

Môn: Lập trình Hướng đối tượng (Object Oriented Programming)

Chương 4. Kết tập và Kế thừa



Mục tiêu bài học

- ✓ Giải thích về khái niệm tái sử dụng mã nguồn
- ✓ Chỉ ra được bản chất, mô tả các khái niệm liên quan đến đến kết tập và kế thừa
- ✓ So sánh kết tập và kế thừa
- ✓ Biểu diễn được kết tập và kế thừa trên UML
- ✓ Giải thích nguyên lý kế thừa và thứ tự khởi tạo, hủy bỏ đối tượng trong kế thừa
- ✓ Áp dụng các kỹ thuật, nguyên lý về kết tập và kế thừa trên ngôn ngữ lập trình Java



Nội dung

1. Tái sử dụng mã nguồn
2. Kết tập (Aggregation)
3. Kế thừa (Inheritance)
4. Ví dụ và bài tập



Nội dung

1. **Tái sử dụng mã nguồn**
2. Kết tập (Aggregation)
3. Kế thừa (Inheritance)
4. Ví dụ và bài tập

1. Tái sử dụng mã nguồn (Re-usability)

- ▣ Tái sử dụng mã nguồn: Sử dụng lại các mã nguồn đã viết
 - ▣ Lập trình cấu trúc: Tái sử dụng hàm/chương trình con
 - ▣ OOP: Khi mô hình thế giới thực, tồn tại nhiều loại đối tượng có các thuộc tính và hành vi tương tự hoặc liên quan đến nhau
 - ▣ Làm thế nào để tái sử dụng lớp đã viết?





1. Tái sử dụng mã nguồn (2)

- ▣ Các cách sử dụng lại lớp đã có:
 - ▣ *Sao chép* lớp cũ thành 1 lớp khác => Dư thừa và khó quản lý khi có thay đổi
 - ▣ Tạo ra lớp mới là sự *tập hợp* hoặc *sử dụng các đối tượng* của lớp cũ đã có => Kết tập (Aggregation)
 - ▣ Tạo ra lớp mới trên cơ sở *phát triển* từ lớp cũ đã có => Kế thừa (Inheritance)

Ưu điểm của tái sử dụng mã nguồn

- ▣ Giảm thiểu công sức, chi phí
- ▣ Nâng cao chất lượng phần mềm
- ▣ Nâng cao khả năng mô hình hóa thế giới thực
- ▣ Nâng cao khả năng bảo trì(maintainability)





Nội dung

1. Tái sử dụng mã nguồn
2. **Kết tập (Aggregation)**
3. Kế thừa (Inheritance)
4. Ví dụ và bài tập

2. Kết tập

✓ Ví dụ:

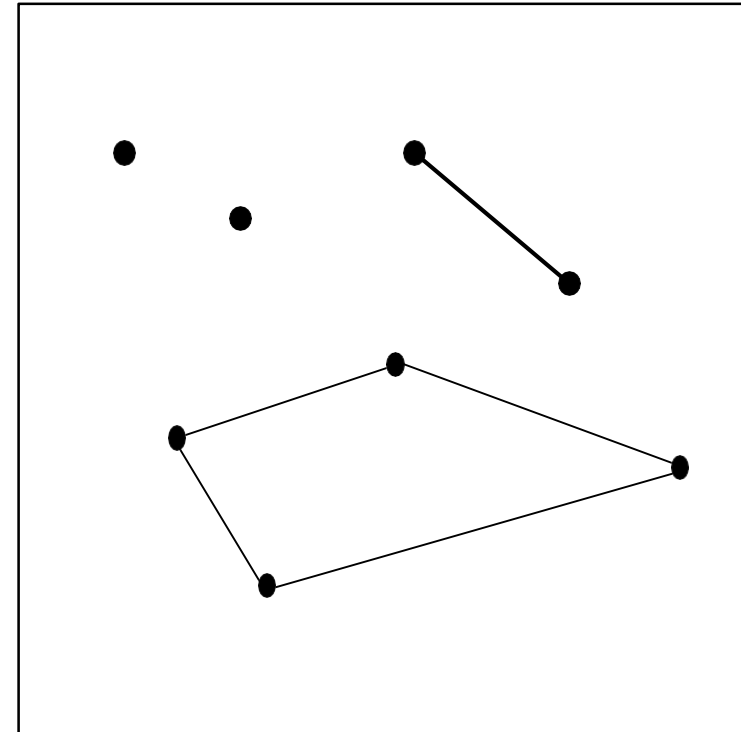
✓ Điểm

✓ Tứ giác gồm 4 điểm

⑦ Kết tập

✓ Kết tập

✓ Quan hệ chứa/có ("has-a") hoặc là một phần (is-a-part-of)



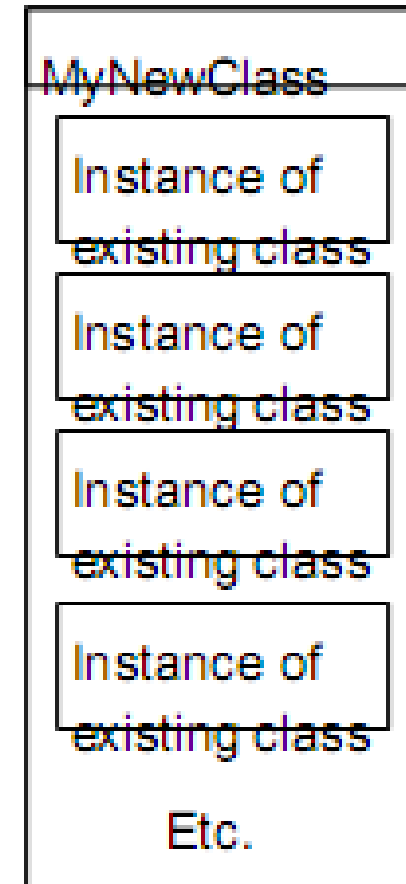


2.1. Bản chất của kết tập

- Kết tập (aggregation)
 - Tạo ra các đối tượng của các lớp có sẵn trong lớp mới □ thành viên của lớp mới.
 - Kết tập tái sử dụng thông qua *đối tượng*
- Lớp mới
 - Lớp toàn thể (Aggregate/Whole),
- Lớp cũ
 - Lớp thành phần (Part).

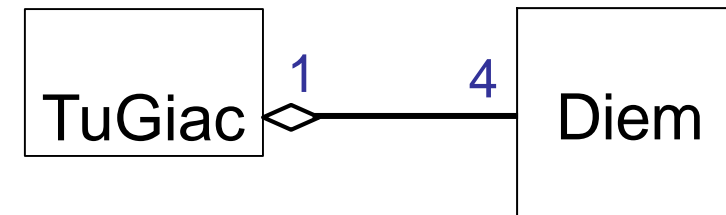
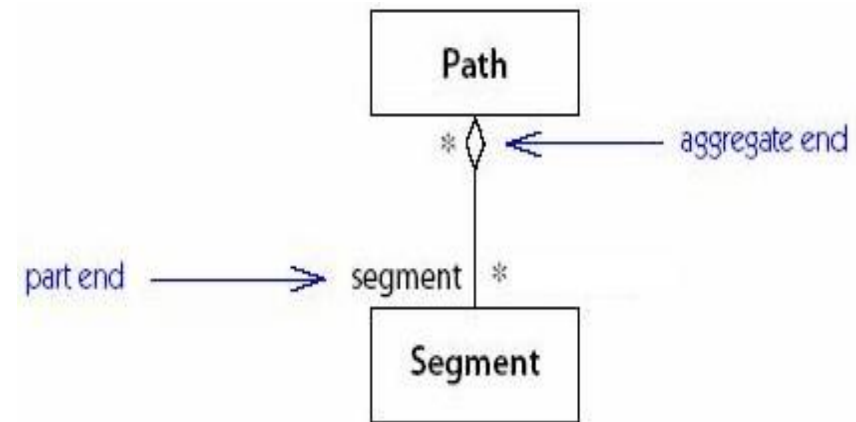
2.1. Bản chất của kết tập (2)

- Lớp toàn thể chứa đối tượng của lớp thành phần
 - Là một phần (is-a-part of) của lớp toàn thể
 - Tái sử dụng các thành phần dữ liệu và các hành vi của lớp thành phần thông qua đối tượng thành phần

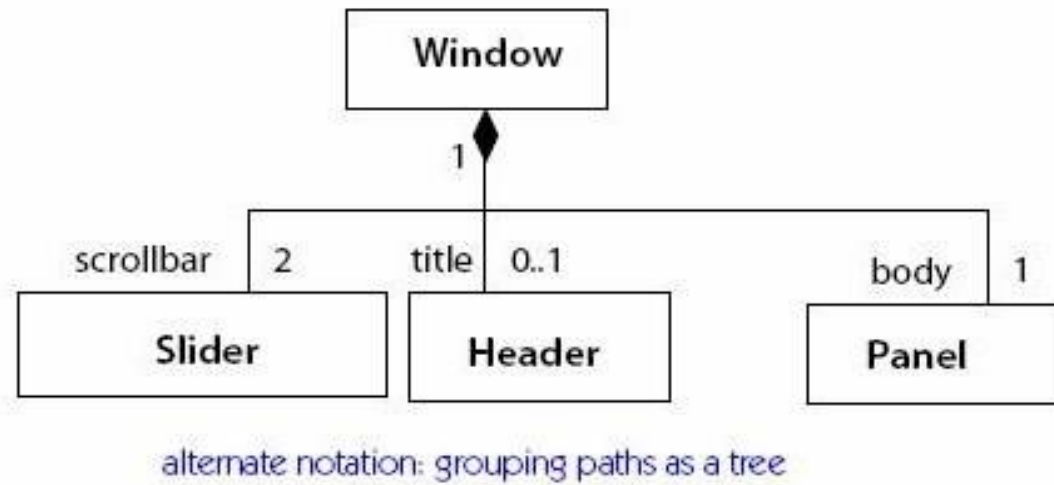
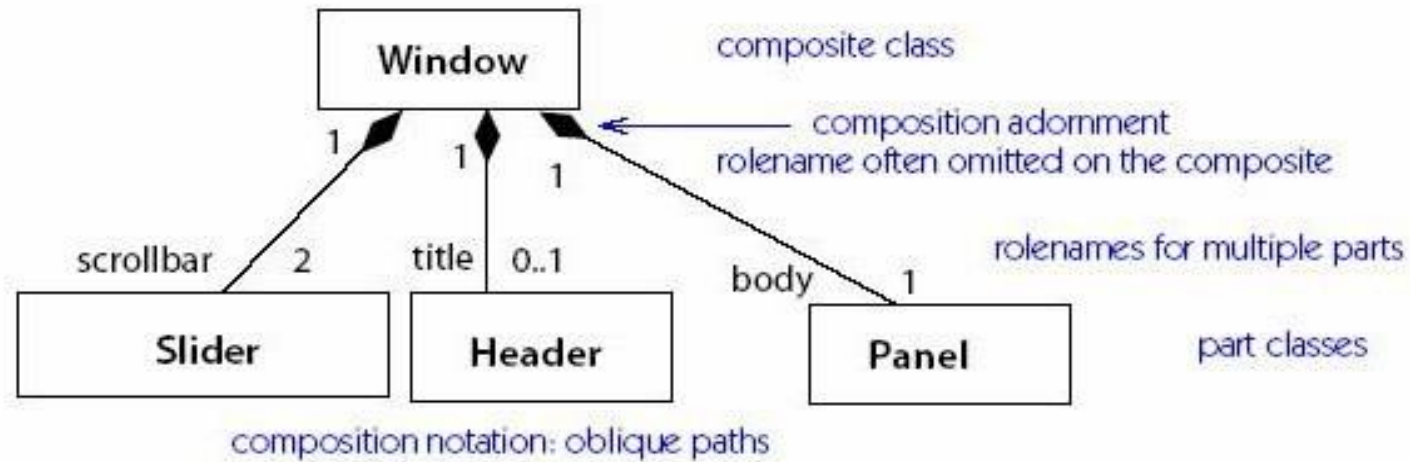


2.2. Biểu diễn kết tập bằng UML

- Sử dụng "hình thoi" tại đầu của lớp toàn thể
- Sử dụng bội số quan hệ (multiplicity) tại 2 đầu
 - 1 số nguyên dương: 1, 2,...
 - Dải số (0..1, 2..4)
 - *: Bất kỳ số nào
 - Không có: Mặc định là 1
- Tên vai trò (rolename)
 - Nếu không có thì mặc định là tên của lớp (bỏ viết hoa chữ cái đầu)



Ví dụ





2.3. Minh họa trên Java

```
class Diem {  
    private int x, y;  
    public Diem(){}  
    public Diem(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 printDiem(){  
        System.out.print("(" + x + ", " + y + ")");  
    }  
}
```

```
class TuGiac {
```

```
    private Diem d1, d2;
```

```
    private Diem d3, d4;
```

```
    public TuGiac(Diem p1, Diem p2,  
                  Diem p3, Diem p4) {
```

```
        d1 = p1; d2 = p2; d3 = p3; d4 = p4;
```

```
    }
```

```
    public TuGiac() {
```

```
        d1 = new Diem();      d2 = new Diem(0,1);
```

```
        d3 = new Diem (1,1); d4 = new Diem (1,0);
```

```
    }
```

```
    public void printTuGiac() {
```

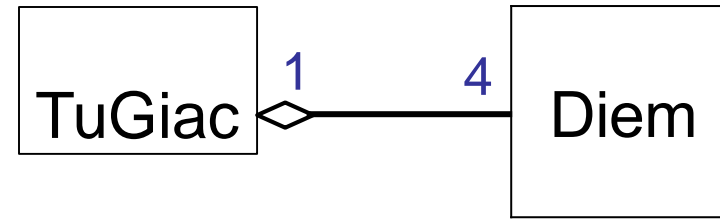
```
        d1.printDiem(); d2.printDiem();
```

```
        d3.printDiem(); d4.printDiem();
```

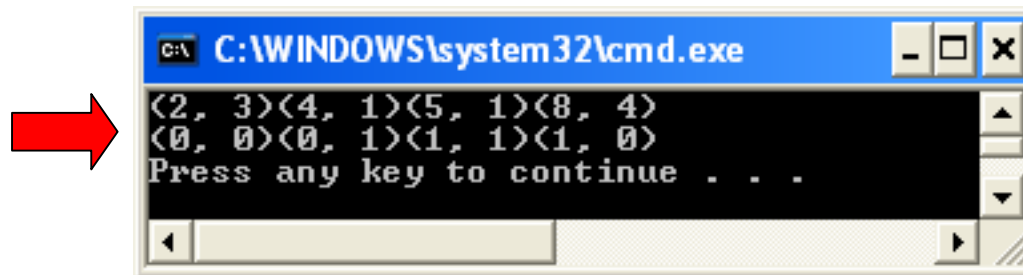
```
        System.out.println();
```

```
    }
```

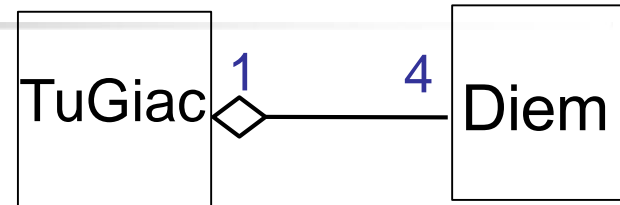
```
}
```



```
public class Test {  
    public static void main(String arg[])  
    {  
        Diem d1 = new Diem(2,3);  
        Diem d2 = new Diem(4,1);  
        Diem d3 = new Diem (5,1);  
        Diem d4 = new Diem (8,4);  
  
        TuGiac tg1 = new TuGiac(d1, d2, d3, d4);  
        TuGiac tg2 = new TuGiac();  
        tg1.printTuGiac();  
        tg2.printTuGiac();  
    }  
}
```



Cách cài đặt khác



```
class TuGiac {
    private Diem[] diem = new Diem[4];
    public TuGiac(Diem p1, Diem p2,
                  Diem p3, Diem p4) {
        diem[0] = p1; diem[1] = p2;
        diem[2] = p3; diem[3] = p4;
    }
    public void printTuGiac() {
        diem[0].printDiem(); diem[1].printDiem();
        diem[2].printDiem(); diem[3].printDiem();
        System.out.println();
    }
}
```



2.4. Thứ tự khởi tạo trong kết tập

- Khi một đối tượng được tạo mới, các thuộc tính của đối tượng đó đều phải được khởi tạo và gán những giá trị tương ứng.
- Các đối tượng thành phần được khởi tạo trước
 - Các phương thức khởi tạo của các lớp của các đối tượng thành phần được thực hiện trước



Nội dung

1. Tái sử dụng mã nguồn
2. Kết tập (Aggregation)
3. **Kế thừa (Inheritance)**
4. Ví dụ và bài tập

3.1. Tổng quan về kế thừa

✓ Ví dụ:

✓ Điểm

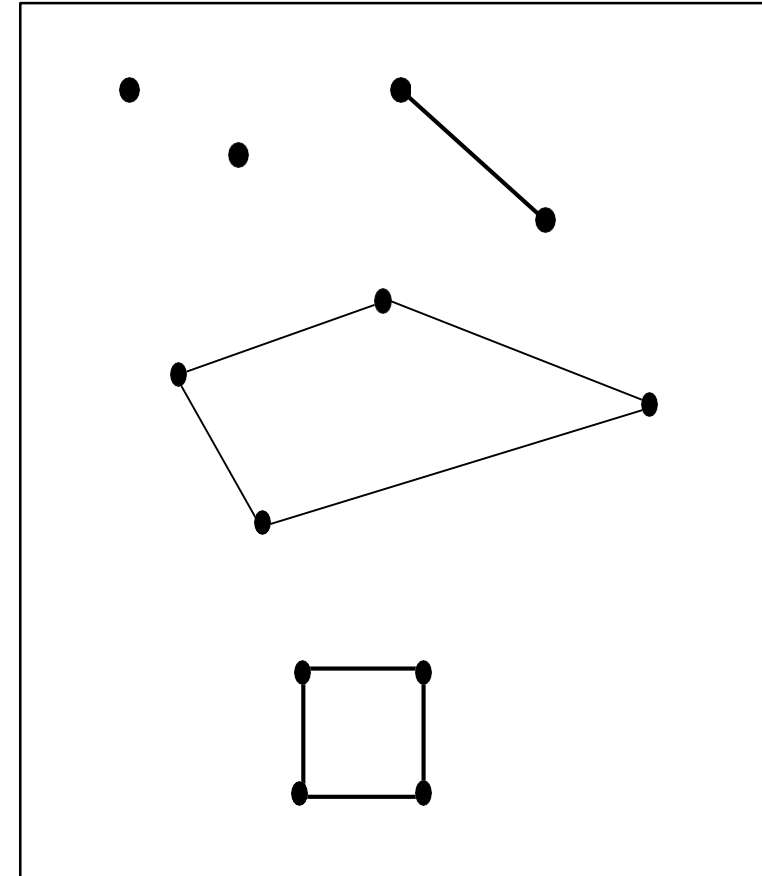
✓ Tứ giác gồm 4 điểm

=> Kết tập

✓ Tứ giác

✓ Hình vuông

=> Kế thừa





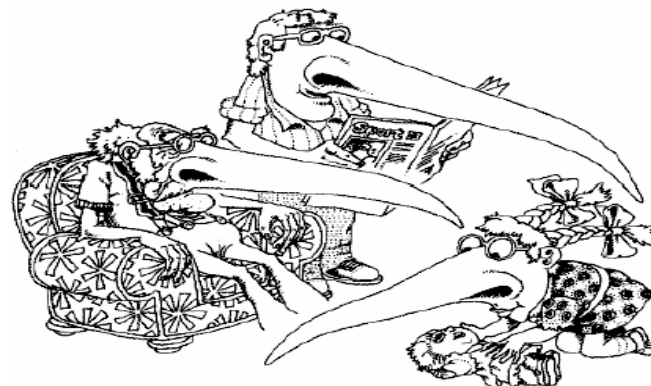
3.1.1. Bản chất kế thừa

- Kế thừa (Inherit, Derive)
 - ▣ Tạo lớp mới bằng cách phát triển lớp đã có.
 - ▣ Lớp mới kế thừa những gì đã có trong lớp cũ và phát triển những tính năng mới.
- Lớp cũ:
 - ▣ Lớp cha (parent, superclass), lớp cơ sở (base class)
- Lớp mới:
 - ▣ Lớp con (child, subclass), lớp dẫn xuất (derived class)

3.1.1. Bản chất kế thừa (2)

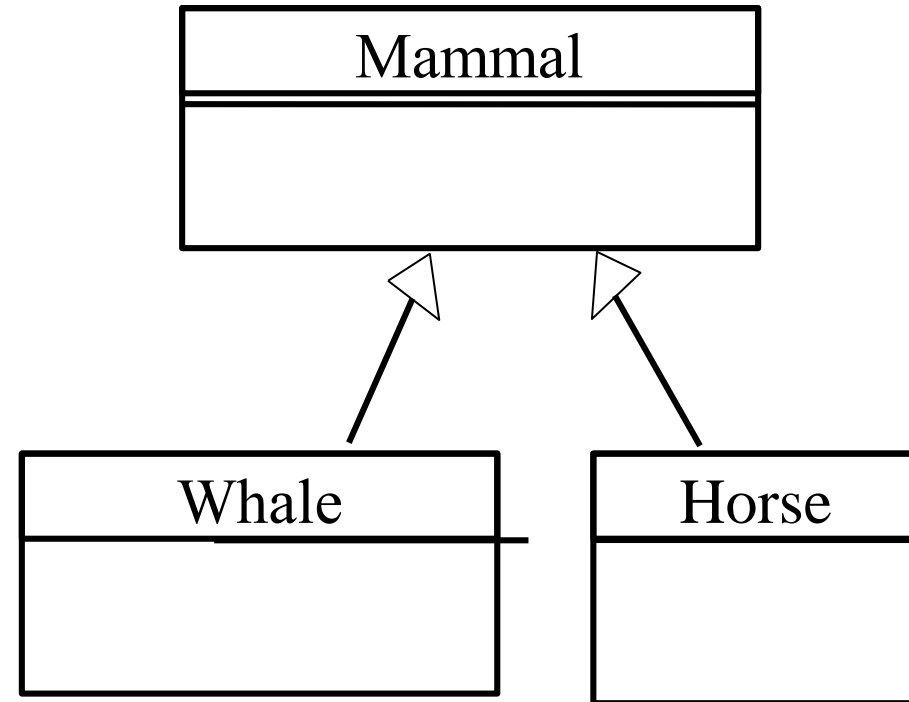
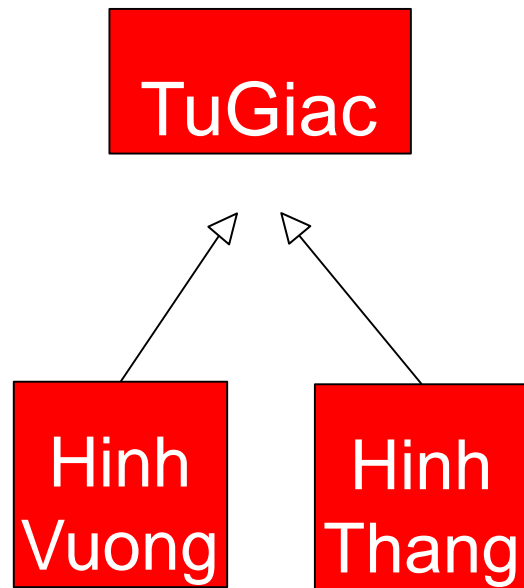
- Lớp con

- Là một loại (is-a-kind-of) của lớp cha
- Tái sử dụng bằng cách kế thừa các thành phần dữ liệu và các hành vi của lớp cha
- Chi tiết hóa cho phù hợp với mục đích sử dụng mới
 - Extension: Thêm các thuộc tính/hành vi mới
 - Redefinition (Method Overriding): Chỉnh sửa lại các hành vi kế thừa từ lớp cha



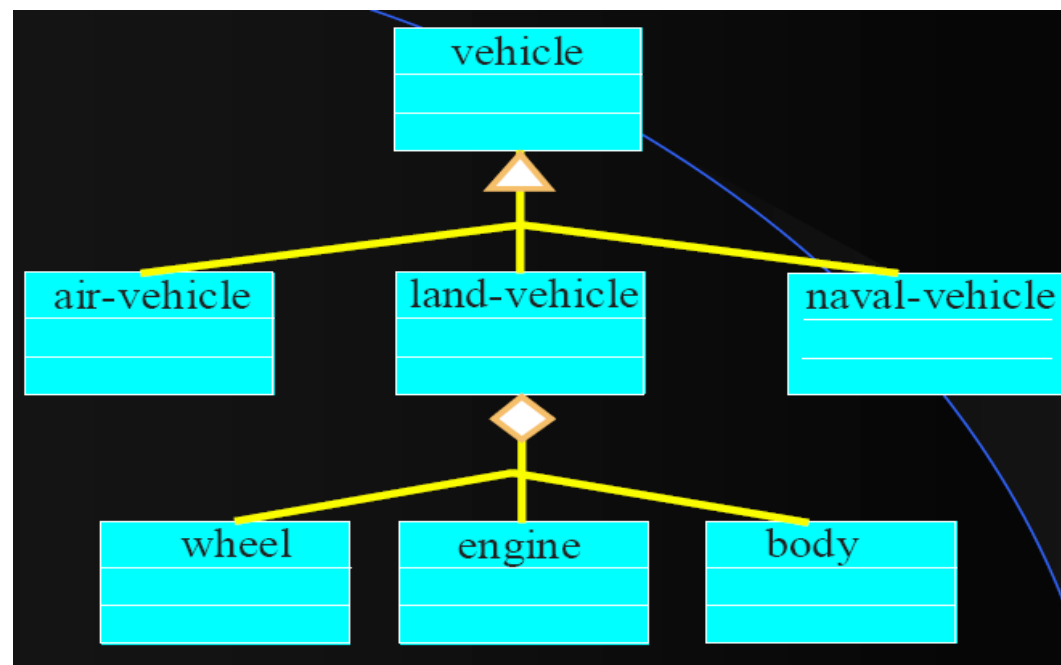
3.1.2. Biểu diễn kế thừa trong UML

- Sử dụng "tam giác rỗng" tại đầu Lớp cha



3.1.3. Kết tập và kế thừa

- So sánh kết tập và kế thừa?
 - ▣ Giống nhau
 - Đều là kỹ thuật trong OOP để tái sử dụng mã nguồn
 - ▣ Khác nhau?



Phân biệt kế thừa và kết tập

Kế thừa

- Kế thừa **tái sử dụng** thông qua **lớp**.
 - Tạo lớp mới bằng cách phát triển lớp đã có
 - Lớp con kế thừa dữ liệu và hành vi của lớp cha
- ✓ Quan hệ **"là một loại"** ("is a kind of")
- ✓ Ví dụ: Ô tô là một loại phương tiện vận tải

Kết tập

- ✓ Kết tập **tái sử dụng** thông qua **đối tượng**.
 - Tạo ra lớp mới là tập hợp các đối tượng của các lớp đã có
 - Lớp toàn thể có thể sử dụng dữ liệu và hành vi thông qua các đối tượng thành phần
- Quan hệ **"là một phần"** ("is a part of")
- Ví dụ: Bánh xe là một phần của Ô tô

3.1.4. Cây phân cấp kế thừa

(Inheritance hierarchy)

- Cấu trúc phân cấp hình cây, biểu diễn mối quan hệ kế thừa giữa các lớp.

