

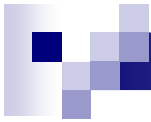


# Kế thừa



# Nội dung

- Vấn đề sử dụng lại
- Sử dụng lại bằng kế thừa
- Kế thừa trong Java
  - định nghĩa lớp kế thừa
  - thêm phương thức, thuộc tính
  - kiểm soát truy cập
  - khởi tạo



# Tài liệu tham khảo

- *Giáo trình lập trình HĐT*, chương 7
- *Thinking in Java*, chapter 6
- *Java how to program*, chapter 9



# Sử dụng lại

- Tồn tại nhiều loại đối tượng có các thuộc tính và hành vi tương tự hoặc liên quan đến nhau
  - Person, Student, Manager,...
- Xuất hiện nhu cầu sử dụng lại các mã nguồn đã viết
  - Sử dụng lại thông qua copy
  - Sử dụng lại thông qua quan hệ *has\_a*
  - Sử dụng lại thông qua cơ chế “kế thừa”



# Sử dụng lại

- Copy mã nguồn

- ☐ Thử công, dễ nhầm
- ☐ Khó sửa lỗi do tồn tại nhiều phiên bản


- Quan hệ *has\_a*

- ☐ Sử dụng lớp đã có như là thành phần của lớp mới
- ☐ Sử dụng lại cài đặt với giao diện mới
  - Phải viết lại giao diện
  - Chưa đủ mềm dẻo



## Ví dụ: *has\_a*

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



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



# Kế thừa

- Dựa trên quan hệ *is\_a*
- Thừa hưởng lại các thuộc tính và phương thức đã có
- Chi tiết hóa/chuyên biệt hóa cho phù hợp với mục đích sử dụng mới
  - Thêm các thuộc tính mới
  - Thêm các phương thức mới
  - Hiệu chỉnh các phương thức đã có
- Thuật ngữ
  - Lớp cơ sở, lớp cha
  - Lớp dẫn xuất, lớp con



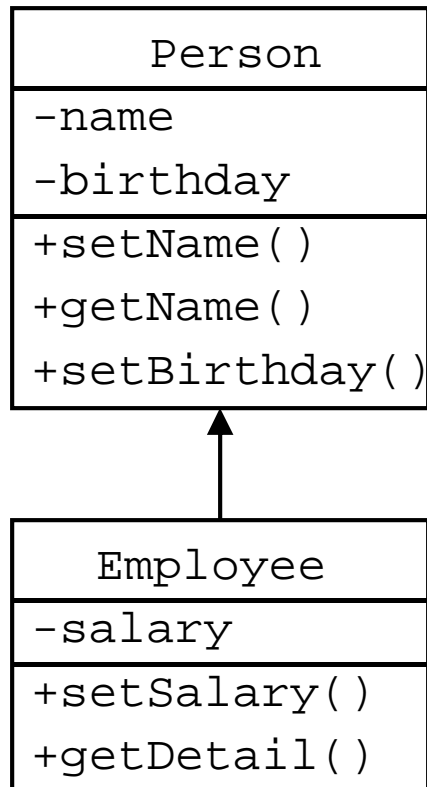


# Kế thừa trong Java

```
[public] class DerivedClass extends BaseClass {  
    /* new features go here */  
}
```

Ví dụ:

```
class Employee extends Person {  
    private double salary;  
    public boolean setSalary(double sal) {  
        ...  
        salary = sal;  
        return true;  
    }  
}
```



```
Employee e = new Employee();

e.setName("John");
e.setSalary(3.0);
System.out.print(e.getName());
```



## Che giấu thông tin giữa lớp cơ sở và lớp dẫn xuất

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



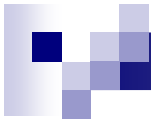
# Che giấu thông tin...

