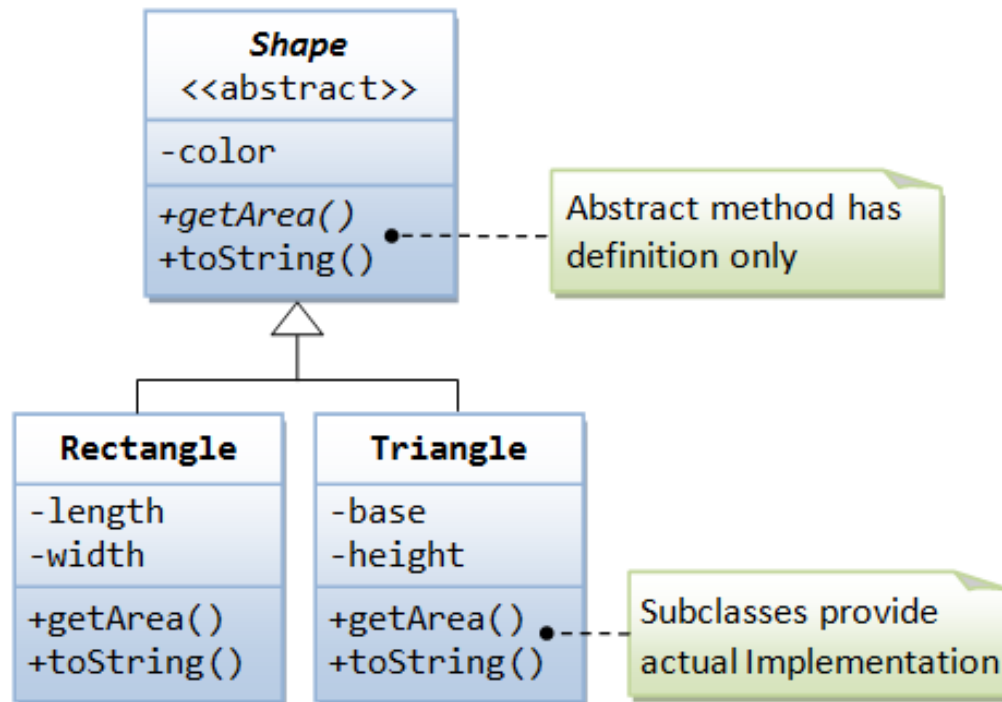
- Lớp trừu tượng cũng có thể kế thừa một lớp khác, nhưng lớp cha cũng phải là một lớp trừu tượng.

Phương thức trừu tượng

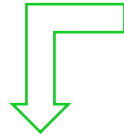
- `[public] abstract <kiểu dữ liệu trả về> <tên phương thức> ([<các tham số>]) [throws <các ngoại lệ>] ;`
- Nếu trong lớp có phương thức **abstract** thì lớp đó phải được khai báo là **abstract**.
- Lớp kế thừa từ lớp trừu tượng thì: **hoặc** chúng phải ghi đè tất cả các phương thức **abstract** của lớp cha, hoặc lớp đó phải là lớp **abstract**.
- **Không** thể tạo ra đối tượng của lớp trừu tượng.
- Lưu ý: các phương thức trừu tượng **bắt buộc phải được** tái định nghĩa trong các **lớp kế thừa** lớp trừu tượng đó.

Lớp trừu tượng

```
public abstract 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



```
public abstract class Shape {  
    double height;  
    double base;  
    public Shape(double a, double b){  
        height = a;  
        base = b;  
    }  
    public abstract double getArea();  
}
```



```
class Triangle extends Shape{  
    public Triangle(double a, double b) {  
        super(a,b);  
    }  
    public double getArea() {  
        return height * base;  
    }  
}
```