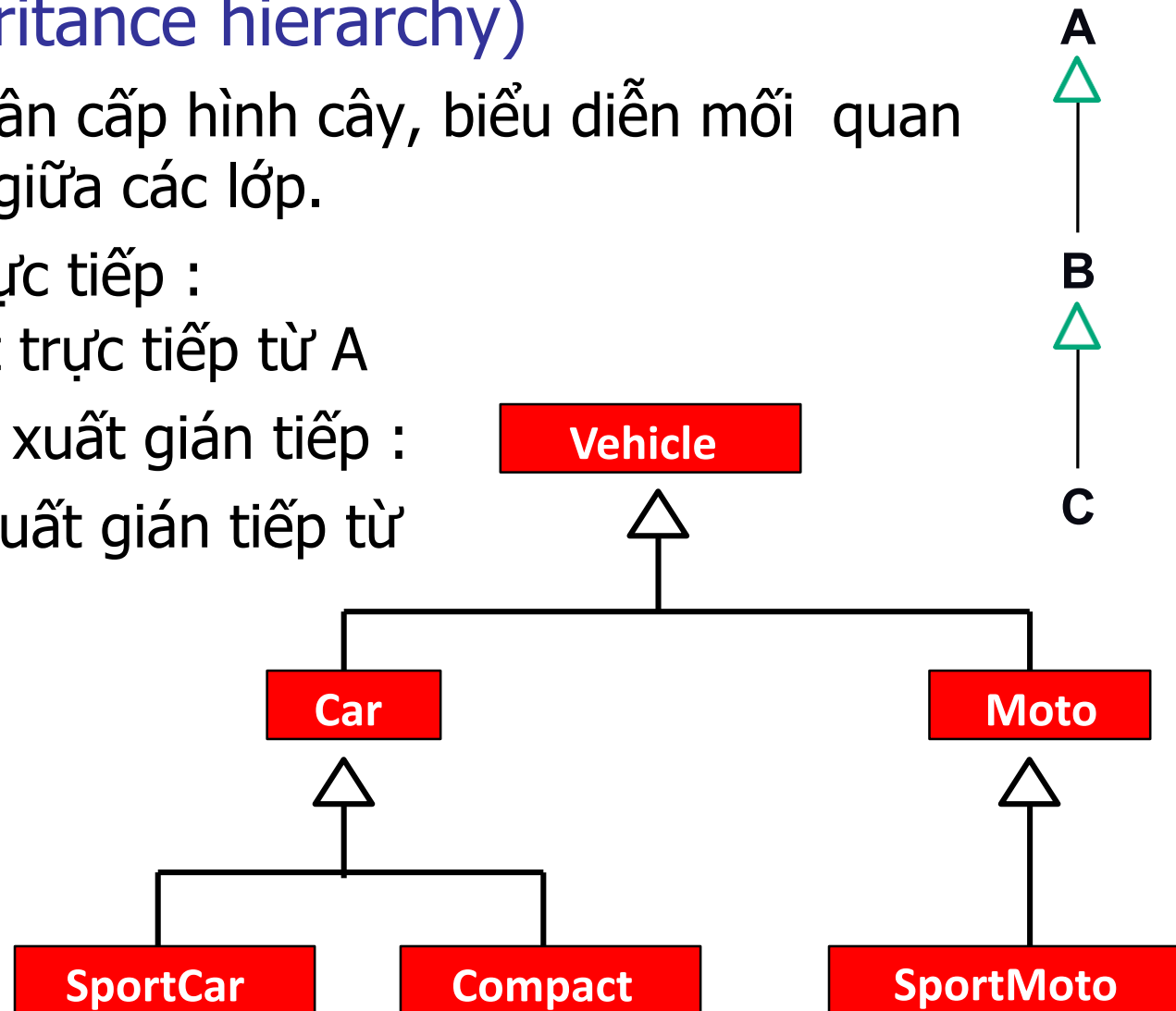
Dẫn xuất trực tiếp :

B dẫn xuất trực tiếp từ A

Dẫn xuất gián tiếp :

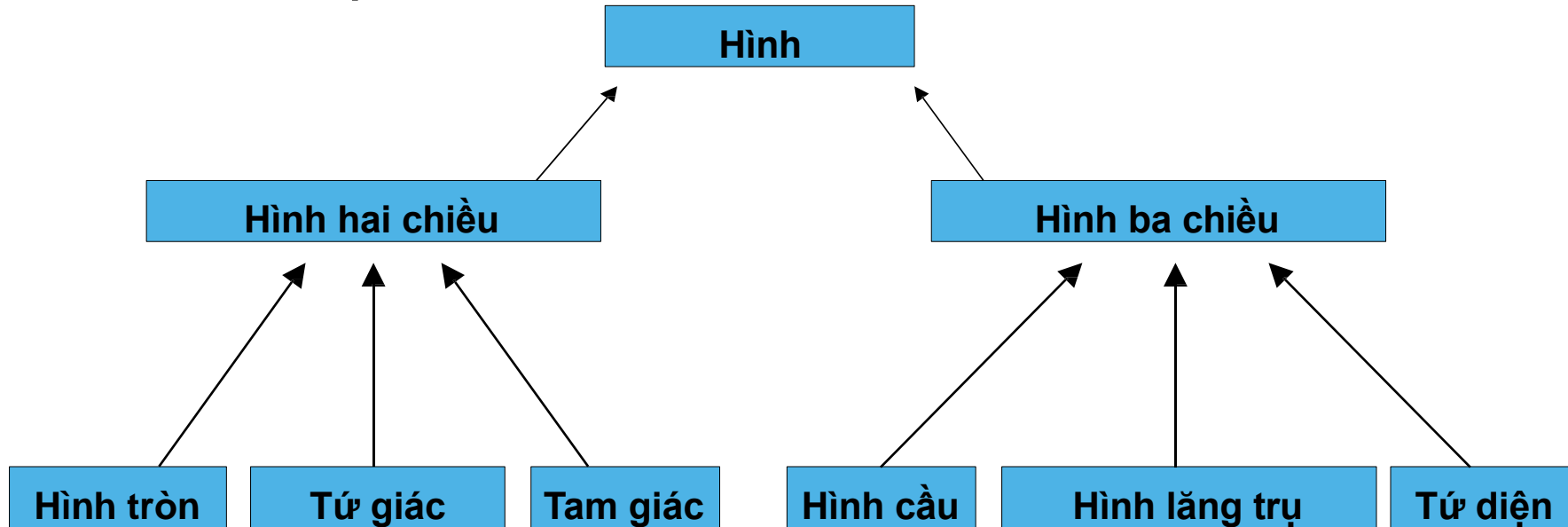
C dẫn xuất gián tiếp từ

A



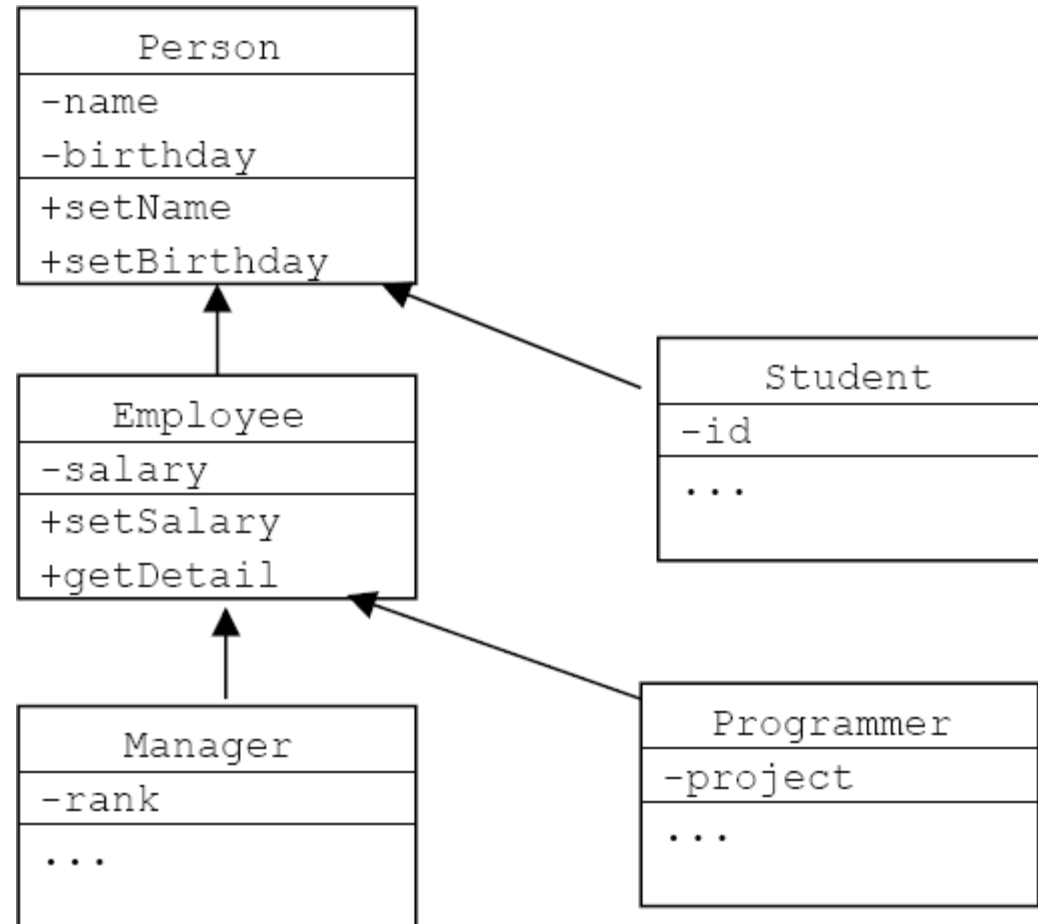
3.1.4. Cây phân cấp kế thừa (2)

- ▢ Các lớp con có cùng lớp cha gọi là anh chị em (siblings)
- ▢ Thành viên được kế thừa sẽ được kế thừa xuống dưới trong cây phân cấp ▢ Lớp con kế thừa tất cả các lớp tổ tiên của nó.



3.1.4. Cây phân cấp kế thừa (2)

Mọi lớp
đều kế thừa từ
lớp gốc Object



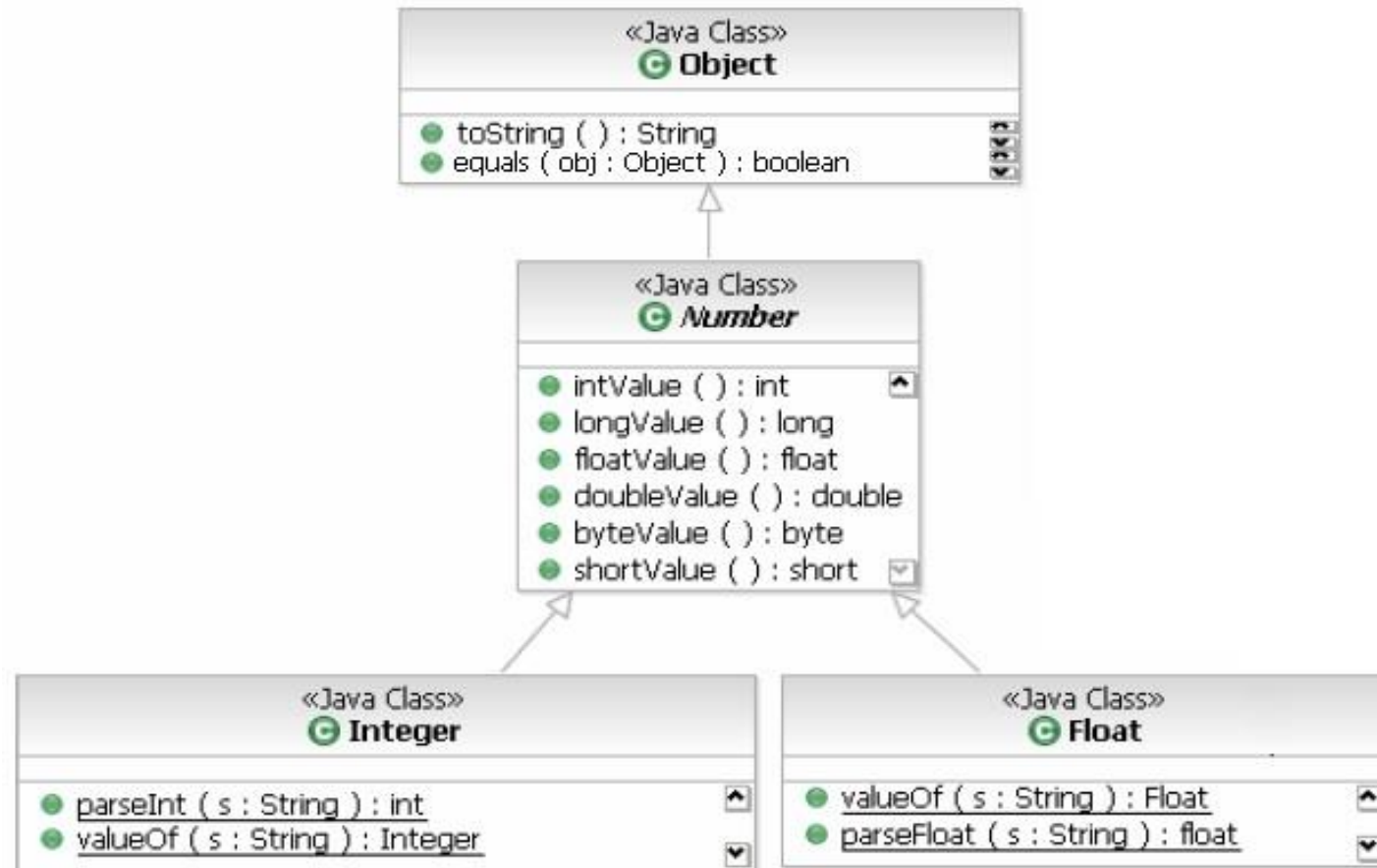


Lớp Object

- ▣ Trong gói java.lang
- ▣ Nếu một lớp không được định nghĩa là lớp con của một lớp khác thì mặc định nó là lớp con trực tiếp của lớp Object.
- ▣ Lớp Object là lớp gốc trên cùng của tất cả các cây phân cấp kế thừa

Lớp Object (2)

- Chứa một số phương thức hữu ích kế thừa lại cho tất cả các lớp, ví dụ: toString(), equals()...





3.2. Nguyên lý kế thừa

- Chỉ định truy cập protected
- Thành viên protected trong lớp cha được truy cập trong:
 - Các thành viên lớp cha
 - **Các thành viên lớp con**
 - Các thành viên các lớp cùng thuộc 1 package với lớp cha
- Lớp con có thể kế thừa được gì?
 - Kế thừa được các thành viên được khai báo là public và protected của lớp cha.
 - Không kế thừa được các thành viên private.
 - Các thành viên có chỉ định truy cập mặc định nếu lớp cha cùng gói với lớp con

3.2. Nguyên lý kế thừa (2)

	public	Không có	protected	private
Cùng lớp cha				
Lớp con cùng gói				
Lớp con khác gói				
Khác gói, non-inher				

3.2. Nguyên lý kế thừa (2)

	public	Không có	protected	private
Cùng lớp cha	Yes	Yes	Yes	Yes
Lớp con cùng gói	Yes	Yes	Yes	No
Lớp con khác gói	Yes	No	Yes	No
Khác gói, non-inher	Yes	No	No	No



3.2. Nguyên lý kế thừa (3)

- ▢ Các trường hợp không được phép kế thừa:
 - ▢ Các phương thức khởi tạo và hủy
 - ▢ Làm nhiệm vụ khởi đầu và gỡ bỏ các đối tượng
 - ▢ Chúng chỉ biết cách làm việc với từng lớp cụ thể
 - ▢ Toán tử gán =
 - ▢ Làm nhiệm vụ giống như phương thức khởi tạo



3.3. Cú pháp kế thừa trên Java

- Cú pháp kế thừa trên Java:
 - `<Lớp con> extends <Lớp cha>`
- Lớp cha nếu được định nghĩa là `final` thì không thể có lớp dẫn xuất từ nó.
- Ví dụ:

```
class HìnhVuong extends TuGiac {  
    ...  
}
```

```

public class TuGiac {
    protected Diem d1, d2, d3, d4;
    public void setD1(Diem _d1) {d1=_d1;}
    public Diem getD1(){return d1;}
    public void printTuGiac(){...}
    ...
}

```

Ví dụ 1.1

Sử dụng các thuộc tính
protected của lớp cha
trong lớp con

```

public class HìnhVuong extends TuGiac {
    public HìnhVuong(){
        d1 = new Diem(0,0); d2 = new Diem(0,1);
        d3 = new Diem(1,0); d4 = new Diem(1,1);
    }
}

```

```

public class Test{
    public static void main(String args[]){
        HìnhVuong hv = new HìnhVuong();
        hv.printTuGiac();
    }
}

```

Gọi phương thức public
lớp cha của đối tượng lớp con

```
public class TuGiac {
    protected Diem d1, d2, d3, d4;
    public void printTuGiac(){...}
    public TuGiac(){...}
    public TuGiac(Diem d1, Diem d2,
                  Diem d3, Diem d4) { ...}
}

public class HìnhVuong extends TuGiac {
    public HìnhVuong(){ super(); }
    public HìnhVuong(Diem d1, Diem d2,
                    Diem d3, Diem d4){
        super(d1, d2, d3, d4);
    }
}

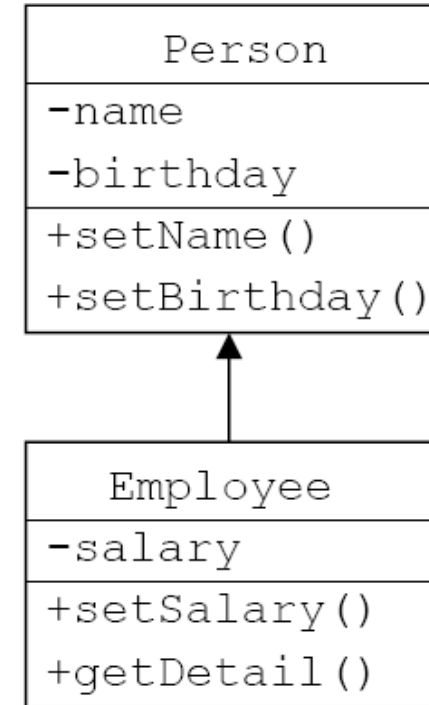
public class Test{
    public static void main(String args[]){
        HìnhVuong hv = new HìnhVuong();
        hv.printTuGiac();
    }
}
```

Ví dụ 1.2

Ví dụ 2

protected

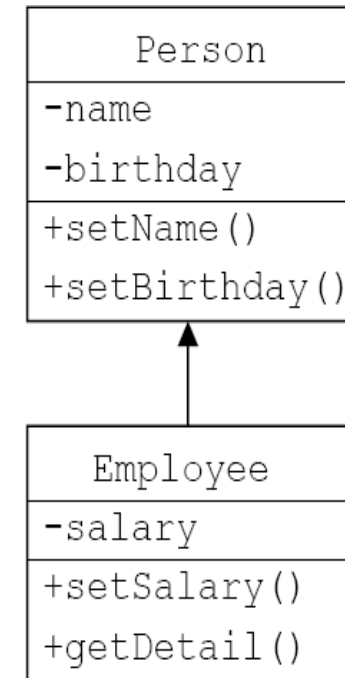
```
class Person {  
    private String name;  
    private Date birthday;  
    public String getName() {return name;}  
    ...  
}  
class Employee extends Person {  
    private double salary;  
    public boolean setSalary(double sal){  
        salary = sal;  
        return true;  
    }  
  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



Ví dụ 2

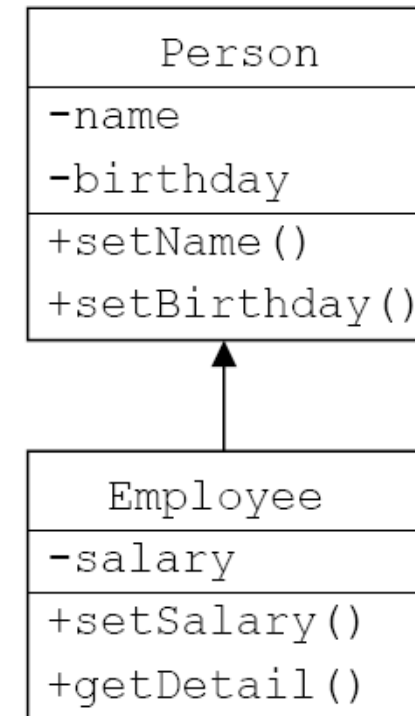
protected

```
class Person {  
    protected String name;  
    protected 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;  
    }  
}
```



Ví dụ 2 (tiếp)

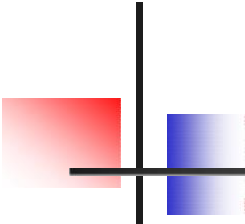
```
public class Test {  
    public static void main(String args[]) {  
        Employee e = new Employee();  
        e.setName("John");  
        e.setSalary(3.0);  
    }  
}
```





Ví dụ 3 – Cùng gói

```
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;  
    }  
}
```



Ví dụ 3 – Khác gói

```
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;
    }
}
```



3.4. Khởi tạo và huỷ bỏ đối tượng

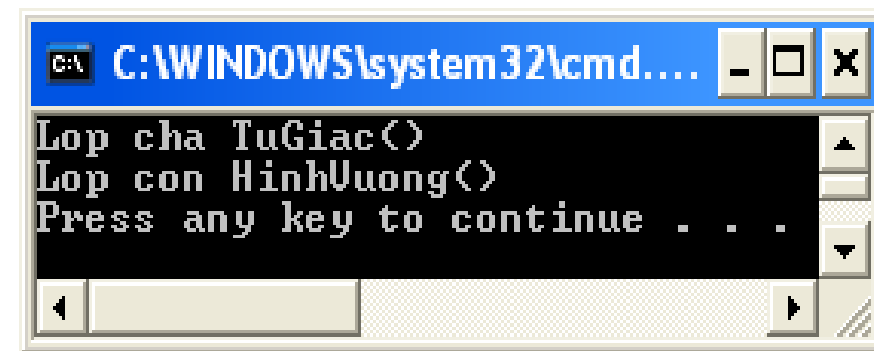
- ▢ Khởi tạo đối tượng:
 - ▢ Lớp cha được khởi tạo trước lớp con.
 - ▢ Các phương thức khởi tạo của lớp con luôn gọi phương thức khởi tạo của lớp cha ở câu lệnh đầu tiên
 - ▢ Tự động gọi (không tường minh - implicit): Khi lớp cha
CÓ phương thức khởi tạo mặc định
 - ▢ Gọi trực tiếp (tường minh - explicit)
- ▢ Huỷ bỏ đối tượng:
 - ▢ Ngược lại so với khởi tạo đối tượng

3.4.1. Tự động gọi constructor của lớp cha

```
public class TuGiac {
    protected Diem d1, d2;
    protected Diem d3, d4;
    public TuGiac() {
        System.out.println
            ("Lop cha TuGiac()");
    }
    //...
}

public class HinhVuong
    extends TuGiac {
    public HinhVuong() {
        //Tu dong gọi TuGiac()
        System.out.println
            ("Lop con HinhVuong()");
    }
}
```

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



```
C:\WINDOWS\system32\cmd...
Lop cha TuGiac()
Lop con HinhVuong()
Press any key to continue . . .
```



3.4.2. Gọi trực tiếp constructor của lớp cha

- Câu lệnh đầu tiên trong phương thức khởi tạo của lớp con có thể gọi phương thức khởi tạo của lớp cha
 - `super(Danh_sach_tham_so) ;`
 - Điều này là bắt buộc nếu lớp cha không có phương thức khởi tạo mặc định
 - Đã viết phương thức khởi tạo của lớp cha với một số tham số
 - Phương thức khởi tạo của lớp con không bắt buộc phải có tham số.

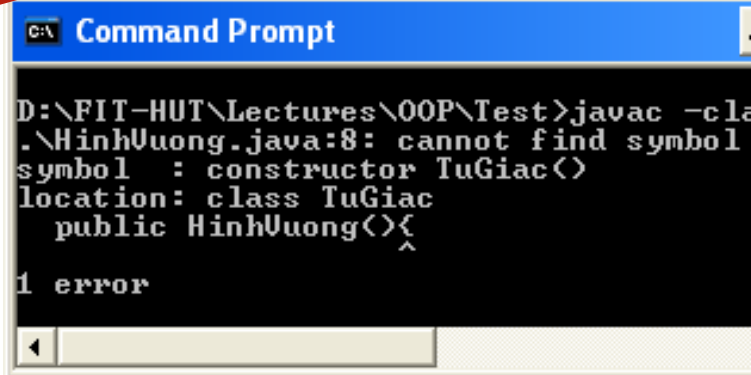
Ví dụ

```
public class TuGiac {  
    protected Diem d1, d2;  
    protected Diem d3, d4;  
    public TuGiac(Diem d1,  
        Diem d2, Diem d3, Diem d4) {  
        System.out.println("Lop cha  
            TuGiac(d1, d2, d3, d4)");  
        this.d1 = d1; this.d2 = d2;  
        this.d3 = d3; this.d4 = d4;  
    }  
}
```

```
public class HìnhVuong extends TuGiac {  
    public HìnhVuong() {  
        System.out.println  
            ("Lop con HìnhVuong()");  
    }  
}
```

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

Lỗi ↓



```
C:\> Command Prompt  
D:\FIT-HUT\Lectures\OOP\Test>javac -cla  
.\HìnhVuong.java:8: cannot find symbol  
symbol : constructor TuGiac()  
location: class TuGiac  
    public HìnhVuong() {  
        ^  
1 error
```

Gọi trực tiếp constructor của lớp cha

Phương thức khởi tạo lớp con **KHÔNG** tham số

```
public class TuGiac {
    protected Diem d1,d2,d3,d4;
    public TuGiac(Diem d1, Diem d2,
        Diem d3, Diem d4) {
        System.out.println("Lop cha
            TuGiac(d1,    d2,  d3,  d4)");
        this.d1 = d1; this.d2    = d2;
        this.d3 = d3; this.d4    = d4;
    }
}
```

```
public class HìnhVuong extends TuGiac {
    public HìnhVuong() {
        super(new Diem(0,0), new Diem(0,1),
            new Diem(1,1),new Diem(1,0));
        System.out.println("Lop con HìnhVuong()");
    }
}
```

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



```
C:\WINDOWS\system32\cmd.exe
Lop cha TuGiac(d1, d2, d3, d4)
Lop con HìnhVuong()
Press any key to continue . . .
```

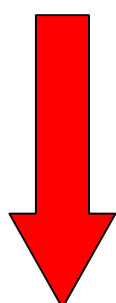
Gọi trực tiếp constructor của lớp cha

Phương thức khởi tạo lớp con **CÓ** tham số

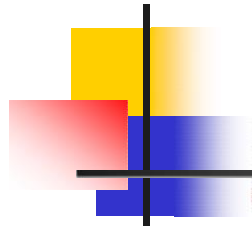
```
public class TuGiac {
    protected Diem d1,d2,d3,d4;
    public TuGiac(Diem d1,
        Diem d2, Diem d3, Diem d4) {
        System.out.println
            ("Lop cha TuGiac(d1,d2,d3,d4)");
        this.d1 = d1; this.d2 = d2;
        this.d3 = d3; this.d4 = d4;
    }
}

public class HìnhVuong extends TuGiac {
    public HìnhVuong(Diem d1, Diem d2,
        Diem d3, Diem d4) {
        super(d1, d2, d3, d4);
        System.out.println("Lop con HìnhVuong(d1,d2,d3,d4)");
    }
}
```

HìnhVuong hv =
new HìnhVuong(
 new Diem(0,0),
 new Diem(0,1),
 new Diem(1,1),
 new Diem(1,0));



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

```
public class TG {  
    private String name;  
    public TG(String name) {  
    }  
}
```

```
public class HV extends TG{  
    public void test(){  
    }  
}
```

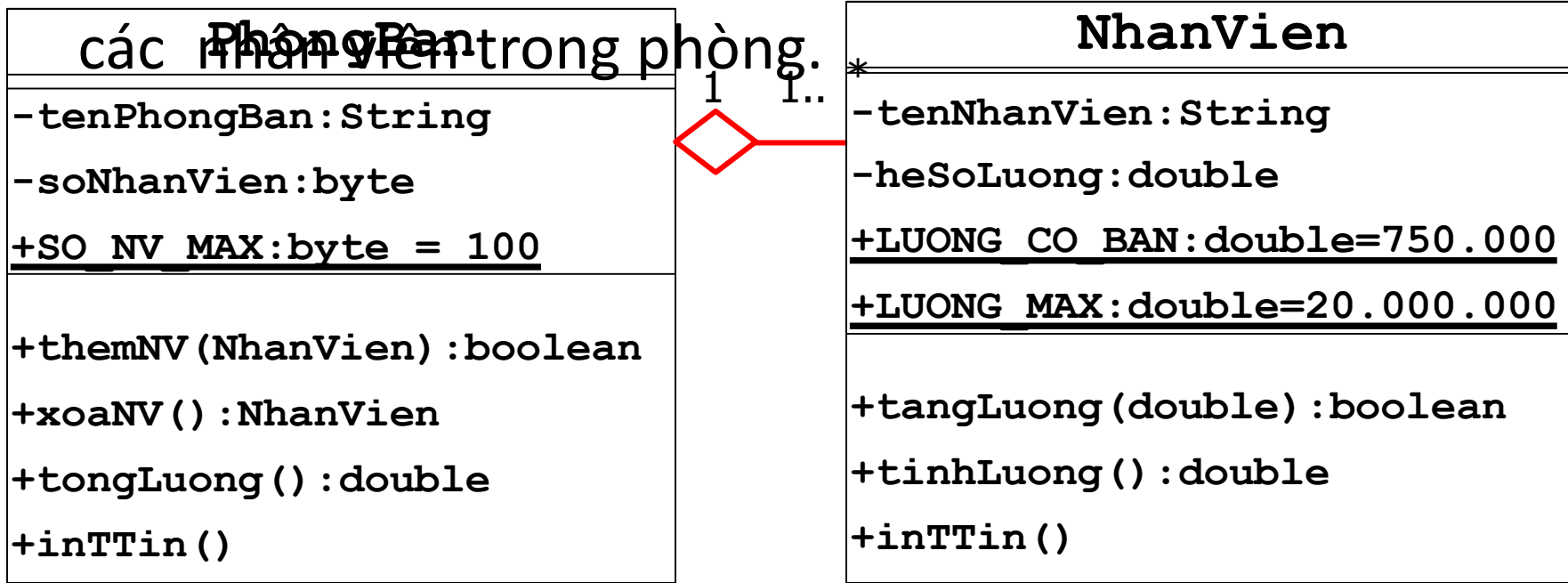


Nội dung

1. Tái sử dụng mã nguồn
2. Kết tập (Aggregation)
3. Kế thừa (Inheritance)
4. **Ví dụ và bài tập**

Bài tập:

- Viết mã nguồn cho lớp **PhòngBan** với các thuộc tính và phương thức như biểu đồ trên cùng phương thức khởi tạo với số lượng tham số cần thiết, biết rằng:
- Việc thêm/xóa nhân viên được thực hiện theo cơ chế của stack
- tongLuong()** trả về tổng lương của các nhân viên trong phòng.
- inTTin()** hiển thị thông tin của phòng và thông tin của



```
public class PhongBan {  
    private String tenPhongBan; private byte soNhanVien;  
    public static final SO_NV_MAX = 100;  
    private NhanVien[] dsnv;  
    public boolean themNhanVien(NhanVien nv){  
        if (soNhanVien < SO_NV_MAX) { dsnv[soNhanVien] = nv;  
            soNhanVien++; return true;  
        } else return false;  
    }  
    public NhanVien xoaNhanVien(){  
        if (soNhanVien > 0) {  
            NhanVien tmp = dsnv[soNhanVien-1];  
            dsnv[soNhanVien-1] = null; soNhanVien--; return tmp;  
        } else return null;  
    }  
    // (cont)...
```

// (cont.)

