

Generics



Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of Software Engineering, FIT, UET,

Vietnam National Univ., Hanoi

Contents

- ❖ Concepts
- ❖ Generic class
- ❖ Generic methods

issues

- ❖ **Basically most of the algorithms** are **independent** on data types of items (e.g., sorting, searching...)
- ❖ **Some data structures** do **not depend** on data types of items either (e.g., stacks, linked list...)
- ❖ How to reuse **the same piece of code** with **different data types**?

Solution

Use inheritance?

- ❖ **All the classes** are inherited from **Object class**
- ❖ Objects are upcasted to **Object**

```
public class MyList { // items could be objects of any classes
    public void add(Object o) {...}
    public Object getFirst() {...}
    ...
}
```

Solution: inheritance

limitation

- ❖ Casting is required all the time

```
MyList myPets = new MyList(); // declare a list of type object  
...  
Animal a = (Animal) myPets.getFirst();
```

- ❖ No mechanism for checking errors

```
myPets.add(new Integer(3));  
...  
Animal a = (Animal) myPets.getFirst(); // runtime error
```

Generics

- ❖ **Generic class** introduced in JDK 5.0
- ❖ enables **classes** to accept **parameters** when defining them, **much like** the *familiar parameters* used in method declarations
- ❖ Defining **a type parameter** for **a class** provides a way for you
 - **to re-use** the same code with **different inputs**
 - The difference is that the **input** to formal parameters are **values**, while the **inputs** to **type parameters** are **types**

Generics...

- ❖ **ArrayList** use **Generics** to allow you **to specify** the data type of the **elements** you're intending to add into that ArrayList
- ❖ The way to do so is by defining that **data type** between **<>** when declaring the ArrayList variable
 - **ArrayList**<String> *listOfString* = **new** ArrayList();

No need for casting

- ❖ Generics eliminates the need for casting
- ❖ E.g.,

Code without generics	rewrite with generics
List list = new ArrayList(); list.add("hello java"); String s= (String) list.get(0);	List <String> list = new ArrayList(); list.add("hello Java"); String s= list.get(0);

Defining a generic type

- ❖ You can define **your own Generic types**
 - by declaring a **generic parameter** when defining your class
 - check the [link](#) for more detail

Defining a generic class

```
public class Pair<K>
{
    private K first;
    private K second;

    public Pair() { first= null; second= null; }
    public Pair(K first, K second) { this.first= first; this.second = second; }
    public void setFirst(K newValue) { first = newValue; }
    public K getFirst() { return first; }

    public void setSecond(K newValue) { second = newValue; }
    public K getSecond() { return second; }

}
```

...

```
Pair<String> o = new Pair<String> ("1st", "2nd");
System.out.println(o.getFirst() + "," + o.getSecond());
```

include more parameter

```
public class Pair<K, V>
{
    private K key;
    private V value;

    public Pair() { key = null; value = null; }
    public Pair(K key, V value)
        { this.key = key; this.value = value; }

    public void setKey(K newValue) { key = newValue; }
    public K getKey() { return key; }

    public void setValue(V newValue) { value = newValue; }
    public V getValue() { return value; }

}

...
Pair<Integer, String> o = new Pair<Integer, String> (1, "1st");
System.out.println(o.getKey() + "," + o.getValue());
```

Generic methods

- ❖ are methods that introduce **their own type parameters**
 - similar to **a generic type**, but **the type parameter's scope** is limited to *the methods* there it is declared
- ❖ **Syntax** for a generic method
 - includes **a list of type parameters** inside angle brackets **<>**
 - appear **before** the method's **return type**

Example

❖ Assume we have the class **Pair<K,V>**

```
class Util{  
    public static <K,V> boolean compare(Pair<K,V> p1, Pair<K,V> p2)  
    {  
        return ( p1.getKey().equals p2.getKey() ) &&  
            ( p1.getKey().equals p2.getKey());  
    }  
}
```

```
Pair<Integer, String> p1= new Pair(1, "apple");  
Pair<Integer, String> p2 = new Pair(2, "pineapple");  
boolean same = Util.<Integer, String>compare(p1, p2);
```