```
class Rectangle extends Shape{  
    public Rectangle(double a, double b) {  
        super(a,b);  
    }  
    public double getArea() {  
        return height * base;  
    }  
}
```

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

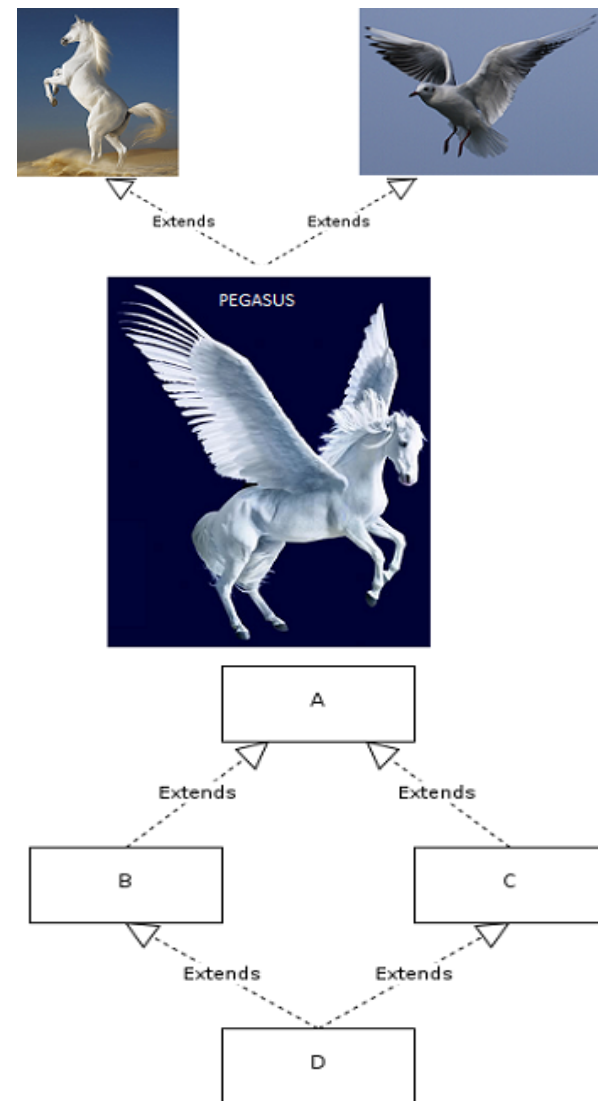
- Cung cấp một chuẩn (standard) cho việc phát triển ứng dụng
- Một phương thức trừu tượng không thể được khai báo `final` hay `private`
 - `Final`: không thể được nạp đè
 - `Private`: không thấy được bởi lớp con nên không thể nạp đè
- Nên lập trình **dựa vào giao diện**
 - Tạo tham chiếu thuộc lớp cha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

Packages và Đa thừa kế (multiple inheritance)

Đa thừa kế

- Đa thừa kế là:
 - Một lớp con thừa kế từ **nhều 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 **Interface**

```
//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>  
{  
    ...  
}
```

Packages

- Packages là một nhóm các lớp có liên quan với nhau sẽ được đặt cùng nhau trong một thư mục.
- Nó giúp cho chương trình có tổ chức hơn và cung cấp một mức nữa của tính bao gói.
- Tất cả các lớp của Java đều phụ thuộc vào 1 gói nào đó.
- Khi không khai báo một lớp thuộc một package nào đó thì default package được sử dụng.
- Để tạo một package thì sử dụng câu lệnh package khai báo ở ngay đầu source file Java. Cấu trúc khai báo như sau:

package pfname;
- Java sử dụng hệ thống file để quản lý các package, mỗi package sẽ ở trong thư mục riêng của nó. Ví dụ, các file *.class* của các lớp nằm trong package ***pfname*** sẽ phải được lưu trữ trong thư mục tên ***pfname***.

Packages

Source file tên là JavaApplication1.java sẽ được đặt trong thư mục có tên test