```
public PhongBan(String tenPB){
    dsnv = new NhanVien[SO_NV_MAX]; tenPhongBan = tenPB; soNhanVien =
    0;
}

public double tongLuong(){
    double tong = 0.0;
    for (int i=0;i<soNhanVien;i++)
        tong += dsnv[i].tinhLuong();
    return tong;
}

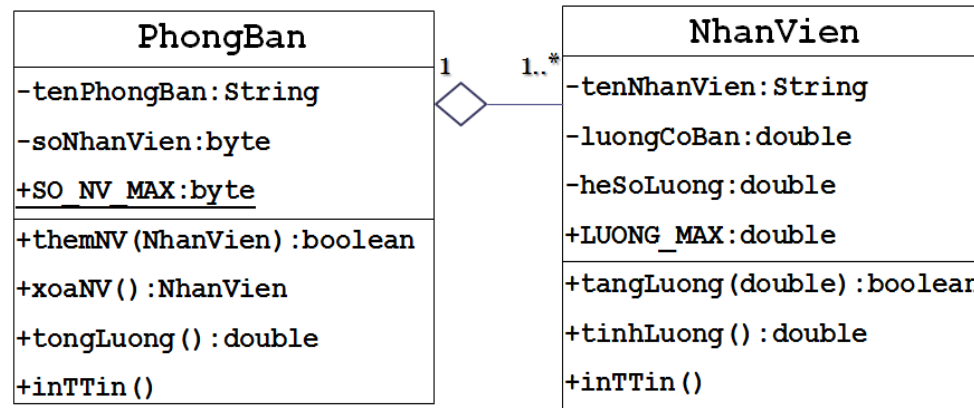
public void inTTin(){
    System.out.println("Ten phong: "+tenPhong);
    System.out.println("So NV: "+soNhanVien);
    System.out.println("Thong tin cac NV");
    for (int i=0;i<soNhanVien;i++)
        dsnv[i].inTTin();
}

}
```

Thảo luận

Trong ví dụ trên

- Lớp cũ? Lớp mới?
 - Lớp cũ: NhanVien
 - Lớp mới: PhongBan
- Lớp mới tái sử dụng lớp cũ thông qua?
 - Mảng đối tượng của lớp NhanVien: dsnv
- Lớp mới tái sử dụng được những gì của lớp cũ?
 - tinhLuong() trong phương thức tongLuong()
 - inTTin() trong phương thức inTTin()



Môn: Lập trình Hướng đối tượng (Object Oriented Programming)

Chương 5. Một số kỹ thuật trong kế thừa



Mục tiêu

- ▣ Trình bày nguyên lý định nghĩa lại trong kế thừa
- ▣ Phân biệt khái niệm đơn kế thừa và đa kế thừa
- ▣ Giới thiệu về giao diện, lớp trừu tượng và vai trò của chúng
- ▣ Ví dụ và bài tập về các vấn đề trên với ngôn ngữ lập trình Java



Nội dung

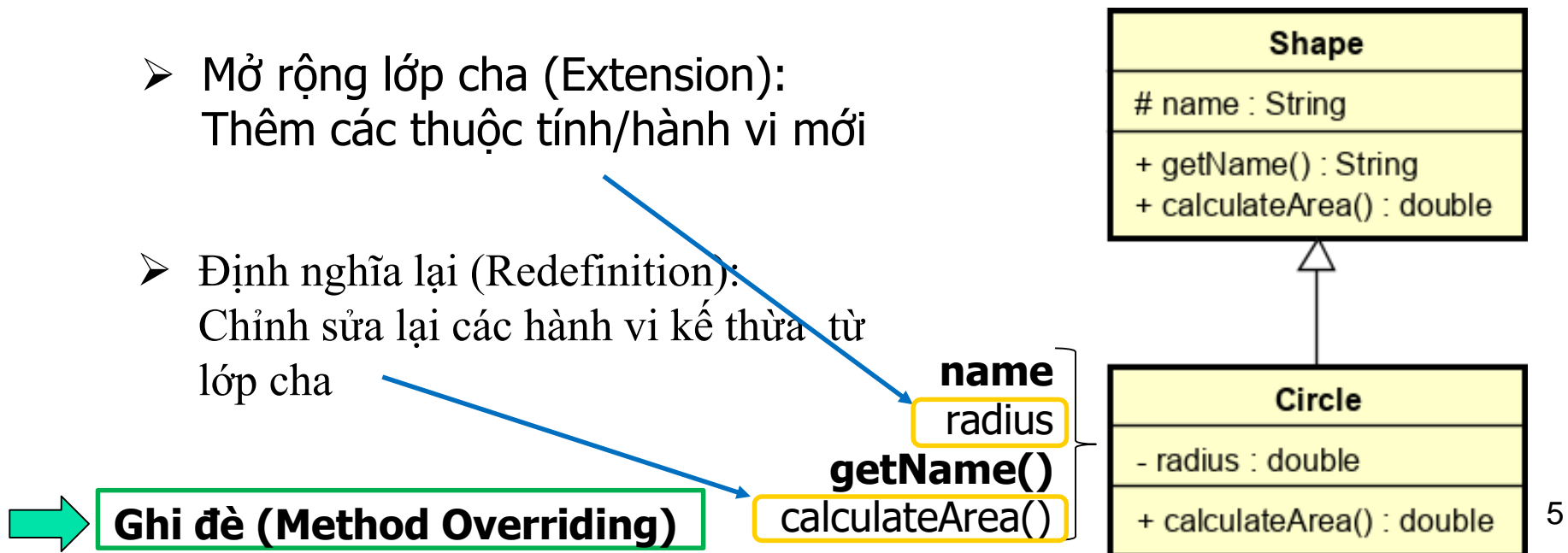
1. Định nghĩa lại/ghi đè (Overriding)
2. Lớp trừu tượng
3. Đơn kế thừa & Đa kế thừa
4. Giao diện (Interface)
5. Vai trò của lớp trừu tượng và giao diện
6. Ví dụ và bài tập

1. Định nghĩa lại/ghi đè (Overriding)

- ✓ Quan hệ kế thừa (inheritance)
 - ✓ Lớp con là một loại (is-a-kind-of) của lớp cha
 - ✓ Kế thừa các thành phần dữ liệu và các hành vi của lớp cha
 - ✓ Chi tiết hóa cho phù hợp với mục đích sử dụng mới:

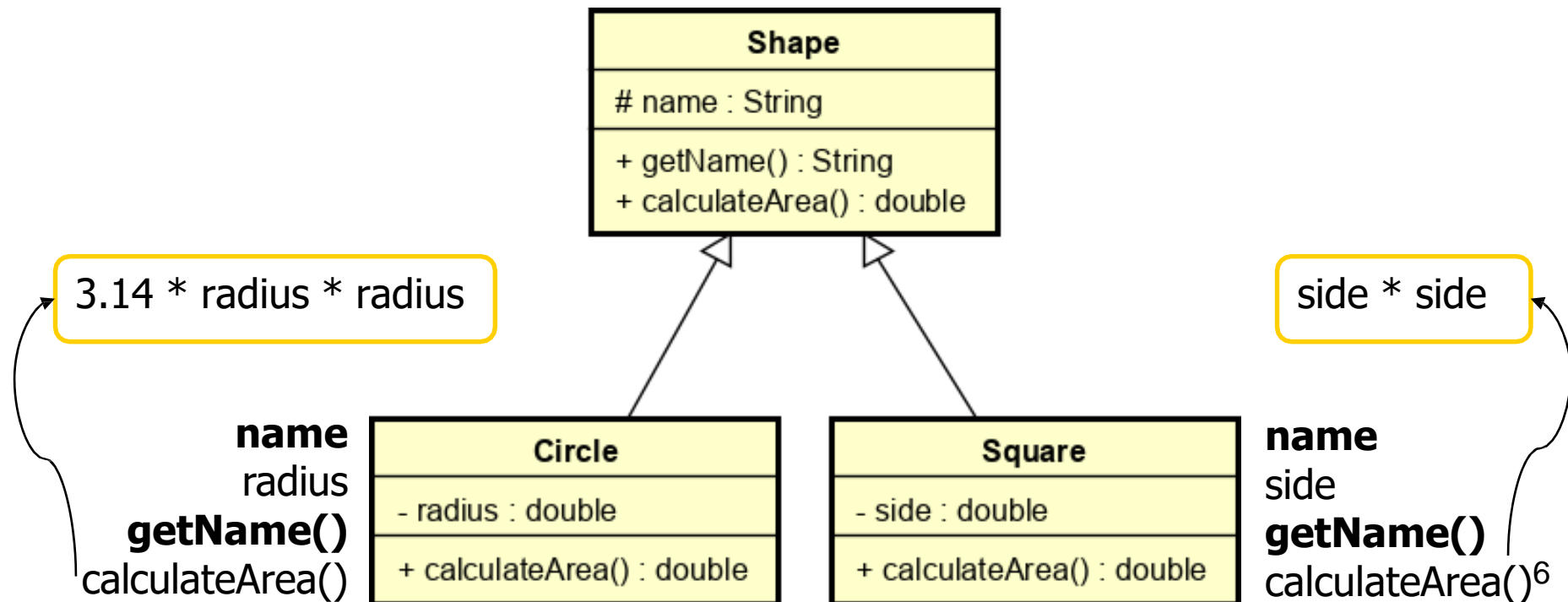
➤ Mở rộng lớp cha (Extension):
Thêm các thuộc tính/hành vi mới

➤ Định nghĩa lại (Redefinition):
Chỉnh sửa lại các hành vi kế thừa từ lớp cha



1. Định nghĩa lại/ghi đè (Overriding)

- Phương thức ghi đè sẽ thay thế hoặc làm rõ hơn cho phương thức cùng tên trong lớp cha
- Đối tượng của lớp con sẽ hoạt động với phương thức mới phù hợp với nó



1. Định nghĩa lại/ghi đè (Overriding)

- Cú pháp: Phương thức ở lớp con hoàn toàn giống về chữ ký với phương thức ở lớp cha
 - Trùng tên & danh sách tham số
 - Mục đích: Để thể hiện cùng bản chất công việc
- Lớp con có thể định nghĩa phương thức trùng tên với phương thức trong lớp cha:

Nếu phương thức mới chỉ trùng tên và khác chữ ký (số lượng hay kiểu dữ liệu của đối số)
→ Chồng phương thức (Method Overloading)

Nếu phương thức mới hoàn toàn giống về giao diện (chữ ký)
→ Định nghĩa lại hoặc ghi đè phương thức (Method Override)

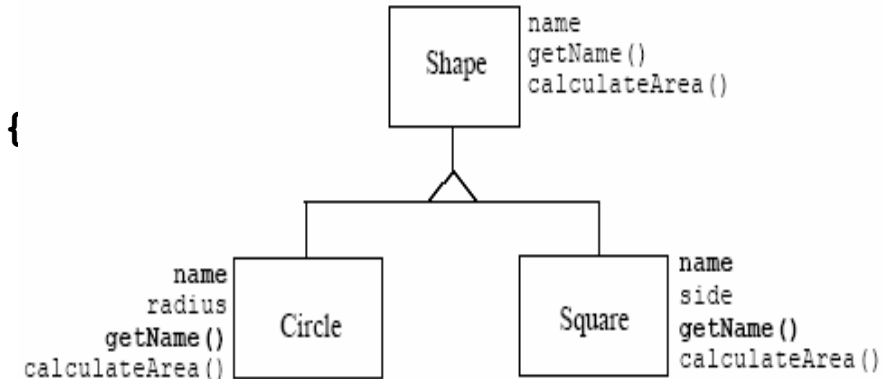
1. Định nghĩa lại/ghi đè (Overriding)

□ Ví dụ:

```
class Shape {  
    protected String name;  
    Shape(String n) { name = n; }  
    public String getName() { return name; }  
    public double calculateArea() { return 0.0; }  
}
```

```
class Circle extends Shape {  
    private double radius;  
    Circle(String n, double r) {  
        super(n);  
        radius = r;  
    }  
}
```

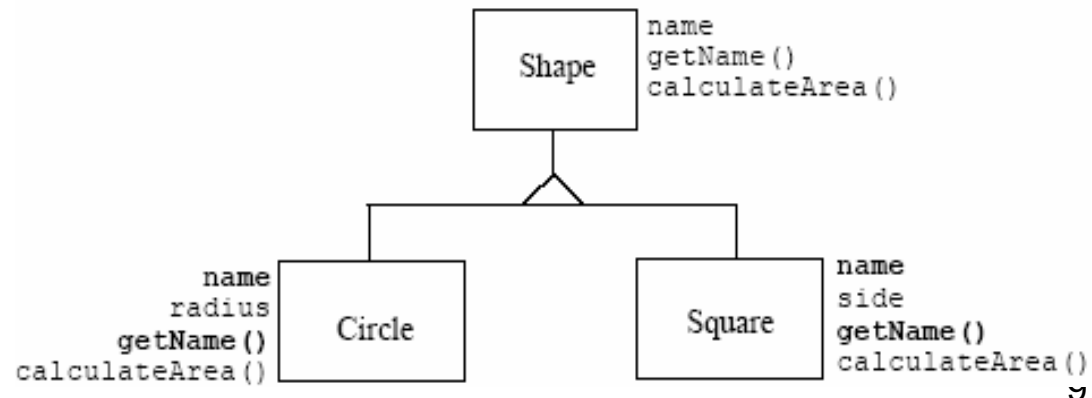
```
    public double calculateArea() {  
        double area = (double) (3.14 * radius * radius);  
        return area;  
    } }
```



1. Định nghĩa lại/ghi đè (Overriding)

- Ví dụ (tiếp theo):

```
class Square extends Shape {  
    private double side;  
    Square(String n, double s) {  
        super(n);  
        side = s;  
    }  
    public double calculateArea() {  
        double area = (double) side * side;  
        return area;  
    }  
}
```



1. Định nghĩa lại/ghi đè (Overriding)

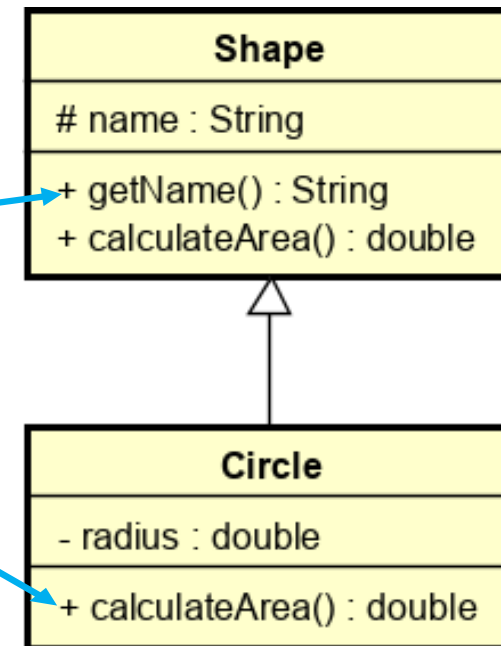
- Ghi đè → phương thức nào được gọi?
 - Máy ảo Java bắt đầu tìm từ lớp của đối tượng, nếu nó không tìm được một phiên bản của phương thức đó tại lớp này thì nó chuyển lên tìm tại lớp cha tiếp theo bên trên ở cây thừa kế,...
 - Cứ như vậy cho đến khi tìm thấy một phiên bản khớp với lời gọi phương thức. Nếu không tìm thấy phiên bản nào hoặc lời gọi có nhiều hơn một phiên bản phù hợp
→ báo lỗi biên dịch!

```
Circle c = new Circle("c");
```

```
String s = c.getName();
```

```
double a = c.calculateArea();
```

Cái gì ở thấp nhất thì được gọi...



1. Định nghĩa lại/ghi đè (Overriding)

- Ví dụ:

```
class MyClass{
    public void myMethod(int a, long b) {
    }
    public void myMethod(long a, int b) {    //overloading
    }
}

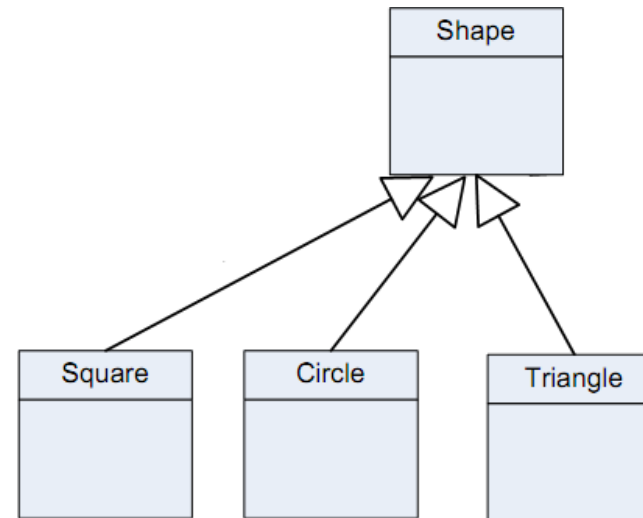
public class Test{
    public static void main(String args[]){
        MyClass m = new MyClass();
        m.myMethod(); //error→không có phiên bản method phù hợp
        m.myMethod(9, 10); //error→2 phiên bản method phù hợp
    }
}
```


1. Định nghĩa lại/ghi đè (Overriding)

- Thêm lớp Triangle:

```
class Triangle extends Shape {  
    private double base, height;  
    Triangle(String n, double b, double h) {  
        super(n);  
        base = b; height = h;  
    }  
    public double calculateArea() {  
        double area = 0.5f * base * height;  
        return area;  
    }  
}
```

Muốn gọi lại các phương thức của
lớp cha đã bị ghi đè ?



Sử dụng từ khóa

super

- Từ khóa **super**: tái sử dụng các đoạn mã của lớp cha trong lớp con
- Gọi phương thức khởi tạo **super(danh sách tham số);**
 - Bắt buộc nếu lớp cha không có phương thức khởi tạo mặc định
- Gọi các phương thức của lớp cha **super.tên_Phương_thức(danh sách tham số);**

Sử dụng từ khóa

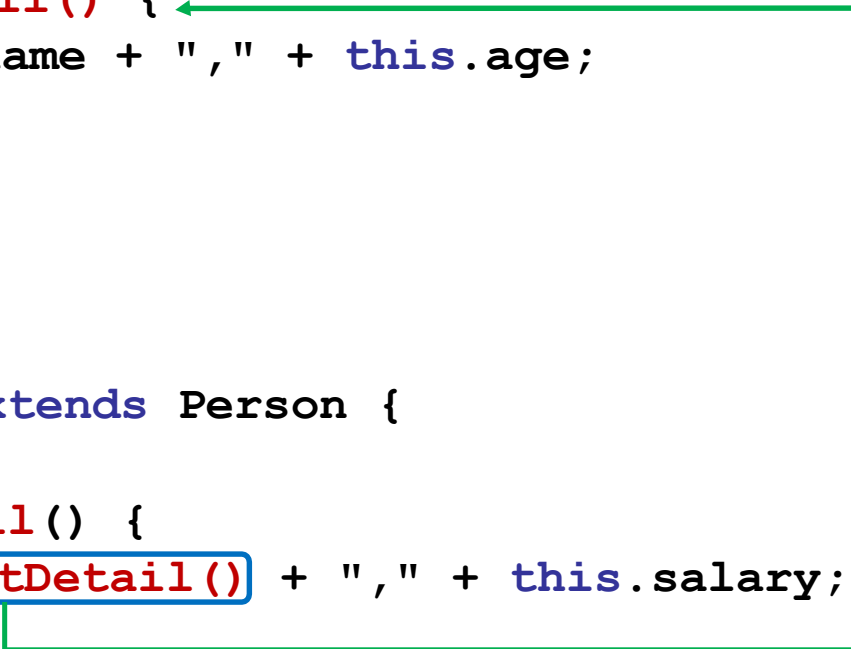
super

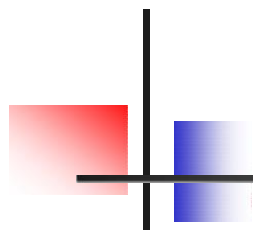
✓ Ví dụ:

```
package abc;

public class Person {
    protected String name;
    protected int age;
    public String getDetail() {
        String s = this.name + "," + this.age;
        return s;
    }
}

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





Quy định trong ghi đề

- ✓ Phương thức ghi đề trong lớp con phải
 - ✓ Có danh sách tham số giống hết phương thức kế thừa trong lớp cha.
 - ✓ Có cùng kiểu trả về với phương thức kế thừa trong lớp cha
- ✓ Các chỉ định truy cập không giới hạn chặt hơn phương thức trong lớp cha
 - ✓ Ví dụ, nếu ghi đề một phương thức protected, thì phương thức mới có thể là protected hoặc public, mà không được là private

Quy định trong ghi đề

- Ví dụ:

```
class Parent {  
    public void doSomething() {}  
    protected int doSomething2() {  
        return 0;  
    }  
}
```

*Không ghi đề được do
không cùng kiểu
trả về*

```
class Child extends Parent {  
    protected void doSomething() {}  
    protected void doSomething2() {}  
}
```

*Không ghi đề được do chỉ định truy cập
yếu hơn (public -> protected)*



Quy định trong ghi đề

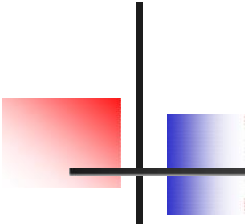
- Không được phép ghi đề:
 - Các phương thức **static** trong lớp cha
 - Các phương thức **private** trong lớp cha
 - Các phương thức hằng (**final**) trong lớp cha



Hạn chế ghi đè – Từ khoá **final**

- Đôi lúc ta muốn hạn chế việc định nghĩa lại vì các lý do sau:
 - Tính đúng đắn: Định nghĩa lại một phương thức trong lớp dẫn xuất có thể làm sai lệch ý nghĩa của nó
 - Tính hiệu quả: Cơ chế kết nối động không hiệu quả về mặt thời gian bằng kết nối tĩnh
- Nếu biết trước sẽ không định nghĩa lại phương thức của lớp cơ sở thì nên dùng từ khóa **final** đi với phương thức. Ví dụ:

```
public final String baseName () {  
    return "Person";  
}
```

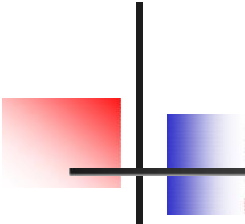


Hạn chế ghi đè – Từ khoá **final**

- Các phương thức được khai báo là **final** không thể ghi đè

```
class A {  
    final void method(){ }  
}
```

```
class B extends A{  
    void method(){           // Báo lỗi!!!  
    }  
}
```

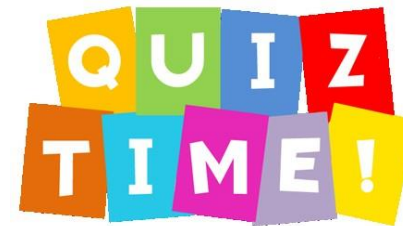



Hạn chế ghi đè – Từ khóa **final**

- ▣ Từ khóa **final** được dùng khi khai báo lớp:
 - ▣ Lớp được khai báo là lớp hằng (không thay đổi), lớp này không có lớp con thừa kế
 - ▣ Được sử dụng để hạn chế việc thừa kế và ngăn chặn việc sửa đổi một lớp

```
public final class A {  
    //...  
}
```

Câu hỏi



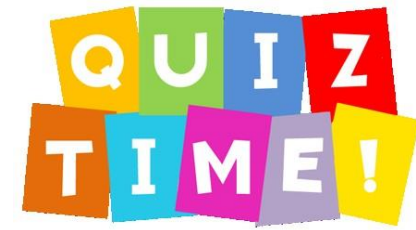
✓ Cho đoạn mã dưới đây:

```
1. class BaseClass {  
2.     private float x = 1.0f;  
3.     float getVar() { return x; }  
4. }  
5. class SubClass extends BaseClass {  
6.     private float x = 2.0f;  
7. // insert code here 8. }
```

■ Lựa chọn nào có thể chèn tại dòng 7 (2 phương án)?

```
1. public double getVar() { return x; }  
2. public float getVar(float f){ return f; }  
3. float getVar() { return x; }  
4. public float getVar() { return x; }  
5. private float getVar() { return x; }
```

Câu hỏi



- Cho đoạn mã dưới đây:

```
1  class Super {  
2      public String getName () { return "Super" ; }  
3  }  
4  public class Sub extends Super {  
5  
6  }
```

- Lựa chọn nào khi đặt vào dòng 5 trong đoạn mã trên gây ra lỗi biên dịch?

