Generics

Hsuan-Tien Lin

Department of CSIE, NTU

OOP Class, May 31-June 1, 2010

Motivation of Generics

```
1  class ANYArray{
2    private ANY[] myarr;
3    public ANYArray(int len){ myarr = new ANY[len]; }
4    protected ANY get(int n){ return myarr[n]; }
5    protected void set(int n, ANY o){ myarr[n] = o; }
6    public void showAll(){
7      for(int i=0;i<myarr.length;i++)
8        System.out.println(myarr[i]);
9    }
10 }
```

- Yes, by identifying the common parts, and then replacing
- manual way: `sed 's/ANY/String/' ANYArray.java > StringArray.java`

C++ Solution (roughly)

```
1   template <class ANY>
2   class Array{
3     private ANY[] myarr;
4     public Array(int len){ myarr = new ANY[len]; }
5     protected ANY get(int n){ return myarr[n]; }
6     protected void set(int n, ANY o){ myarr[n] = o; }
7     public void showAll(){
8       for(int i=0;i<myarr.length;i++)
9         System.out.println(myarr[i]);
10    }
11  }
12
13  {
14    Array<String> sarr(5);
15    sarr.set(3, "lalala");
16  }
```

- basically, the step `sed 's/ANY/String/' ANYArray.cpp > StringArray.cpp` done by compiler
- code automatically **duplicates** during complication as you use `Array<String>`, `Array<Integer>`, `Array<Double>`, ...

Java Solution (roughly)

```
1   class Array<ANY>{
2     private ANY[] myarr;
3     public Array(int len){ myarr = (ANY[])(new Object[len]); }
4     protected ANY get(int n){ return myarr[n]; }
5     protected void set(int n, ANY o){ myarr[n] = o; }
6     public void showAll(){
7       for(int i=0;i<myarr.length;i++)
8         System.out.println(myarr[i]);
9     }
10  }
11
12  {
13    Array<String> sarr(5);
14    sarr.set(3, "lalala");
15  }
```

- the ANY $\rightarrow$ Object step is automatically done by compiler: a true **one-class** solution

How does duplicating solution compare with one-class solution?

type ctrl: duplicating (strong) vs one-class (weak)
downcast: duplicating (no need) vs one-class (need)
overhead: duplicating (high) vs one-class (low)

How can we write one class for arbitrary sets of arbitrary size **while keeping type information**?

Yes, because every String is an Object (6)

No, because of men's instinct

StringArray ObjectArray

Should StringSet extend ObjectSet?

Not necessarily, because need to consider private method/var

Java Solution: Generics (since 1.4)

- no manual duplicating (as opposed to old languages): save coding efforts
- no automatic duplicating (as opposed to C++): save code size and re-compiling efforts
- check type information very strictly by compiler (as opposed to single-object polymorphism): ensure type safety in JVM

Note: type information **erased** after compilation

Type Erasure: Mystery 1

```
1 class Set<T>{
2   Set(){
3     T[] arr = new T[10];
4     arr[0] = new T();
5   }
6 }
```

- cannot new with an “undetermined type” T (no T in runtime)

Type Erasure: Mystery 2

Professor[] parr = new Professor[20];

```
1  class Set<T>{
2  }
3  public class Fun{
4    public static void main(String[] argv){
5      Set<String>[] arr = new Set<String>[20];
6      arr[0].addElement(new Integer(3));
7    }
8  }
```

- cannot create generic array (after type erasure, no type guarantee)

Use of Generics: Java Collection Framework

- interfaces: Collection (Set, List) and Map
- abstract classes: AbstractCollection (AbstractSet, AbstractList) and AbstractMap
- concrete classes: HashSet, ArrayList, HashMap

SimpleArray I

```
1  class SimpleArray<E>{
2      E[] arr; int count;
3
4      void init1(int init_size){
5          arr = new E[init_size];
6      }
7      void init2(int init_size){
8          arr = new Object[init_size];
9      }
10     void init3(int init_size){
11         arr = (E[])(new Object[init_size]);
12     }
13     @SuppressWarnings("unchecked")
14     void init4(int init_size){
15         arr = (E[])(new Object[init_size]);
16     }
17     Object[] arrObj;
18     void init5(int init_size){
19         arrObj = new Object[init_size];
20     }
21
```

E[] != Object[]

SimpleArray II

```
22
23      void add1(E element){
24          arr[count++] = element;
25      }
26
27      void add2(E element){
28          arrObj[count++] = element;
29      }
30
31
32
33
34
35
36
37
38
39
40
41
42
43
```

SimpleArray III

```
44      E pop1(){
45          return arr[count--];
46      }
47
48      E pop2(){
49          return arrObj[count--];
50          return null;
51      }
52
53      E pop3(){
54          return (E)arrObj[count--];
55      }
56
57      @SuppressWarnings("unchecked")
58      E pop4(){
59          return (E)arrObj[count--];
60      }
61
62
63
64
65
```

count < 0?

E != Object?

SimpleArray IV

```
66      boolean equals1(SimpleArray<E> another){
67          if (another.count != count) return false;
68          for(int i=0;i<another.count;i++)
69              if (arr[i] != another.arr[i])
70                  return false;
71          return true;
72      }
73      boolean equals2(SimpleArray<?> another){
74          if (another.count != count) return false;
75          for(int i=0;i<another.count;i++)
76              if (arr[i] != another.arr[i])
77                  return false;
78          return true;
79      }
80
81
82      Object o = new String();
83      SArray<Object> a = new SArray<String>(10);
84
85      a.add(new Professor());
86
87
        Object[] oarr = new String[10];
```

SimpleArray<Object>?

SimpleArray V

```
88      void join1(SimpleArray<E> another){
89          for(int i=0;i<another.count;i++)
90              arr[count++] = another.arr[i];
91      }
92      void join2(SimpleArray<? extends E> another){
93          for(int i=0;i<another.count;i++)
94              arr[count++] = another.arr[i];
95      }
96      void join3(SimpleArray<?> another){
97          for(int i=0;i<another.count;i++)
98              arr[count++] = another.arr[i];
99      }
100
101
102
103
104
105
106
107
108
109
```

E E

E != Object

<E, T, D, L>

<? extends A & B & C>

SimpleArray VI

```
110     E[] toArray1(){
111         return arr;
112     }
113
114     @SuppressWarnings("unchecked")
115     E[] toArray2(){
116         return (E[])arrObj;
117     }
118
119     Object[] toObjectArray1(){
120         return arr;
121     }
122     Object[] toObjectArray2(){
123         return arr;
124     }           arrObj;
125
126     @SuppressWarnings("unchecked")
127     <T> T[] toTArray1(T[] type){
128         return (T[])arr;
129     }
130
131
```

generic method

SimpleArray VII

```
132     void join2(SimpleArray<? extends E> another){
133         for(int i=0;i<another.count;i++)
134             arr[count++] = another.arr[i];
135     }
136
137     <T extends E> void join2withT(SimpleArray<T> another){
138         for(int i=0;i<another.count;i++)
139             arr[count++] = another.arr[i];
140     }
141
142 }
```

More on Type Erasure

Yes:13; No: 7 YES

```
1  ArrayList<String> l1 = new ArrayList<String>();
2  ArrayList<Integer> l2 = new ArrayList<Integer>();
3  System.out.println(l1.getClass() == l2.getClass());
4  System.out.println(l1 instanceof Collection<String>);
```

l1: Yes: 5; No: 11 HAHAHA

l2: Yes: 2; No: 12 HAHAHA