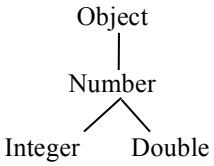
Lower-bounded type parameters

We give examples of generic lower-bounded type parameters.



Function add

Procedure add, to the right, adds the elements of int array b to ArrayList c. One can use add, for example, to put initial values into an array list.

```
/** Add elements of b to c*/
public static void add(int[] b,
            ArrayList<? super Integer> c) {
   for (int e : b) {
     c.add(e);
   }
}
```

The type of the elements of ArrayList c is

    ? super Integer

which is one of Integer, Number, and Object. We could have used simply ArrayList<Integer> for the type of c, but the use of the wild-card lower-bounded type provides flexibility. Below is an example of its use:

```
ArrayList<Integer> al= new ArrayList<>();
int[] ia= {1, 5, 6, 4};
add(ia, al);
```

A more flexible function add

```
/** Add elements of b to c*/
public static <T> void add(T[] b,
            ArrayList<? super T> c) {
   for (T e : b) {
     c.add(e);
   }
}
```

But why not make the procedure more flexible and have it work not only for type Integer but for any class/type T? We rewrite add as a generic procedure, as shown to the right.

Since parameter array b is an in-parameter —its values are extracted and used but the array is not changed— you might think you could write the declaration of b as

    <? extends T>[] b

But this notation is not allowed for arrays, and it is in fact unnecessary. If T1 is a subclass/subtype of T, then T1[] is a subtype of T[]. Consider the code below. For the call add(ia, al), type T of method add is inferred to be Number, because argument al has type ArrayList<Number>. Thus, the type Integer[] of argument ia is cast upward to Number[]. This code prints the characters "[1, 5, 6, 4]".

```
ArrayList<Number> al= new ArrayList<>();
Integer[] ia= {1, 5, 6, 4};
add(ia, al);
System.out.println(al);
```

Copy an ArrayList

```
/** Copy src to dest. */
public static <T> void copy(
            ArrayList<? extends T> src,
            ArrayList<? super T> dest) {
   dest.clear();
   for (T e : src) dest.add(e);
}
```

This example shows the use of both an upper-bounded wildcard and a lower-bounded wildcard.

Parameter src is an in-parameter: values are extracted and the data structure is not changed. Therefore, the corresponding argument can be an ArrayList of any subclass of T.

Parameter dest is an out-parameter: the ArrayList is changed, but its elements are not extracted and used. Therefore, the corresponding argument can be an ArrayList of any superclass or superinterface of T.

Here is an example of a call on copy, using procedure add, above, to create an initial ArrayList.

```
ArrayList<Integer> x1= new ArrayList<>();
Integer[] xa= {1, 5, 6, 4};
add(xa, x1);
ArrayList<Object> x2= new ArrayList<>();
copy(x1, x2);
```

Lower-bounded type parameters

Finding the maximum value in an ArrayList

This example shows the contortions one can go through to make a method as flexible as possible.

Generic function max returns the largest element in its ArrayList parameter b. The obvious generic type parameter to use is:

(1) <T extends Comparable<T>>

Why? Class T has to implement interface Comparable so that function compareTo is available.

Instead, however, the type parameter is

(2) <T extends Comparable<? super T>>

We concoct an example to show that type parameter (2) allows more calls than type parameter (1). Look at classes Person and Student to the right. Note that Student implements Comparable<Person>, not Comparable<Student>. So, with variable as declared as

ArrayList<Student> as

the call

max(as)

is syntactically incorrect if type parameter (1) is used and syntactically correct if (2) is used.

```
/** Return the largest element in b.
  * Precondition: b contains at least one element. */
public static <T extends Comparable<? super T>>
                                    T max(ArrayList<T> b) {
   Iterator<T> it= b.iterator();
   T w= it.next();
   while (it.hasNext()) {
        T e= it.next();
        if (e.compareTo(w) > 0) w= e;
   }
   return w;
}
```

```
public class Person {
   public int age;
}

public class Student extends Person
                    implements Comparable<Person> {
   public @Override int compareTo(Person p) {
      return age - p.age;
   }
}
```

We urge you to pop all this into a Java program and play with it.

The example given above seems far-fetched. Why didn't class Person implement Comparable<Person> so that class Student didn't have to? Well, maybe someone else wrote class Person and it can't be changed.

But class java.util.Collections contains several generic method that use notation (2) to provide utmost flexibility, most notably, the following sort procedure. So it's good to have seen and studied this use of notation (2).

```
/** Sort li according to the order induced by c. */
public static <T> void sort(List<T> li, Comparator<? super T> c)
```