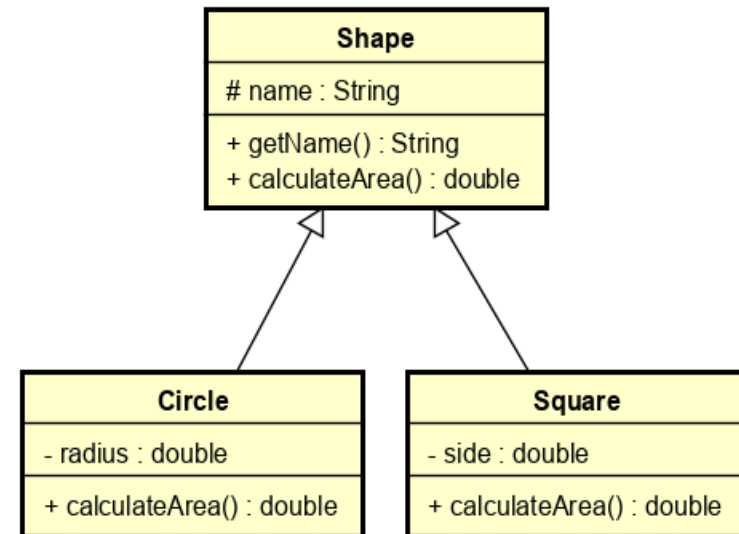
1. `public String getTen () { }`
2. `public void getName(String str) { }`
3. `public String getName() {return "Sub"; }`
4. `public void getName() { }`

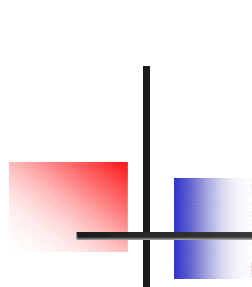
2. Lớp trừu tượng

- ✓ Các ngôn ngữ lập trình hướng đối tượng cung cấp các cơ chế kiểu trừu tượng (abstract type)
 - ✓ Các kiểu trừu tượng có cài đặt không đầy đủ hoặc không có cài đặt
 - ✓ Nhiệm vụ chính của chúng là giữ vai trò kiểu tổng quát hơn của một số các kiểu khác

- ✓ Xét ví dụ: lớp Shape

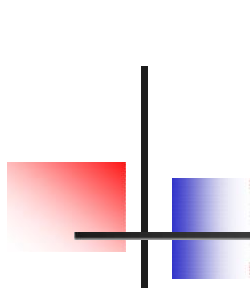
- ✓ Là một lớp "không rõ ràng", khó hình dung ra các đối tượng cụ thể
 - Không thể thể hiện hóa (instantiate của lớp) trực tiếp
 - tạo đối tượng





2. Lớp trừu tượng

- Đặc điểm của lớp trừu tượng
 - Không thể tạo đối tượng trực tiếp từ các lớp trừu tượng
 - Thường lớp trừu tượng được dùng để định nghĩa các "khái niệm chung", đóng vai trò làm lớp cơ sở (base class) cho các lớp "cụ thể" khác (concrete class)
 - Chưa đầy đủ, thường được sử dụng làm lớp cha. Lớp con kế thừa nó sẽ hoàn thiện nốt.
 - Lớp trừu tượng thường chứa các *phương thức trừu tượng* (phương thức không được cài đặt)



2. Lớp trừu tượng

- Phương thức trừu tượng
 - Là các phương thức “không rõ ràng” / chưa hoàn thiện, khó đưa ra cách cài đặt cụ thể
 - Chỉ có chữ ký mà không có cài đặt cụ thể
 - Các lớp dẫn xuất có thể làm rõ - định nghĩa lại (overriding) các phương thức trừu tượng này



Từ khoá **abstract**

- Lớp trừu tượng
 - Khai báo với từ khóa **abstract**
public abstract class Shape {
 // Nội dung lớp
}
- Phương thức trừu tượng
 - Khai báo với từ khóa **abstract**
public abstract float calculateArea();

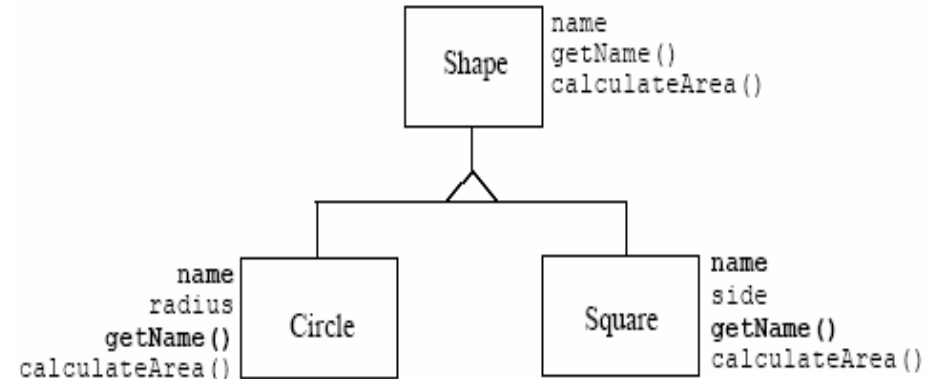
Shape a = new Shape(); //Compile error

2. Lớp trừu tượng

- Ví dụ:

```
abstract class Shape {  
    protected String name;  
    Shape(String n) { name = n; }  
    public String getName() { return name; }  
    public abstract double calculateArea();  
}
```

```
class Circle extends Shape {  
    private double radius;  
    Circle(String n, double r){  
        super(n);  
        radius = r;  
    }  
    public double calculateArea() {  
        double area = (double) (3.14 * radius * radius);  
        return area;  
    }  
}
```

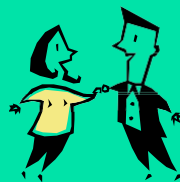


Lớp con bắt buộc phải override tất cả các phương thức abstract của lớp cha

2. Lớp trừu tượng

- Nếu một lớp có một hay nhiều phương thức trừu tượng thì nó phải là lớp trừu tượng
 - Lớp con khi kế thừa phải cài đặt cụ thể cho các phương thức trừu tượng của lớp cha
 - Nếu không ghi đè các phương thức này thì lớp con cũng trở thành một lớp trừu tượng
- Phương thức trừu tượng không thể khai báo là **final** hoặc **static**

Kết hợp cho phép
abstract public
abstract protected



Kết hợp KHÔNG cho phép
abstract private
abstract static
abstract final



2. Lớp trừu tượng

■ Ví dụ:

```
abstract class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    public void move(int dx, int dy) {  
        x += dx;  
        y += dy;  
        plot();  
    }  
    public abstract void plot();  
    // phương thức trừu tượng không có  
    // phần code thực hiện  
}
```

2. Lớp trừu tượng

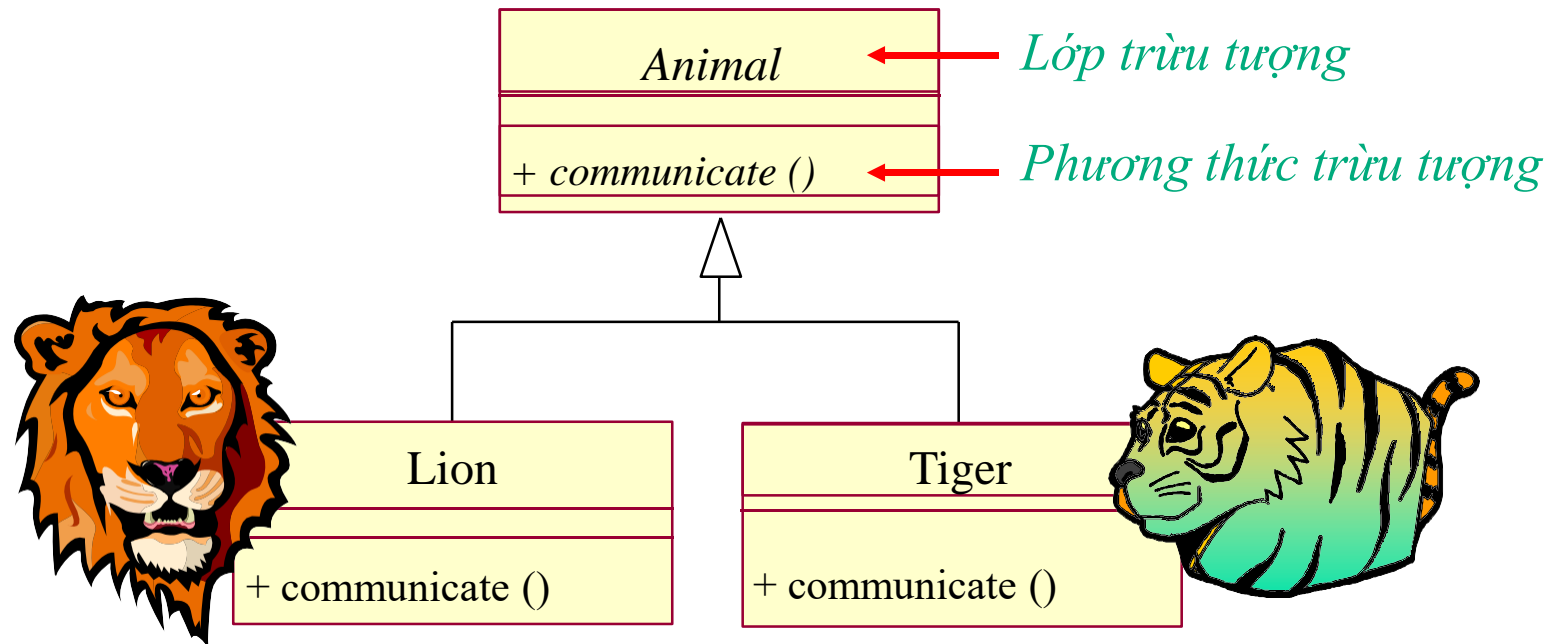
- Ví dụ:

```
abstract class ColoredPoint extends Point {  
    int color;  
    public ColoredPoint(int x, int y, int color) {  
        super(x, y); this.color = color;  
    }  
}  
class SimpleColoredPoint extends ColoredPoint {  
    public SimpleColoredPoint(int x, int y, int color) {  
        super(x, y, color);  
    }  
    public void plot() { ... }  
    // code to plot a SimplePoint  
}
```

2. Lớp trừu tượng

▣ Biểu diễn trong UML

- ▣ Lớp trừu tượng (không thể tạo đối tượng cụ thể)
 - ▣ Chứa phương thức trừu tượng
 - ▣ Tên lớp / tên phương thức: Chữ nghiêng



Tất cả các đối tượng là sư tử hoặc hổ

Câu hỏi



- 1. Đoạn mã dưới đây có lỗi gì không?

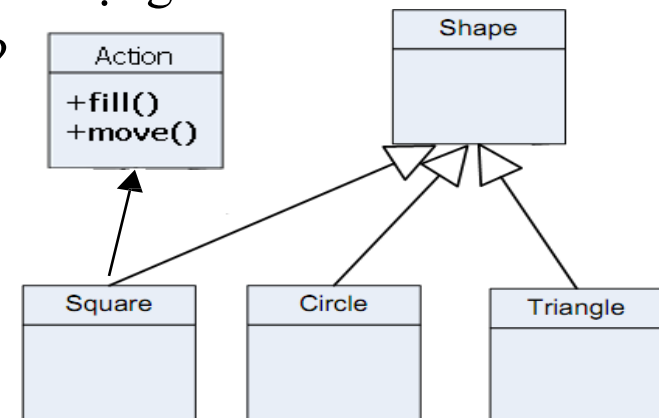
```
abstract class ABC {  
    void firstMethod() {  
        System.out.println("First Method");  
    }  
    void secondMethod() {  
        System.out.println("Second Method");  
    }  
}
```

- 2. Lớp nào là lớp trừu tượng, lớp nào có thể tạo đối tượng?

```
abstract class A {  
    }  
  
class B extends A {  
    }
```

3. Đơn kế thừa & Đa kế thừa

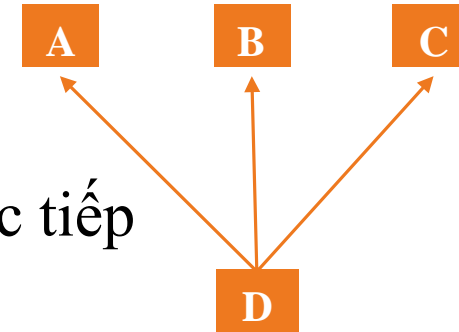
- Giả sử trong bài toán các lớp đối tượng Hình học, lớp Square cần thiết kế bổ sung thêm những hành vi mới Fill (tô màu), Move (di chuyển) mà chỉ có các đối tượng của nó sử dụng
 - Giải pháp 1: thêm các hành vi này vào lớp cha Shape → ảnh hưởng đến các đối tượng của lớp con Circle và Triangle (các đối tượng này không sử dụng đến các hành vi trên)
 - Giải pháp 2: đặt các hành vi này trực tiếp tại lớp Square → tương lai có thể có thêm lớp mới Hình thang cũng sử dụng các hành vi trên → cần phải cài đặt lại, không tái sử dụng
- ...cần HAI lớp cha trong cây thừa kế?



3. Đơn kế thừa & Đa kế thừa

■ Đa kế thừa (Multiple Inheritance)

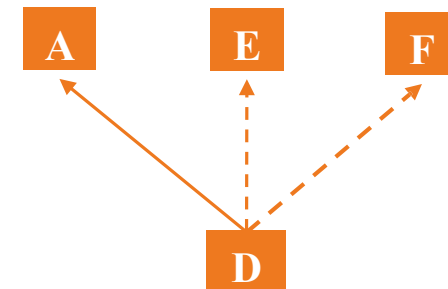
- ✓ Một lớp có thể kế thừa nhiều lớp cha trực tiếp
- ✓ C++ hỗ trợ đa kế thừa



■ Đơn kế thừa (Single Inheritance)

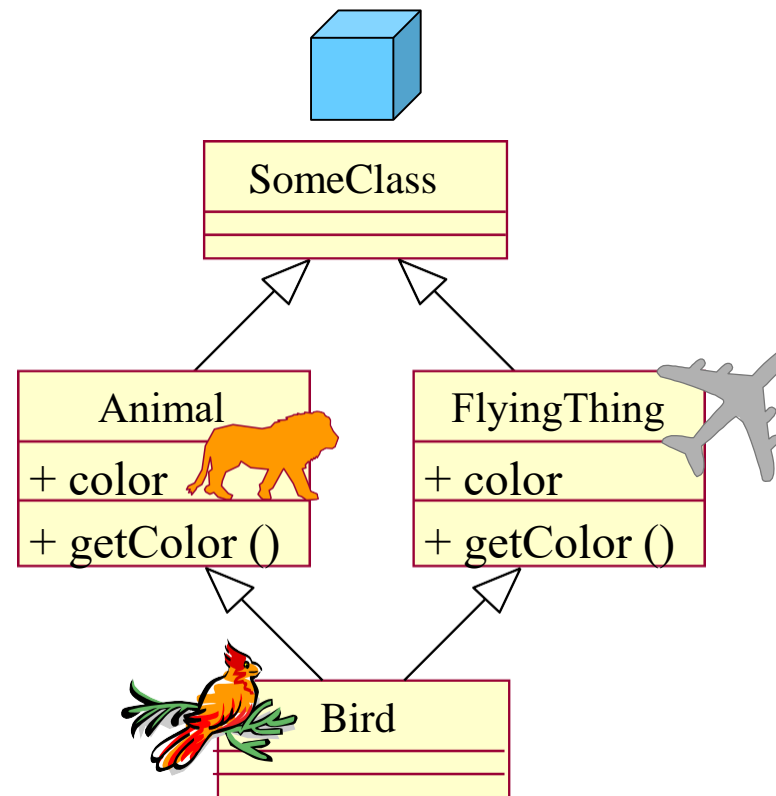
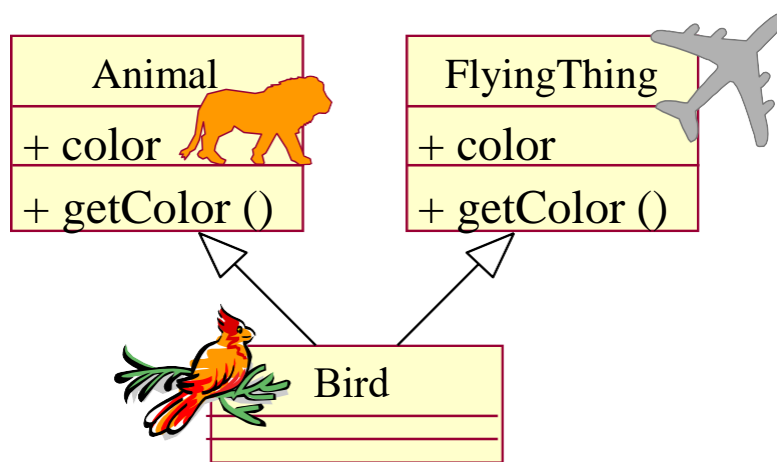
- Một lớp chỉ được kế thừa từ một lớp cha trực tiếp
- Java chỉ hỗ trợ đơn kế thừa

→ Đưa thêm khái niệm Giao diện (Interface)



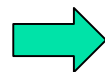
3. Đơn kế thừa & Đa kế thừa

- Vấn đề gặp phải trong đa kế thừa
 - Name collision
 - "Diamond shape" problem



4. Giao diện

- Giao diện (interface)
 - Phương tiện để giao tiếp
 - Không phải quan tâm đến mã bên trong, chỉ cần thống nhất về giao diện
 - Thư viện lập trình hoặc các dịch vụ
- Giao diện trong Java
 - Một cấu trúc lập trình của Java được định nghĩa với từ khóa **interface**
 - Giải quyết bài toán đa thừa kế, tránh các rắc rối nhập nhằng ngữ nghĩa



Phương thức nào cũng phải trừu tượng!

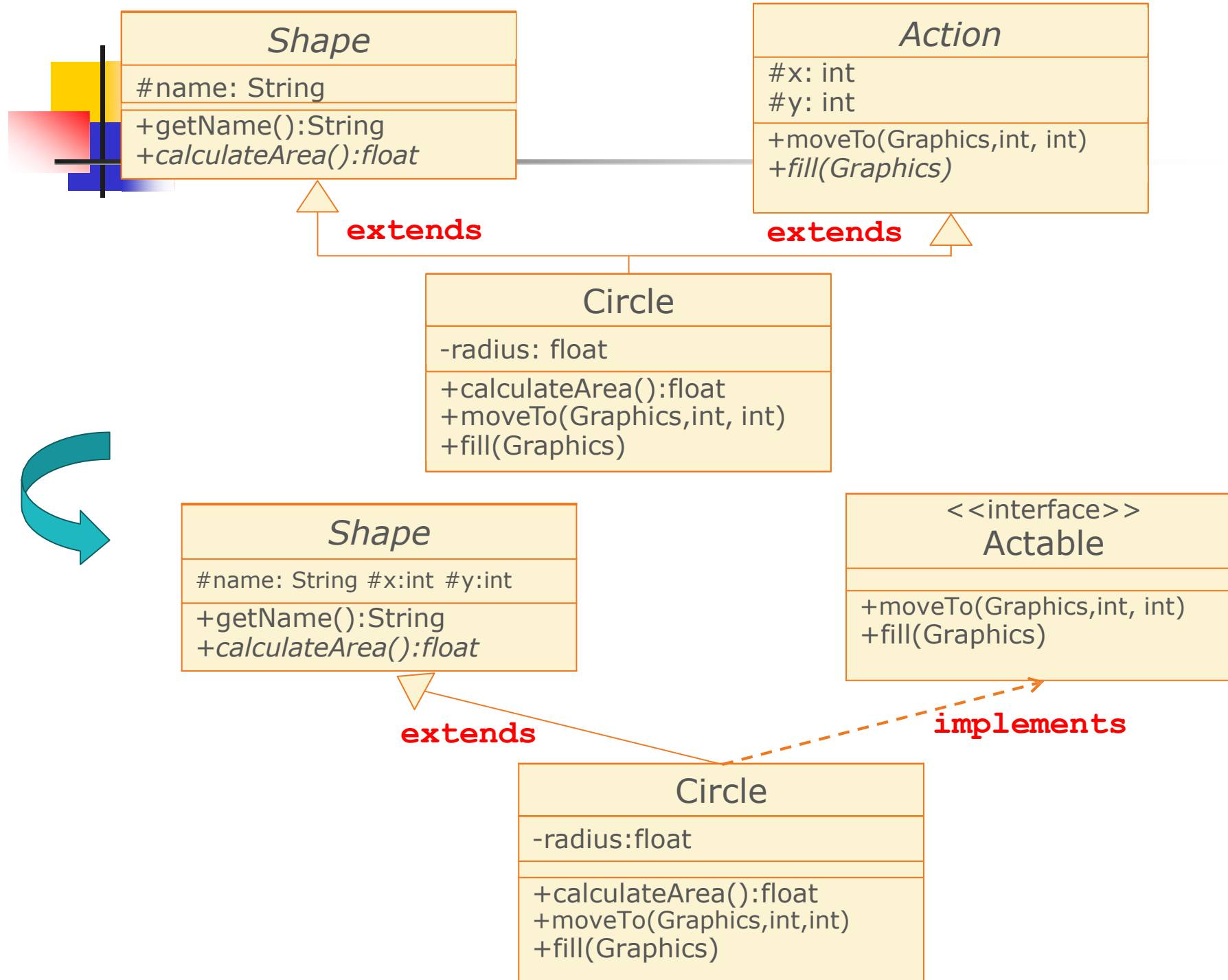
4. Giao diện

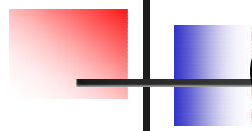
- ▢ Sử dụng từ khóa **interface** để định nghĩa
 - ▢ Một giao diện chỉ được bao gồm:
 - ▢ Chữ ký các phương thức (method signature)
 - ▢ Các thuộc tính khai báo hằng (static & final)
 - ▢ Không có thể hiện
 - ▢ Chỉ được thực thi và mở rộng
- ▢ Cú pháp khai báo giao diện trên Java
 - interface** <Tên giao diện> { }
 - <Giao diện con> **extends** <Giao diện cha> { }
- ▢ Ví dụ
 - public interface DoiXung { ... }**
 - public interface Can extends DoiXung { ... }**
 - public interface DiChuyen { ... }**

4. Giao diện

- Lớp thực thi giao diện
 - Hoặc là lớp trừu tượng (abstract class)
 - Hoặc là bắt buộc phải cài đặt chi tiết toàn bộ các phương thức trong giao diện nếu là lớp cụ thể
- Một lớp có thể thực thi nhiều giao diện
<Lớp con> [*extends* <Lớp cha>]
implements <Danh sách giao diện>
- Ví dụ:

```
public class HìnhVuong extends TuGiac
implements DoiXung, DiChuyen {
}
```





4. Giao diện

- Ví dụ:

```
import java.awt.Graphics;
abstract class Shape {
    protected String name;
    protected int x, y;
    Shape(String n, int x, int y) {
        name = n; this.x = x; this.y = y;
    }
    public String getName() {
        return name;
    }
    public abstract float calculateArea();
}

interface Actable {
    public void moveTo(Graphics g, int x1, int y1);
    public void fill(Graphics g);
}
```

```
class Circle extends Shape implements Actable {
    private int radius;
    public Circle(String n, int x, int y, int r) {
        super(n, x, y); radius = r;
    }
    public float calculateArea() {
        float area = (float) (3.14 * radius * radius);
        return area;
    }
    public void draw(Graphics g) {
        System.out.println("Draw circle at ("
                            + x + "," + y + ")");
        g.drawOval(x-radius,y-radius,2*radius,2*radius);
    }
    public void moveTo(Graphics g, int x1, int y1) {
        x = x1; y = y1; draw(g);
    }
    public void fill(Graphics g) {
        System.out.println("Fill circle at ("
                            + x + "," + y + ")");
        // paint the region with color...
    }
}
```

4. Giao diện

- ✓ Giao diện có thể được sử dụng như một kiểu
- ✓ Các đối tượng gán cho biến giao diện phải thuộc lớp thực thi giao diện
- ✓ Ví dụ:

```
public interface I {}
```

```
public class A implements I {}
```

```
public class B {}
```

```
A a = new A();
```

```
B b = new B();
```

```
I i = (I) a; //đúng
```

```
I i2 = (I) b; //sai
```

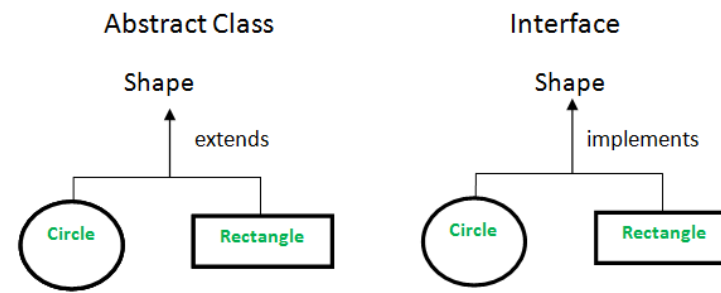
4. Giao diện

- Một interface có thể được coi như một dạng “class” mà:
 - Phương thức và thuộc tính là public không tường minh
 - Các thuộc tính là static và final
 - Các phương thức là abstract
- Không thể thể hiện hóa (instantiate) trực tiếp

4. Giao diện

- ✓ Góc nhìn quan niệm
 - ✓ Interface không cài đặt bất cứ một phương thức nào nhưng để lại cấu trúc thiết kế trên bất cứ lớp nào sử dụng nó
 - ✓ Một interface: 1 contract – trong đó các nhóm phát triển phần mềm thống nhất sản phẩm của họ tương tác với nhau như thế nào, mà không đòi hỏi bất cứ một tri thức về cách thức tiến hành của nhau
 - ✓ Interface: đặc tả cho các bản cài đặt (implementation) khác nhau.
 - ✓ Phân chia ranh giới:
 - ✓ Cái gì (What) và như thế nào (How)
 - ✓ Đặc tả và Cài đặt cụ thể.

4. Giao diện



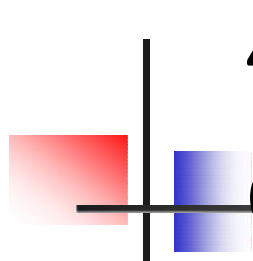
■ Lớp trừu tượng

- Cần có ít nhất một phương thức abstract, có thể chứa các phương thức instance
- Có thể chứa các phương thức protected và static
- Có thể chứa các thuộc tính final và non-final
- Một lớp chỉ có thể kế thừa một lớp trừu tượng

vs.

• Giao diện

- Chỉ có thể chứa chữ ký phương thức (danh sách các phương thức)
- Chỉ có thể chứa các phương thức public mà không có mã nguồn
- Chỉ có thể chứa các thuộc tính hằng
- Một lớp có thể thực thi (kế thừa) nhiều giao diện



4. Giao diện

- ▢ Nhược điểm

- ▢ Không cung cấp một cách tự nhiên cho các tình huống không có sự đụng độ về kế thừa xảy ra
- ▢ Kế thừa nhằm tăng tái sử dụng mã nguồn nhưng giao diện không làm được điều này

Câu hỏi



- 1. Khai báo nào là hợp lệ trong một interface?
 - a. **public static int answer = 42;**
 - b. **int answer;**
 - c. **final static int answer = 42;**
 - d. **public int answer = 42;**
 - e. **private final static int answer = 42;**
- ✓ 2. Một lớp có thể kế thừa chính bản thân nó không?
- ✓ 3. Chuyện gì xảy ra nếu lớp cha và lớp con đều có thuộc tính trùng tên?
- ✓ 4. Phát biểu “Các phương thức khởi tạo cũng được thừa kế xuống các lớp con” là đúng hay sai?
- ✓ 5. Có thể xây dựng các phương thức khởi tạo cho lớp trừu tượng không?
- ✓ 6. Có thể khai báo phương thức protected trong một giao diện không?