```
package test;  
public class JavaApplication1{  
    public static void main (String []args){  
        System.out.print ("Hello Java");  
    }  
}
```

- Khi muốn sử dụng các lớp của một package khác thì có thể dùng từ khóa import để sử dụng các lớp trực tiếp mà không phải thông qua các tên package.
- Cấu trúc câu lệnh import như sau: *import pkname.classname;*
- pkname là tên của package và classname là tên của lớp cần được import.
- Nếu muốn sử dụng tất cả các lớp của một package: *import pkname.*;*

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) **nhiều** giao diện:
 - Cài đặt tất cả các phương thức 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

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
- Cú pháp: thêm từ khóa **default** trước khai báo

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

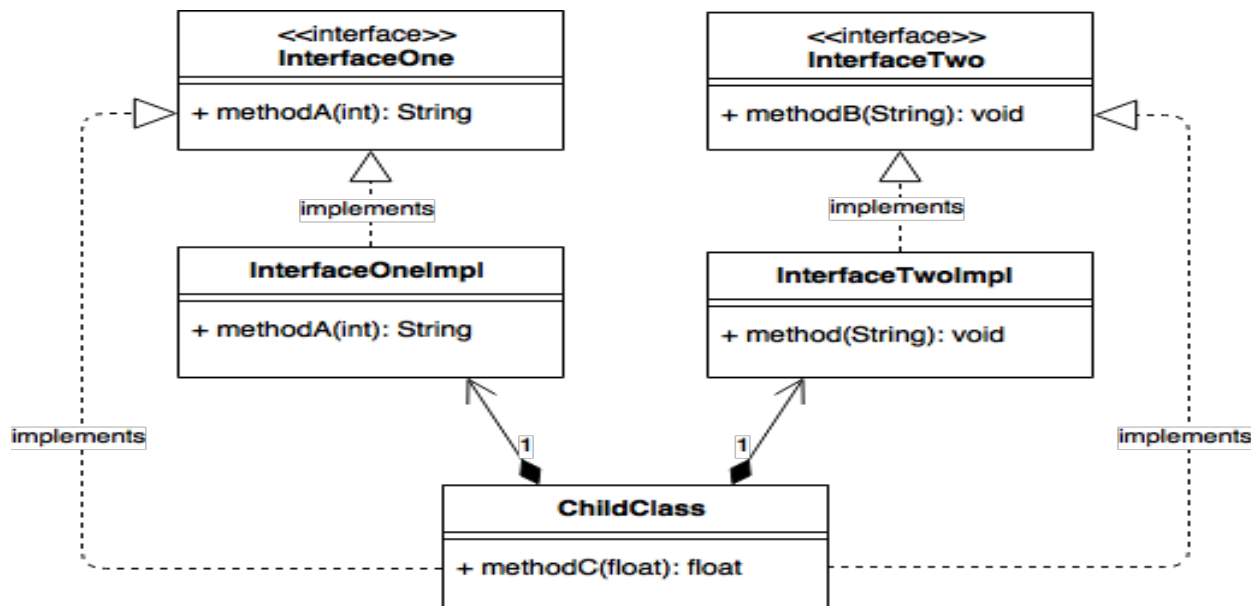
```
public class Button implements Clickable, Accessible {  
    public static void main(String[] args) {  
        Button button = new Button();  
        button.click();  
        button.access();  
    }  
}
```

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

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

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) {  
        //...  
    }  
}
```

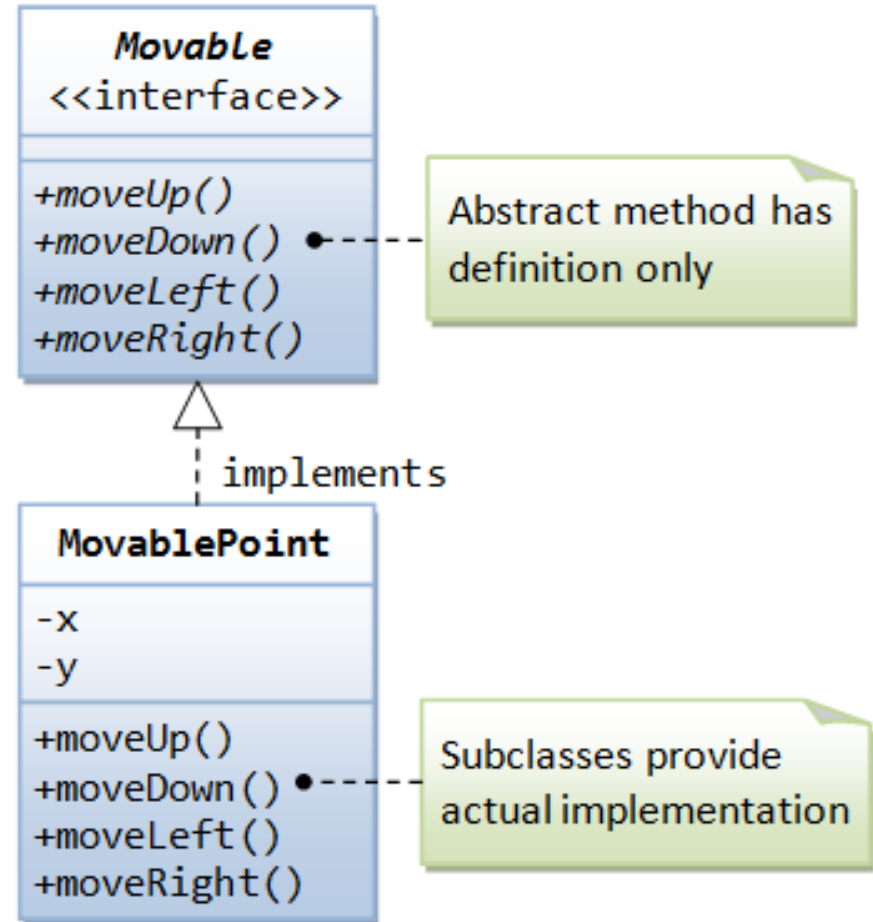
```
interface InterfaceOne {  
    String methodA(int a);  
}  
  
interface InterfaceTwo {  
    void methodB(String s);  
}
```