- Sử dụng lại dễ dàng mà không phụ thuộc vào cách cài đặt cụ thể
- Người lập trình lớp dẫn xuất có thể khác người lập trình lớp cơ sở
  - ☐ người lập trình lớp cơ sở có thể thay đổi thiết kế nội tại
- *Che giấu không có nghĩa là không nhìn thấy mã nguồn*



# Mức truy cập protected

- Để đảm bảo che giấu thông tin, thông thường các thuộc tính được khai báo là private
  - Đối tượng thuộc lớp dẫn xuất phải truy cập tới chúng thông qua các phương thức get và set.
- Mức truy cập protected giải quyết vấn đề này
  - Đối tượng thuộc lớp dẫn xuất truy cập được các *protected members* của lớp cơ sở
  - Đối tượng thuộc lớp khác không truy cập được



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

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



# Các mức kiểm soát truy cập

Modifier	Same class	Same package	Subclass	Universe
private	Yes			
package ( <i>default</i> )	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes



# Trong cùng gói

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

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



# 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;  
        s += "," + salary;  
        return s;  
    }  
}
```



# Kế thừa: từ gói khác

- Có thể kế thừa từ gói khác
  - ☐ Kế thừa từ thư viện của Java: ví dụ từ Applet
  - ☐ Kế thừa từ gói của hãng khác
- Kế thừa mà không cần biết mã nguồn
  - ☐ Bảo mật mã nguồn
  - ☐ Nâng cao khả năng sử dụng lại



# Định nghĩa lại phương thức

- Có thể định nghĩa lại các phương thức của lớp cơ sở
  - Chi tiết hóa cho phù hợp với bài toán mới
- Đối tượng của lớp dẫn xuất sẽ hoạt động với phương thức *mới* phù hợp với nó
  - Cơ chế liên kết động (dynamic binding)
- Có thể tái sử dụng phương thức cùng tên của lớp cơ sở bằng từ khóa **super**

# Ví dụ

```
package abc;

public class Person {
    protected Date birthday;
    protected String name;
    public String getDetail() {...}
    ...
}
```


```
import abc;

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



## Định nghĩa lại...

- Phải có quyền truy cập không *chặt* hơn phương thức được định nghĩa lại
- Phải có kiểu giá trị trả lại như nhau
- *Chỉ có tác dụng với phương thức không phải là private*
  - *Phương thức private được che giấu với lớp dẫn xuất*




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

```
class Child extends Parent {  
    protected void doSomething() {} //error  
    private void doSomething2() {} // ok  
}
```



# Khởi tạo của lớp dẫn xuất

- Lớp dẫn xuất kế thừa mọi thuộc tính và phương thức của lớp cơ sở
- *Không kế thừa phương thức khởi tạo*
  - *Về mặt cú pháp, có thể thấy chúng có tên khác nhau*
- Có hai giải pháp gọi constructor của lớp cơ sở
  - sử dụng constructor mặc định
  - gọi constructor của lớp cơ sở một cách tường minh



```
class Point {  
    protected int x, y;  
    public Point() {}  
    public Point(int xx, int yy) {  
        x = xx;  
        y = yy;  
    }  
}
```

```
class Circle extends Point {  
    protected int radius;  
    public Circle() {}  
}
```


```
Point p = new Point(10, 10);  
Circle c1 = new Circle();  
Circle c2 = new Circle(10, 10); // error
```






# Gọi constructor của lớp cơ sở

- 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 mặc định thì bắt buộc phải gọi constructor tường minh




```
class Point {
    protected int x, y;
    public Point() {}
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() {}
    public Circle(int xx, int yy, int r) {
        super(xx, yy);
        radius = r;
    }
}
```



```
class Point {
    protected int x, y;
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() { super(0, 0); }
    public Circle(int xx, int yy, int r) {
        super(xx, yy);
        radius = r;
    }
}
```



```
class Point {
    protected int x, y;
    public Point() {}
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() { }
    public Circle(int xx, int yy, int r) {
        // super(xx, yy);
        radius = r;
    }
}
```