5. Lớp trừu tượng & Giao diện

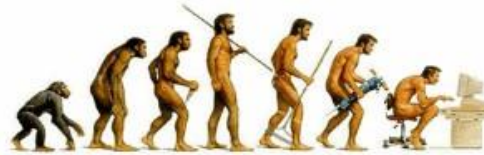
- Khi nào nên cho một lớp là lớp độc lập, lớp con, lớp trừu tượng, hay nên biến nó thành interface?
 - Một lớp nên là lớp độc lập, nghĩa là nó không thừa kế lớp nào (ngoại trừ Object) nếu nó không thỏa mãn quan hệ IS-A đối với bất cứ loại nào khác
 - Một lớp nên là lớp con nếu cần cho nó làm một phiên bản chuyên biệt hơn của một lớp khác và cần ghi đè hành vi có sẵn hoặc bổ sung hành vi mới

5. Lớp trừu tượng & Giao diện

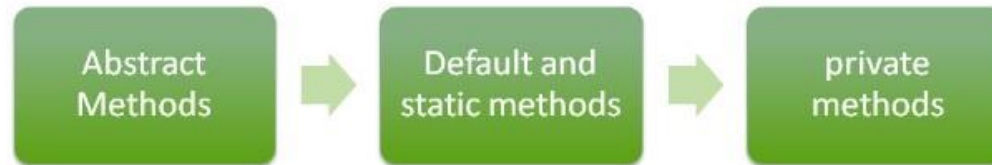
- Khi nào nên cho một lớp là lớp độc lập, lớp con, lớp trừu tượng, hay nên biến nó thành interface?
 - Một lớp nên là lớp cha nếu muốn định nghĩa một khuôn mẫu cho một nhóm các lớp con, và có mã cài đặt mà tất cả các lớp con kia có thể sử dụng
 - Cho lớp đó làm lớp trừu tượng nếu muốn đảm bảo rằng không ai được tạo đối tượng thuộc lớp đó
 - Dùng một interface nếu muốn định nghĩa một vai trò mà các lớp khác có thể nhận, bất kể các lớp đó thuộc cây thừa kế nào

Mở rộng

- Khái niệm giao diện trong các phiên bản của Java



Java7 vs Java8 vs Java9

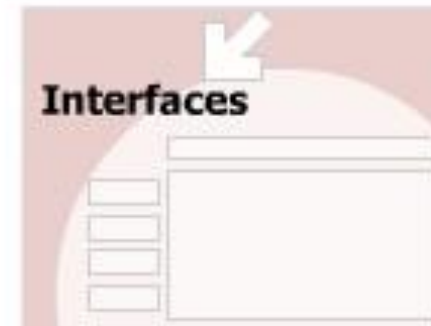
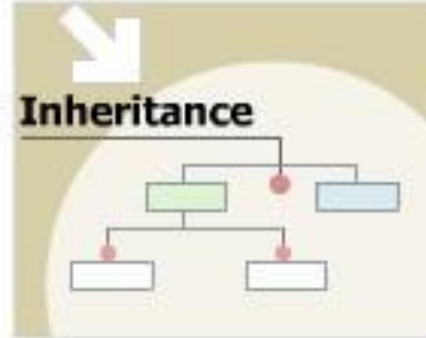


- Chữ ký các phương thức (method signature)
- Các thuộc tính khai báo hằng (static & final)

- Chữ ký các phương thức (method signature)
- Các thuộc tính khai báo hằng (static & final)
- Phương thức mặc định (default method)
- Phương thức tĩnh (Static method)

- Chữ ký các phương thức (method signature)
- Các thuộc tính khai báo hằng (static & final)
- Phương thức mặc định (default method)
- Phương thức tĩnh (Static method)
- Private methods

Tổng kết



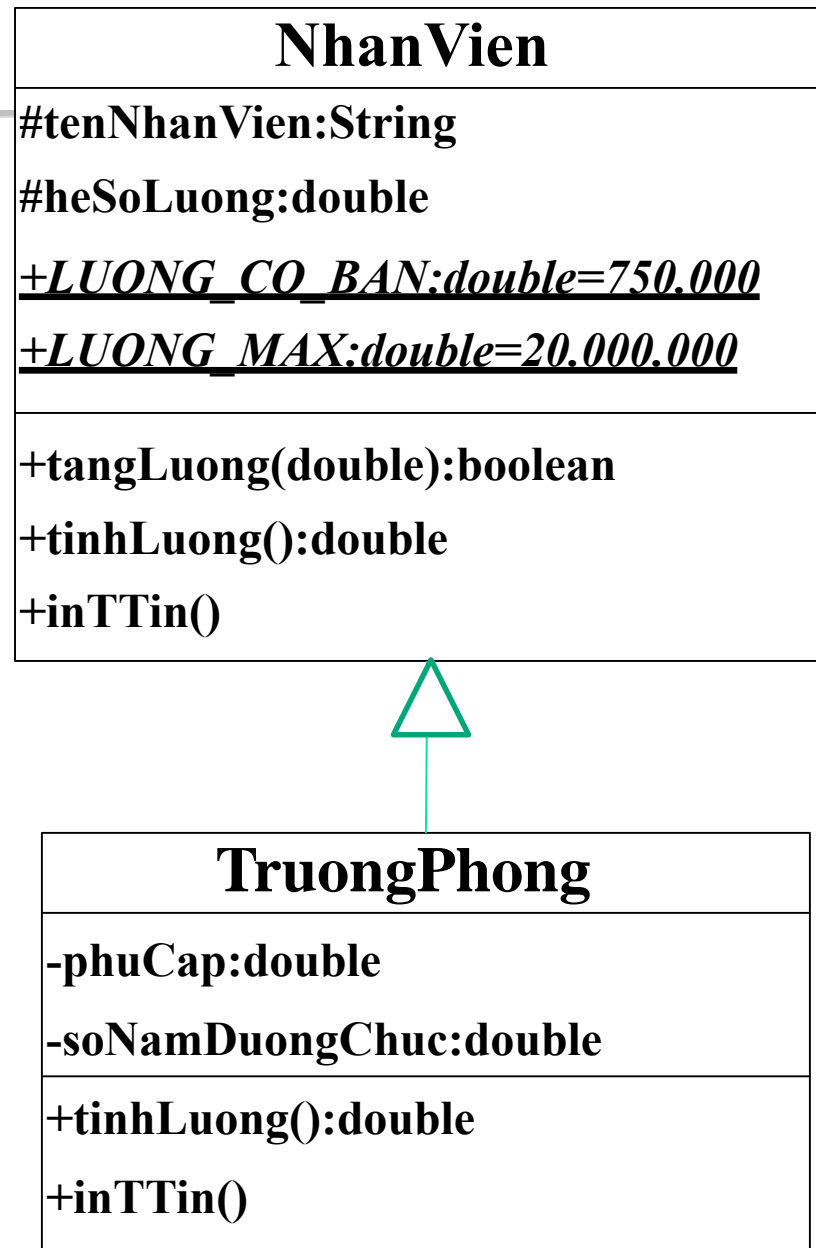
Tổng kết

- ✓ Ghi đè
 - ✓ Các phương thức ở lớp con có cùng chữ ký và danh sách tham số với phương thức ở lớp cha, được tạo ra để định nghĩa lại các hành vi ở lớp con
- ✓ Lớp trừu tượng
 - ✓ Các lớp không được khởi tạo đối tượng, được tạo ra làm lớp cơ sở cho các lớp con định nghĩa rõ hơn
 - ✓ Có ít nhất một phương thức trừu tượng
- ✓ Giao diện
 - ✓ Định nghĩa các phương thức mà lớp thực thi phải cài đặt
 - ✓ Giải quyết vấn đề đa kế thừa

Bài tập

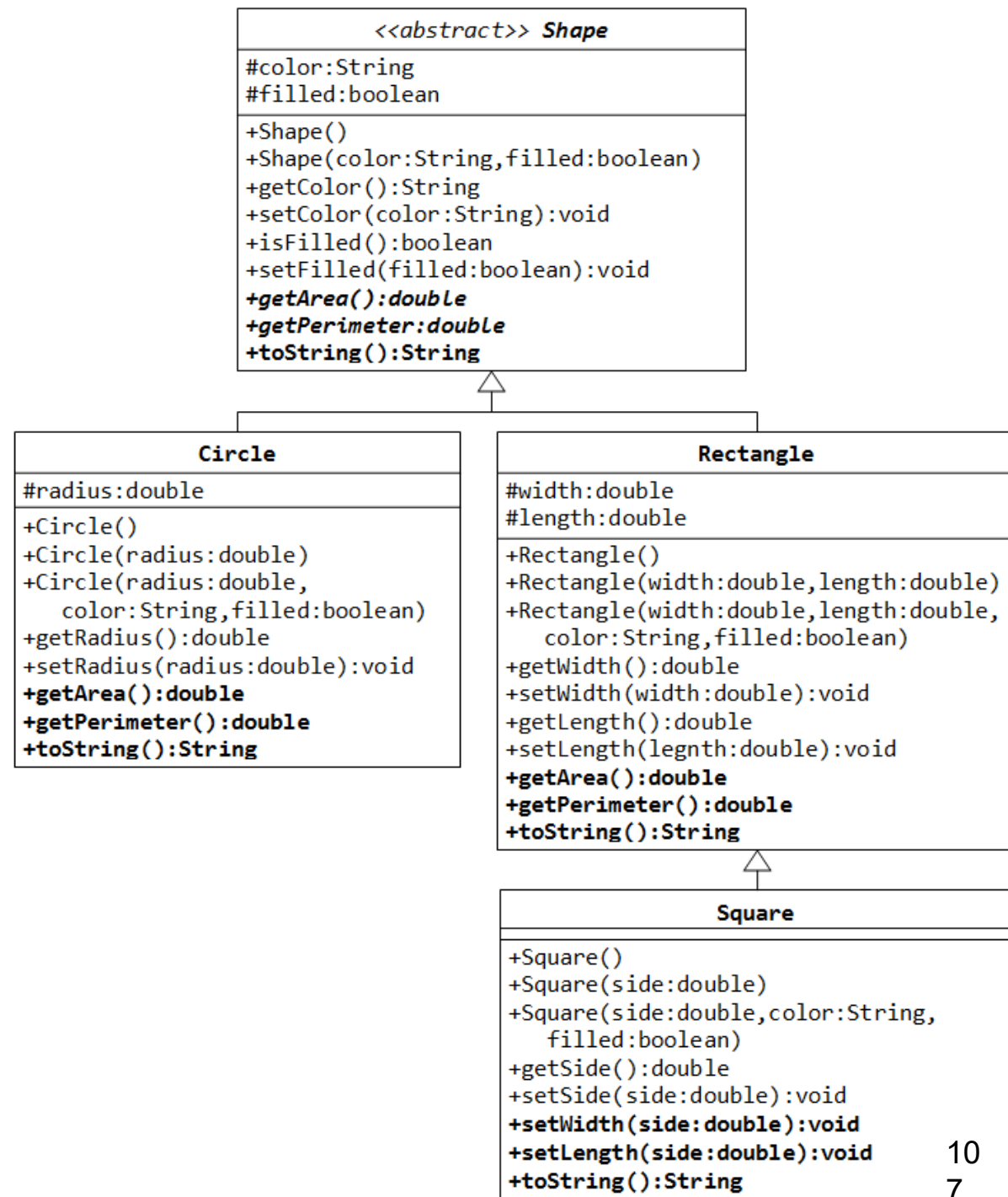
1

- Sửa lại lớp NhanVien:
 - 3 thuộc tính không hằng của NhanVien kế thừa lại cho lớp TruongPhong
- Viết mã nguồn của lớp TruongPhong như hình vẽ
 - Viết các phương thức khởi tạo cần thiết để khởi tạo các thuộc tính của lớp TruongPhong
 - Lương của trưởng phòng = Lương Cơ bản * hệ số lương + phụ cấp



Bài tập 2

- Xây dựng các lớp theo sơ đồ lớp



Môn: Lập trình Hướng đối tượng (Object Oriented Programming)

Chương 6. Đa hình



Mục tiêu

- ✓ Giới thiệu về upcasting và downcasting
- ✓ Phân biệt liên kết tĩnh và liên kết động
- ✓ Nắm vững kỹ thuật đa hình
- ✓ Ví dụ và bài tập về các vấn đề trên với ngôn ngữ lập trình Java



Nội dung

1. Upcasting và Downcasting
2. Liên kết tĩnh và Liên kết động
3. Đa hình (Polymorphism)
4. Ví dụ và bài tập



1. Upcasting và Downcasting

- Chuyển đổi kiểu dữ liệu nguyên thủy
 - Java tự động chuyển đổi kiểu khi
 - Kiểu dữ liệu tương thích
 - Chuyển đổi từ kiểu hẹp hơn sang kiểu rộng hơn

int i;
double d = i;

 - Phải ép kiểu khi
 - Kiểu dữ liệu tương thích
 - Chuyển đổi từ kiểu rộng hơn sang kiểu hẹp hơn
- int i;**
~~**byte b = i;**~~ **byte b = (byte)i;**

1. Upcasting và Downcasting

- Chuyển đổi kiểu dữ liệu tham chiếu

- Kiểu dữ liệu tham chiếu có thể được chuyển đổi kiểu khi

- Kiểu dữ liệu tham chiếu (lớp) *tương thích*

- Nằm trên cùng một cây phân cấp kế thừa

A var1 = **new** B();

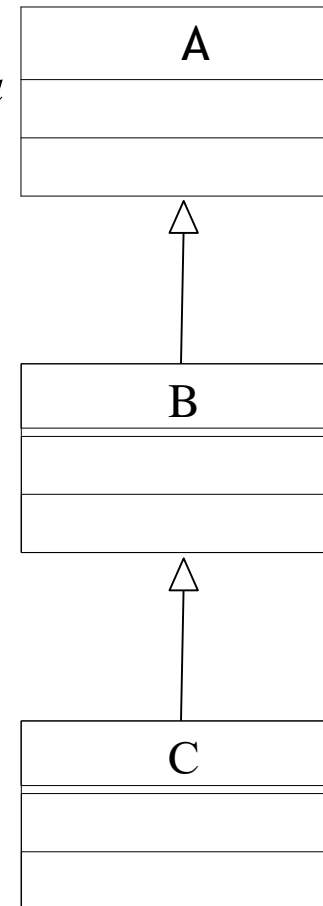
A var1 = **new** A();

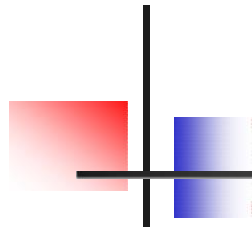
C var2 = (C)var1;

- Hai cách chuyển đổi

- Up-casting

- Down-casting





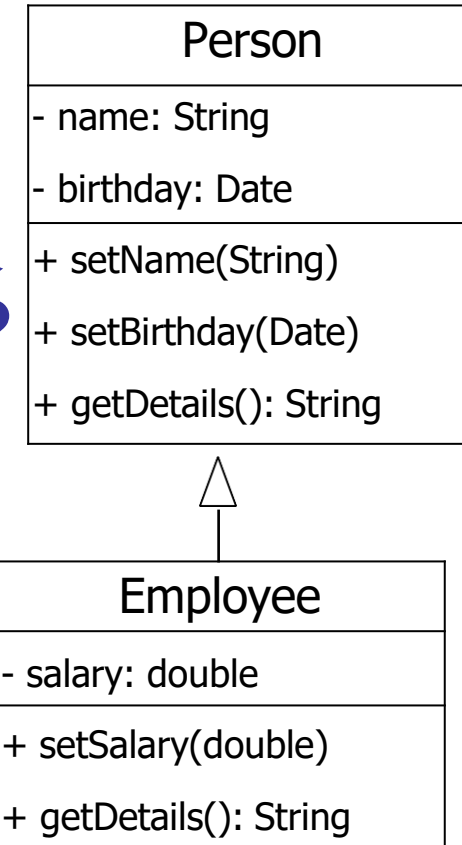
1.1 Upcasting

- Up casting: đi lên trên cây phân cấp thừa kế (moving up the inheritance hierarchy)
- Up casting là khả năng nhìn nhận đối tượng thuộc lớp dẫn xuất như là một đối tượng thuộc lớp cơ sở.
- Tự động chuyển đổi kiểu

1.1 Upcasting

- Ví dụ:

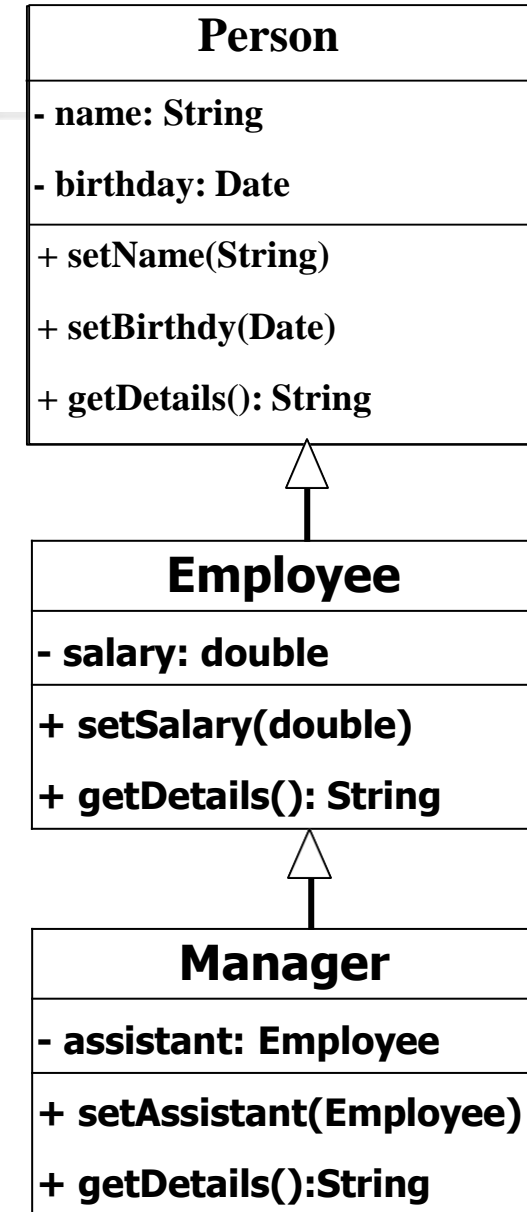
```
public class Test1 {  
    public static void main(String arg[]) {  
        Employee e = new Employee();  
        Person p;  
        p = e;  
        p.setName("Hoa");  
        p.setSalary(350000);  
        // compile error  
    }  
}
```



1.1 Upcasting

■ Ví dụ:

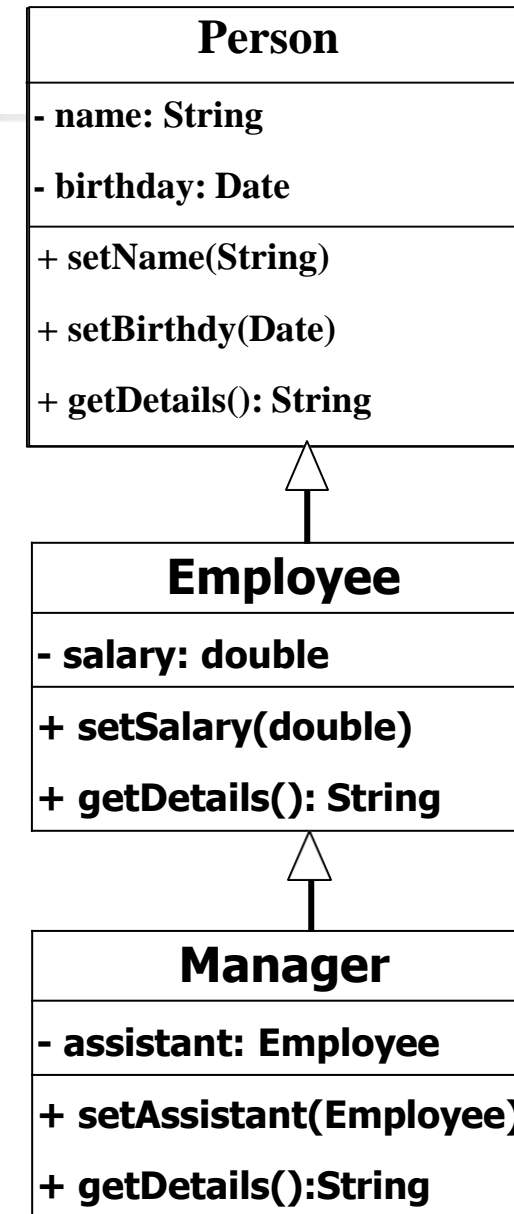
```
class Manager extends Employee {  
    Employee assistant;  
    // ...  
  
    public void setAssistant(Employee e) {  
        assistant = e;  
    }  
    // ...  
}  
  
public class Test2 {  
    public static void main(String arg[]) {  
        Manager junior, senior;  
        // ...  
        senior.setAssistant(junior);  
    }  
}
```



1.1 Upcasting

■ Ví dụ:

```
public class Test3 {  
    String static teamInfo(Person p1, Person p2) {  
        return "Leader: " + p1.getName() + ",  
            member: " + p2.getName();  
    }  
    public static void main(String arg[]) {  
        Employee e1, e2;  
        Manager m1, m2;  
        // ...  
        System.out.println(teamInfo(e1, e2));  
        System.out.println(teamInfo(m1, m2));  
        System.out.println(teamInfo(m1, e2));  
    }  
}
```





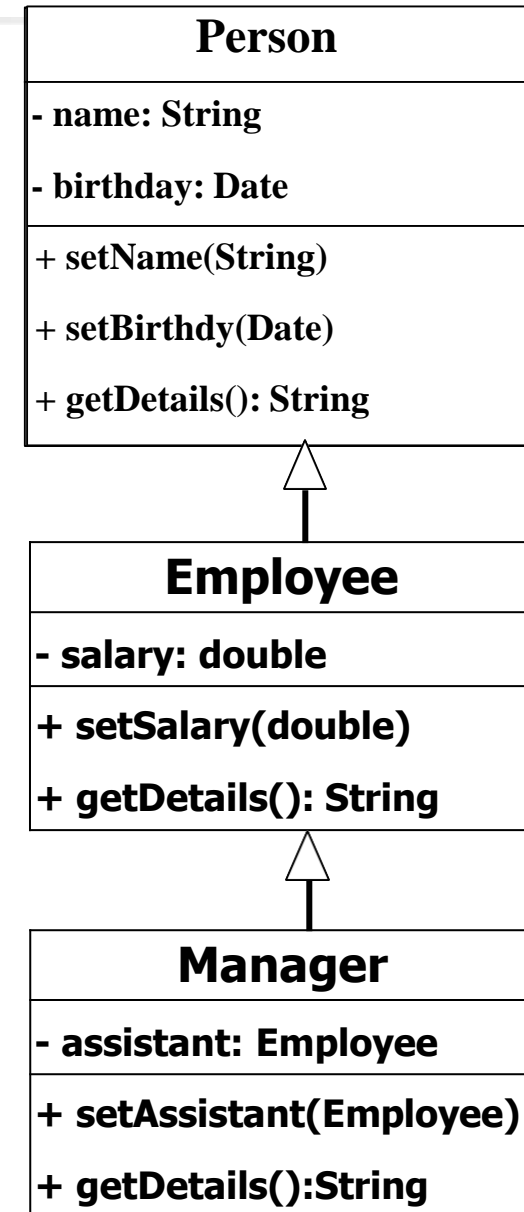
1.2 Downcasting

- ✓ Down casting: đi xuống cây phân cấp thừa kế (move back down the inheritance hierarchy)
- ✓ Down casting là khả năng nhìn nhận một đối tượng thuộc lớp cơ sở như một đối tượng thuộc lớp dẫn xuất.
- ✓ Không tự động chuyển đổi kiểu
→ Phải ép kiểu.

1.2 Downcasting

- Ví dụ:

```
public class Test2 {  
    public static void main(String arg[]) {  
        Employee e = new Employee();  
        Person p = e; // up casting  
        Employee e1 = (Employee) p;  
        // down casting  
        Manager m = (Manager) e;  
        // run-time error  
        Person p2 = new Manager();  
        Employee e2 = (Employee) p2;  
    }  
}
```





Toán tử instanceof

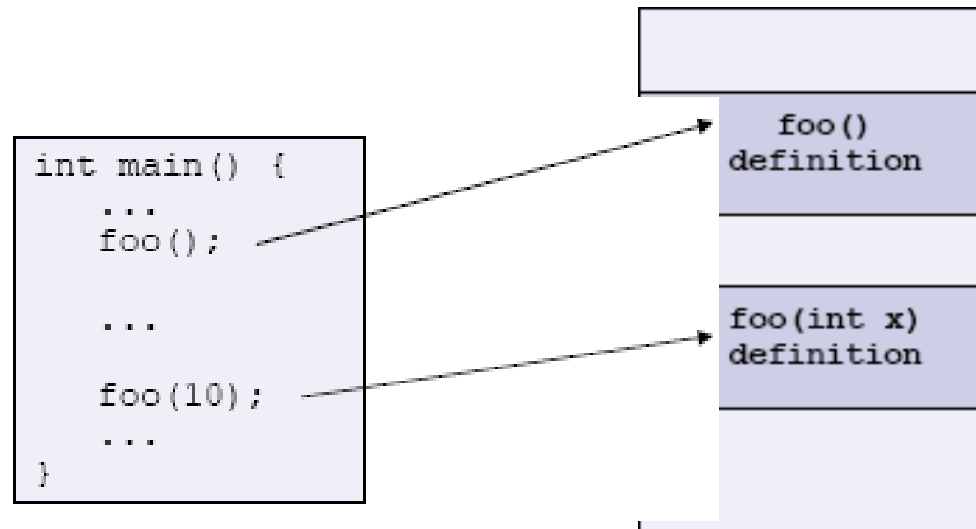
- ✓ Kiểm tra xem một đối tượng có phải là thể hiện của một lớp nào đó không
- ✓ Trả về: true | false (nếu đối tượng là null thì trả về false)

```
public class Employee extends Person {}  
public class Student extends Person {}
```

```
public class Test{  
    public doSomething(Person e) {  
        if (e instanceof Employee) {...  
        }  
        else if (e instanceof Student) {...  
        }  
        else {...}  
    }  
}
```

Liên kết lời gọi hàm

- ✓ Liên kết lời gọi hàm (function call binding) là quy trình xác định khối mã hàm cần chạy khi một lời gọi hàm được thực hiện
 - ✓ C: đơn giản vì mỗi hàm có duy nhất một tên
 - ✓ C++: chồng hàm, phân tích chữ ký kiểm tra danh sách tham số

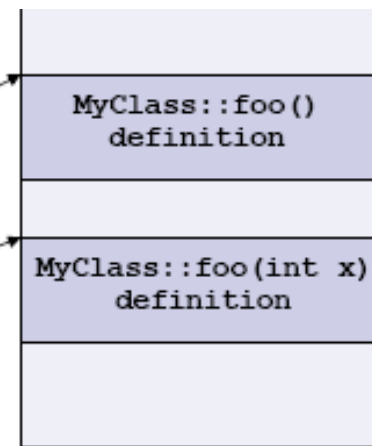


Trong ngôn ngữ Hướng đối tượng

- ✓ Liên kết lời gọi phương thức
- ✓ Đối với các lớp độc lập (không thuộc cây thừa kế nào), quy trình này gần như không khác với function call binding
 - ✓ so sánh tên phương thức, danh sách tham số để tìm định nghĩa tương ứng
 - ✓ một trong số các tham số là tham số ẩn: con trỏ this

bar.foo (); → lời gọi này bị ràng buộc với định nghĩa của phương thức mà nó gọi

```
int main() {  
    ...  
    MyClass bar;  
    ...  
    bar.foo();  
    ...  
    bar.foo(10);  
    ...  
}
```





2.1 Liên kết tĩnh

- ✓ Liên kết tại thời điểm biên dịch
 - ✓ Early Binding/Compile-time Binding
 - ✓ Lời gọi phương thức được quyết định khi biên dịch, do đó chỉ có một phiên bản của phương thức được thực hiện
 - ✓ Nếu có lỗi thì sẽ có lỗi biên dịch
 - ✓ Ưu điểm về tốc độ
- ✓ C/C++ function call binding, và C++ method binding cơ bản đều là ví dụ của liên kết tĩnh (static function call binding)



2.1 Liên kết tĩnh

- ✓ Thích hợp cho các lời gọi hàm thông thường
 - ✓ Mỗi lời gọi hàm chỉ xác định duy nhất một định nghĩa hàm, kể cả trường hợp hàm chồng.
- ✓ Phù hợp với các lớp độc lập không thuộc cây thừa kế nào
 - ✓ Mỗi lời gọi phương thức từ một đối tượng của lớp hay từ con trỏ đến đối tượng đều xác định duy nhất một phương thức



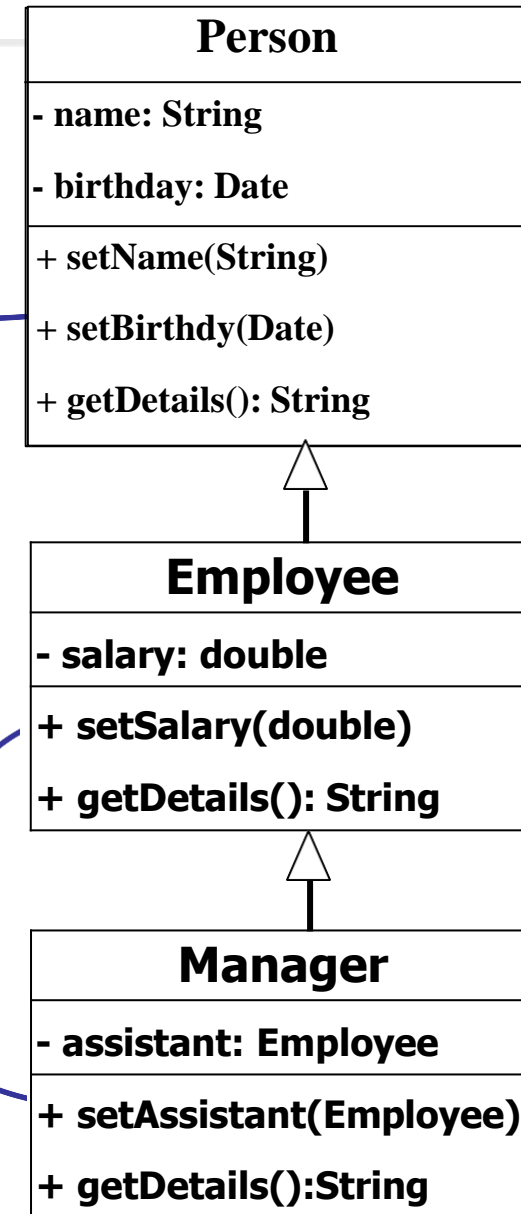
2.2 Liên kết động

- ✓ Lời gọi phương thức được quyết định khi thực hiện (run-time)
 - ✓ Late binding/Run-time binding
 - ✓ Phiên bản của phương thức phù hợp với đối tượng được gọi
 - ✓ Java trì hoãn liên kết phương thức cho đến thời gian chạy (run-time) - đây được gọi là liên kết động hoặc liên kết trễ
 - Java mặc định sử dụng liên kết động

Ví dụ

```
public class Test {  
    public static void main(String arg[]){  
        Person p = new Person();  
        // ...  
        Employee e = new Employee();  
        // ...  
        Manager m = new Manager();  
        // ...  
        Person pArr[] = {p, e, m};  
        for (int i=0; i< pArr.length; i++){  
            System.out.println(  
                pArr[i].getDetail());  
        }  
    }  
}
```

Tùy thuộc vào đối tượng
gọi tại thời điểm thực thi
chương trình (run-time)



Câu hỏi

- Giả sử lớp Sub kế thừa từ lớp cha Sandwich. Tạo hai đối tượng từ các lớp này:

Sandwich x = new Sandwich();

Sub y = new Sub();

- Phép gán nào sau đây là hợp lệ?