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);  
    }  
}
```

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);
    }
}
```

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 **<is-a>**
- 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 không thể 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 lớp
 - Như là một lớp hoàn toàn ảo: chỉ có khai báo phương thức, không có định nghĩa phương thức và các thuộc tính

Xử lý ngoại lệ

Xử lý ngoại lệ (exception handling)

- Ngoại lệ: là một lỗi xảy ra khi thực thi chương trình:
 - Ví dụ: chia cho 0, mở tập tin không tồn tại,...
- Khi một lỗi xảy ra, một **đối tượng** ngoại lệ sẽ sinh ra và chương trình phải “bắt” và xử lý ngoại lệ
- Bắt ngoại lệ: dùng câu lệnh `try...catch...finally`

Xử lý ngoại lệ (exception handling)

Xử lý ngoại lệ: có 3 hoạt động chính: đặc tả ngoại lệ, ném ra ngoại lệ và bắt ngoại lệ.

- Các ngoại lệ có thể phát sinh cần được mô tả chi tiết trong lệnh khai báo của phương thức, việc khai báo này được gọi là **đặc tả ngoại lệ**.
- Khi một câu lệnh trong phương thức gây lỗi, mà người lập trình không cung cấp mã xử lý lỗi, thì ngoại lệ được chuyển đến phương thức gọi phương thức đó, việc này được gọi là **ném ra ngoại lệ**.
- Java runtime sẽ tìm mã xử lý lỗi bằng cách lần ngược trở lại chuỗi các phương thức gọi nhau, bắt đầu từ phương thức hiện tại. Chương trình sẽ kết thúc nếu không tìm thấy mã xử lý ngoại lệ. Quá trình tìm kiếm này gọi là **bắt ngoại lệ**.

Xử lý ngoại lệ (exception handling)

Đặc tả ngoại lệ:

- Đặc tả ngoại lệ là khai báo cho trình biên dịch biết rằng phương thức này có thể gây ra ngoại lệ lúc thi hành.
- Để khai báo ngoại lệ ta sử dụng từ khoá throws trong khai báo phương thức, ví dụ:

public void myMethod() throws

IOException, RemoteException

- từ khoá throws chỉ cho trình biên dịch java biết rằng phương thức này có thể ném ra ngoại lệ `IOException` và `RemoteException`, nếu một phương thức ném ra nhiều ngoại lệ thì các ngoại lệ được khai báo cách nhau bởi dấu phẩy ‘,’.

Xử lý ngoại lệ (exception handling)

Ném ra ngoại lệ:

- Một phương thức sau khi đã khai báo các ngoại lệ, thì bạn (hoặc chương trình thực thi java) có thể ném ra các đối tượng ngoại lệ, có kiểu mà ta đã khai báo trong danh sách throws. Cú pháp của lệnh ném ra ngoại lệ:

throw *ExceptionObject*;

- Chú ý:
 - Chú ý giữa lệnh khai báo ngoại lệ và lệnh ném ra ngoại lệ

Xử lý ngoại lệ (exception handling)

Bắt ngoại lệ:

- Khi một ngoại lệ xảy ra, đối tượng tương ứng với ngoại lệ đó được tạo ra. Đối tượng này sau đó được truyền cho phương thức là nơi mà ngoại lệ xảy ra. Đối tượng này chứa thông tin chi tiết về ngoại lệ. Thông tin này có thể được nhận về và được xử lý.
- Vấn đề đối với người lập trình java là phải biết được đoạn mã nào của có thể gây ra lỗi. Khi đã khoanh vùng được đoạn mã có thể gây ra lỗi ta sẽ đặt đoạn mã có khả năng gây ra lỗi này trong khối *try*, và đặt đoạn mã xử lý lỗi trong khối *catch*.

