# i219 Software Design Methodology
## 6. Object-oriented programming language 3

Kazuhiro Ogata (JAIST)



## Outline of lecture

- Primitive wrapper class
- Generics

# Primitive wrapper class

- For each primitive type, such as int and double, a wrapper class, such as Integer and Double, is prepared.
- An object, such as new Integer("4") of such a wrapper class represents a value, such as 4 of a primitive type, and the object & value are interchangeable.

```
int four = new Integer("4");
Integer five = 5;
```

- But, five can be set to null, while four cannot.
- There are some cases such that a wrapper class should be used instead of a primitive type (explained later).



# Generics (1)

- Classes & interfaces can have type parameters.

*cModifiers* class *ClassName*<$T_1$,…,$T_n$> { … }

type parameters (type variables)

*iModifiers* interface *InterfaceName*<$U_1$,…,$U_m$> { … }

allowing us to design generic data structures, such as generic lists.

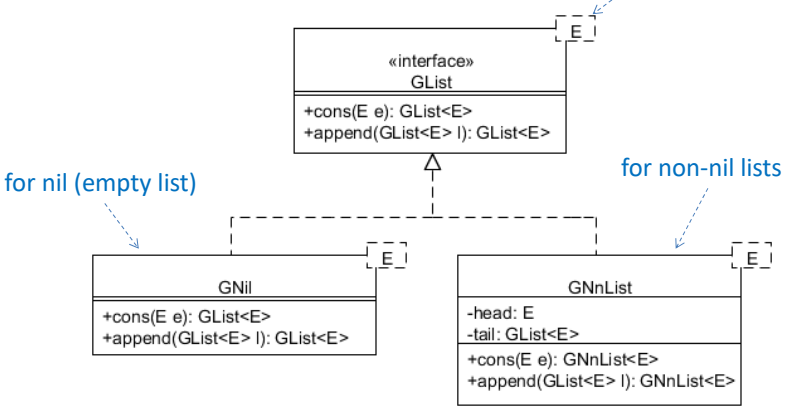
*ClassName*<$T_1$,…,$T_n$> & *InterfaceName*<$U_1$,…,$U_m$> are called generic types; a type variable can be any non-primitive type.

Any name can be used for a type parameter but conventionally one capital letter is used such as E for elements, K for keys, V for values, N for numbers, T for types, …

# Generics (2)

- Let us design generic lists.

type parameter for list elements






# Generics (3)

putting e into the receiver (list) at top

```
public interface GList<E> {
  GList<E> cons(E e);
  GList<E> append(GList<E> l);
}
```

concatenating the receiver (list) and l

<> can also be used instead of <E>, provided that the compiler can infer the type correctly

```
public class GNil<E> implements GList<E> {
  public GList<E> cons(E e) { return new GNnList<E>(e,this); }
  public GList<E> append(GList<E> l) { return l; }
}
```

this refers to the object executing the method, representing nil; making an object representing a singleton list that consists of e.

# Generics (4)

```
public class GNnList<E> implements GList<E> {
  private E head;
  private GList<E> tail;
  public GNnList(E e,GList<E> l) { head = e; tail = l; }
  public GNnList<E> cons(E e) {
     return new GNnList<E>(e,this);
  }

  public GNnList<E> append(GList<E> l) {
     return (GNnList<E>) tail.append(l).cons(head);
  }
}
```

the 1st element of the list

the remainder list

making a list whose head is e and whose tail is this (the object that is executing this method)

need the type cast because of the type of the return type of cons

concatenating tail and l, and putting head into the concatenation



# Generics (5)

```
GList<String> l1 = new GNnList<String>("C++",new GNil<String>());
GList<Double> l4 = new GNnList<Double>(1.41,new GNil<Double>());
l4.cons(1.73).cons(2.43);
GList<Double> l5 = l4.append(l4);
```

type arguments

Invocation (such as GList<String> & GList<Double>) of generic types with type arguments (such as String & Double) are called parameterized types.