1. **x = y;**

2. **y = x;**

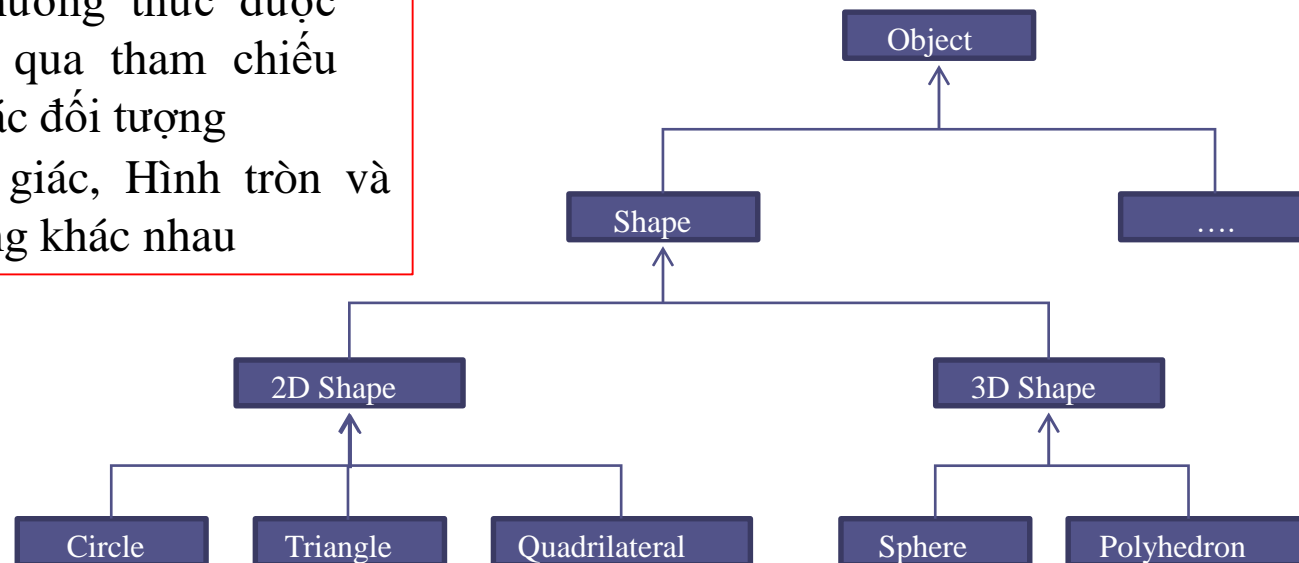
3. **y = new Sandwich ();**

4. **x = new Sub ();**

3. Đa hình

- ✓ Ví dụ: Một hoạt động có thể được thực hiện trên một đối tượng 2DShape cũng có thể được thực hiện trên một đối tượng thuộc một trong ba lớp Tam giác, Hình tròn, Tứ giác.
- ✓ Lớp cha 2DShape định nghĩa giao diện chung
- ✓ Các lớp con Tam giác, Vòng tròn, Tứ giác phải theo giao diện này (kế thừa), nhưng cũng được phép cung cấp các triển khai riêng của chúng (ghi đè)

→ Khi một phương thức được yêu cầu thông qua tham chiếu lớp 2DShape, các đối tượng 2DShape, Tam giác, Hình tròn và Tứ giác phản ứng khác nhau



3. Đa hình

Ví dụ:

```
public class 2DShape {  
    public void display() {  
        System.out.println("2D Shape");  
    }  
}
```

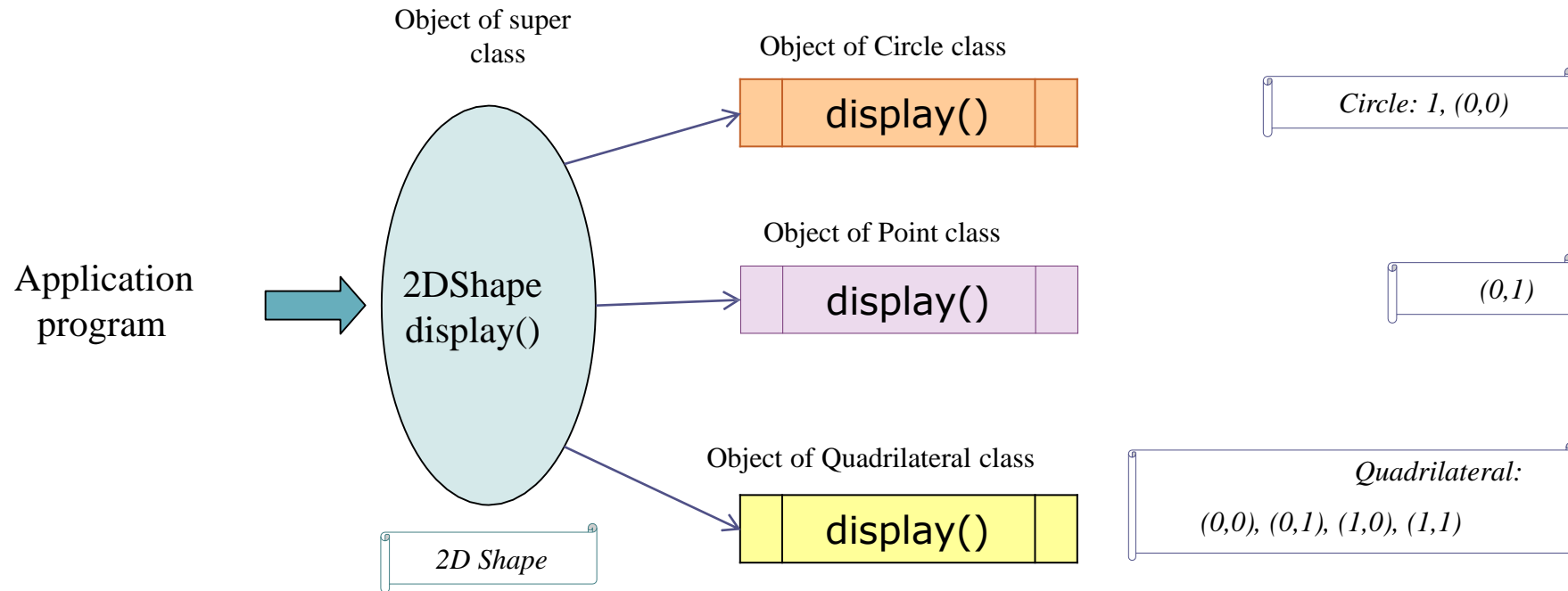
```
public class Point extends 2DShape {  
    private int x, y;  
    ...  
    public void display(){  
        System.out.print("(" + x + "," + y + ")");  
    }  
}
```

```
public class Circle extends 2DShape{  
    public static final double PI = 3.14159;  
    private Point p;  
    private double r; //radius  
  
    ...  
    public void display(){  
        System.out.print("Circle: " + r + ",");  
        p.display();  
        System.out.println();  
    }  
}
```

```
public class Quadrilateral extends 2DShape { private  
    Point p1, p2, p3, p4;  
  
    ....  
  
    public void display(){  
        System.out.println("Quadrilateral: ");  
        p1.display(); p2.display();  
        p3.display(); p4.display();  
        System.out.println();  
    }  
}
```


3. Đa hình

- Ví dụ: Có nhiều sự lựa chọn một khi một phương thức được gọi thông qua một tham chiếu lớp cha.





3. Đa hình

- ✓ Polymorphism: Nhiều hình thức thực hiện, nhiều kiểu tồn tại
 - ✓ Khả năng của một biến tham chiếu thay đổi hành vi theo đối tượng mà nó đang giữ.
- ✓ Đa hình trong lập trình
 - ✓ Đa hình phương thức:
 - ✓ Phương thức trùng tên, phân biệt bởi danh sách tham số.
 - ✓ Đa hình đối tượng
 - ✓ Nhìn nhận đối tượng theo nhiều kiểu khác nhau
 - ✓ Các đối tượng khác nhau cùng đáp ứng chung danh sách các thông điệp có giải nghĩa thông điệp theo cách thức khác nhau.



3. Đa hình

- ✓ Polymorphism: gia tăng khả năng tái sử dụng những đoạn mã nguồn được viết một cách tổng quát và có thể thay đổi cách ứng xử một cách linh hoạt tùy theo loại đối tượng
 - ✓ Tính đa hình (*Polymorphism*) trong Java được hiểu là trong từng trường hợp, hoàn cảnh khác nhau thì đối tượng có hình thái khác nhau tùy thuộc vào từng ngữ cảnh
- ✓ Để thể hiện tính đa hình:
 - ✓ Các lớp phải có quan hệ kế thừa với 1 lớp cha nào đó
 - ✓ Phương thức được ghi đè (override) ở lớp con

3. Đa hình

- ✓ Ví dụ:
- ✓ Các đối tượng khác nhau giải nghĩa các thông điệp theo các cách thức khác nhau

■ Liên kết động (Java)

```
Person p1 = new Person();
```

```
Person p2 = new Employee();
```

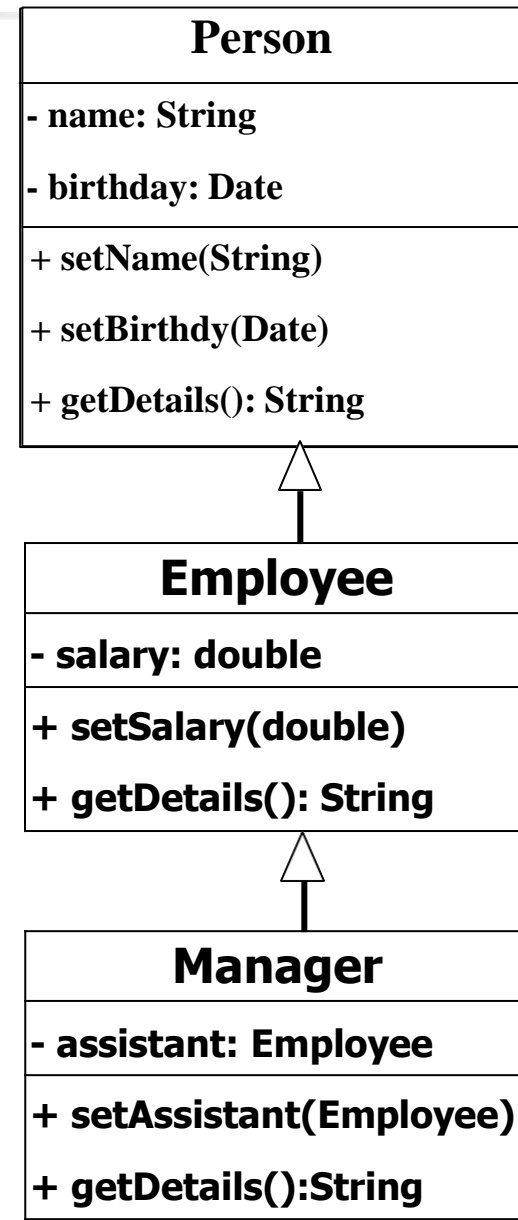
```
Person p3 = new Manager();
```

```
// ...
```

```
System.out.println(p1.getDetail());
```

```
System.out.println(p2.getDetail());
```

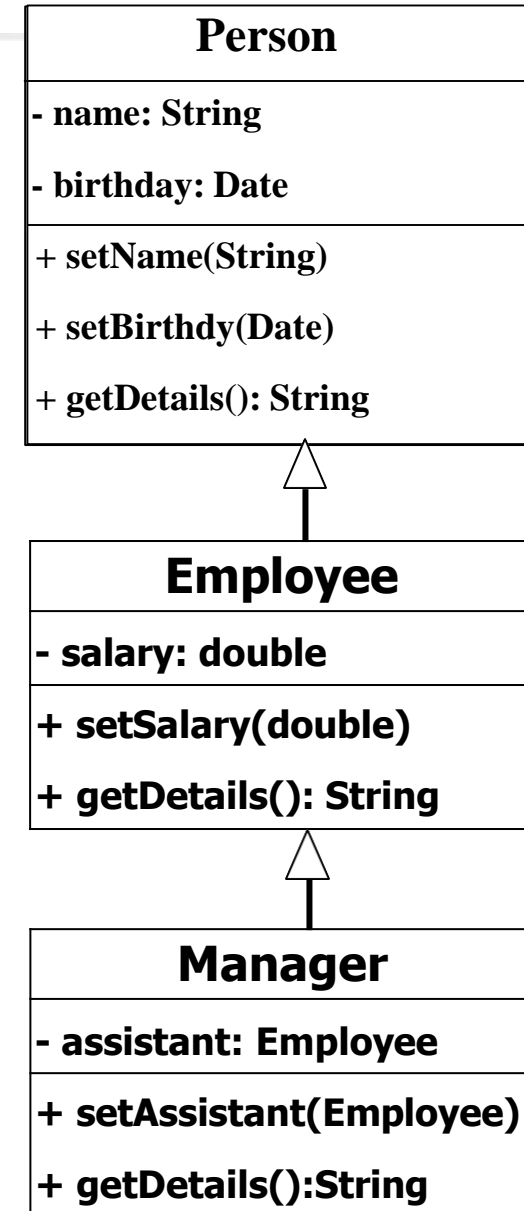
```
System.out.println(p3.getDetail());
```



3. Đa hình

■ Ví dụ:

```
class EmployeeList {  
    Employee list[];  
    ...  
    public void add(Employee e) {...}  
    public void print() {  
        for (int i=0; i<list.length; i++)  
        {  
            System.out.println(list[i].getDetail());  
        }  
    }  
  
    EmployeeList list = new EmployeeList();  
    Employee e1; Manager m1;  
    ...  
    list.add(e1);  
    list.add(m1);  
    list.print();  
}
```



3. Đa hình

- Ví dụ: Các đối tượng Triangle, Rectangle, Circle đều là các đối tượng Shape

...

```
public static void handleShapes(Shape[] shapes){
```

```
    // Vẽ các hình theo cách riêng của mỗi hình
```

```
    for( int i = 0; i < shapes.length; ++i) {
```

```
        shapes[i].draw();
```

```
    }
```

...

```
    // Gọi đến phương thức xóa,
```

```
    // không cần quan tâm đó là hình gì
```

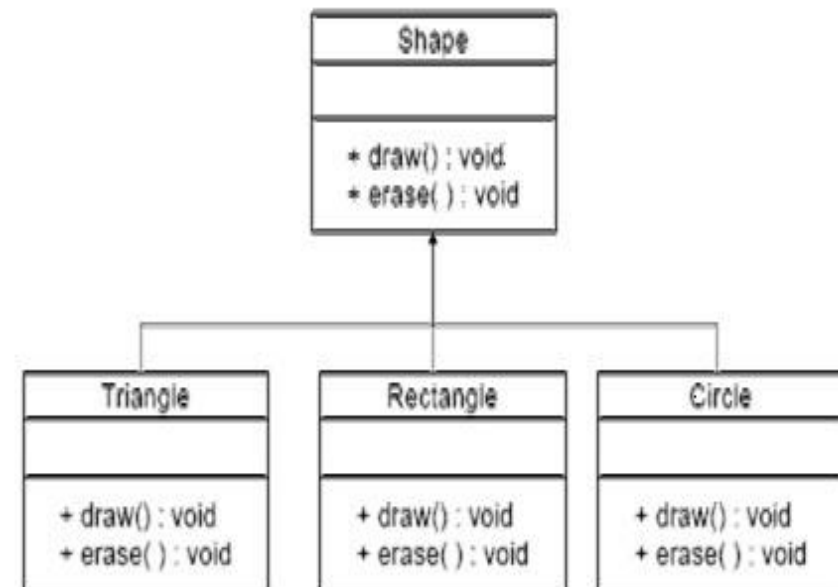
```
    for( int i = 0; i < shapes.length; ++i) {
```

```
        shapes[i].erase();
```

```
    }
```

```
}
```

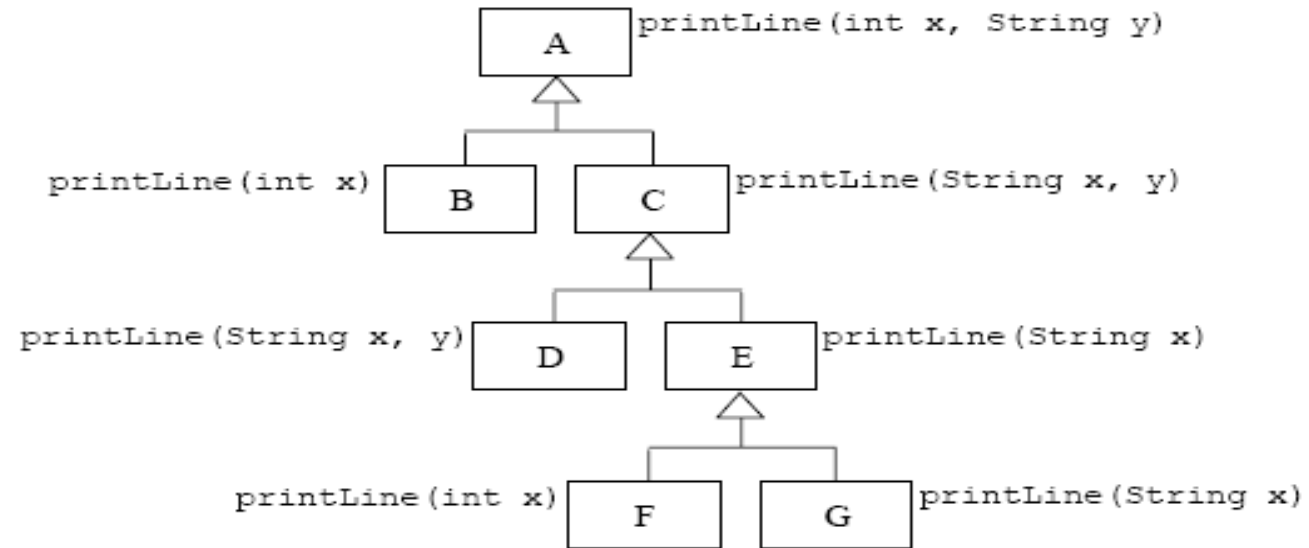
...



Câu hỏi



- Cho biểu đồ lớp:

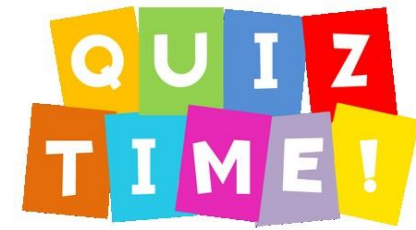


Phương thức **printLine()** của lớp nào sẽ được sử dụng trong mỗi trường hợp dưới đây, biết rằng **z** là một đối tượng của lớp **F**? Giải thích ngắn gọn?

1. **z.printLine(1)**
2. **z.printLine(2, "Object-Oriented Programming")**
3. **z.printLine("Java")**
4. **z.printLine("Object-Oriented Programming", "Java")**
5. **z.printLine("Object-Oriented Programming", 3)**



Câu hỏi



- Những điều kiện nào trả về **true**? (Có thể xem Java documentation để biết các quan hệ thừa kế giữa các lớp)
Biết rằng System.out là một đối tượng của lớp PrintStream.
 1. `System.out instanceof PrintStream`
 2. `System.out instanceof OutputStream`
 3. `System.out instanceof LogStream`
 4. `System.out instanceof Object`
 5. `System.out instanceof String`
 6. `System.out instanceof Writer`



Tổng kết

- ✓ Upcasting và downcasting
 - ✓ Nhìn nhận các đối tượng thuộc lớp cơ sở như đối tượng thuộc lớp dẫn xuất (upcasting) và ngược lại (down-casting)
- ✓ Liên kết tĩnh và liên kết động
 - ✓ Liên kết lời gọi hàm lúc biên dịch (liên kết tĩnh) hay lúc chạy chương trình (liên kết động)
- ✓ Đa hình
 - ✓ Nhìn nhận một đối tượng dưới nhiều kiểu khác nhau



Bài tập 1

- Kiểm tra các đoạn mã sau đây và vẽ sơ đồ lớp tương ứng

```
abstract public class Animal {  
    abstract public void greeting();  
}  
public class Cat extends Animal {  
    public void greeting() {  
        System.out.println("Meow!");  
    }  
}  
public class Dog extends Animal {  
    public void greeting() {  
        System.out.println("Woof!");  
    }  
}
```

```
public void greeting(Dog another) {  
    System.out.println("Woooooooooooooof!");  
}  
}  
public class BigDog extends Dog {  
    public void greeting() {  
        System.out.println("Woow!");  
    }  
    public void greeting(Dog another) {  
        System.out.println("Woooooooooowwwww!");  
    }  
}
```

Bài tập 2

- Giải thích các đầu ra (hoặc các lỗi nếu có) cho chương trình thử nghiệm sau:

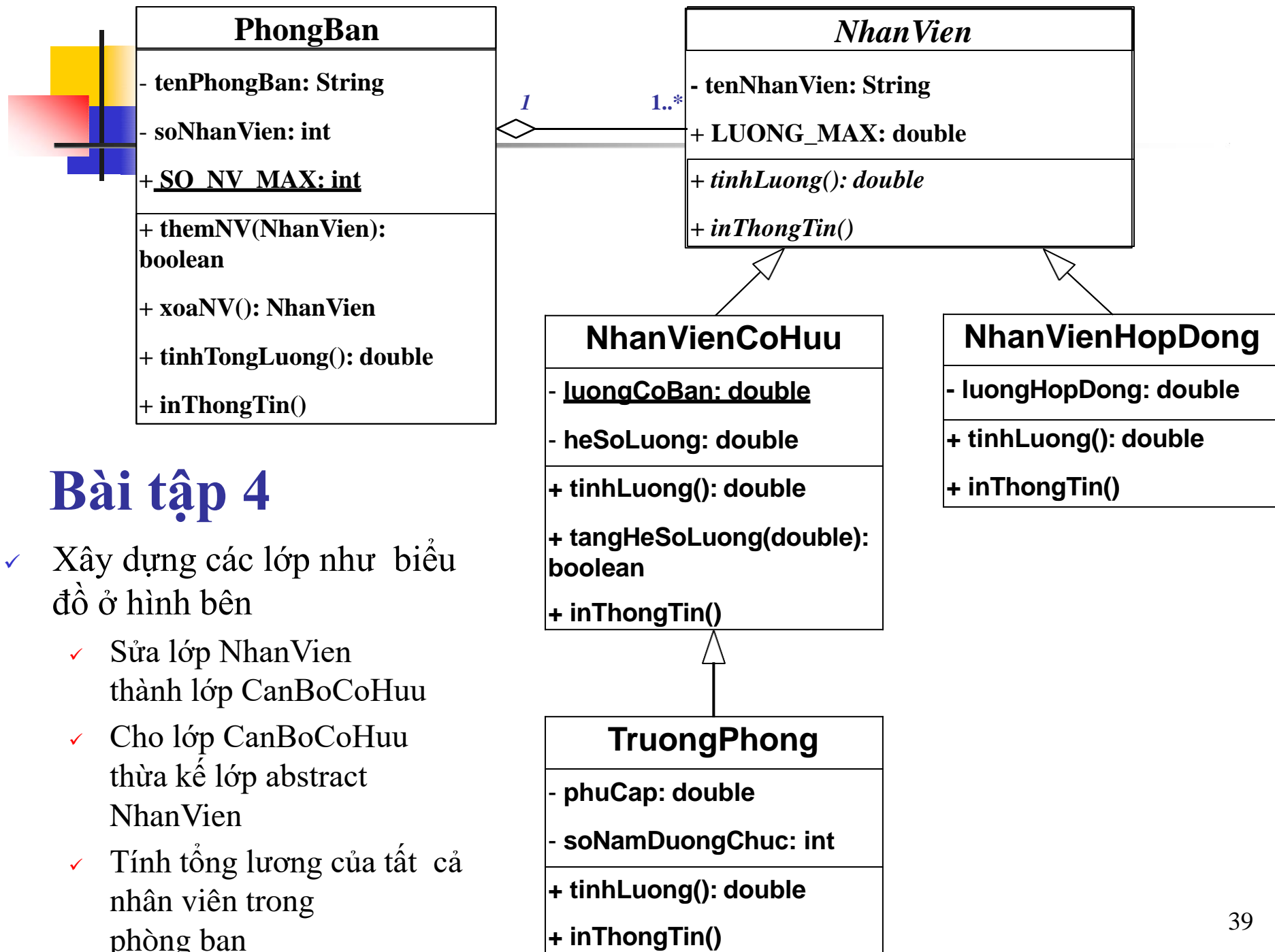
```
public class TestAnimal {  
    public static void main(String[] args) {  
        // Using the subclasses  
        Cat cat1 = new Cat();  
        cat1.greeting();  
        Dog dog1 = new Dog();  
        dog1.greeting();  
        BigDog bigDog1 = new BigDog();  
        bigDog1.greeting();  
  
        // Using Polymorphism  
        Animal animal1 = new Cat();  
        animal1.greeting();  
        Animal animal2 = new Dog();  
        animal2.greeting();  
        Animal animal3 = new BigDog();
```

```
        animal3.greeting();  
        Animal animal4 = new Animal();  
  
        // Downcast  
        Dog dog2 = (Dog)animal2;  
        BigDog bigDog2 = (BigDog)animal3;  
        Dog dog3 = (Dog)animal3;  
        Cat cat2 = (Cat)animal2;  
        dog2.greeting(dog3);  
        dog3.greeting(dog2);  
        dog2.greeting(bigDog2);  
        bigDog2.greeting(dog2);  
        bigDog2.greeting(bigDog1);  
    }  
}
```



Bài tập 3

- ✓ Phân tích xây dựng các lớp như mô tả sau:
 - ✓ Hàng điện máy <mã hàng, tên hàng, nhà sản xuất, giá, thời gian bảo hành, điện áp, công suất>
 - ✓ Hàng sành sứ < mã hàng, tên hàng, nhà sản xuất, giá, loại nguyên liệu>
 - ✓ Hàng thực phẩm <mã hàng, tên hàng, nhà sản xuất, giá, ngày sản xuất, ngày hết hạn dùng>
- ✓ Viết chương trình tạo mỗi loại một mặt hàng cụ thể. Xuất thông tin về các mặt hàng này.



