

OBJECT-ORIENTED LANGUAGE AND THEORY
8. POLYMOPHISM

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn



10/10

2

Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
3. Polymorphism
4. Generic programming

10/10

3

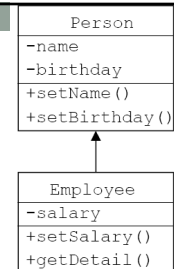
1.1. Upcasting

- Moving up the inheritance hierarchy
- Up casting is the capacity to view an object of a derived class as an object of its basic class.
- Automatic type conversion

10/10

Example

```
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  
    }  
}
```



5

Example (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);
    }
}
```



6

Example (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));
    }
}
```



7

1.2. Downcasting

- Move back down the inheritance hierarchy
- Down casting is the capacity to view an object of a basic class as an object of its derived class.
- Does not convert types automatically
 - Must cast types explicitly.



8

Example

```
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;
    }
}
```



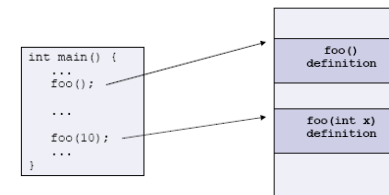
Outline

1. Upcasting and Downcasting
- ➡ 2. Static and dynamic bindings
3. Polymorphism
4. Generic programming



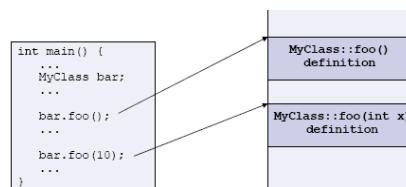
Function call binding

- Function call binding is a procedure to **specify the piece of code that need to be executed** when calling a function
- E.g. C language: a function has a unique name



OOP languages (method call binding)

- For independent classes (are not in any inheritance tree), the procedure is almost the same as function call binding
 - Compare function name, argument list to find the corresponding definition



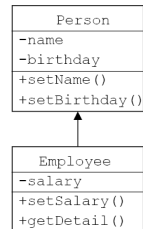
2.1. Static Binding

- Binding at the compiling time
 - Early Binding/Compile-time Binding
 - Function call is done when compiling, hence there is only one instance of the function
 - Any error will cause a compiling error
 - Advantage of speed
- C/C++ function call binding, and C++ method binding are basically examples of static function call binding



Example

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

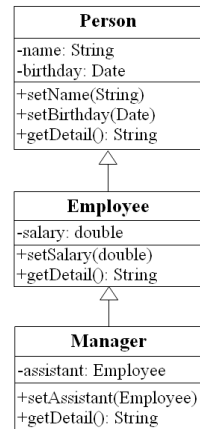


2.2. Dynamic binding

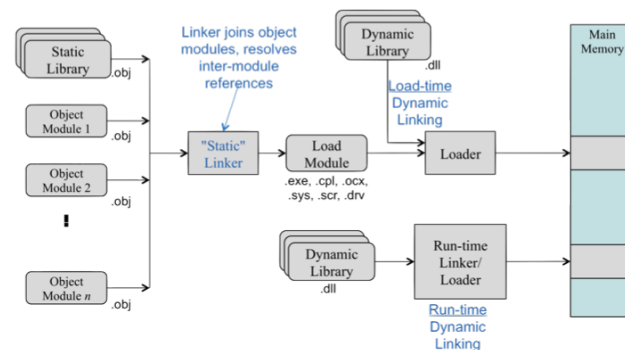
- The method call is done at run-time
- Late binding/Run-time binding
- Instance of method is suitable for called object.
- Java uses dynamic binding by default

Example

```
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());
        }
    }
}
```



Linker and Loader



Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
- ➔ 3. Polymorphism
4. Generic programming

3. Polymorphism

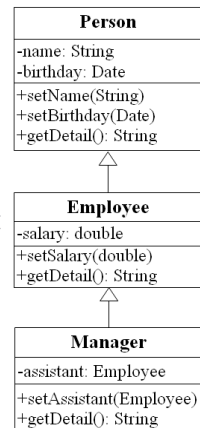
- Polymorphism: many ways of performance, of existence
- Polymorphism in OOP
 - Method polymorphism:
 - Methods with the same name, only difference in argument lists.
 - Object polymorphism
 - **A single symbol to represent multiple different types**
 - **A single interface to entities of different types**