For each parameterized type, a new class is not made, but there exists only one class for each generic type (class).

becomes true

```
boolean b = (l1.getClass() == l4.getClass());
```

(the object representing ) the class of an object is returned

# Generics (6)

*ClassName* & *interfaceName* of *ClassName*<$T_1$,…,$T_n$> & *InterfaceName*<$U_1$,…,$U_m$> are called raw types of the generic types.

Assigning a parameterized type to its raw type is permitted.

```
GList l6 = l4;
```
<- no warning is generated (the type of l4 is GList<Double>)

But, javac generates warning for the reverse direction.

```
GList<Double> l7 = l6;
l4 = l6;
```
<- warning is generated, although compilation is done

```
GNnList<Double> l8
      = new GNnList<Double>(2.44,new GNil<Double>());
l4 = l8;
```

the assignment is allowed b/c GNnList<Double> is a subtype of GList<Double>



# Generics (7)

```
GNnList<Double> l9 = l4;
```
<- compilation error

the assignment is not allowed b/c a supertype GList<Double> is assigned to a subtype GNnList<Double>

```
GNnList<double> l10;
```
<- compilation error b/c a primitive type such as double cannot be used as a type argument

# Generics (8)

- Let us design generic binary search trees. Keys should be comparable, however, although there are some classes (such as System) whose objects are not comparable.
- A type parameter can be constrained such that any non-primitive type that extends/implements a class/interface should be used as a type argument of the parameter.

V can be any non-primitive type, but K should be any non-primitive type that extends/implements Comparable<K>

```
public interface TBSTree<K extends Comparable<K>,V> {
   TBSTree<K,V> put(K k,V v);
   V get(K k);
}
```

Comparable<K> is a generic interface in which method int compareTo(K k) is declared



# Generics (9)

V can be any non-primitive type, but K should be any non-primitive type that extends/implements Comparable<K>

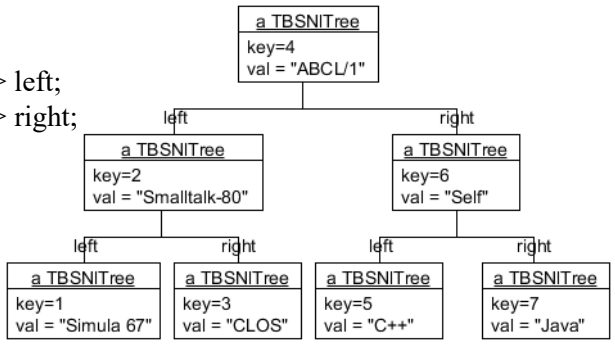
```
public class TBSLeaf<K extends Comparable<K>,V>
   implements TBSTree<K,V> {
  public TBSNlTree<K,V> put(K k,V v) {
    return new TBSNlTree<K,V>(k,v,this,this);
  }
  public V get(K k) { return null; }
}
```

this refers to the object executing the method, representing a leaf; making a singleton binary search tree that consists of k and v

since any keys are not registered in a leaf, null is returned

# Generics (10)

```
public class TBSNlTree<K extends Comparable<K>,V>
   implements TBSTree<K,V> {
  private K key;
  private V val;
  private TBSTree<K,V> left;
  private TBSTree<K,V> right;
```




```
  public TBSNlTree(K k,V v,TBSTree<K,V> l,TBSTree<K,V> r) {
    key = k;  val = v;  left = l;  right = r;  }
```



# Generics (11)

```
  public TBSNlTree<K,V> put(K k,V v) {
    int cmp = k.compareTo(key);
    if (cmp == 0) {  val = v;
    } else if (cmp < 0) {  left = left.put(k,v);
    } else {  right = right.put(k,v); }
    return this;  }

  public V get(K k) {
    int cmp = k.compareTo(key);
    if (cmp == 0) {  return val;
    } else if (cmp < 0) {  return left.get(k);
    } else {  return right.get(k);  }  }
}
```