Bài tập 4

- ✓ Xây dựng các lớp như biểu đồ ở hình bên
 - ✓ Sửa lớp NhanVien thành lớp CanBoCoHuu
 - ✓ Cho lớp CanBoCoHuu thừa kế lớp abstract NhanVien
 - ✓ Tính tổng lương của tất cả nhân viên trong phòng ban

Môn: Lập trình Hướng đối tượng (Object Oriented Programming)

Chương 7. Collections



Mục tiêu

- ✓ Giới thiệu về lập trình tổng quát và cách thực hiện trong các ngôn ngữ lập trình
- ✓ Giới thiệu về collection framework với các cấu trúc tổng quát: List, HashMap, Tree, Set, Vector,...
- ✓ Định nghĩa và sử dụng Template và ký tự đại diện (wildcard)
- ✓ Ví dụ và bài tập về các vấn đề trên với ngôn ngữ lập trình Java



Nội dung

1. Giới thiệu về lập trình tổng quát
2. Lập trình tổng quát trong Java
 - ✓ Giới thiệu về collection framework
 - ✓ Giới thiệu về các cấu trúc tổng quát List, HashMap, Tree, Set, Vector
3. Định nghĩa và sử dụng Template
4. Ký tự đại diện (Wildcard)
5. Ví dụ và bài tập

1. Giới thiệu về lập trình tổng quát

- Tổng quát hóa chương trình để có thể hoạt động với các kiểu dữ liệu khác nhau, kể cả kiểu dữ liệu trong tương lai

- Thuật toán đã xác định

**Tổng quát hoá
chương trình**

- Ví dụ:

Phương thức **sort()**

- Số nguyên int
- Xâu ký tự String
- Đối tượng số phức
Complex object

**Thuật toán giống nhau, chỉ
khác về kiểu dữ liệu**

Lớp lưu trữ kiểu
ngăn xếp (Stack)

- • ...
- Lớp IntegerStack → đối tượng Integer
- Lớp StringStack → đối tượng String
- Lớp AnimalStack → đối tượng animal,...

**Các lớp có cấu
trúc tương tự,
khác nhau về
kiểu đối tượng
xử lý**

1. Giới thiệu về lập trình tổng quát

- ✓ Lập trình Generic có nghĩa là lập trình mà có thể tái sử dụng cho nhiều kiểu dữ liệu
 - ✓ Cho phép trừu tượng hóa kiểu dữ liệu
- ✓ Giải pháp trong các ngôn ngữ lập trình:
 - ✓ C: dùng con trỏ không định kiểu (con trỏ void)
 - ✓ C++: dùng template
 - ✓ Java 1.5 trở về trước: lợi dụng upcasting và kiểu tổng quát object
 - ✓ Java 1.5: đưa ra khái niệm về template

1. Giới thiệu về lập trình tổng quát

- Ví dụ C: hàm `memcpy()` trong thư viện `string.h`

```
void* memcpy(void* region1, const void* region2, size_t n);
```

- ✓ Hàm `memcpy()` bên trên được khai báo tổng quát bằng cách sử dụng các con trỏ `void*`
- ✓ Điều này giúp cho hàm có thể sử dụng với nhiều kiểu dữ liệu khác nhau
 - ✓ Dữ liệu được truyền vào một cách tổng quát thông qua địa chỉ và kích thước kiểu dữ liệu
 - ✓ Hay nói cách khác, để sao chép dữ liệu, ta chỉ cần địa chỉ và kích cỡ của chúng

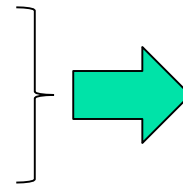
1. Giới thiệu về lập trình tổng quát

- Ví dụ: Lập trình Generic từ trước Java 1.5

```
public class ArrayList {  
    public Object get(int i) { ... }  
    public void add(Object o) { ... }  
    ...  
    private Object[] elementData;  
}
```

- Lớp Object là lớp cha tổng quát nhất → có thể chấp nhận các đối tượng thuộc lớp con của nó

```
List myList = new ArrayList();  
myList.add("Fred");  
myList.add(new Dog());  
myList.add(new Integer(42));
```



Các đối tượng
trong một danh
sách khác hẳn
nhau

- Hạn chế: Phải ép kiểu => có thể ép sai kiểu (run-time error)

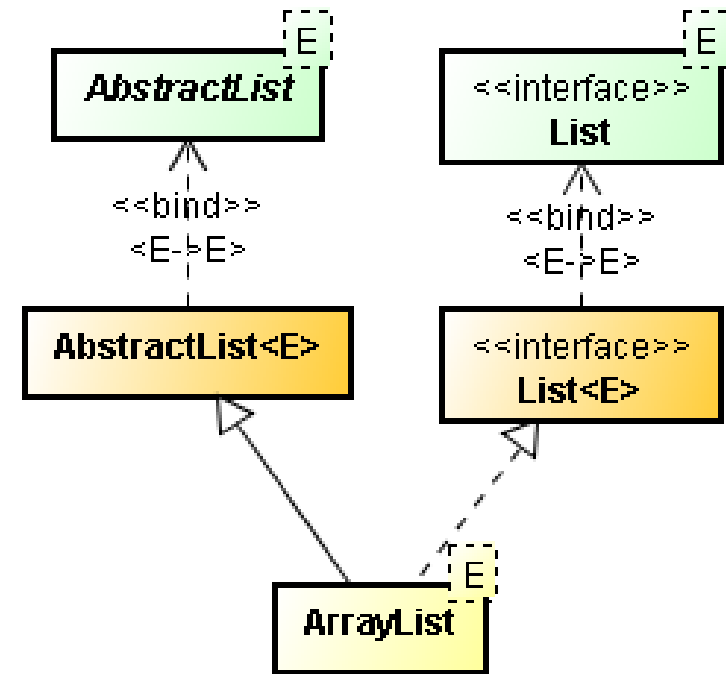
```
String name = (String) myList.get(1); //Dog!!!
```

1. Giới thiệu về lập trình tổng quát

- Ví dụ: Lập trình Generic từ Java 1.5
 - Java 1.5 Template

Danh sách chỉ chấp nhận
các đối tượng có kiểu là
Integer

```
List<Integer> myList =  
    new LinkedList<Integer>();  
myList.add(new Integer(0));  
Integer x = myList.iterator().next(); //Không cần ép kiểu  
myList.add(new String("Hello")); //Compile Error
```



2. Lập trình tổng quát trong Java

- ✓ Collection – tập hợp: Nhóm các đối tượng lại thành một đơn vị duy nhất
- ✓ Java Collections Framework:
 - ✓ Biểu diễn các tập hợp
 - ✓ Cung cấp giao diện tiêu chuẩn cho hầu hết các tập hợp cơ bản
 - ✓ Xây dựng dựa trên
 - ✓ Interface: thể hiện tính chất của các kiểu tập hợp khác nhau như List, Set, Map
 - ✓ Class: các lớp cụ thể thực thi các giao diện
 - ✓ Thuật toán: cài đặt một số thao tác đơn giản, là các phương thức tĩnh để xử lý trên collection như tìm kiếm, sắp xếp...

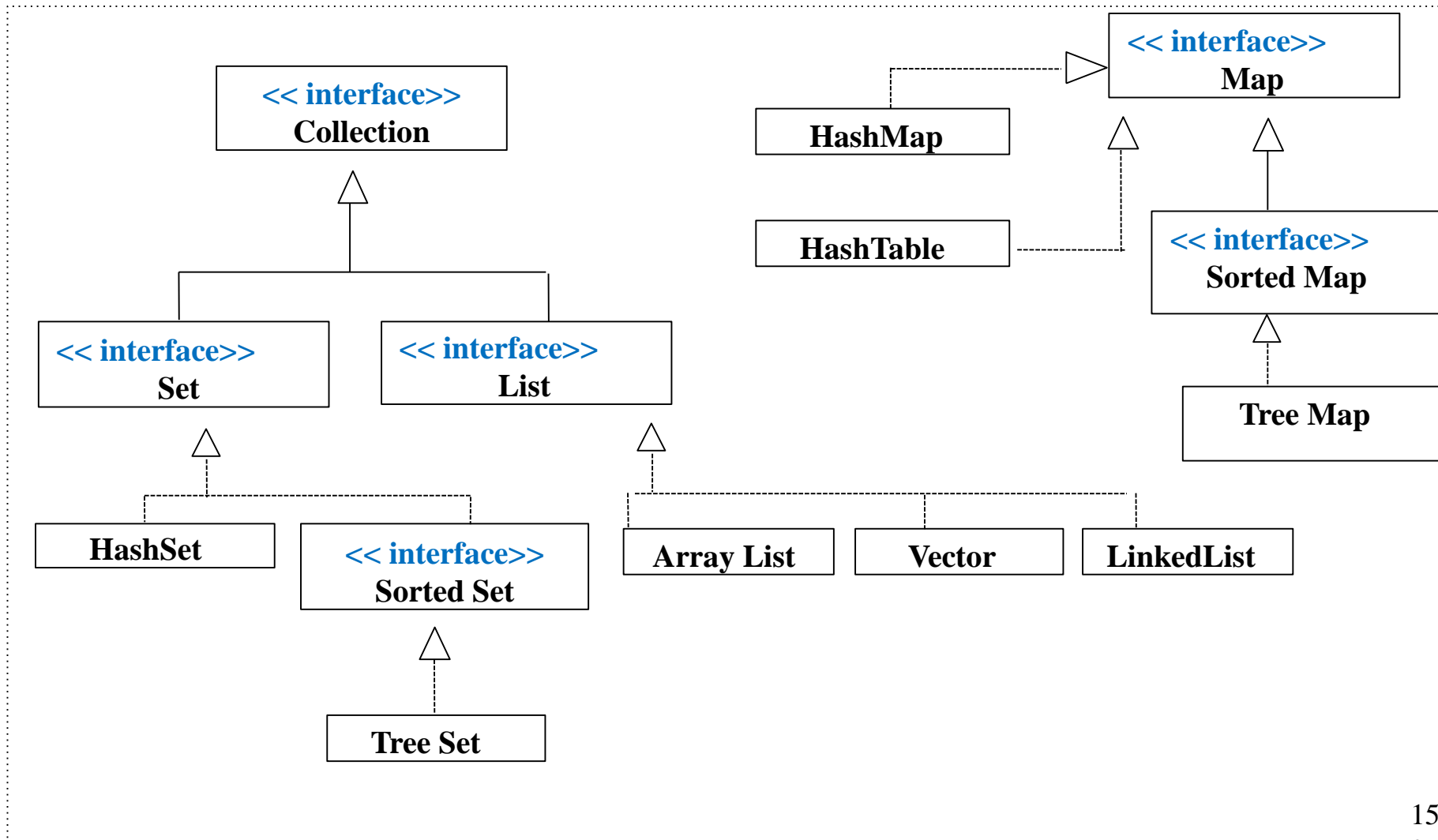
2. Lập trình tổng quát trong Java

- ✓ Java Collections Framework:
 - ✓ List: Tập các đối tượng tuần tự, kế tiếp nhau, có thể lặp lại
 - ✓ Set: Tập các đối tượng không lặp lại
 - ✓ Map: Tập các cặp khóa-giá trị (key-value) và không cho phép khóa lặp lại
 - Liên kết các đối tượng trong tập này với đối các đối tượng trong tập khác như tra từ điển/danh bạ điện thoại.

2. Lập trình tổng quát trong

Java

■ Cây cấu trúc giao diện Collection



2. Lập trình tổng quát trong Java

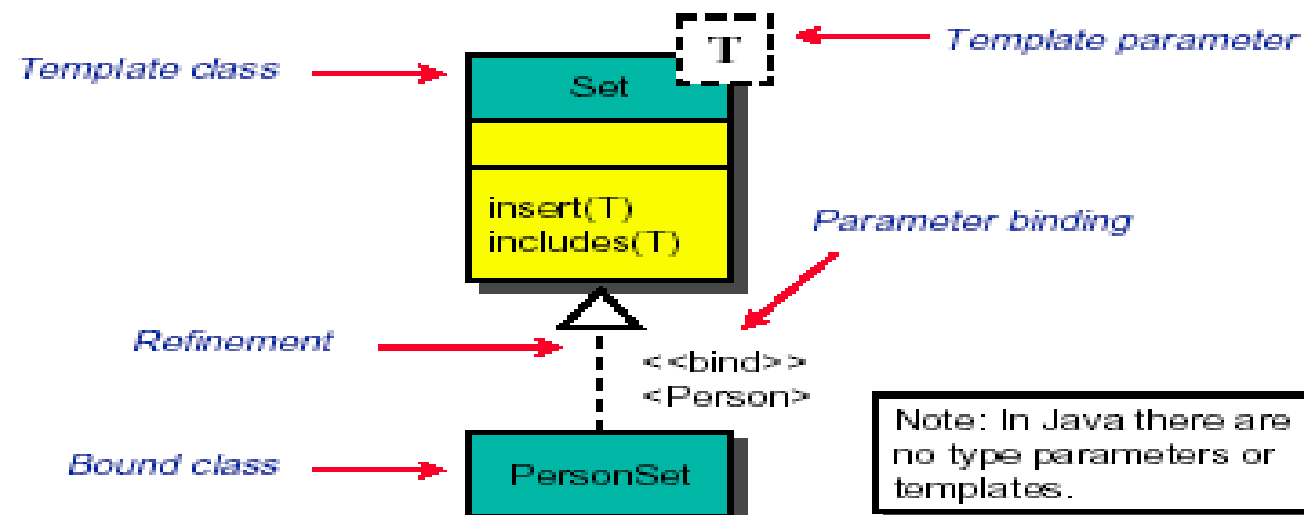
■ So sánh Tập hợp và mảng

Tập hợp	Mảng
Tập hợp (có thể) truy xuất theo dạng ngẫu nhiên	Mảng truy xuất 1 cách tuần tự
Tập hợp có thể chứa nhiều loại đối tượng/dữ liệu khác nhau	Mảng chứa 1 loại đối tượng/dữ liệu nhất định
Dùng theo kiểu tập hợp xây dựng sẵn của Java chỉ khai báo và gọi những phương thức đã được định nghĩa	Dùng tổ chức dữ liệu theo mảng phải lập trình hoàn toàn
Duyệt các phần tử tập hợp thông qua Iterator	Duyệt các phần tử mảng tuần tự thông qua chỉ số mảng

2. Lập trình tổng quát trong

Java

- ✓ Các giao diện và lớp thực thi trong Collection framework của Java đều được xây dựng theo template
 - → cho phép xác định **tập hợp các phần tử cùng kiểu** nào đó bất kỳ
- ✓ Cho phép chỉ định kiểu dữ liệu của các Collection □ hạn chế việc thao tác sai kiểu dữ liệu



2. Lập trình tổng quát trong

Java

- Ví dụ

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

- Ví dụ

```
List<String> myList = new ArrayList<String>();  
myList.add("Fred");           // OK  
myList.add(new Dog());        //Compile error!  
String s = myList.get(0);
```

Giao diện Collection

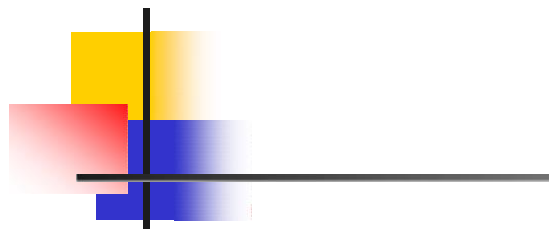
- ✓ Xác định giao diện cơ bản cho các thao tác với một tập các đối tượng
 - ✓ Thêm vào tập hợp
 - ✓ Xóa khỏi tập hợp
 - ✓ Kiểm tra có là thành viên
- ✓ Chứa các phương thức thao tác trên các phần tử riêng lẻ hoặc theo khối
- ✓ Cung cấp các phương thức cho phép thực hiện duyệt qua các phần tử trên tập hợp (lặp) và chuyển tập hợp sang mảng

«Java Interface» I <i>Collection</i>	
●	size () : int
●	isEmpty () : boolean
●	contains (o : Object) : boolean
●	iterator () : Iterator
●	toArray () : Object [*]
●	toArray (a : Object [*]) : Object [*]
●	add (o : Object) : boolean
●	remove (o : Object) : boolean
●	containsAll (c : Collection) : boolean
●	addAll (c : Collection) : boolean
●	removeAll (c : Collection) : boolean
●	retainAll (c : Collection) : boolean
●	clear () : void
●	equals (o : Object) : boolean
●	hashCode () : int

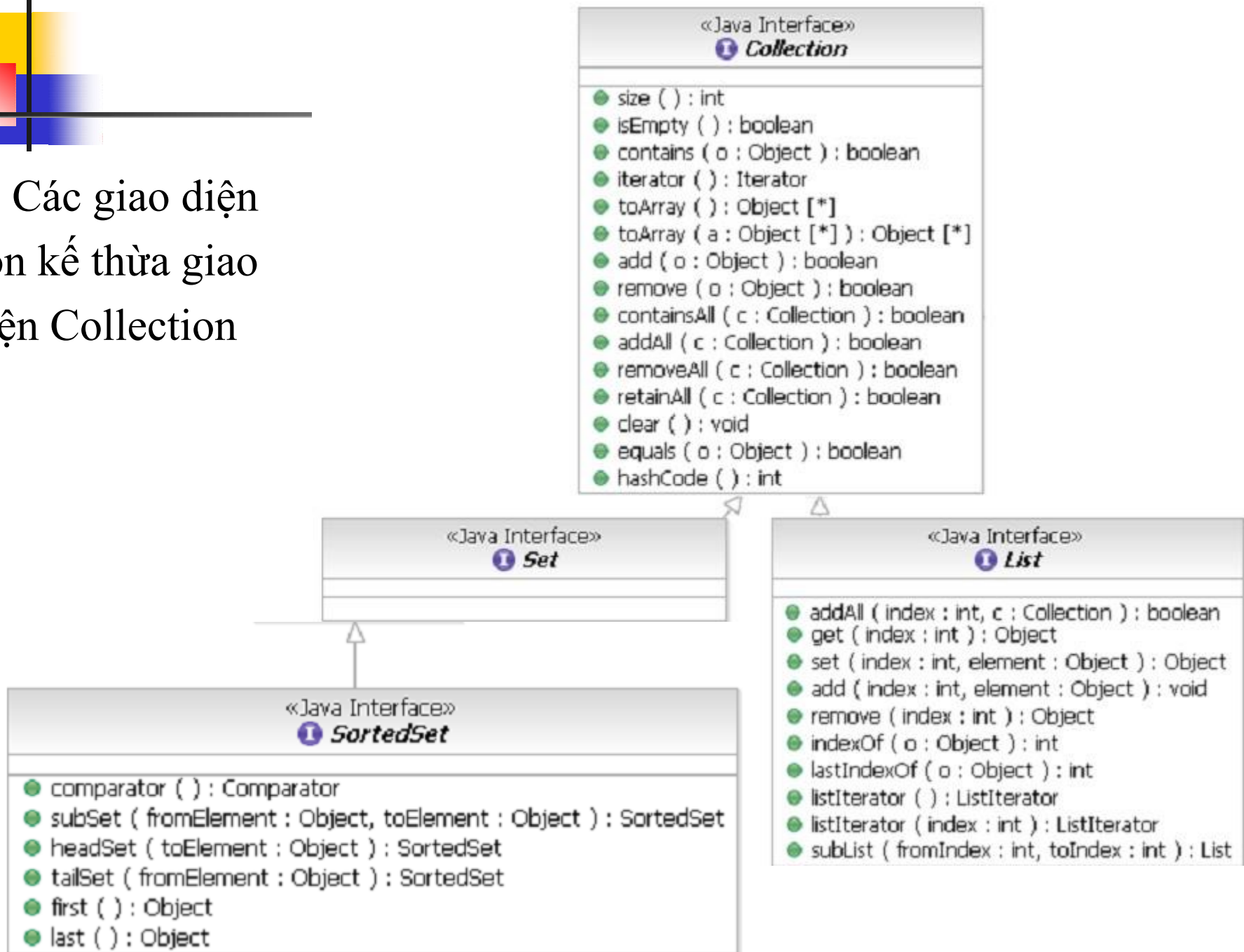


Giao diện Collection

```
public interface Collection {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);  
    boolean remove(Object element);  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean addAll(Collection c);  
    boolean removeAll(Collection c);  
    boolean retainAll(Collection c);  
    ....  
    // Array Operations  
    Object[] toArray();  
    Object[] toArray(Object a[]);  
}
```



■ Các giao diện con kế thừa giao diện Collection



Giao diện

Set

- ✓ Set kế thừa từ Collection, hỗ trợ các thao tác xử lý trên collection kiểu tập hợp
- ✓ Ví dụ:
 - ✓ Set of cars:
 - ✓ {BMW, Ford, Jeep, Chevrolet, Nissan, Toyota, VW}
 - ✓ Nationalities in the class
 - ✓ {Chinese, American, Canadian, Indian}
- ✓ Một tập hợp các phần tử **không được trùng lặp**.
- ✓ Set không có thêm phương thức riêng ngoài các phương thức kế thừa từ Collection.

Giao diện SortedSet

- ✓ SortedSet : kế thừa giao diện **Set**
 - ✓ Các phần tử được sắp xếp theo một thứ tự
 - ✓ Không có các phần tử trùng nhau
 - ✓ Cho phép một phần tử là **null**
 - ✓ Các đối tượng đưa vào trong một SortedSet phải cài đặt giao diện Comparable hoặc lớp cài đặt SortedSet phải nhận một Comparator trên kiểu của đối tượng đó
- ✓ Phương thức: tương tự **Set**, thêm 2 phương thức
 - ✓ **first()**: lấy phần tử đầu tiên (nhỏ nhất)
 - ✓ **last()**: lấy phần tử cuối cùng (lớn nhất)
 - ✓ **SortedSet subSet(Object e1, Object e2)**: lấy một tập các phần tử nằm trong khoảng từ e1 tới e2

Giao diện

List

- ✓ List kế thừa từ Collection, nó cung cấp thêm các phương thức để xử lý collection kiểu danh sách
 - ✓ Danh sách là một collection với các phần tử được xếp theo chỉ số
- ✓ Một số phương thức của List
 - ✓ Object get(int index);
 - ✓ Object set(int index, Object o);
 - ✓ void add(int index, Object o);
 - ✓ Object remove(int index);
 - ✓ int indexOf(Object o);
 - ✓ int lastIndexOf(Object o);

Giao diện Map

- ✓ Xác định giao diện cơ bản để thao tác với một tập hợp bao gồm cặp khóa-giá trị
 - ✓ Thêm một cặp khóa-giá trị
 - ✓ Xóa một cặp khóa-giá trị
 - ✓ Lấy về giá trị với khóa đã có
 - ✓ Kiểm tra có phải là thành viên (khóa hoặc giá trị)
- ✓ Cung cấp 3 cách nhìn cho nội dung của tập hợp:
 - ✓ Tập các khóa
 - ✓ Tập các giá trị
 - ✓ Tập các ánh xạ khóa-giá trị

«Java Interface» I Map	
●	size () : int
●	isEmpty () : boolean
●	containsKey (key : Object) : boolean
●	containsValue (value : Object) : boolean
●	get (key : Object) : Object
●	put (key : Object, value : Object) : Object
●	remove (key : Object) : Object
●	putAll (t : Map) : void
●	clear () : void
●	keySet () : Set
●	values () : Collection
●	entrySet () : Set
●	equals (o : Object) : boolean
●	hashCode () : int

Giao diện

Map

- ✓ Giao diện Map cung cấp các thao tác xử lý trên các bảng ánh xạ
 - ✓ Bảng ánh xạ lưu các phần tử theo khoá và không được có 2 khoá trùng nhau
- ✓ Một số phương thức của Map
 - ✓ `Object put(Object key, Object value);`
 - ✓ `Object get(Object key);`
 - ✓ `Object remove(Object key);`
 - ✓ `boolean containsKey(Object key);`
 - ✓ `boolean containsValue(Object value);`
 - ...

Giao diện SortedMap

- ✓ Giao diện SortedMap
 - ✓ thừa kế giao diện **Map**
 - ✓ các phần tử được sắp xếp theo thứ tự
 - ✓ tương tự **SortedSet**, tuy nhiên việc sắp xếp được thực hiện với các khóa
- ✓ Phương thức: Tương tự **Map**, bổ sung thêm:
 - ✓ **firstKey()**: returns the first (lowest) value currently in the map
 - ✓ **lastKey()**: returns the last (highest) value currently in the map

Các lớp thực thi giao diện Collection

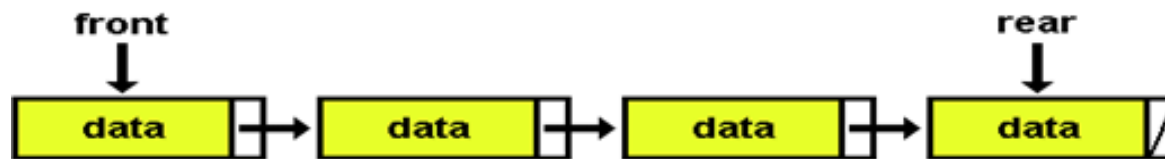
- Java đã xây dựng sẵn một số lớp thực thi các giao diện Set, List và Map và cài đặt các phương thức tương ứng

		IMPLEMENTATIONS				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Legacy
I N T E R F A C E S	Set	HashSet		TreeSet		
	List		ArrayList		LinkedList	Vector, Stack
	Map	HashMap		TreeMap		HashTable, Properties

Các lớp thực thi giao diện

Collection

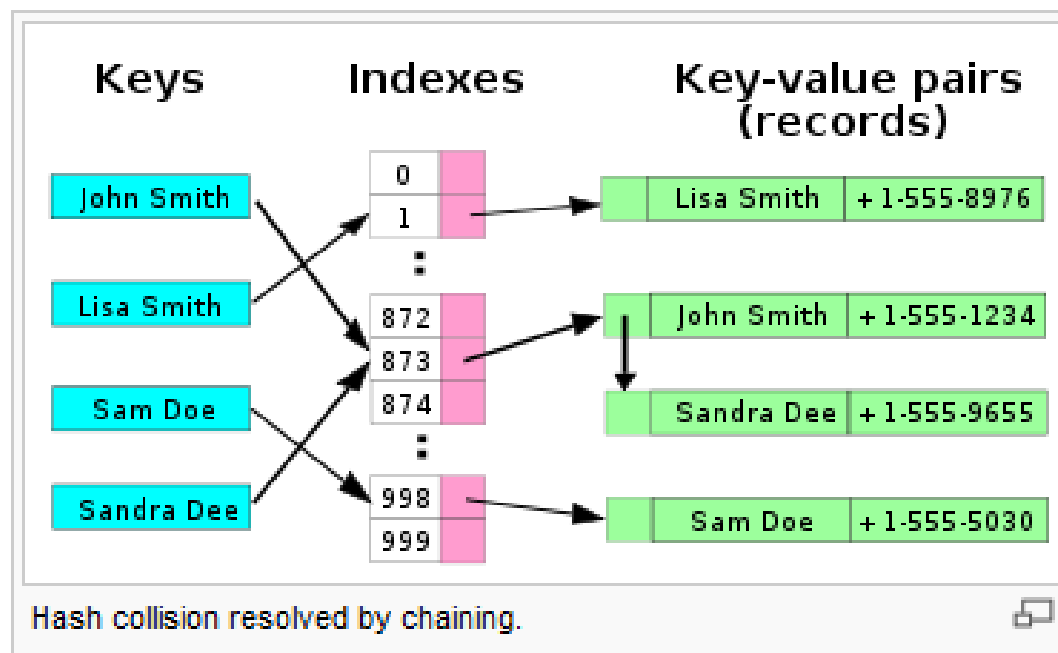
- ✓ **ArrayList**: Mảng động, nếu các phần tử thêm vào vượt quá kích cỡ mảng, mảng sẽ tự động tăng kích cỡ
- ✓ **LinkedList**: Danh sách liên kết
 - ✓ Hỗ trợ thao tác trên đầu và cuối danh sách
 - ✓ Được sử dụng để tạo ngăn xếp, hàng đợi, cây...

