

GENERIC PROGRAMMING: ADVANCED MATERIAL

ITERATORS

An **iterator** is an object that allows a user to process a collection of items in a one-by-one fashion. Iterators have two methods: `next` (which returns the next item in the collection) and `hasNext` (which returns `true` if there are still items to be processed, and `false` otherwise). Iterators are objects constructed from classes that implement the `Iterator` interface. Thus, a reference variable of type `Iterator` can point to an iterator object. Some predefined classes (`ArrayList` is one example) have a method named `iterator` that returns the iterator for that class. For example, consider the program in Fig. 1.

```
 1 import java.util.*;    // need to import ArrayList and Iterator
 2 class IteratorExample
 3 {
 4    public static void main(String[] args)
 5    {
 6       ArrayList<String> sal;
 7       sal = new ArrayList<String>();
 8
 9       sal.add("Bert");
10       sal.add("Ernie");
11       sal.add("Grover");
12
13       Iterator<String> itr;
14       itr = sal.iterator();     // get iterator
15
16       while (itr.hasNext())
17          System.out.println(itr.next());
18    }
19 }
```

Figure 1

The call of the `iterator` method in the `ArrayList` object returns an iterator for the `ArrayList`:

```
14       itr = sal.iterator();     // get iterator
```

We then use the `hasNext` and `next` methods in the `itr` object in a `while` loop to traverse the data in the `ArrayList`:

```
16       while (itr.hasNext())
17          System.out.println(itr.next());
```

`Iterator` is a generic interface. Thus, when we declare `itr`, we specify the base type within angle brackets:

```
13       Iterator<String> itr;
```

We specify the base type `String` here because the base type of the `sal` `ArrayList` is `String`.

What You Can and Cannot Do in a Generic Class

The type you pass to a generic class must be a class. It cannot be a primitive type. For example, it is illegal to specify int when you delcar a reference variable to a generic class or call its constructor:

```
OneThing<int> p1;            // Illegal--cannot use primitive type
p1 = new OneThing<int>();   // Illegal--cannot use primitive type
```

This restriction is not serious—we can simply use the wrapper class in place of the primitive type.

The `BadGeneric1` class in Fig. 2 illustrates another restriction on generic classes.

```
 1 class TestBadGeneric1
 2 {
 3    public static void main(String[] args)
 4    {
 5       Hasf h1 = new Hasf();
 6       BadGeneric1<Hasf> g1 = new BadGeneric1<Hasf>(h1);
 7       g1.m();
 8       BadGeneric1<String> g2 = new BadGeneric1<String>("hello");
 9       g2.m();
10    }
11 }
12 //==================================================
13 class Hasf
14 {
15    public void f()
16    {
17       System.out.println("hello");
18    }
19 }
20 //==================================================
21 class BadGeneric1<T>
22 {
23    T r;
24    //----------------------------------
25    public BadGeneric1(T rr)
26    {
27       r = rr;
28    }
29    //----------------------------------
30    public void m()
31    {
32       System.out.println(r.toString());  // legal
33       r.f();                             // illegal
34    }
35 }
```

Figure 2

`r` (see line 23) points to an object of some type. All objects necessarily have a `toString` method because `toString` is in the `Object` class. Thus, it is legal to invoke the `toString` method via `r`, as we do on line 32:

```
32       System.out.println(r.toString());  // legal
```

However, not all objects have an `f` method. Thus, it is illegal to invoke `f` via `r`, as we do on line 33:


```
33       r.f();                              // illegal
```

On line 6 in Fig. 2, the class `Hasf` is passed to the type parameter `T` in the generic class `BadGeneric`:

```
 6       BadGeneric1<Hasf> g1 = new BadGeneric1<Hasf>(h1);
```

`Hasf` has an `f` method (see lines 13 to 19). Nevertheless, the call of `f` via `r` on line 33 is illegal. Remember that `BadGeneric1` is a *generic* class. It should work properly regardless of the class we pass it. If the call of `f` via `r` on line 33 were legal, then the `BadGeneric1` class would not work properly if we passed it a class without an `f` method, as on line 8:

```
 8       BadGeneric1<String> g2 = new BadGeneric1<String>("hello");
```

We can, however, make a change to `BadGeneric1` that makes line 33 legal. We simply change line 21 to

```
class BadGeneric1<T extends Hasf>
```

The phrase "`extends Hasf`" here tells the compiler that the class passed to `T` will be `Hasf` or some subclass of `Hasf`. Because `Hasf` has an `f` method, the object to which `r` points will now necessarily have an `f` method. For this reason, the compiler will now allow the call on line 33 of `f` via `r`. By adding "`extends Hasf`", however, we have restricted the types we can pass to `BadGeneric1`. We can now pass only `Hasf` or a subclass of `Hasf`. Thus, our modification causes line 8 to become illegal because it passes `String`, which is not a subclass of `Hasf`, to `BadGeneric1`. In this example, we are extending the type parameter T with a class (`Hasf`). We can also extend a type parameter with an abstract class or an interface. In all cases, we use the reserved word `extends`, even if we are extending the type parameter with an interface.