$o_1$.compareTo($o_2$) =

- a negative integer such as $-1$ if $o_1$ is less than $o_2$
- 0 if $o_1$ equals than $o_2$
- a positive integer such as 1 if $o_1$ is greater than $o_2$

# Generics (12)

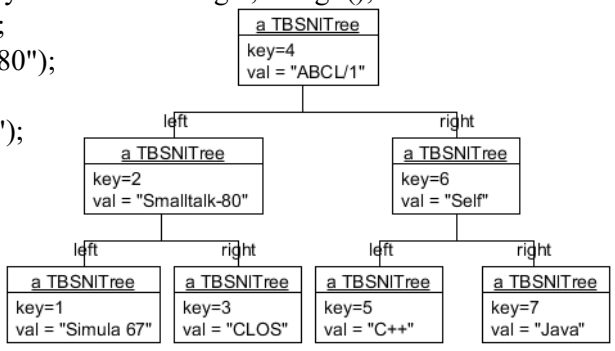
a utility class for use of TBSTree, TBSLeaf & TBSNlTree

```
public class TentativeBinarySearchTree<K extends Comparable<K>,V> {
  private TBSTree<K,V> bst;

  public TentativeBinarySearchTree() { bst = new TBSLeaf<K,V>(); }

  public void put(K k,V v) {
    bst = bst.put(k,v); }
  }

  public V get(K k) {
    return bst.get(k);
  }
}
```



# Generics (13)

```
TentativeBinarySearchTree<Integer,String> oopls
   = new TentativeBinarySearchTree<Integer,String>();
oopls.put(4,"ABCL/1");
oopls.put(2,"Smalltalk-80");
oopls.put(6,"Self");
oopls.put(1,"Simula 67");
oopls.put(3,"CLOS");
oopls.put(5,"C++");
oopls.put(7,"Java");
```




```
TentativeBinarySearchTree<System,String> tbst;
```

compilation error b/c System does not implement Comparable<System>

# Generics (14)

an implementation of natural numbers a la Peano

```
public interface Nat extends Comparable<Nat> { Nat plus(Nat y); }
        an object of Zero represents zero
public class Zero implements Nat { public Nat plus(Nat y) { return y; }
  public int compareTo(Nat y) { if (y instanceof Zero) { return 0;}
                                  else { return -1; } } }
        an object of NzNat represents n+1
public class NzNat implements Nat { private Nat n;
  public NzNat(Nat y) { this.n = y; }
  public Nat pred() { return n; }
  public NzNat plus(Nat y) { return new NzNat(n.plus(y)); }
  public int compareTo(Nat y) { if (y instanceof Zero) { return 1; }
                  else { return n.compareTo(((NzNat) y).pred()); } } }
```

TentativeBinarySearchTree<NzNat,String> oopls2;

compilation error b/c NzNat does not implement Comparable<NzNat> but Comparable<Nat>, although objects of NzNat are comparable



# Generics (15)

lower bounded wildcard representing an arbitrary type that is the same as K or a supertype of K

K extends Comparable<K> ➡ K extends Comparable<? super K>

✓ ? extends K: upper bound wildcard representing an arbitrary type that is the same as K or a subtype of K

✓ ?: unbounded wildcard, the same as ? extends Object

as follows

```
public interface BSTree<K extends Comparable<? super K>,V> { … }
public class BSLeaf<K extends Comparable<? super K>,V>
   implements BSTree<K,V> { … }
public class BSNlTree<K extends Comparable<? super K>,V>
   implements BSTree<K,V> { … }
public class BinarySearchTree<K extends Comparable<? super K>,V> { … }
```

then it is possible to use NzNat as the 1st argument

```
BinarySearchTree<NzNat,String> oopls
        = new BinarySearchTree<NzNat,String>();
```

# Summary

- Primitive wrapper class
- Generics