This can be left out, the compiler will refer the type that is needed

Bounded type parameters

```
class Util {  
    public static <T> T min(T[] a) //finding the smallest element in a  
    {  
        if (a == null || a.length == 0) return null;  
  
        T smallest = a[0];  
        for (int i = 1; i < a.length; i++)  
            if (smallest > a[i]) //compile error  
                smallest = a[i];  
        return smallest;  
    }  
}
```

*the operator > applies only
primitive types, cannot use
it to compare objects!*

➡ Use a type parameter bounded by the Comparable<T> **interface**

```
public interface Comparable<T>{  
    public int CompareTo(T o);  
}
```

```
public static <T extends Comparable<T>> T min(T[] a){  
    ...  
    smallest > a[i] is replaced with smallest.compareTo(a[i]) > 0  
    //The others are the same as before  
}
```

Bounded type

- ❖ Syntax: `<T extends BoundingType>`
- ❖ `T` and `BoundingType` can be either **interface** or **class**
- ❖ `T` is *a subtype* of `BoundingType`
- ❖ can include multiple `BoundingTypes`
 - `<T extends superClassNames & Interfaces>`
 - We **can't** have more than **one class** in multiple bounds
 - e.g., `<T extends Comparable & Serializable>`

```

class ArrayAlg {
    /** Gets the minimum and maximum of an array of objects of type T.
     * @param a an array of objects of type T
     * @return a pair with the min and max value,
     *         or null if a is null or empty
     */
    public static <T extends Comparable<T>> Pair<T> minmax(T[] a)
    {
        if (a == null || a.length == 0) return null;
        T min = a[0];
        T max = a[0];
        for (int i = 1; i < a.length; i++)
        {
            if (min.compareTo(a[i]) > 0) min = a[i];
            if (max.compareTo(a[i]) < 0) max = a[i];
        }
        return new Pair<T>(min, max);
    }
}

```

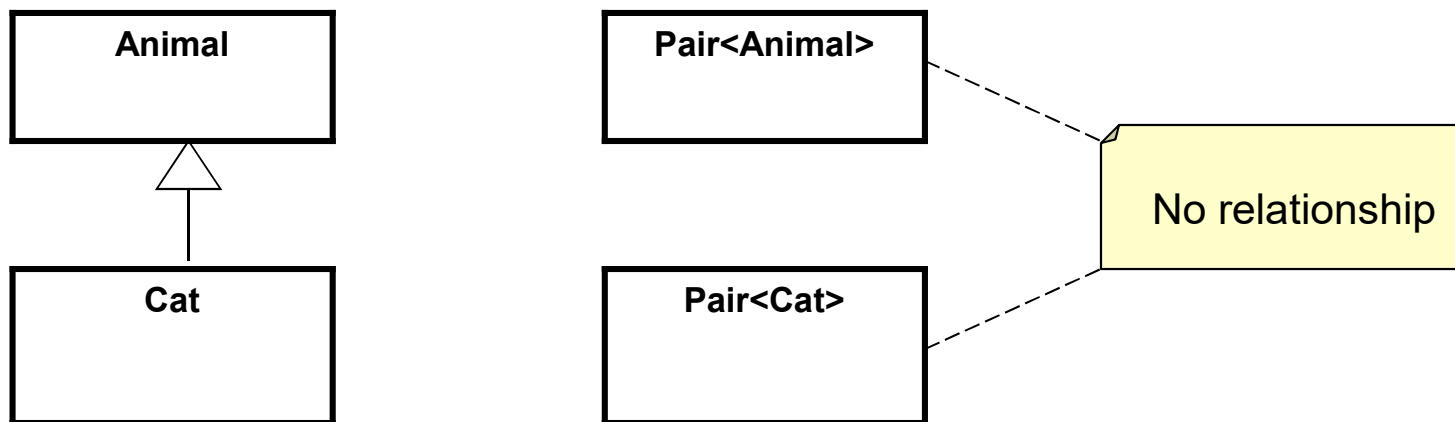
```

...
String[] words = { "Mary", "had", "a", "little", "lamb" };
Pair<String> o = ArrayAlg.minmax(words);
System.out.println("min = " + o.getFirst());
System.out.println("max = " + o.getSecond());