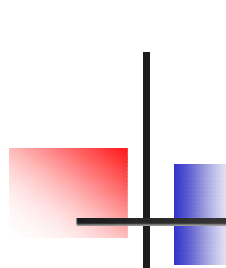


Các lớp thực thi giao diện Collection

✓ HashSet: Bảng băm

- ✓ Lưu các phần tử trong một bảng băm
- ✓ Không cho phép lưu trùng lặp
- ✓ Cho phép phần tử null





Các lớp thực thi giao diện Collection

- ✓ **LinkedHashSet**: Bảng băm kết hợp với linked list nhằm đảm bảo thứ tự các phần tử
 - ✓ Thừa kế HashSet và thực thi giao diện Set
 - ✓ Khác HashSet ở chỗ nó lưu trữ trong một danh sách móc nối đôi
 - ✓ Thứ tự các phần tử được sắp xếp theo thứ tự được insert vào tập hợp
- ✓ **TreeSet**: Cho phép lấy các phần tử trong tập hợp theo thứ tự đã sắp xếp
 - ✓ Các phần tử được thêm vào TreeSet tự động được sắp xếp
 - ✓ Thông thường, ta có thể thêm các phần tử vào HashSet, sau đó convert về TreeSet để duyệt theo thứ tự nhanh hơn

Các lớp thực thi giao diện Collection

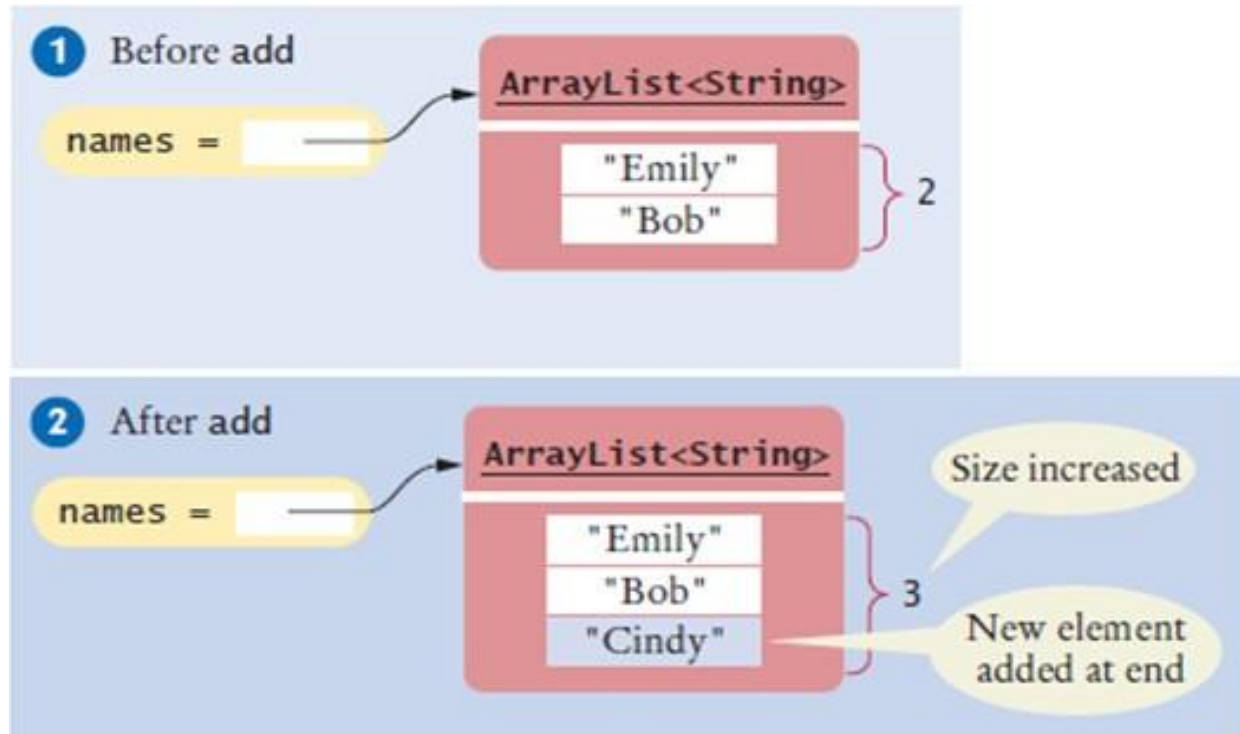
- ✓ **HashMap**: Bảng băm (cài đặt của Map)
- ✓ **LinkedHashMap**: Bảng băm kết hợp với linked list nhằm đảm bảo thứ tự các phần tử (cài đặt của Map)
- ✓ **TreeMap**: Cây (cài đặt của Map)
- ✓ **Legacy Implementations**
 - ✓ Là các lớp cũ được cài đặt bổ sung thêm các collection interface.
 - ✓ **Vector**: Có thể thay bằng ArrayList
 - ✓ **Hastable**: Có thể thay bằng HashMap

Các lớp thực thi giao diện

Collection

■ Ví dụ:

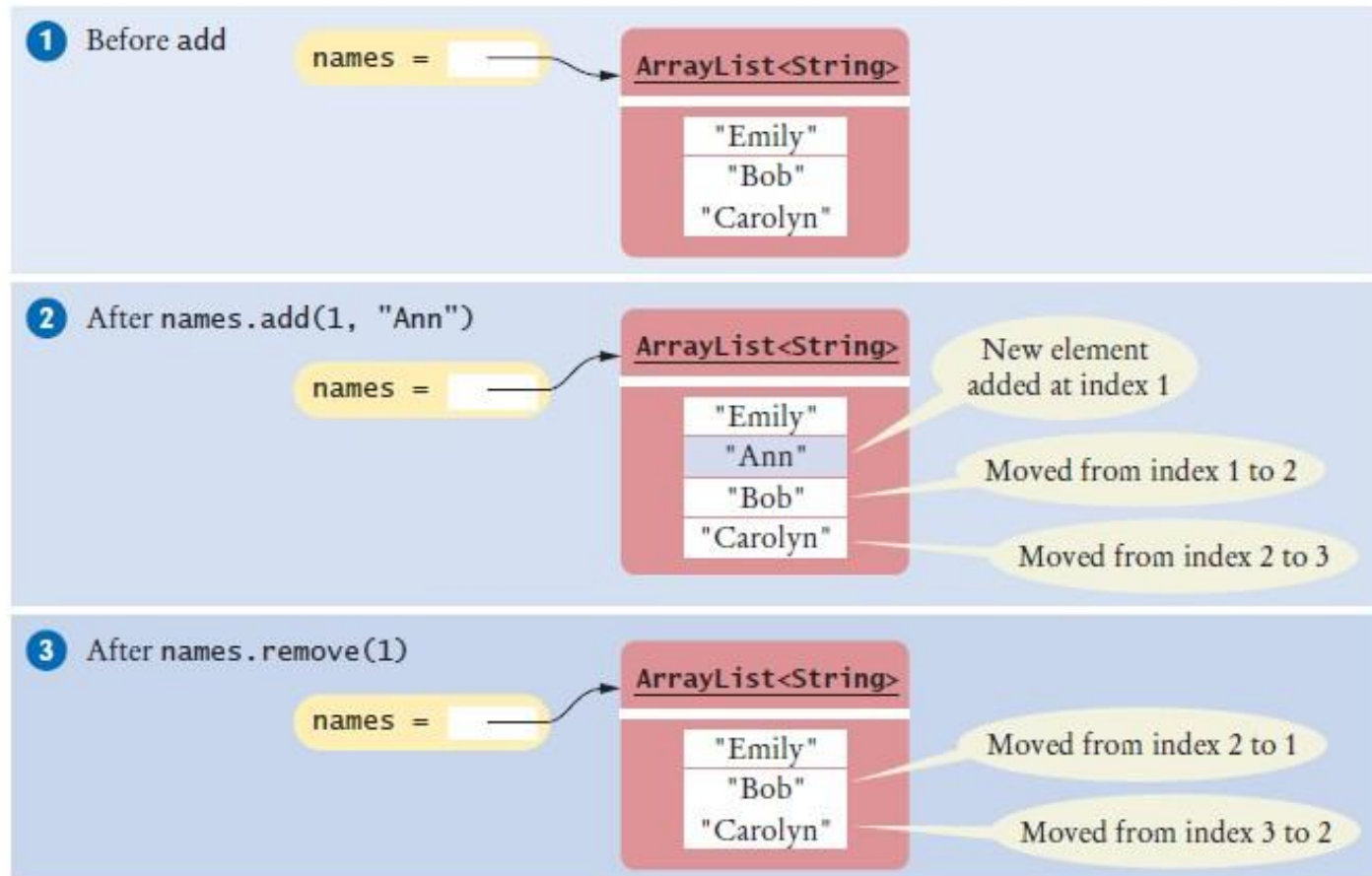
```
ArrayList<String> names = new ArrayList<String>();  
names.add("Emily");  
names.add("Bob");  
names.add("Cindy");
```



Các lớp thực thi giao diện

Collection

- Ví dụ: **String name = names.get(0);**
names.add(1, "Ann");
names.remove(1);



Câu hỏi

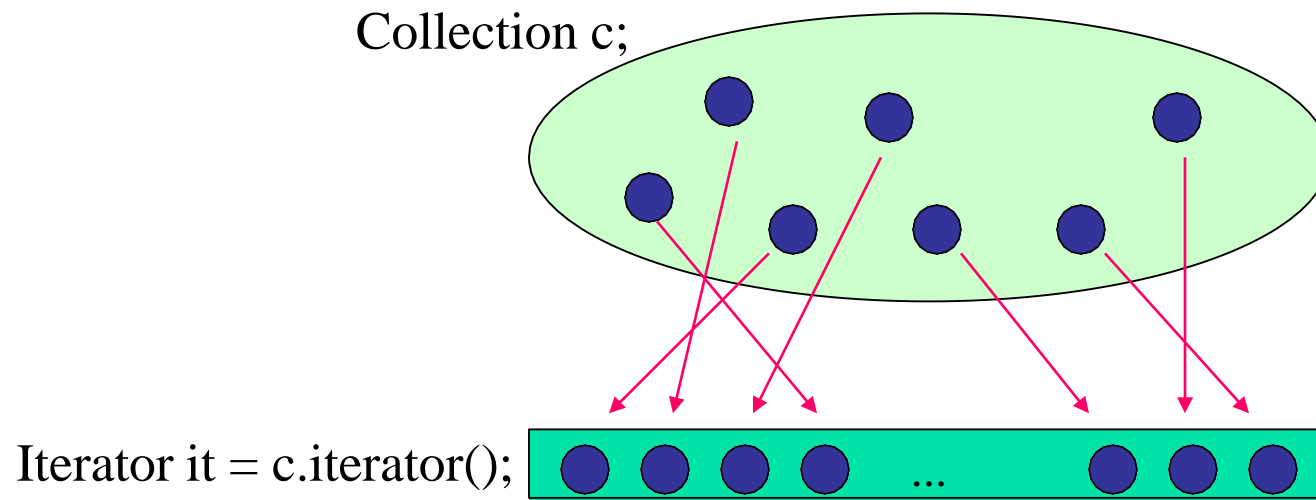


- Sau khi thực hiện đoạn chương trình sau, danh sách **names** có chứa các phần tử nào?

```
ArrayList<String> names = new ArrayList<String>;  
names.add("Bob");  
names.add(0, "Ann");  
names.remove(1);  
names.add("Cal");
```

Giao diện Iterator và Comparator

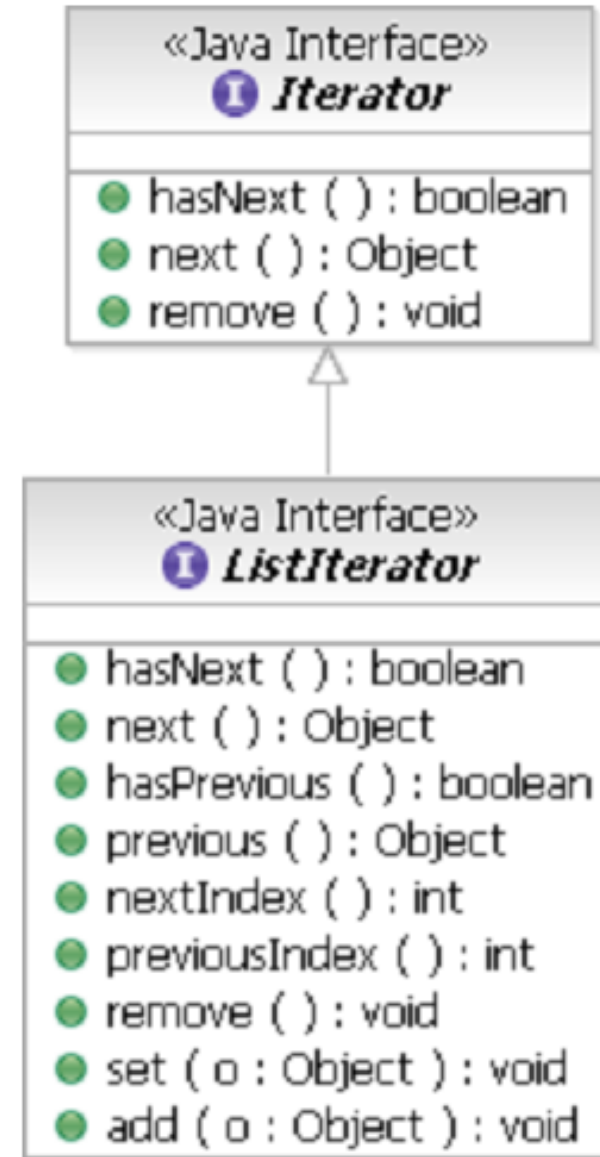
- ✓ Sử dụng để duyệt và so sánh trên các Collection
- ✓ Iterator
 - Các phần tử trong collection có thể được duyệt thông qua Iterator



Giao diện Iterator và Comparator

✓ Iterator

- ✓ Cung cấp cơ chế thuận tiện để duyệt (lặp) qua toàn bộ nội dung của tập hợp, mỗi lần là một đối tượng trong tập hợp
 - Giống như SQL cursor
- ✓ Iterator của các tập hợp đã sắp xếp duyệt theo thứ tự tập hợp
- ✓ ListIterator thêm các phương thức đưa ra bản chất tuần tự của danh sách cơ sở





Giao diện Iterator và Comparator

- ✓ Iterator : Các phương thức
 - ✓ `iterator()`: yêu cầu container trả về một iterator
 - ✓ `next()`: trả về phần tử tiếp theo
 - ✓ `hasNext()`: kiểm tra có tồn tại phần tử tiếp theo hay không
 - ✓ `remove()`: xóa phần tử gần nhất của iterator

Giao diện Iterator và Comparator

- Iterator: Ví dụ

- Định nghĩa iterator

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

- Sử dụng iterator

```
Collection c;  
  
Iterator i = c.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    // Process this object  
}
```

Tương tự vòng lặp **for**

```
for (String name : names){  
    System.out.println(name);  
}
```




Giao diện Iterator và Comparator

- ✓ Giao diện **Comparator** được sử dụng để cho phép so sánh hai đối tượng trong tập hợp
- ✓ Một **Comparator** phải định nghĩa một phương thức **compare()** lấy 2 tham số **Object** và trả về -1, 0 hoặc 1
- ✓ Không cần thiết nếu tập hợp đã có khả năng so sánh tự nhiên (vd. String, Integer...)



Giao diện Iterator và Comparator

■ Ví dụ lớp Person:

```
class Person {  
    private int age;  
    private String name;  
  
    public void setAge(int age){  
        this.age=age;  
    }  
    public int getAge(){  
        return this.age;  
    }  
    public void setName(String name){  
        this.name=name;  
    }  
    public String getName(){  
        return this.name;  
    }  
}
```



Giao diện Iterator và Comparator

- Ví dụ Cài đặt AgeComparator :

```
class AgeComparator implements Comparator {  
    public int compare(Object ob1, Object ob2) {  
        int ob1Age = ((Person)ob1).getAge();  
        int ob2Age = ((Person)ob2).getAge();  
  
        if(ob1Age > ob2Age)  
            return 1;  
        else if(ob1Age < ob2Age) return  
            -1;  
        else  
            return 0;  
    }  
}
```



Giao diện Iterator và Comparator

- Ví dụ Sử dụng AgeComparator :

```
public class ComparatorExample {  
    public static void main(String args[]) {  
        ArrayList<Person> lst = new ArrayList<Person>();  
        Person p = new Person();  
        p.setAge(35);  
        p.setName("A");  
        lst.add(p);  
        p = new Person();  
        p.setAge(30);  
        p.setName("B");  
        lst.add(p);  
        p = new Person();  
        p.setAge(32);  
        p.setName("C");  
        lst.add(p);  
    }  
}
```



Giao diện Iterator và Comparator

■ Ví dụ Sử dụng AgeComparator :

```
System.out.println("Order before sorting");
for (Person person : lst) {
    System.out.println(person.getName() +
                        "\t" + person.getAge());
}

Collections.sort(lst, new AgeComparator());
System.out.println("\n\nOrder of person" +
                  "after sorting by age");

for (Iterator<Person> i = lst.iterator(); i.hasNext();) {
    Person person = i.next();
    System.out.println(person.getName() + "\t" +
                        person.getAge());
} //End of for
} //End of main
} //End of class
```

Lớp tổng quát

- ✓ Lớp tổng quát (generic class) là lớp có thể nhận kiểu dữ liệu là một lớp bất kỳ
- ✓ Cú pháp
Tên Lớp <kiểu 1, kiểu 2, kiểu 3...> {

}
- ✓ Các phương thức hay thuộc tính của lớp tổng quát có thể sử dụng các kiểu được khai báo như mọi lớp bình thường khác

Lớp tổng quát

quát

■ Ví dụ:

Tên kiểu, sẽ được thay thế bằng một kiểu cụ thể khi sử dụng

```
public class Information<T> {  
    private T value;  
    public Information(T value) { this.value  
        = value;  
    }  
    public T getValue() {  
        return value;  
    }  
}
```

```
Information<String> mystring =  
    new Information<String>("hello");  
Information<Circle> circle =  
    new Information<Circle>(new Circle());  
Information<2DShape> shape =  
    new Information<>(new 2DShape());
```

Lớp tổng quát

- Quy ước đặt tên kiểu

<i>Tên kiểu</i>	<i>Mục đích</i>
E	Các thành phần trong một collection
K	Kiểu khóa trong Map
V	Kiểu giá trị trong Map
T	Các kiểu thông thường
S, U	Các kiểu thông thường khác

- Chú ý: Không sử dụng các kiểu dữ liệu nguyên thủy cho các lớp tổng quát

```
Information<int> integer =  
    new Information<int>(2012); //Error  
Information<Integer> integer =  
    new Information<Integer>(2012); //OK
```


Phương thức tổng quát

- ✓ Phương thức tổng quát (generic method) là các phương thức tự định nghĩa kiểu tham số của nó
- ✓ Có thể được viết trong lớp bất kỳ (tổng quát hoặc không)
- ✓ Cú pháp
(chỉ định truy cập) **<kiểu1, kiểu 2...>** (kiểu trả về) tên phương thức (danh sách tham số)
{
 //...
}

■ Ví dụ

```
public static <E> void print(E[] a) { ... }
```

Phương thức tổng quát

- Ví dụ:

```
public class ArrayTool {  
    // Phương thức in các phần tử trong mảng String  
    public static void print(String[] a) {  
        for (String e : a) System.out.print(e + " ");  
        System.out.println();  
    }  
    // Phương thức in các phần tử trong mảng với kiểu  
    // dữ liệu bất kỳ  
    public static <E> void print(E[] a) {  
        for (E e : a) System.out.print(e + " ");  
        System.out.println();  
    }  
}
```

Phương thức tổng quát

■ Ví dụ:

...

```
String[] str = new String[5];
```

```
Point[] p = new Point[3];
```

```
int[] intnum = new int[2];
```

```
ArrayTool.print(str);
```

```
ArrayTool.print(p);
```

```
// Không dùng được với kiểu dữ liệu nguyên thủy
```

```
ArrayTool.print(intnum);
```

Giới hạn kiểu dữ liệu tổng quát

- ✓ Có thể giới hạn các kiểu dữ liệu tổng quát sử dụng phải là dẫn xuất của một hoặc nhiều lớp
- ✓ Giới hạn 1 lớp
`<type_param extends bound>`
- ✓ Giới hạn nhiều lớp
`<type_param extends bound_1 & bound_2 & ..>`

Giới hạn kiểu dữ liệu tổng quát

quát

Chấp nhận các kiểu là lớp
con của 2DShape

■ Ví dụ:

```
public class Information<T extends 2DShape> {  
    private T value;  
    public Information(T value) {  
        this.value = value;  
    }  
    public T getValue() {  
        return value;  
    }  
}  
  
Information<Point> pointInfo =  
    new Information<Point>(new Point()); // OK  
  
Information<String> stringInfo =  
    new Information<String>();
```

// error 51

4. Ký tự đại diện (Wildcard)

- ✓ Quan hệ thừa kế giữa hai lớp không có ảnh hưởng gì đến quan hệ giữa các cấu trúc tổng quát dùng cho hai lớp đó.
- ✓ Ví dụ:
 - ✓ Dog và Cat là các lớp con của Animal
 - → Có thể đưa các đối tượng Dog và Cat vào một `ArrayList<Animal>`
 - ✓ Tuy nhiên, `ArrayList<Dog>`, `ArrayList<Cat>` lại không có quan hệ gì với `ArrayList<Animal>`

4. Ký tự đại diện (Wildcard)

✓ Generic

- ✓ Kiểu khai báo trong lớp tổng quát (template) khi khởi tạo phải cùng với kiểu của các đối tượng thực sự.
- ✓ Nếu khai báo `List<Foo>` \Rightarrow Danh sách chỉ chấp nhận các đối tượng lớp `Foo`, các đối tượng là cha hoặc con của lớp `Foo` sẽ không được chấp nhận.

```
class Parent { }
```

```
class Child extends Parent { }
```

```
List<Parent> myList = new ArrayList<Child>();
```

4. Ký tự đại diện

(Wildcard)

- ✓ Làm thế nào để xây dựng các tập hợp dành cho kiểu bất kì là lớp con của lớp cụ thể nào đó?
→ Giải pháp là sử dụng **kí tự đại diện** (wildcard)
- ✓ Ký tự đại diện: **?** dùng để hiển thị cho một kiểu dữ liệu chưa biết trong collection

```
void printCollection(Collection<?> c) {  
    for (Object e : c) {  
        System.out.println(e);  
    }  
}
```

- Khi biên dịch, dấu ? có thể được thay thế bởi bất kì kiểu dữ liệu nào.

4. Ký tự đại diện (Wildcard)

- Tuy nhiên viết như thế này là không hợp lệ
`Collection<?> c = new ArrayList<String>();`
`c.add("a1"); //compile error, null`
- Vì không biết c đại diện cho tập hợp kiểu dữ liệu nào => không thể thêm phần tử vào c

4. Ký tự đại diện (Wildcard)

- ✓ "? extends Type": Xác định một tập các kiểu con của Type. Đây là wildcard hữu ích
- ✓ "? super Type": Xác định một tập các kiểu cha của Type
- ✓ "?": Xác định tập tất cả các kiểu hoặc bất kỳ kiểu nào

4. Ký tự đại diện (Wildcard)

✓ Ví dụ:

✓ **? extends Animal** có nghĩa là kiểu gì đó thuộc loại Animal

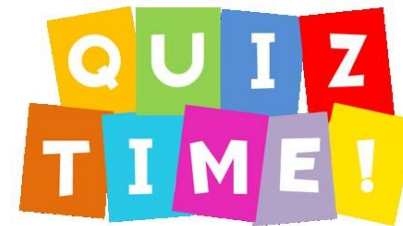
✓ Hai cú pháp sau là tương đương:

```
public void foo( ArrayList<? extends Animal> a)
```

```
public <T extends Animal> void foo( ArrayList<T> a)
```

✓ Dùng "T", thường được sử dụng khi còn muốn T xuất hiện ở các vị trí khác

Câu hỏi



```
public void draw(List<Shape> shape) {  
    for(Shape s: shape) {  
        s.draw(this);  
    }  
}
```

→ Khác như thế nào với:

```
public void draw(List<? extends Shape> shape){  
  
    for(Shape s: shape) {  
        s.draw(this);  
    } }  
}
```

Tổng kết

- ✓ Generic programming: tổng quát hóa chương trình để có thể hoạt động với các kiểu dữ liệu khác nhau, kể cả kiểu dữ liệu trong tương lai với thuật toán đã xác định
 - Trong Java sử dụng Template
- ✓ Collection – tập hợp: Nhóm các đối tượng lại thành một đơn vị duy nhất
- ✓ Java Collections Framework: biểu diễn các tập hợp, cung cấp giao diện tiêu chuẩn (giao diện, lớp thực thi, thuật toán)
- ✓ Lớp tổng quát (generic class) là lớp có thể nhận kiểu dữ liệu là một lớp bất kỳ

Bài

1

- ✓ Trừu tượng hoá mô tả sau: một quyển sách là tập hợp các chương, chương là tập hợp các trang.
 - ✓ Phác hoạ các lớp Book, Chapter, và Page
 - ✓ Tạo các thuộc tính cần thiết cho các lớp, hãy tận dụng tập hợp như là thuộc tính của lớp
 - ✓ Tạo các phương thức cho lớp Chapter cho việc thêm trang và xác định một chương có bao nhiêu trang
 - ✓ Tạo các phương thức cho lớp Book cho việc thêm chương và xác định quyển sách có bao nhiêu chương, và số trang cho quyển sách

Bài

2

- Xây dựng lớp Stack tổng quát với các kiểu dữ liệu

StackOfChars
- elements: char[] - size: int
+ StackOfChars() + StackOfChars (capacity: int) + isEmpty(): boolean + isFull(): boolean + peak(): char + push(value:char): void + pop(): char + getSize(): int

StackOfIntegers
- elements: int[] - size: int
+ StackOfIntegers() + StackOfIntegers (capacity: int) + isEmpty(): boolean + isFull(): boolean + peak(): int + push(value:int): void + pop(): int + getSize(): int

• • •