Xử lý ngoại lệ (exception handling)

Bắt ngoại lệ:

```
try {  
    //các câu lệnh có khả năng  
    //sinh ra lỗi  
}  
catch (<loại ngoại lệ 1> e) {  
    //xử lý ngoại lệ 1  
}  
...  
catch (<loại ngoại lệ n> en) {  
    //xử lý ngoại lệ n  
}  
finally {  
    //các lệnh sẽ được thực hiện  
    //cả trong trường hợp có và  
    //không có ngoại lệ xảy ra  
}
```

• Lưu ý:

- Khối lệnh **finally** là không bắt buộc
- Các loại ngoại lệ trong Java được tổ chức theo dạng cây phân cấp với gốc cây phân cấp là **Exception**

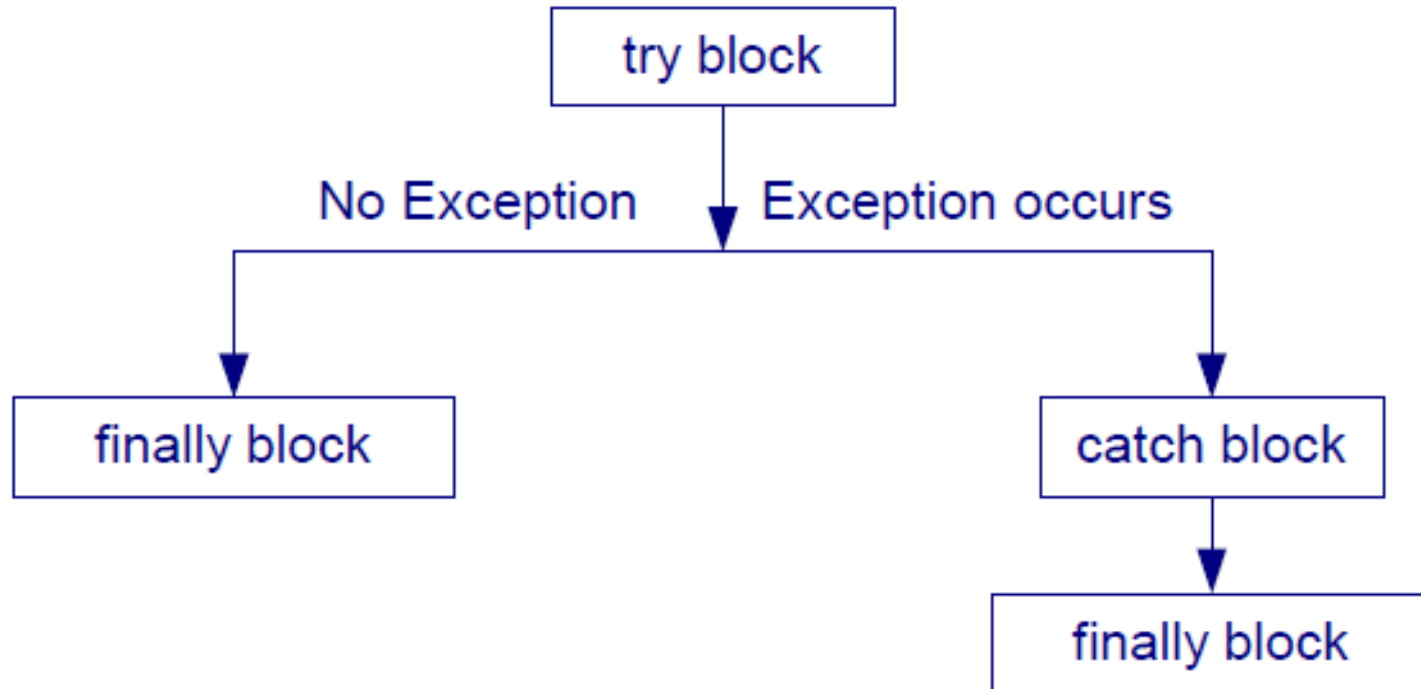
Xử lý ngoại lệ (exception handling)

Bắt ngoại lệ:

