

BỘ MÔN CÔNG NGHỆ PHẦN MỀM
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG


Bài 07. Đa hình (Polymorphism)

Cao Tuấn Dũng
dungct-fit@mail.hut.edu.vn
caotuandung@gmail.com

Nội dung

1. Upcasting và Downcasting
2. Liên kết tĩnh và Liên kết động
3. Đa hình (Polymorphism)
4. Lập trình tổng quát (generic prog.)

Nội dung

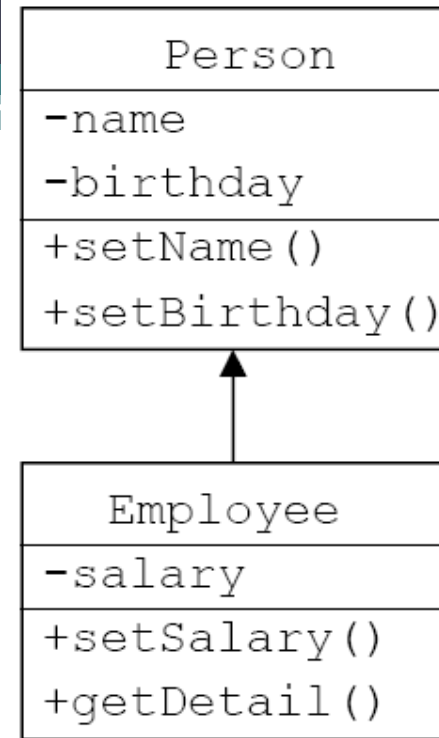
- 
1. Upcasting và Downcasting
 2. Liên kết tĩnh và Liên kết động
 3. Đa hình (Polymorphism)
 4. Lập trình tổng quát (generic prog.)

1.1. Upcasting

- 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

Ví dụ

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



Ví dụ (2)

```
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);
    }
}
```

Ví dụ (3)

```
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

- 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.

Ví dụ

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

Nội dung

1. Upcasting và Downcasting



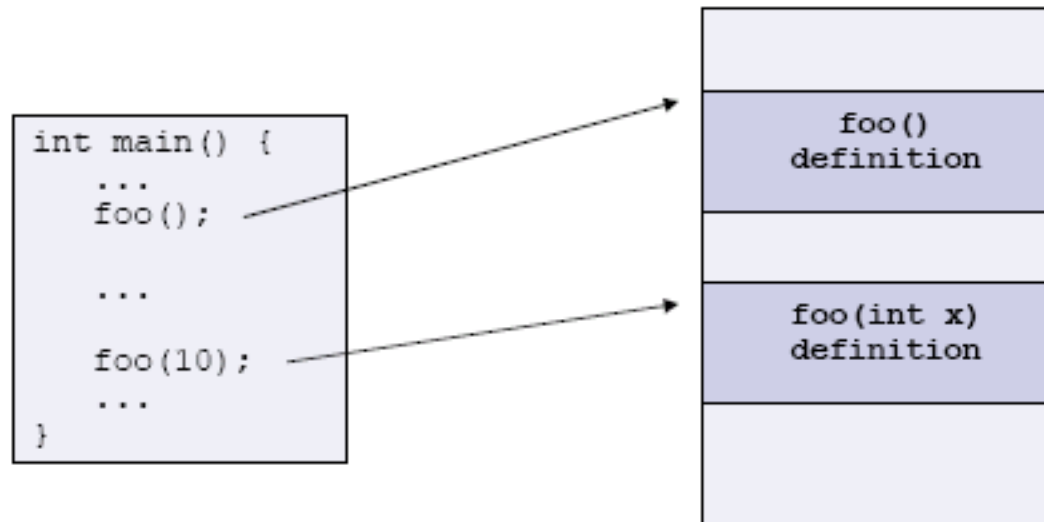
2. Liên kết tĩnh và Liên kết động

3. Đa hình (Polymorphism)

4. Lập trình tổng quát (generic prog.)

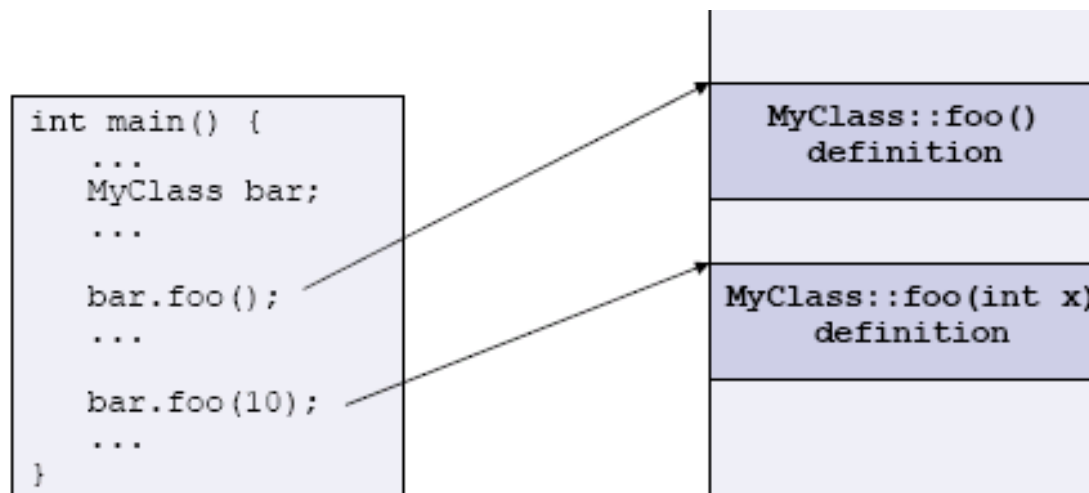
Liên kết lời gọi hàm function call binding

- 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.



Ngôn ngữ HĐT (method call binding)

- 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**

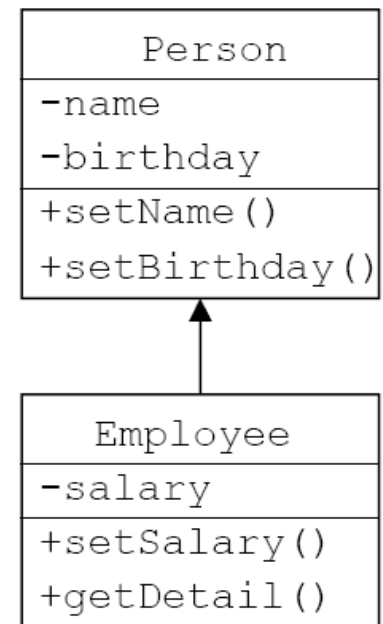


2.1. Liên kết tĩnh (Static Binding)

- 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 static function call binding

Ví dụ

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

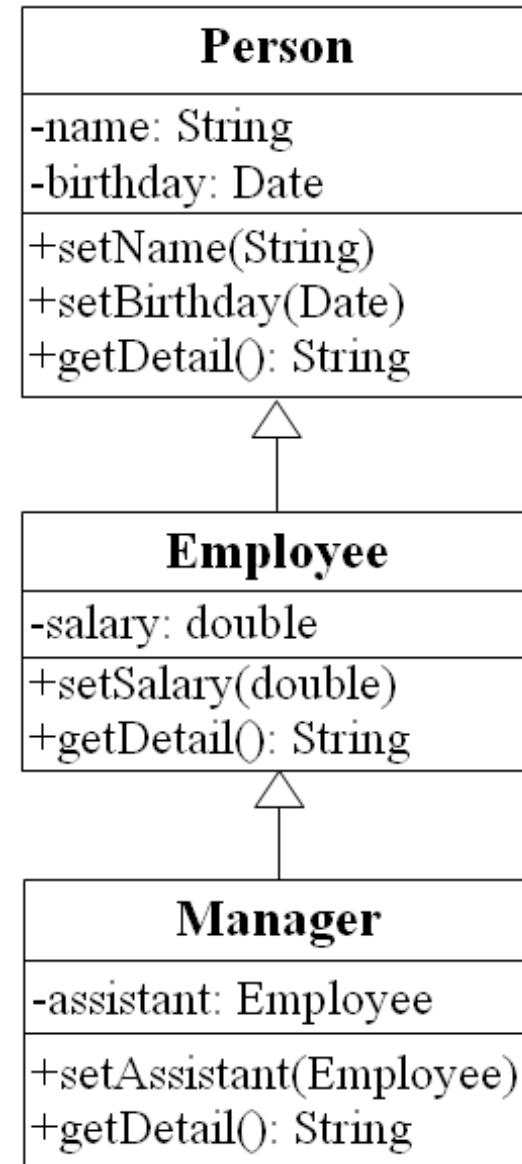


2.2. Liên kết động (Dynamic binding)

- 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 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());
        }
    }
}
```



Nội dung

1. Upcasting và Downcasting
2. Liên kết tĩnh và Liên kết động
- ⇒ 3. Đa hình (Polymorphism)
4. Lập trình tổng quát (generic prog.)

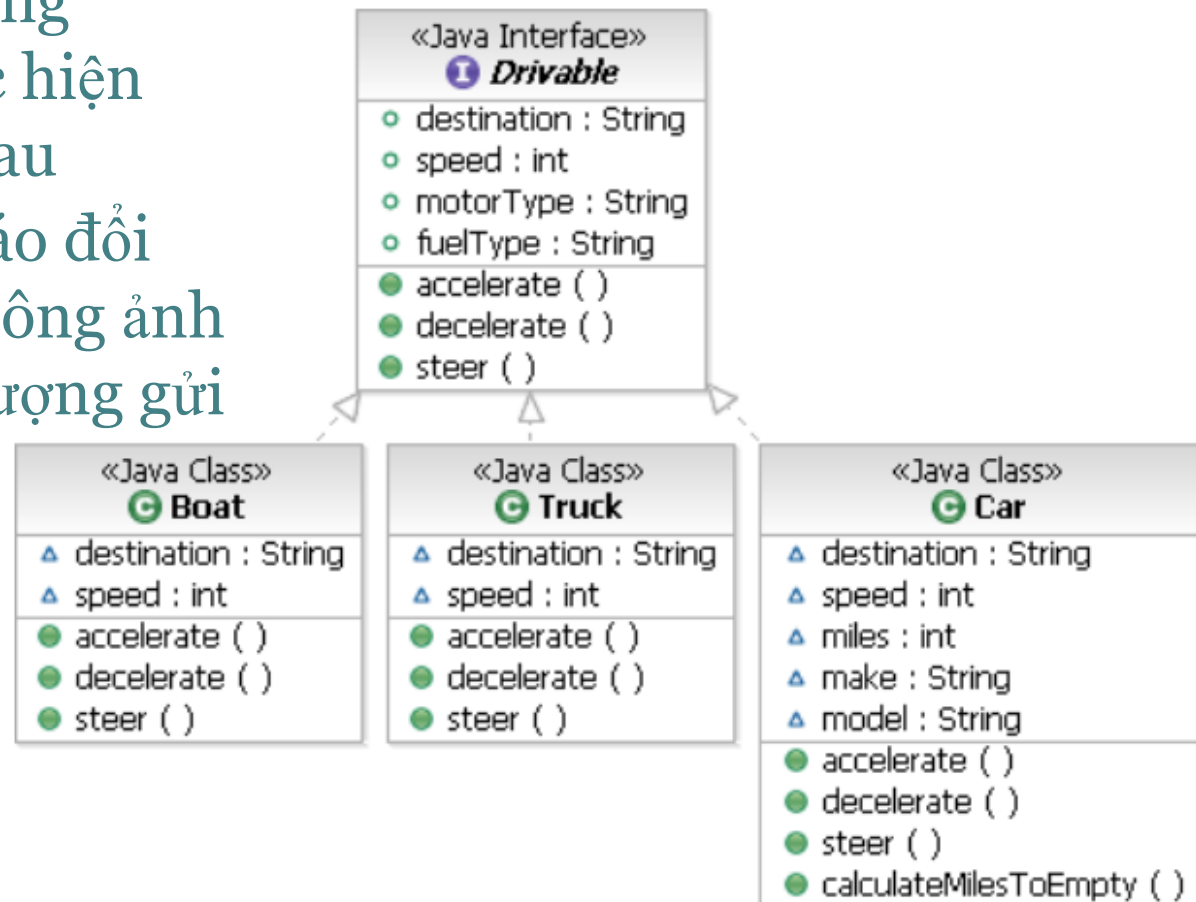
3. Đa hình (Polymorphism)

- Ví dụ: Nếu đi du lịch, bạn có thể chọn ô tô, thuyền, hoặc máy bay
 - Dù đi bằng phương tiện gì, kết quả cũng giống nhau là bạn đến được nơi cần đến
 - Cách thức đáp ứng các dịch vụ có thể khác nhau



3. Đa hình (2)

- Các lớp khác nhau có thể đáp ứng danh sách các thông điệp giống nhau, vì vậy cung cấp các dịch vụ giống nhau
 - Cách thức đáp ứng thông điệp, thực hiện dịch vụ khác nhau
 - Chúng có thể trao đổi cho nhau mà không ảnh hưởng đến đối tượng gửi thông điệp
- Đa hình



3. Đa hình (3)

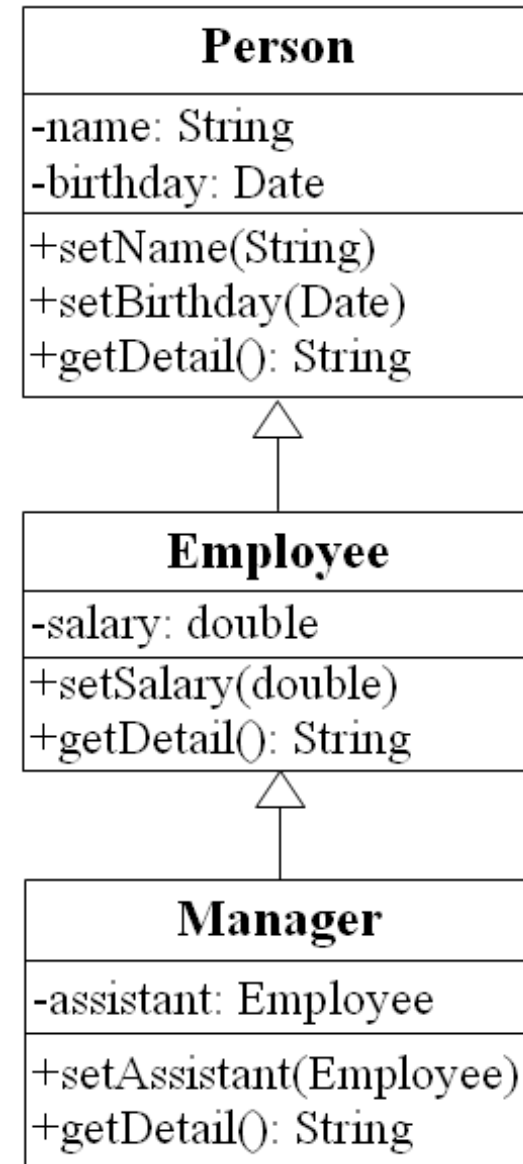
- Polymorphism: Nhiều hình thức thực hiện, nhiều kiểu tồn tại
- Đa hình trong lập trình
 - Đa hình phương thức:
 - Phương thức trùng tên, phân biệt bởi danh sách tham số.
 - Đa hình đối tượng
 - Nhìn nhận đối tượng theo nhiều kiểu khác nhau
 - Các đối tượng khác nhau cùng đáp ứng chung danh sách các thông điệp có giải nghĩa thông điệp theo cách thức khác nhau.

3. Đa hình (4)

- Nhìn nhận đối tượng theo nhiều kiểu khác nhau → Upcasting và Downcasting

```
public class Test3 {
    public static void main(String
                           args[]) {
        Person p1 = new Employee();
        Person p2 = new Manager();

        Employee e = (Employee) p1;
        Manager m = (Manager) p2;
    }
}
```



3. Đa hình (5)

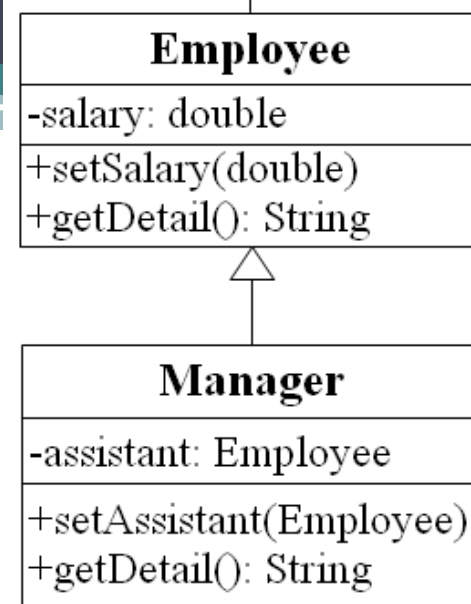
- Các đối tượng khác nhau giải nghĩa các thông điệp theo các cách thức khác nhau → Liên kết động (java)
- Ví dụ:

```
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());
```

Ví dụ khác

```
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());
        }
    }
}

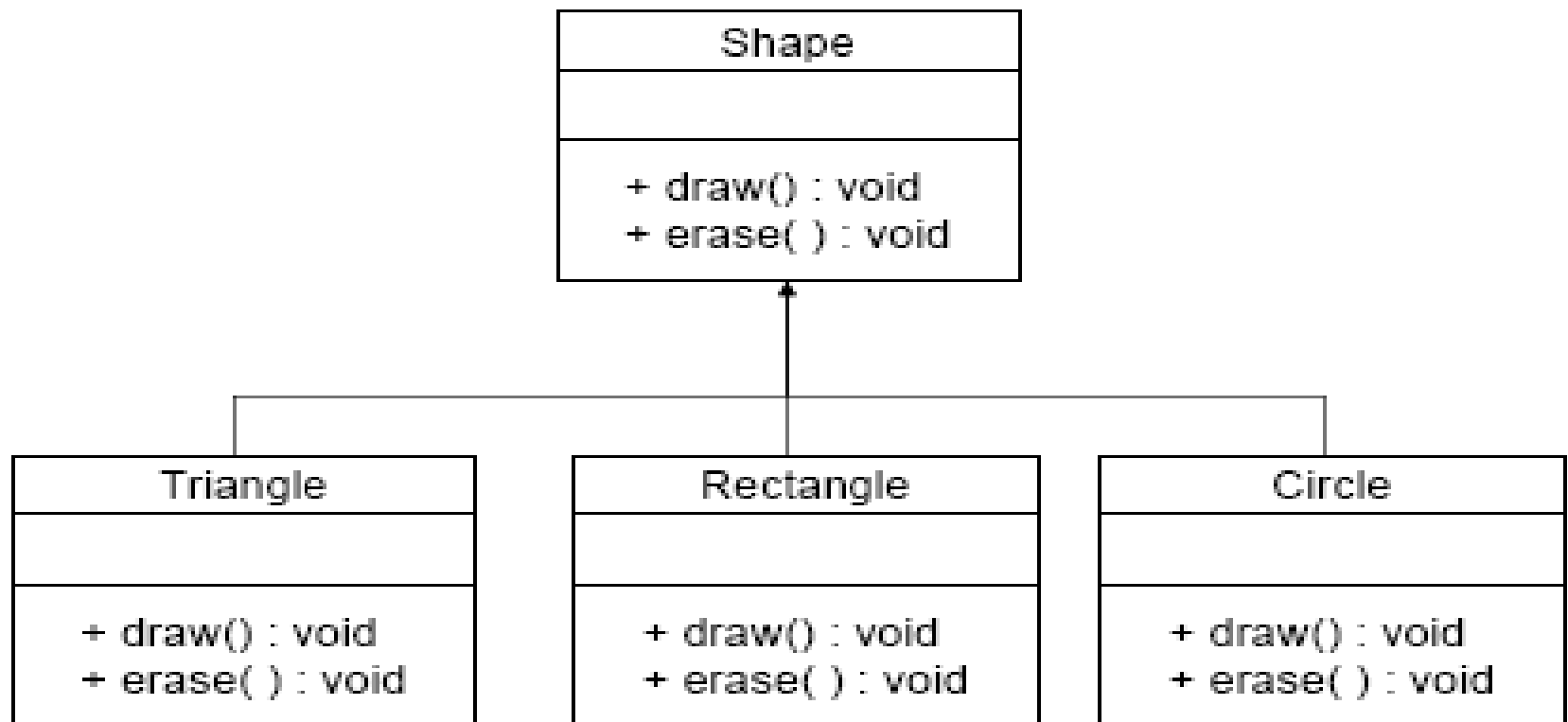
...
EmployeeList list = new EmployeeList();
Employee e1; Manager m1;
...
list.add(e1); list.add(m1);
list.print();
```



Toán tử instanceof

```
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 {...}  
    }  
}
```


Ví dụ



Ví dụ

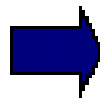
```
...
public static void handleShapes(Shape[] shapes){
// Cac hình vẽ sẽ được vẽ theo cách riêng của chúng
for( int i = 0; i < shapes.length; ++i) {
shapes[i].draw();
}
...
//Ta đơn giản gọi phương thức xóa không quan tâm đến
// đối tượng thuộc lớp nào
for( int i = 0; i < shapes.length; ++i) {
shapes[i].erase();
}
}
...
```

Đa hình trong C++

Ví dụ về đa hình: Bò điên

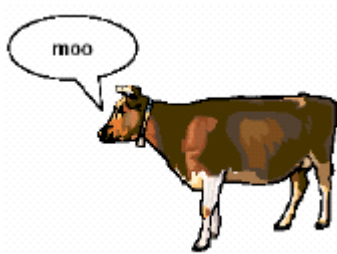
**Một con bò – Kêu
Moo**

| COW |
|-------------------|
| - name - color |
| + make a sound() |



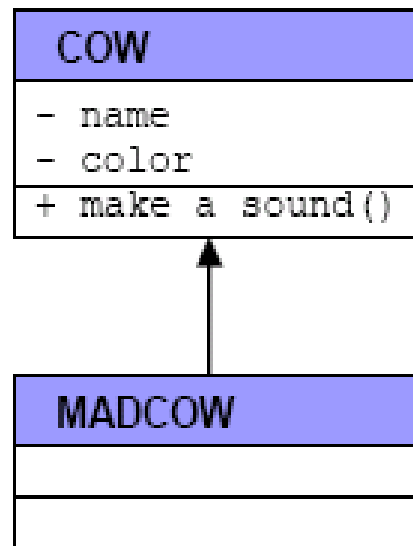
```
class Cow {  
public:  
    Cow(string name, string color);  
    ~Cow();  
    void makeASound();  
private:  
    string name;  
    string color;  
};
```

```
...  
void Cow::makeASound();  
{  
    cout << "Moo" << endl;  
}
```



Bò điên

- Bò điên – cũng là bò:
nhưng tưởng mình là ếch
kêu oap



```
class MadCow : public Cow {  
public:  
    MadCow(string name, string color);  
    ~MadCow();  
  
    void makeASound();  
};
```

```
...  
void MadCow::makeASound() {  
    cout << "oap" << endl;  
}
```

Bò điên

- Như vậy, nếu tạo một đối tượng **Cow**, và gọi hành vi **makeASound()** của nó, nó sẽ kêu “moo”

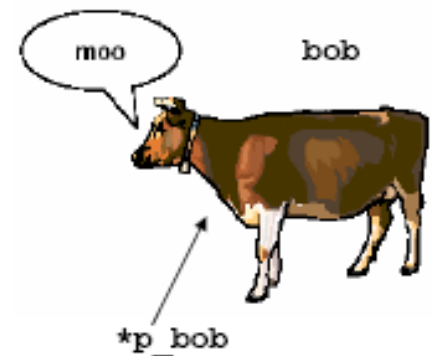
```
Cow bob(“Bob”, “brown”);
```

```
bob.makeASound();
```

- Ta còn có thể tạo một con trỏ tới đối tượng **Cow**, và làm cho nó kêu “moo”

```
p_bob = &bob;
```

```
p_bob->makeASound();
```



Nhập nhằng

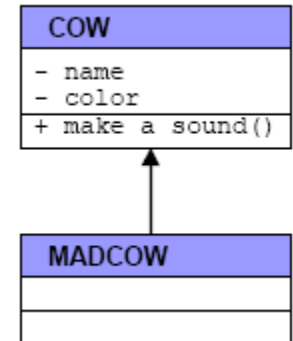
- Nếu coi nó là bò điên (**MadCow**), nó sẽ hành động đúng là bò điên
`MadCow maddy("Maddy", "white");`
`maddy.makeASound(); // Go "oap"`
`MadCow * maddy = new MadCow("Maddy", "white");`
`maddy->makeASound(); // Go "oap"`
- Còn khi tưởng nó chỉ là bò thường (Cow), nó lại có vẻ bình thường
`MadCow* maddy = new madCow("Maddy", "white");`
`Cow* cow = maddy; // Upcasting`
`cow->makeASound(); // Go "moo" ???`
- Làm thế nào để bò điên lúc nào cũng điên (kêu "oap")?

Liên kết tĩnh: trường hợp có cây thừa kế

Kiểu tĩnh:
MadCow

Kiểu tĩnh:
Cow

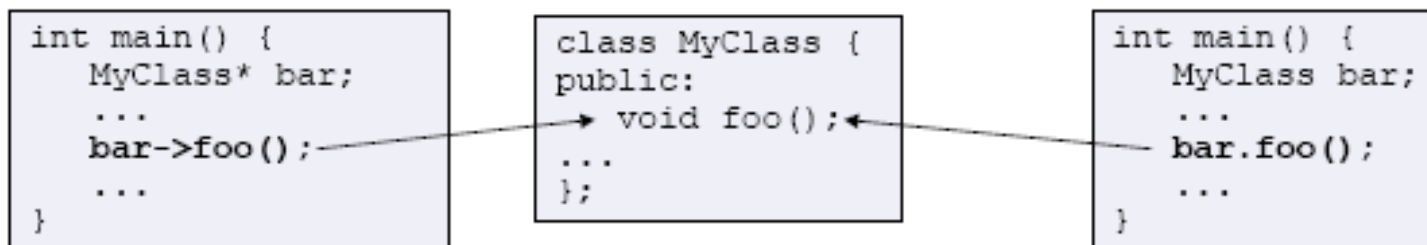
```
Cow bob("Bob", "brow");  
MadCow maddy("Maddy", "white");  
bob.makeASound(); // go "moo"  
maddy.makeASound(); // go "oap"  
MadCow& rMaddy = maddy;  
rMaddy.makeASound(); // Go "oap"  
MadCow* pMaddy = &maddy;  
pMaddy->makeASound(); // Go "oap"  
Cow& rCow = maddy;  
rCow.makeASound(); // Go "moo"  
Cow* pCow = &maddy;  
pCow->makeASound(); // Go "moo"
```



Sao không
kêu oap???

Liên kết tĩnh

- Với liên kết tĩnh, quyết định “định nghĩa hàm nào được chạy” được đưa ra tại thời điểm biên dịch - rất lâu trước khi chương trình chạy.
- 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



Đa hình tĩnh

- Đa hình tĩnh thích hợp cho các phương thức:
 - được định nghĩa tại một lớp, và được gọi từ một thể hiện của chính lớp đó (trực tiếp hoặc gián tiếp qua con trỏ)
 - được định nghĩa tại một lớp cơ sở và được thừa kế public nhưng không bị override tại lớp dẫn xuất, và được gọi từ một thể hiện của lớp dẫn xuất đó
 - trực tiếp hoặc gián tiếp qua con trỏ tới lớp dẫn xuất
 - hoặc qua một con trỏ tới lớp cơ sở
 - được định nghĩa tại một lớp cơ sở và được thừa kế public và bị override tại lớp dẫn xuất, và được gọi từ một thể hiện của lớp dẫn xuất đó (trực tiếp hoặc gián tiếp qua con trỏ tới lớp dẫn xuất)

Đa hình động

Khi chạy đến đây,
chương trình nhận ra
pCow đang trỏ tới
MadCow, nên gọi
MadCow::makeASound

```
MadCow maddy("Maddy", "white");  
...  
Cow* pCow = &maddy;  
pCow->makeASound();           // Go "oap"
```

- Dynamic function call binding (dynamic binding – liên kết động) là quy trình liên kết một lời gọi hàm với một định nghĩa hàm tại thời gian chạy
 - còn gọi là “run-time” binding hoặc “late binding”
- Với liên kết động, quyết định chạy định nghĩa hàm nào được đưa ra tại thời gian chạy, khi ta biết chắc con trỏ đang trỏ đến đối tượng thuộc lớp nào.
- Đa hình động (dynamic polymorphisme) là loại đa hình cài đặt bằng liên kết động

Hàm ảo

- **Hàm/phương thức ảo – virtual function/method** là cơ chế của C++ cho phép cài đặt đa hình động
- Nếu khai báo một hàm thành viên (phương thức) là **virtual**, trình biên dịch sẽ đẩy lùi việc liên kết các lời gọi phương thức đó với định nghĩa hàm cho đến khi chương trình chạy.
 - nghĩa là, ta bảo trình biên dịch sử dụng liên kết động thay cho liên kết tĩnh đối với phương thức đó
- Để một phương thức được liên kết tại thời gian chạy, nó phải khai báo là phương thức ảo (từ khoá **virtual**) tại *lớp cơ sở*

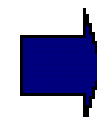
Quay lại ví dụ Bò điên

```
class Cow {  
public:  
    Cow(string name, string color);  
    ~Cow();  
  
    virtual void makeASound();  
    ...  
};
```

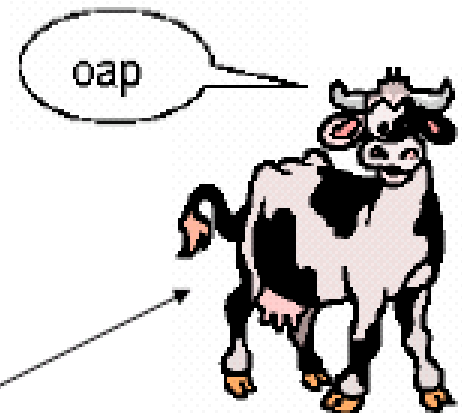
Khai báo hàm ảo,
yêu cầu trình biên dịch
dùng liên kết động

```
int main()  
{  
    MadCow* maddy =  
        new MadCow("Maddy", "white");  
    Cow* cow = maddy;    // Upcasting  
    ...  
    cow->makeASound();    // Go "oap"  
    ...  
}
```

makeASound đã được khai báo là hàm ảo,
Khi chạy, cow trỏ tới MadCow, nên
MadCow::makeASound() được gọi



*cow



Hàm ảo

- Khai báo với từ khoá `virtual` trong định nghĩa lớp
- Không cần tái định nghĩa hàm, nếu định nghĩa hàm nằm ngoài định nghĩa lớp
- Một khi một phương thức được khai báo là hàm ảo tại lớp cơ sở, nó sẽ *tự động* là hàm ảo tại mọi lớp dẫn xuất trực tiếp hoặc gián tiếp.

Các hàm `makeASound()` của `Cow` và các lớp dẫn xuất đều là hàm ảo

```
class Cow {  
public:  
    ...  
    virtual void makeASound();  
    ...  
};  
...  
void Cow::makeASound()  
{... }
```

```
class MadCow : public Cow {  
public:  
    ...  
    virtual void makeASound();  
    ...  
};  
...  
void MadCow::makeASound()  
{... }
```

Không cần
nhưng nên có

Nội dung

1. Upcasting và Downcasting
2. Liên kết tĩnh và Liên kết động
3. Đa hình (Polymorphism)
- ⇒ 4. Lập trình tổng quát (generic prog.)

4. Lập trình tổng quát (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
 - thuật toán đã xác định
- Ví dụ:
 - C: dùng con trỏ void
 - C++: dùng template
 - Java: lợi dụng upcasting
 - Java 1.5: template

Ví dụ: C dùng con trỏ void

- Hàm memcpy:

```
void* memcpy(void* region1,  
             const void* region2, size_t n){  
    const char* first = (const char*)region2;  
    const char* last = ((const char*)region2) + n;  
    char* result = (char*)region1;  
    while (first != last)  
        *result++ = *first++;  
    return result;  
}
```

Ví dụ: C++ dùng template

Khi sử dụng, có thể thay thế ItemType bằng int, string,... hoặc bất kỳ một đối tượng của một lớp nào đó

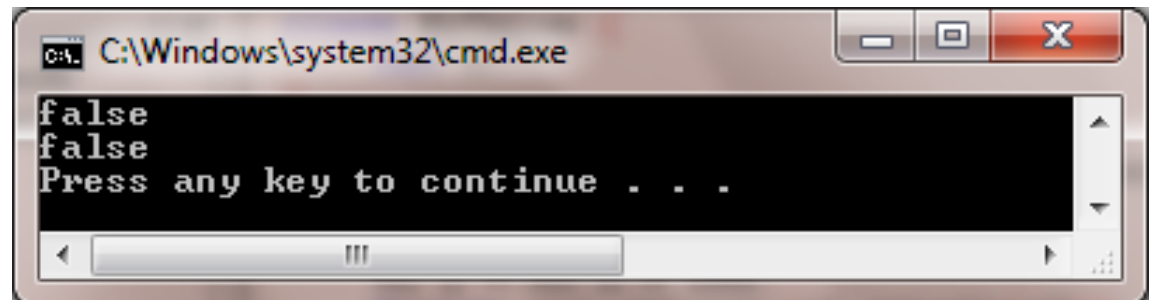
```
template<class ItemType>
void sort(ItemType A[], int count ) {
    // Sort count items in the array, A, into increasing order
    // The algorithm that is used here is selection sort
    for (int i = count-1; i > 0; i--) {
        int index_of_max = 0;
        for (int j = 1; j <= i ; j++)
            if (A[j] > A[index_of_max]) index_of_max = j;
        if (index_of_max != i) {
            ItemType temp = A[i];
            A[i] = A[index_of_max];
            A[index_of_max] = temp;
        }
    }
}
```

Ví dụ: Java dùng upcasting và Object

```
class MyStack {  
    ...  
    public void push(Object obj) {...}  
    public Object pop() {...}  
}  
public class TestStack{  
    MyStack s = new MyStack();  
    Point p = new Point();  
    Circle c = new Circle();  
    s.push(p); s.push(c);  
    Circle c1 = (Circle) s.pop();  
    Point p1 = (Point) s.pop();  
}
```

Nhắc lại - equals của lớp tự viết

```
class MyValue {  
    int i;  
}  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        MyValue v1 = new MyValue();  
        MyValue v2 = new MyValue();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
        System.out.println(v1==v2);  
    }  
}
```



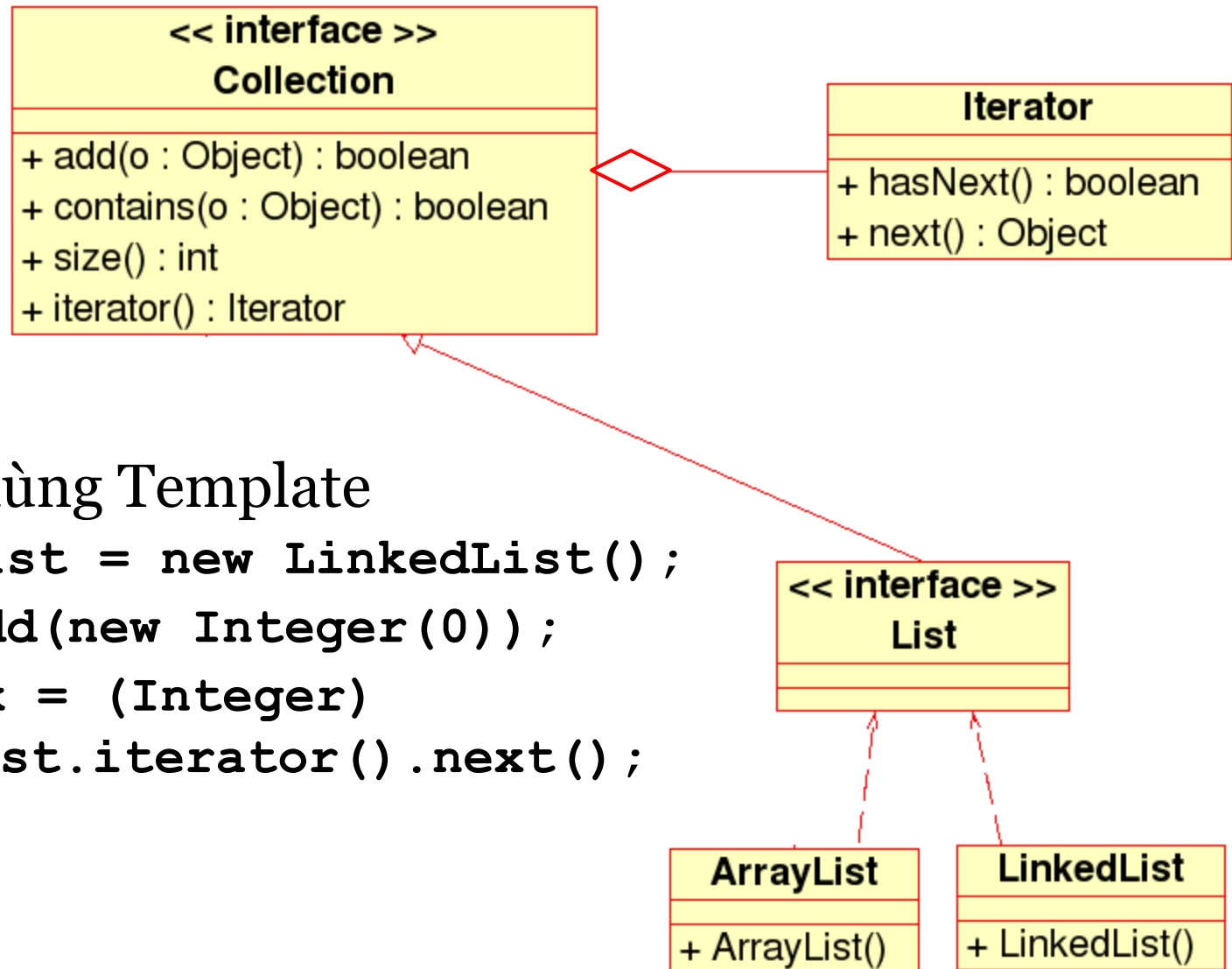
Bài tập

- Viết lại phương thức equals cho lớp MyValue (phương thức này kế thừa từ lớp Object)

```
class MyValue {
    int i;
    public boolean equals(Object obj) {
        return (this.i == ((MyValue) obj).i);
    }
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        MyValue v1 = new MyValue();
        MyValue v2 = new MyValue();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
        System.out.println(v1==v2);
    }
}
```

Ví dụ: Java 1.5: Template



- Không dùng Template

```

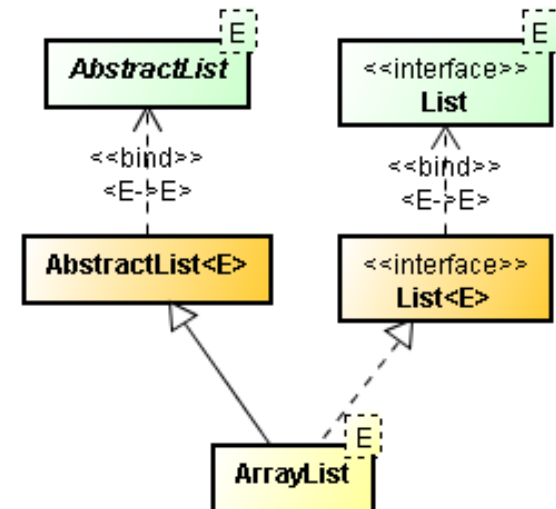
List myList = new LinkedList();
myList.add(new Integer(0));
Integer x = (Integer)
    myList.iterator().next();
  
```

Ví dụ: Java 1.5: Template (2)

- Dùng Template:

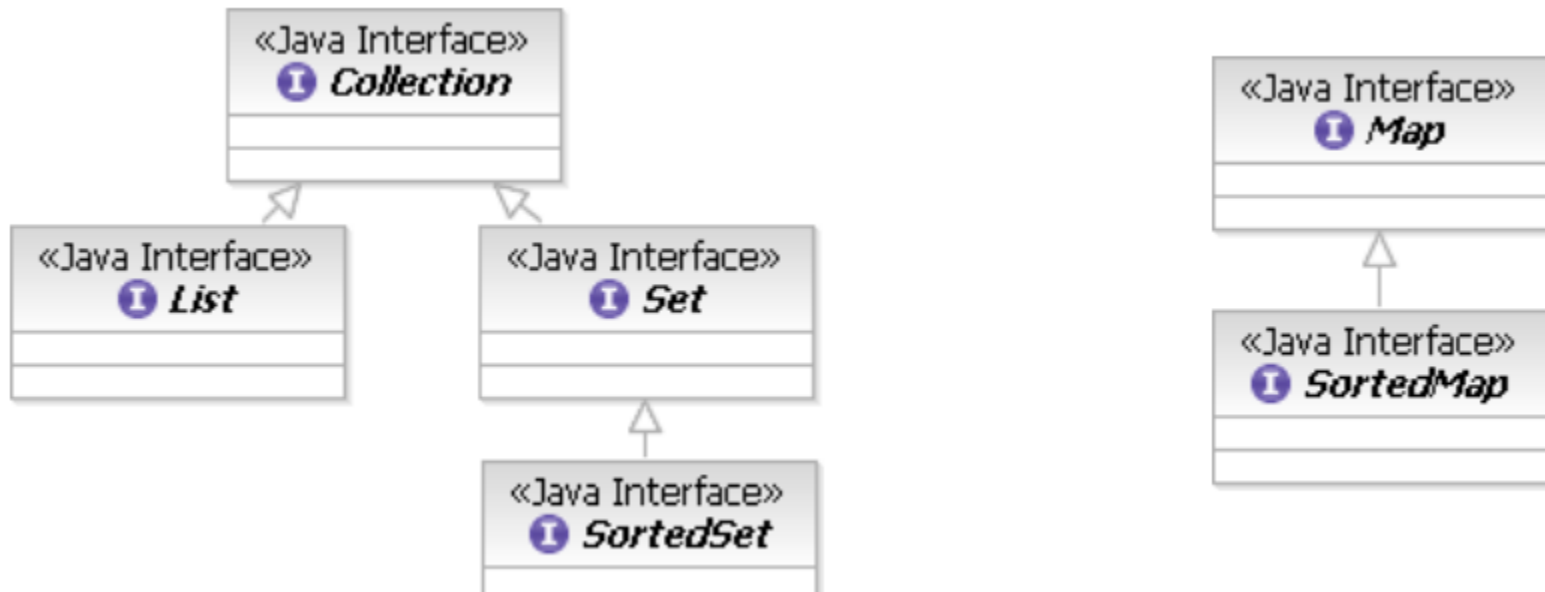
```
List<Integer> myList = new LinkedList<Integer>();
myList.add(new Integer(0));
Integer x = myList.iterator().next();
```

```
//myList.add(new Long(0)); → Error
```



4.1. Java generic data structure

- Collection: Tập các đối tượng
 - 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.

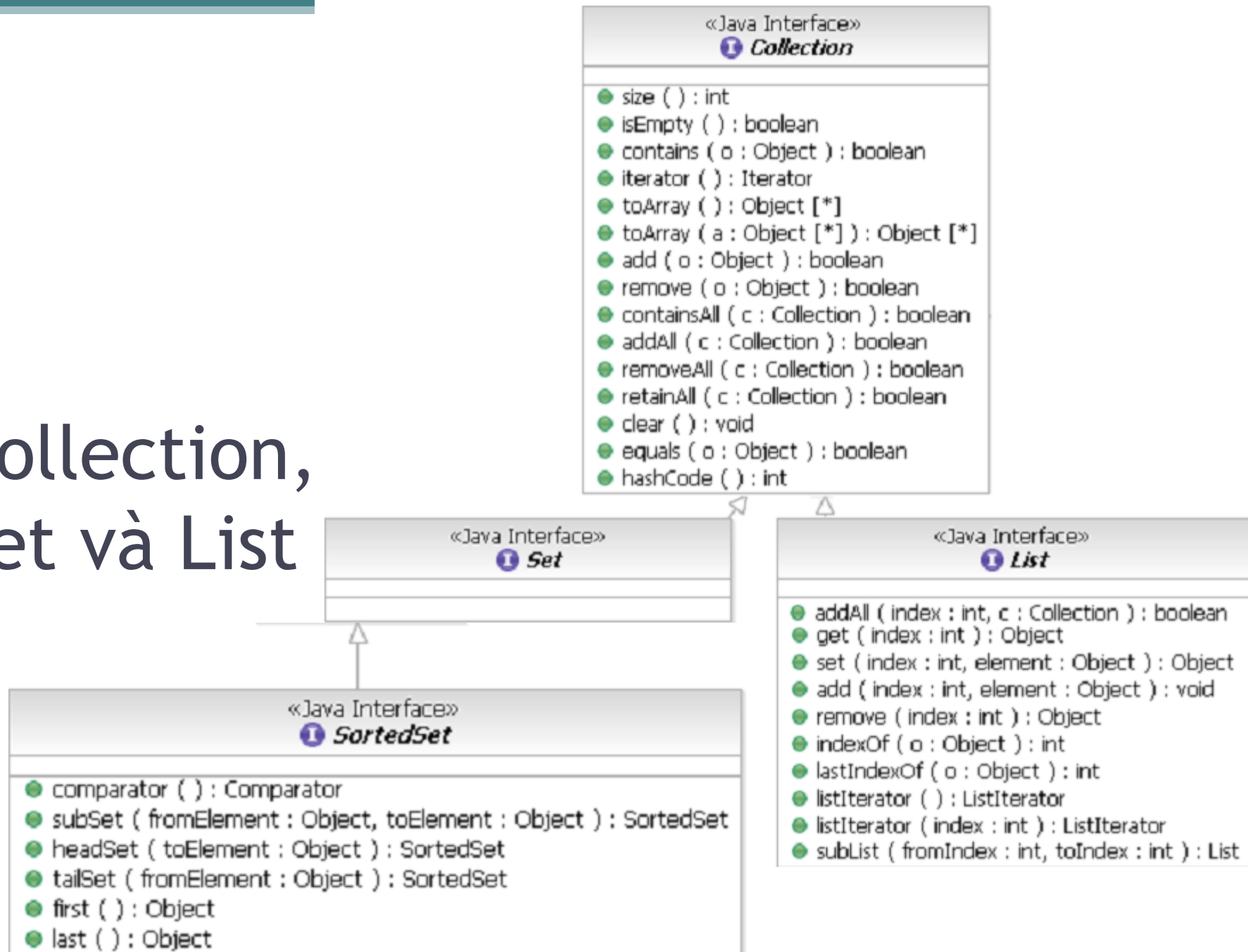


a. Giao diện Collection

- 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>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 |

Collection, Set và List



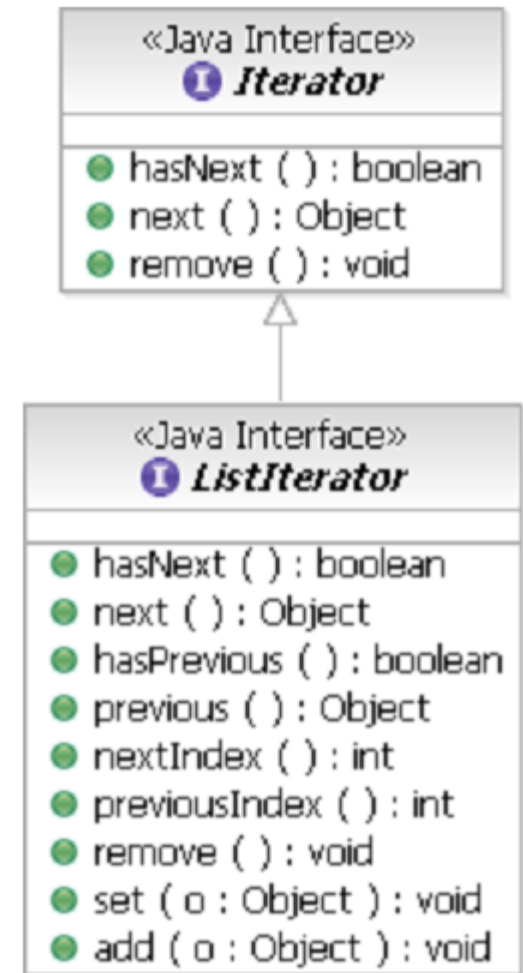
b. Giao diện Map

- 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



c. 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
- ListIterator thêm các phương thức đưa ra bản chất tuần tự của danh sách cơ sở
- Iterator của các tập hợp đã sắp xếp duyệt theo thứ tự tập hợp



Mẫu mã nguồn Iterator

```
Collection c;  
// Some code to build the collection  
  
Iterator i = c.iterator();  
while (i.hasNext()) {  
    Object o = i.next();  
    // Process this object  
}
```

Các giao diện và các cài đặt (Implementation - các lớp thực thi)

| | | 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 |

```
public class MapExample {  
    public static void main(String args[]) {  
        Map map = new HashMap();  
        Integer ONE = new Integer(1);  
        for (int i=0, n=args.length; i<n; i++) {  
            String key = args[i];  
            Integer frequency =(Integer)map.get(key);  
            if (frequency == null) { frequency = ONE; }  
            else {  
                int value = frequency.intValue();  
                frequency = new Integer(value + 1);  
            }  
            map.put(key, frequency);  
        }  
        System.out.println(map);  
        Map sortedMap = new TreeMap(map);  
        System.out.println(sortedMap);  
    }  
}
```


4.2. Định nghĩa và sử dụng Template

```
class MyStack<T> {  
    ...  
    public void push(T x) {...}  
    public T pop() {  
        ...  
    }  
}
```

Sử dụng template

```
public class Test {  
    public static void main(String args[]) {  
  
        MyStack<Integer> s1 = new MyStack<Integer>();  
        s1.push(new Integer(0));  
        Integer x = s1.pop();  
  
        //s1.push(new Long(0)); → Error  
  
        MyStack<Long> s2 = new MyStack<Long>();  
        s2.push(new Long(0));  
        Long y = s2.pop();  
  
    }  
}
```

Định nghĩa Iterator

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

```
public interface Iterator<E>{  
    E next() ;  
    boolean hasNext() ;  
}
```

```
class LinkedList<E> implements List<E> {  
    // implementation  
}
```

4.3. Ký tự đại diện (Wildcard)

```
public class Test {  
    public static void main(String args[]) {  
        List<String> lst0 = new LinkedList<String>();  
        //List<Object> lst1 = lst0; → Error  
        //printList(lst0); → Error  
    }  
  
    void printList(List<Object> lst) {  
        Iterator it = lst.iterator();  
        while (it.hasNext())  
            System.out.println(it.next());  
    }  
}
```

Ví dụ: Sử dụng Wildcards

```
public class Test {  
    void printList(List<?> lst) {  
        Iterator it = lst.iterator();  
        while (it.hasNext())  
            System.out.println(it.next());  
    }  
  
    public static void main(String args[]) {  
        List<String> lst0 =  
            new LinkedList<String>();  
        List<Employee> lst1 =  
            new LinkedList<Employee>();  
  
        printList(lst0);    // String  
        printList(lst1);    // Employee  
    }  
}
```

Các ký tự đại diện Java 1.5

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

Ví dụ wildcard (1)

```
public void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for(int k = 0;k<c.size();k++) {  
        System.out.println(i.next());  
    }  
}
```

→ Sử dụng wildcard:

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

Ví dụ wildcard (2)

```
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) {  
    // rest of the code is the same  
}
```


Template Java 1.5 vs. C++

- Template trong Java không sinh ra các lớp mới
- Kiểm tra sự thống nhất về kiểu khi biên dịch
 - Các đối tượng về bản chất vẫn là kiểu Object