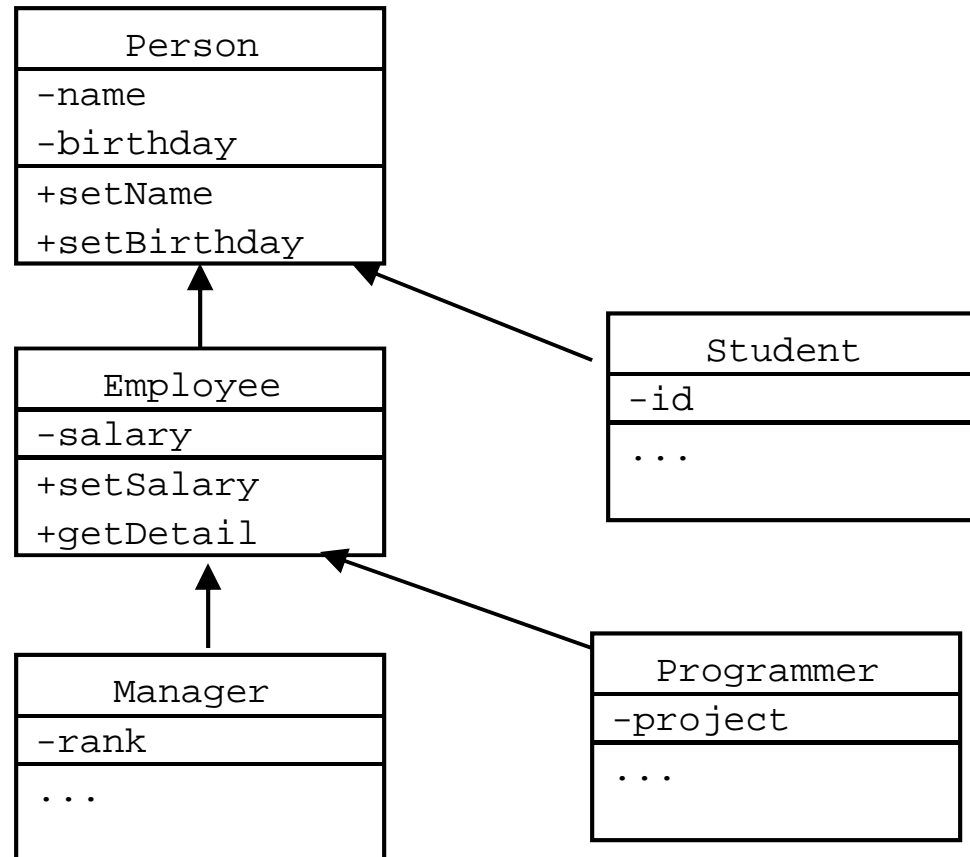


# Thứ tự khởi tạo

```
class Point {  
    protected int x, y;  
    public Point() {  
        System.out.println("Point constructor");  
    }  
}  
  
class Circle extends Point {  
    protected int radius;  
    public Circle() {  
        System.out.println("Circle constructor");  
    }  
}  
  
...  
Circle c = new Circle();
```