```


Inheritance and generics

- ❖ There's **no** inheritance relationship in generics



How to make generic class/method with **Pair<Animal>** accept **Pair<Cat>** ?

wildcards

- ❖ Create a type **Pair** to be able to work with a subtype of *Animal* as follows
 - **Pair**<? extends *Animal*> *aPair* = ...;

Example 1

```
public class TestAnimal{
    static void makeASymphony( ArrayList<Animal> a){
        for( Animal anAnimal: a){
            anAnimal.makeNoise();
        }
    }
    public static void main(String [] args){
        ArrayList<Animal> pets = new ArrayList<Animal>();
        pets.add(new Dog()); pets.add(new Cat());
        makeASymphony(pets);
    }
}
```

```
class Animal{
```

```
    void makeNoise(){
```

```
        System.out.println("Make noise...");
```

```
    }
```

```
}
```

```
class Dog extends Animal{
```

```
    void makeNoise(){
```

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

```
    }
```

```
}
```

```
class Cat extends Animal{
```

```
    void makeNoise(){
```

```
        System.out.println("Meow")
```

```
    }
```

```
}
```

Example 2

```
public class TestAnimal{
    static void makeASymphony( ArrayList<Animal> a){
        ...
    }
    public static void main(String [] args){
        ArrayList<Dog> dogs = new ArrayList<Dog>();
        dogs.add(new Dog()); dogs.add(new Dog());
        makeASymphony(dogs);
    }
}
```

types don't
match

ArrayList<Animal> cannot be changed
to ArrayList<Dog> because in
Generics there's no such inheritance

```
% javac TestAnimal.java
TestAnimal.java:44: cannot find symbol
symbol   : method
makeASymphony(java.util.ArrayList<Dog>)
location: class TestAnimal
    makeASymphony(dogs);
    ^
1 error
```

Using Upper bounded wildcards

```
static void makeASymphony( ArrayList<? extends Animal> a){  
    for( Animal anAnimal: a){  
        anAnimal.makeNoise();  
    }  
}
```

- ❖ Upper bounded wildcard is used **to relax** the restrictions on a variable
 - E.g., to write a method that works on `List<Integer>`, `List<Double>`, `List<Number>`, we can achieve this using *an upper bounded wildcard*
- ❖ To declare an upper bounded card, use the wildcard character “?” followed by the keyword “`extends`”, followed by **its upper bound**
 - e.g., `List<? extends Number>`

Example

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

List & Iterator interface in java

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

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

Example 1

```
void printList(List<Object> list) { // goals is to print a list of any type,  
                                // but fail  
  
    Iterator it = list.iterator();  
    while (it.hasNext())  
        System.out.println(it.next());  
}
```

```
List<String> list_1 = new LinkedList<String>();
```

```
List<Object> list_2 = list_1; // error
```

```
printList(list_1); // error, trying to print list of String not Object
```


use Unbounded wildcards

```
void printList(List<?> lst) { // allow to print a list of any type  
    Iterator it = lst.iterator();  
    while (it.hasNext())  
        System.out.println(it.next());  
}
```

```
List<Integer> l1 = Arrays.asList(1, 2, 3);
```

```
List<String> l2 = Arrays.asList("one", "two", "three");
```

```
printList(l1);
```

```
printList(l2);
```

Unbounded wildcards

- ❖ The unbounded wildcard type is specified
 - using *the wildcard character* “?”
 - e.g., List<?>
- ❖ **2 Scenarios** where an unbounded wildcard is a useful approach
 1. if you're writing *a method* that can be implemented using **functionality** provided in the **Object class**
 2. When **the code** is using **methods** in **the generic class** that **don't** depend on **the type parameter**
 - E.g, List.size(). List.clear()
 - In fact, Class<?> is so often used because most of the methods in **Class<?>** don't depend on **T**

Lower bounded wildcards

- ❖ Lower bounded wildcard **restricts** the unknown type to be a **super type** of that type
- ❖ It is expressed using
 - The wildcard character “?”, followed by the **super** keyword, followed by its lower bound
- ❖ E.g., to write a method that puts **Integer objects** into a list
 - to maximize **flexibility**, you would like the method to work on `List<Integer>`, `List<Number>`, `List<Object>`
 - You would specify **List<? super Integer>**



THE END