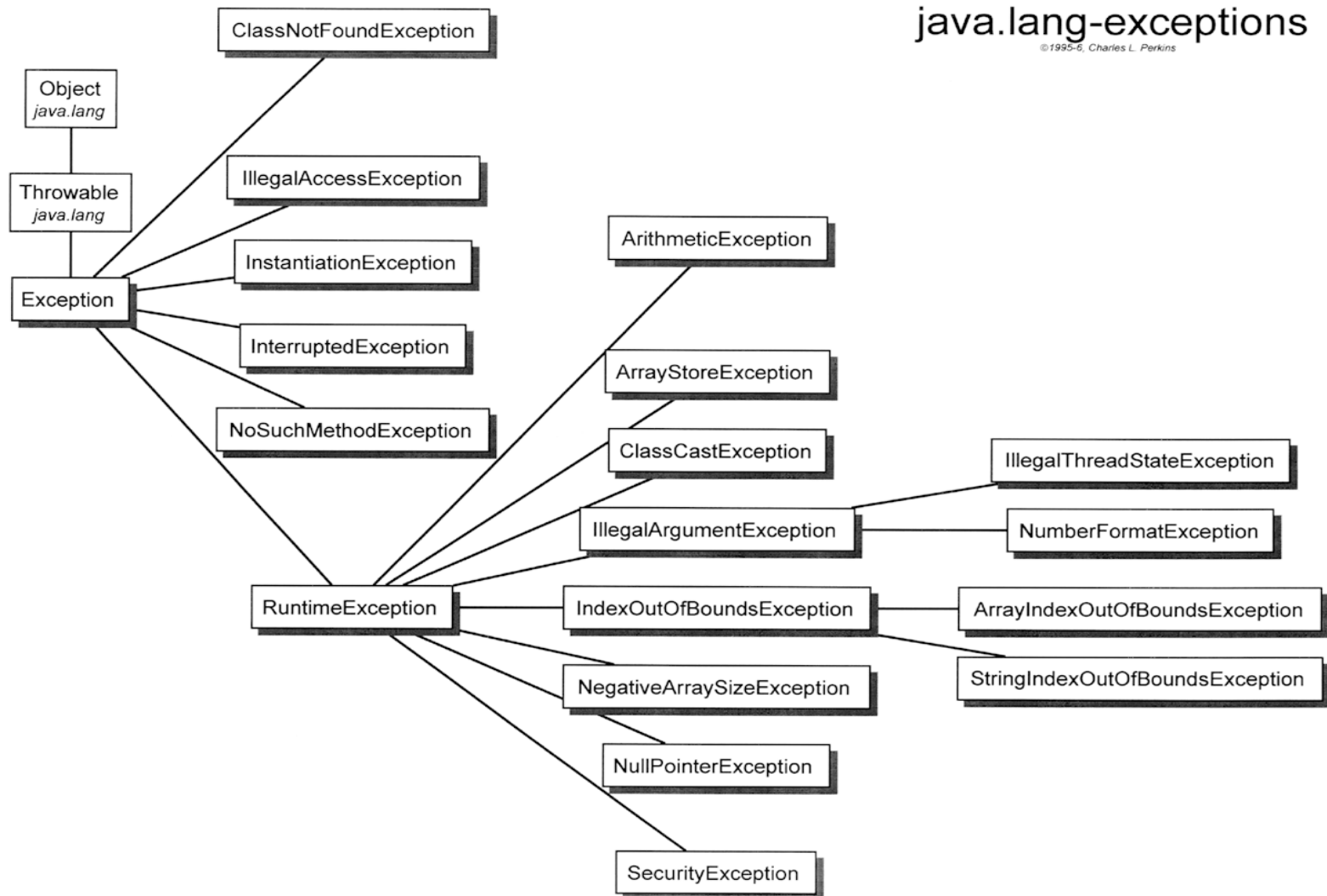
- Nếu không có một ngoại lệ nào phát sinh trong khối try thì các mệnh đề catch sẽ bị bỏ qua.
- Trong trường hợp một trong các câu lệnh bên trong khối try gây ra một ngoại lệ thì, thì java sẽ bỏ qua các câu lệnh còn lại trong khối try để đi tìm mã xử lý ngoại lệ.
 - Nếu kiểu ngoại lệ so khớp với kiểu ngoại lệ trong mệnh đề catch, thì mã lệnh trong khối catch đó sẽ được thực thi.
 - Nếu không tìm thấy một kiểu ngoại lệ nào được so khớp java sẽ kết thúc phương thức đó và chuyển ngoại lệ đó ra phương thức đã gọi phương thức này quá trình này được tiếp tục cho đến khi tìm thấy mã xử lý ngoại lệ.
 - Nếu không tìm thấy mã xử lý ngoại lệ trong chuỗi các phương thức gọi nhau, chương trình có thể chấm dứt và in thông báo lỗi ra luồng lỗi chuẩn System.err