# Kế thừa nhiều tầng

Trong Java, mọi lớp đối tượng đều kế thừa từ lớp gốc Object



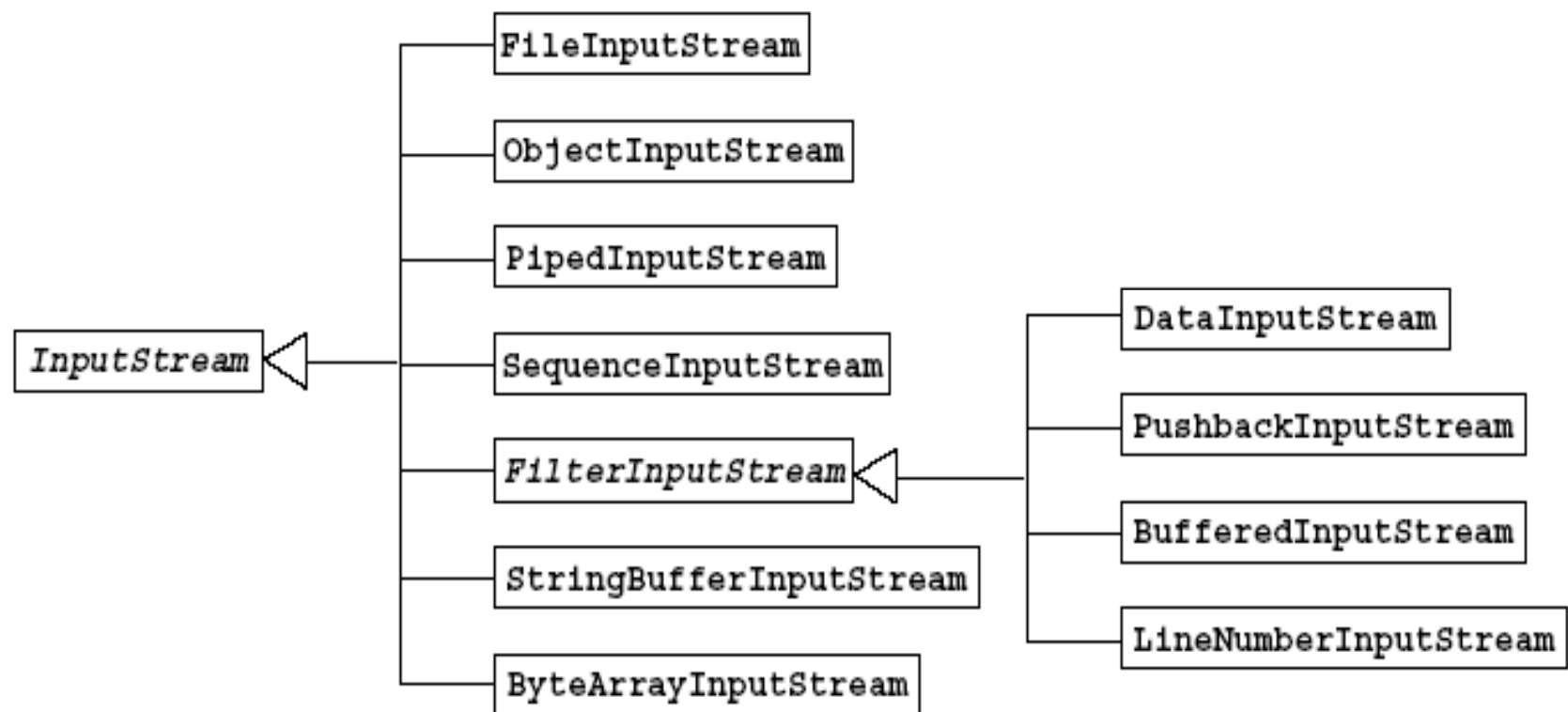
## Phương thức toString()

- Kế thừa từ lớp Object, trả lại kết quả là context của đối tượng

```
public class Person {  
    protected Date birthday;  
    protected String name;  
    public String toString() {...}  
    ...  
}
```

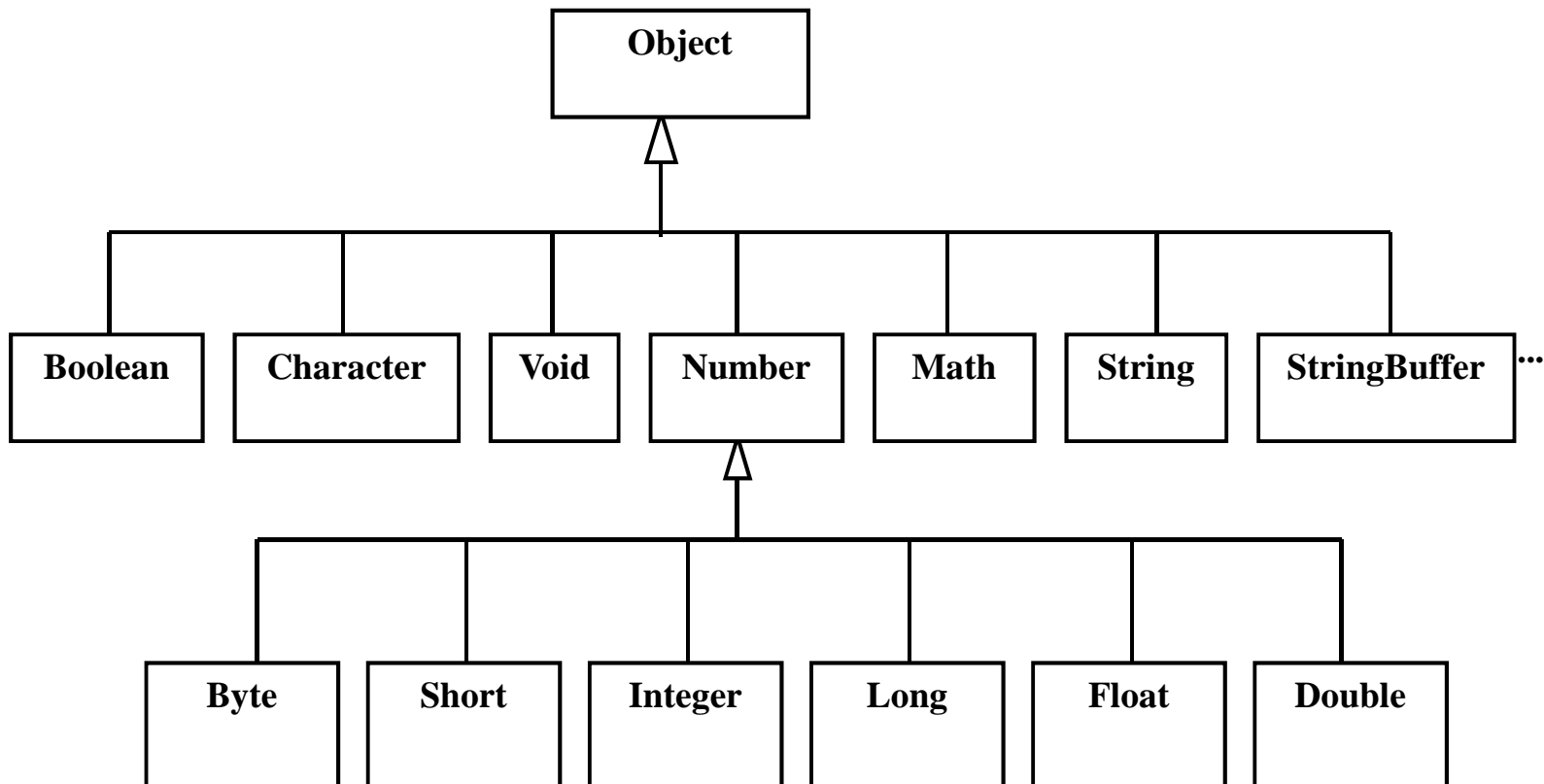
```
public class Employee extends Person {  
    ...  
    public String toString() {  
        String s;  
        s = super.toString() + "," + salary;  
        return s;  
    }  
}  
...  
Employee e = new Employee();  
System.out.println(e);
```

# Ví dụ: Phả hệ của InputStream





# Ví dụ: Một số lớp cơ bản của Java





# Từ khóa `final`

- Thuộc tính `final`

- ☐ **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



# Tham số final

```
class MyDate {  
    int year, month, day;  
    public MyDate(int y, int m, int d) {  
        year = y; month = m; day = d;  
    }  
    public void copyTo(final MyDate d) {  
        d.year = year;  
        d.month = month;  
        d.day = day;  
        // d = new MyDate(year, month, day);  
    }  
    ...  
}
```



# Tổng kết

- Kế thừa là một ưu điểm quan trọng của lập trình hướng đối tượng
- Cho phép dễ dàng sử dụng lại
  - sử dụng lại ngay trong một chương trình
  - sử dụng lại giữa các chương trình
- Thích hợp với các bài toán phức tạp, tồn tại nhiều loại đối tượng (vd. đồ họa,...)
- Cần có cách nhìn mới khi phân tích thiết kế hệ thống



# Bài tập

- Hãy phân tích và chỉ ra các khả năng kế thừa trong bài toán quản lý các đối tượng đồ họa.
- Hãy phân tích và chỉ ra các khả năng kế thừa trong bài toán quản lý con người trong trường đại học.
- Xét riêng bài toán quản lý sinh viên. Có cần sử dụng quan hệ kế thừa cho đối tượng sinh viên hay không?



# Bài tập

- Cài đặt các lớp Person, Employee, Manager sử dụng kế thừa
  - Bổ sung phương thức, thuộc tính, sửa đổi phương thức.
- Thành thạo các phương thức constructor, các mức truy cập
- Xây dựng **lớp MyStack** “*tổng quát*” có thể chứa đối tượng bất kỳ.