3. Polymorphism (2)

- A single symbol to represent multiple different types
- ➔ Upcasting and 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. Polymorphism (5)

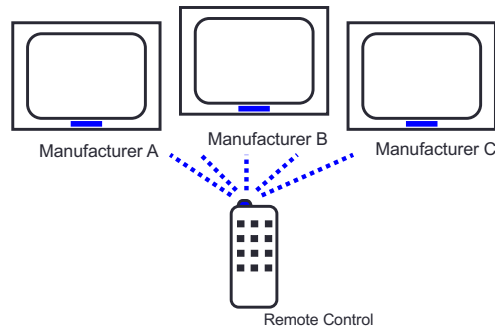
- A single interface to entities of different types
- ➔ Dynamic binding (Java)

- Example:

```
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());
```

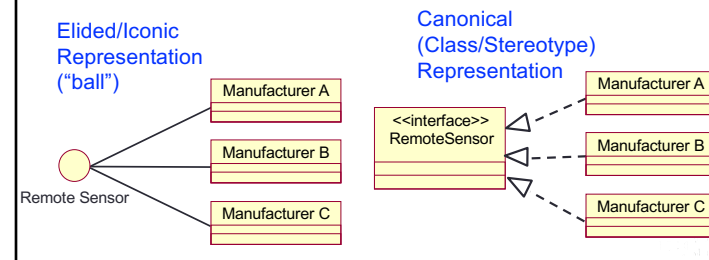
Why Polymorphism?

- The ability to hide many different implementations behind a single interface



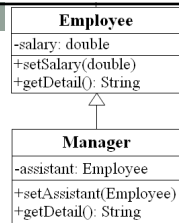
What Is an Interface?

- A declaration of a coherent set of public features and obligations
 - A contract between providers and consumers of services



Other examples

```
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();
}
```



Operator 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 {...}
    }
}
```

Outline

1. Upcasting and Downcasting
2. Static and dynamic bindings
3. Polymorphism
- ➔ 4. Generic programming

4. Generic programming

- Generalizing program so that it can work with different data types, including some future data types
 - Algorithm is already defined
- Example:
 - C: using pointer void
 - C++: using template
 - Java: take advantage of upcasting
 - Java 1.5: Template

Example: C using void pointer

- Malloc function:

```
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;
}
```

Example: C++ using template

When using, we can replace
ItemType by int, string,... or any
object of any class

```
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;
        }
    }
}
```

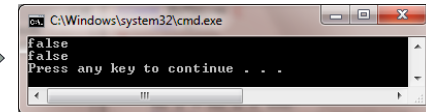
Example: Java using upcasting and Object

```
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();
}
```



Recall – equals of a custom class