Xử lý ngoại lệ (exception handling)

Finally:

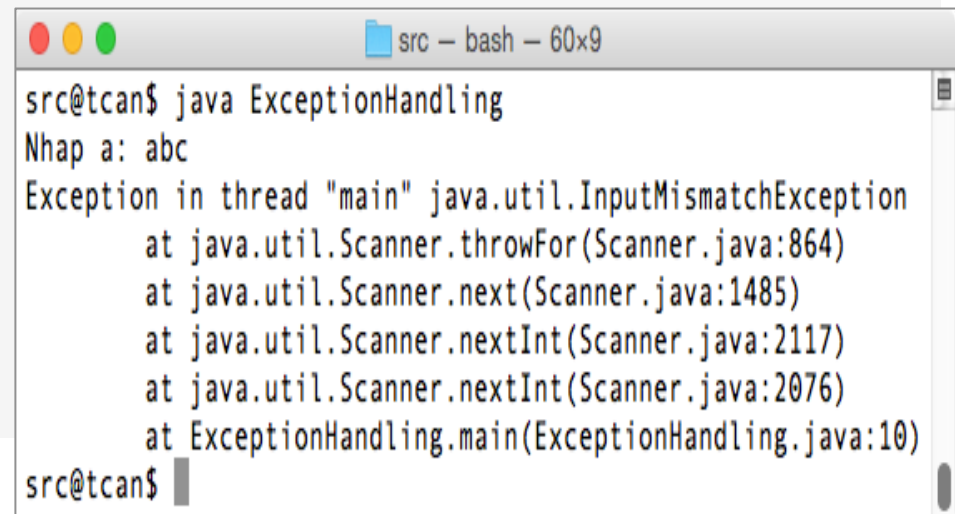


Xử lý ngoại lệ (exception handling)



Xử lý ngoại lệ (exception handling)

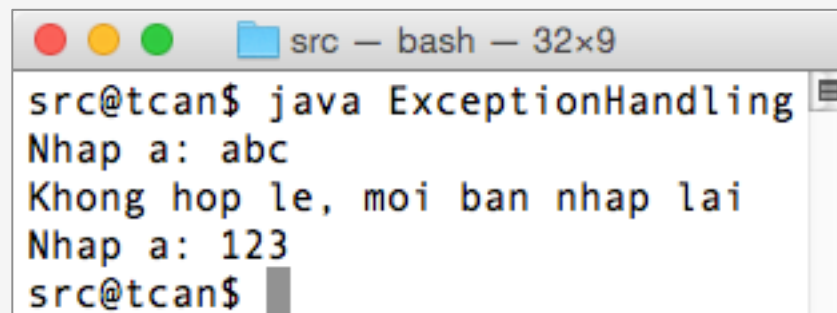
```
import java.util.Scanner;  
import java.util.InputMismatchException;  
  
class ExceptionHandling {  
    public static void main(String args[]) {  
        int a, b;  
        Scanner s = new Scanner(System.in);  
  
        System.out.print("Nhập a: ");  
        a = s.nextInt();  
        s.nextLine();  
  
        System.out.print("Nhập b: ");  
        b = s.nextInt();  
        s.nextLine();  
  
        //. . .  
    }  
}
```



```
src@tcan$ java ExceptionHandling  
Nhập a: abc  
Exception in thread "main" java.util.InputMismatchException  
    at java.util.Scanner.throwFor(Scanner.java:864)  
    at java.util.Scanner.next(Scanner.java:1485)  
    at java.util.Scanner.nextInt(Scanner.java:2117)  
    at java.util.Scanner.nextInt(Scanner.java:2076)  
    at ExceptionHandling.main(ExceptionHandling.java:10)  
src@tcan$
```


Xử lý ngoại lệ (exception handling)

```
import java.util.Scanner;
import java.util.InputMismatchException;
class ExceptionHandling {
    public static void main(String args[]) {
        int a, b;
        Scanner s = new Scanner(System.in);
        while (true) {
            try {
                System.out.print("Nhập a: ");
                a = s.nextInt();
            }
            catch (InputMismatchException e) {
                System.out.println("Không hợp lệ, mời bạn nhập lại");
                continue;
            }
            finally {
                s.nextLine();
            }
            break;
        } //while
        //tiếp tục nhập b, ...
    } //main
}
```