You cannot use the type parameter in a generic class to create an object or an array. For example, the statements on lines 3 and 4 of Fig. 3 are illegal.

```
 1  class BadGeneric2<T>
 2  {
 3     T t  = new T();                 // illegal
 4     T[] a = (T[]) new T[20];        // illegal
 5  }
 6  //=================================================
 7  class GoodGeneric1<T>
 8  {
 9     T t  = (T)new Object();         // legal
10     T[] a = (T[]) new Object[20];   // legal
11  }
12  //=================================================
13  class GoodGeneric2<T extends C>    // C is a class
14  {
15     T t  = (T)new C();              // legal
16     T[] a = (T[]) new C[20];        // legal
17  }
```

Figure 3

There is a good reason why this usage is illegal: the compiler cannot translate a statement that constructs an object or array without knowing the type of the object or array. The code that the compiler generates depends of this type. For example, the code to which the compiler translates

```
Integer t = new Integer(3);
```

is different from the code to which the compiler translates

```
Double t = new Double(7.5);
```

For one thing, the objects have different sizes (it takes eight bytes to store 7.5 within a `Double` object, but only four bytes to store 3 within an `Integer` object). So how could the compiler correctly translate

```
T t = new T();
```

in a generic class with type parameter `T`? In fact, there is no translation that would be correct. If the compiler assumes `T` is a specific type and generates code accordingly, then this code would be correct for that type only, and we would no longer have a generic class.

Now consider the code on line 9 Fig. 3:

```
 9     T t  = (T)new Object();         // legal
```

It looks similar to the code on line 3:

```
 3     T t  = new T();                 // illegal
```

However, unlike the code on line 3, it is legal. On line 9, we are calling the constructor for a *specific* class (`Object`). Thus, the compiler knows what code to generate to create the object. For the same reason, the code on lines 10, 15, and 16 is also legal.

Extending the Type Parameter with the Comparable Interface

The `Comparable` interface is a predefined interface in the `java.lang` package. It contains only the abstract method compareTo`(Object obj)`. Thus, any class that implements `Comparable` must have a `compareTo` method, or else it would be an abstract class. The predefined classes whose objects have a linear order—such as `Integer`, `Double`, and `String`—implement `Comparable`. Thus, they all contain a `compareTo` method.

Now consider the `OrderedPair` class in Fig. 4. An object created from `OrderedPair` itself contains two objects: one pointed to by `x`, the other pointed to by `y`. The `smaller` method (lines 20-26) returns `x` or `y` depending on which object is smaller


```
 1 class TestOrderedPair
 2 {
 3    public static void main(String[] args)
 4    {
 5       OrderedPair<Integer> p = new OrderedPair<Integer>(1, 2);
 6       System.out.println(p.smaller());
 7    }
 8 }
 9 //==================================================
10 class OrderedPair<T>
11 {
12    private T x, y;
13    //------------------------------------
14    public OrderedPair(T xx, T yy)
15    {
16       x = xx;
17       y = yy;
18    }
19    //------------------------------------
20    public T smaller()
21    {
22       if (x.compareTo(y) < 0)     // illegal
23          return x;
24       else
25          return y;
26    }
27 }
```

Figure 4

To determine which object is smaller, the `smaller` method invokes the `compareTo` method via `x`:

```
22       if (x.compareTo(y) < 0)     // illegal
```

This invocation, however, is illegal. Because the type specified for `x` is the type parameter, we can invoke via `x` only those methods that are in the `Object` class (and, therefore, would necessarily be in the object to which `x` points). We can fix this problem simply by changing line 1 to

```
class OrderedPair<T extends Comparable>
```

The phrase “`extends Comparable`” here tells the compiler that the class passed to `OrderedPair` implements the `Comparable` interface. Thus, it will necessarily have the `compareTo` method. This modification makes line 22 legal. However, it also restricts the classes that can be passed to those that implement Comparable. For example, if `r1` and `r2` are assigned `Random` objects with

```
Random r1 = new Random();
Random r2 = new Random();
```

then the following statement would be illegal because `Random` does not implement `Comparable`:

```
OrderedPair<Random> p = new OrderedPair<Random>(r1, r2);
```

HOMEWORK PROBLEMS

1) Compile the program `TestBadGeneric1.java`. What is the error message generated by the compiler?

2) Write a program that reads integer numbers from a file, placing each on a linked list. It then traverses the linked list, displaying each number on the list. Use the predefined generic `LinkedList` class. Implement the traverse of the linked list using the iterator provided by `LinkedList`. The iterator provided by `LinkedList` is constructed from a class that implements the generic interface `ListIterator`—not `Iterator`. Like `Iterator`, `ListIterator` has the `next` and `hasNext` methods. It also has some methods not in `Iterator`. Both `LinkedList` and `ListIterator` are in `java.util`. Test your program on a file that contains 1 to 10 in ascending order.