```
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);
    }
}
```

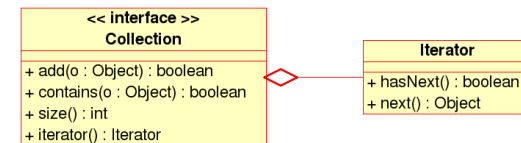


Exercise

- Re-write method **equals** for the class **MyValue** (this method is inherited from the class **Object**)

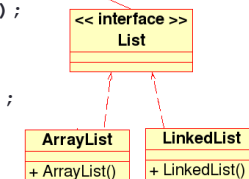


Example: Java 1.5: Template



- Without Template

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

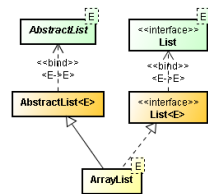


Example: Java 1.5: Template (2)

- Using Template:

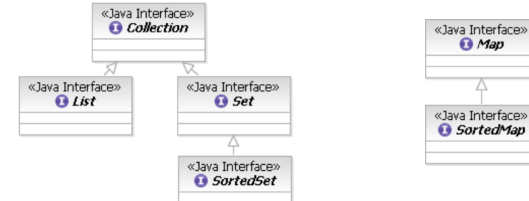
```
List<Integer> myList = new LinkedList<Integer>();
myList.add(new Integer(0));
Integer x = myList.iterator().next();
```

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



4.1. Java generic data structure

- Collection: a set of objects
 - List: a collection of objects that are sequential, consecutive and repeatable
 - Set: a collection of objects that are not repeatable
 - Map: Set of pairs key-value without repetition
- Linking objects in this set to other sets as a dictionary/a telephone book.

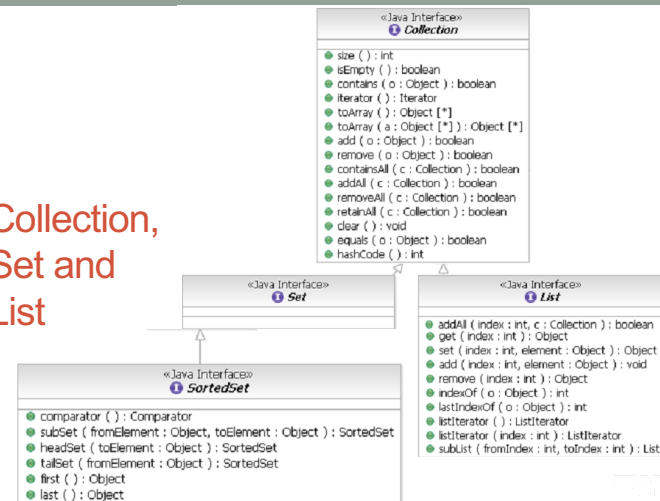


a. Interface of Collection

- Specifies basic interface for manipulating a set of objects
 - Add to collection
 - Remove from collection
 - Check if existing
- Contains methods to manipulate individual objects or a set of objects
- Provide methods to traverse objects in a repeatable collection and convert a collection to an array

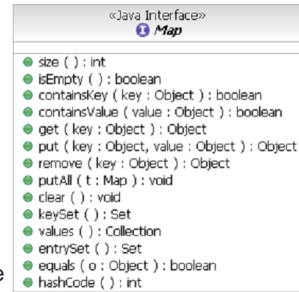


Collection, Set and List



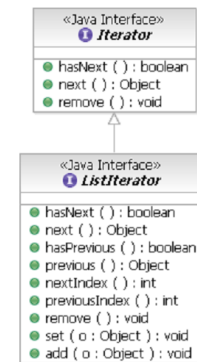
b. Interface of Map

- A basic interface for manipulating a set of pairs key-value
 - Add a pair key-value
 - Remove a pair key-value
 - Get a value of a given key
 - Check if existing (key or value)
- 3 views for the content of collections:
 - Key collection
 - Value collection
 - Mapping collection of key-value



c. Iterator

- Provide a mechanism to visit (repeat) all the members of a collection
 - Similar to SQL cursor
- ListIterator has methods to show the sequential attribute of the basic list
- Iterator of a sorted collection will visit in the sorting order



Source code for Iterator

```

Collection c;
// Some code to build the collection

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

Interface and Implementation

		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

Exercise

- Write a program that get a String array from command line arguments then count the frequency of distinct tokens in the array.



4.2. Defining and using Template

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



Using 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();

    }
}
```



Defining 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. 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());
    }
}
```



Example: Using 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
    }
}
```



Wildcards of Java 1.5

- "? extends Type": Specifies a set of children types of Type. This is the most useful wildcard.
- "? super Type": Specifies a set of parent types of Type
- "?": Specifies all the types or any types.



Example of wildcard (1)

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

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



Example of wildcard (2)

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

→ What is the difference compared with:

```
public void draw(List<? extends Shape> shape) {
    // rest of the code is the same
}
```

Template Java 1.5 vs. C++

- Template in Java does not create new classes
- Check the consistency of types when compiling
 - All the objects are basically of the type Object

Function call vs. Message passing

- Call function
 - Indicate the exact piece of code to be executed.
 - Has only an execution of a function with some specific name.
 - There are no functions with the same name
- Message passing
 - **Request a service from an object and the object will decide what to do**
 - **Different objects will have different reactions/behaviors for a message.**

Message vs. Method

- Message
 - Is sent from an object to another object and does not contain any piece of code to be executed
- Method
 - Method/function in structure programming languages
 - Is an execution of service that is requested in the message
 - Is a piece of code to be executed in order to respond to a message sent to an object