A terminal window titled "src — bash — 32x9" showing the execution of the Java program. The prompt is "src@tcan\$". The user enters "java ExceptionHandling". The program outputs "Nhập a: abc". The user enters "abc". The program outputs "Không hợp lệ, mời bạn nhập lại". The user enters "Nhập a: 123". The user enters "123". The prompt returns to "src@tcan\$".

```
src@tcan$ java ExceptionHandling
Nhập a: abc
Không hợp lệ, mời bạn nhập lại
Nhập a: 123
src@tcan$
```

Xử lý ngoại lệ (exception handling)

```
class ExcDemo3 {  
    public static void main (String args[]) {  
        int numer[] = { 4, 8, 16, 32, 64, 128 };  
        int denom[] = { 2, 0, 4, 4, 0, 8 };  
        for (int i=0; i<denom.length; i++) {  
            try {  
                System.out.println (numer[i] + " / " + denom[i] + " is "  
                    + numer[i]/denom[i]);  
            }  
            catch (ArithmeticException exc) {  
                System.out.println ("Can't divide by Zero!");  
            }  
        }  
    }  
}
```

4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16

Xử lý ngoại lệ (exception handling)

Ví dụ:

```
import java.lang.ArithmeticException;

public class TryClass{
    public static void main(String args[]) {
        int n=0;
        try{
            System.out.println(1/n);
        }
        catch(ArithmeticException ex){
            System.out.println("Loi chia cho 0");
        }
    }
}
```

Xử lý ngoại lệ (exception handling)

```
class ExcDemo1 {  
    public static void main (String args[]) {  
        int nums[] = new int[4];  
        try {  
            System.out.println ("Before exception is generated.");  
            // Generate an index out-of-bounds exception.  
            nums[7] = 10;  
            System.out.println ("this won't be displayed");  
        }  
        catch (ArrayIndexOutOfBoundsException exc) {  
            // catch the exception  
            System.out.println ("Index out-of-bounds!");  
        }  
        System.out.println ("After catch statement.");  
    }  
}
```

Before exception is generated.
Index out-of-bounds!
After catch statement.

Xử lý ngoại lệ (exception handling)

Before exception is generated.

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
at ExcTest.genException() (ExcDemo2.java:6)
at ExcDemo2.main (ExcDemo2.java:13)

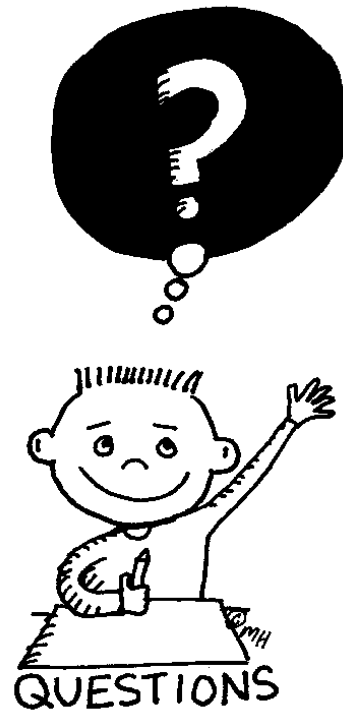
```
public class ExcDemo2 {  
    public static void main (String args[]) {  
        try {  
            ExcTest.genException();    }  
        catch (ArrayIndexOutOfBoundsException exc) {  
            System.out.println ("Index out-of-bounds!");    }  
            System.out.println ("After catch statement.");  
        }  
    }  
}
```

Xử lý ngoại lệ (exception handling)

- Có thể sử dụng nhiều hơn một mệnh đề catch với một mệnh đề try tuy nhiên mỗi catch phải bắt một loại khác nhau của exception.

```
class ExcDemo3 {
    public static void main (String args[]) {
        int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512};
        int denom[] = { 2, 0, 4, 4, 0, 8 };
        for (int i=0; i<numer.length; i++) {
            try {
                System.out.println (numer[i] + " / " + denom[i] + " is "
                    + numer[i]/denom[i]);
            }
            catch (ArithmeticException exc) {
                System.out.println ("Can't divide by Zero!");
            }
            catch (ArrayIndexOutOfBoundsException exc) {
                System.out.println ("No matching element found.");
            }
        }
    }
}
```

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.
```



Question?

CT176 – LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG