

Generics

Hsuan-Tien Lin

Department of CSIE, NTU

OOP Class, May 23, 2013

How can we write a class for an Integer set of arbitrary size?

How can we write a class for a String set of arbitrary size?

How can we write classes for Integer/String/Double/Professor sets of arbitrary size?

How can we write **one class** for arbitrary sets of arbitrary size?

Motivation of Generics (1/3)

```
1  class StringArray{
2      private String[] myarr;
3      public StringArray(int len){ myarr = new String[len]; }
4      public String get(int n){ return myarr[n]; }
5      public void set(int n, String s){ myarr[n] = s; }
6      public void showAll(){
7          for(int i=0;i<myarr.length;i++)
8              System.out.println(myarr[i]);
9      }
10 }
11 class ProfessorArray{
12     private Professor[] myarr;
13     public ProfessorArray(int len){ myarr = new Professor[len]; }
14     public Professor get(int n){ return myarr[n]; }
15     public void set(int n, Professor p){ myarr[n] = p; }
16     public void showAll(){
17         for(int i=0;i<myarr.length;i++)
18             System.out.println(myarr[i]);
19     }
20 }
```

- Can we avoid writing the same boring things again and again?

Motivation of Generics (2/3)

```
1  class ObjectArray{
2      private Object[] myarr;
3      public ObjectArray(int len){ myarr = new Object[len]; }
4      protected Object get(int n){ return myarr[n]; }
5      protected void set(int n, Object 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 class StringArray extends ObjectArray{
13     public StringArray(int len){ super(len); }
14     public String get(int n){ return (String)super.get(n); }
15     public void set(int n, String s){ super.set(n, s); }
16 }
```

- Yes, by inheritance and polymorphism—everything is an Object

Motivation of Generics (3/3)

```
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
- `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 compilation 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 → Object step is automatically done by compiler: a true **one-class** solution

How does duplicating solution compare with one-class solution?

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

Should `StringSet` extend `ObjectSet`?

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

```
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