L62

Monday, February 7, 2022 6:41 PM

Generics - Writing Custom Generic class, Bounded Types, Wildcards and Types

Writing Custom Generic Class

Q1. We can write custom generic classes/interfaces where the complete class/interface is parameterized. For example:

```
class A<T> {
	private T t;
	public A(T t) {
		this.t = t;
	}
	public T getValue() {
		return t;
	}
	public void setValue(T t) {
		this.t = t;
	}
}
```

In the above code, the **type parameter** <T> is visible in the entire scope/body of class A.

We can also have a normal non-parameterized class contain one or more generic methods and generic constructors.

Below is a crazy example of a class with generic constructor and a generic method, only to explain the syntax:

```
public class CrazyGenericExample {
	public <Q> CrazyGenericExample(Q q) {			//statement 1System.out.println("q = " + q);
	}
	public <X, Y, Z> Z doSomething(X x, Y y, Z z) {		//statement 2System.out.println("x = " + x + ", y = " + y + ");
		return z; // this method simply returns the value in the parameter z}
}
```

In the above code, since the class CrazyGenericExample is not parameterized, the generic constructor and the generic method have to explicitly mention the type parameters they use.

Note the **type parameter <Q>** in **statement 1** used by the constructor.

Similarly, note the **type parameters <X Y Z>** in **statement 2** in the method declaration.

These **type parameters** which are explicitly declared by constructors and methods are visible only to their respective scopes unlike the **type parameters** declared in the class declaration statement.

Also note that **a generic method** can be both **non-static** and **static**.

We normally do not write custom generic classes unless we are developing framework level APIs which will be used by others. However, it is good to know the syntax.

See and retype the below code.

```java
package q11394;
public class CustomGenericClassExample {
    public static void main(String[] args) {
        A<String> a1 = new A<>("Ganga");
        System.out.println("a1.getValue() = " + a1.getValue());
        A<Boolean> a2 = new A<>(true);
        System.out.println("a2.getValue() = " + a2.getValue());
    }
}
class A<T> {
    private T t;
    public A(T t) {
        this.t = t;
    }
    public T getValue() {
        return t;
    }
    public void setValue(T t) {
        this.t = t;
    }
}

```

Bounded Types

Q1. There would be situations when we want to have a generic class or a method, whose types are restricted.

For example, if we want to write a method which accepts two collections and returns the collection with more elements, we **cannot** use a unrestricted type parameter as given in the below code:

```
class Util {
	public static <T>T largerCollection(T collection1, T collection2) {
		...
	}
}
```

In the above code, the actual **type argument** passed into **T** can be a String or an Integer or any other type. There will be no enforcement by the compiler that it should be an instance of type Collection.

There is a way in generics by which we can restrict the type passed in **T** to be a subtypes of Collection. Below is the syntax:

```
class Util {
	public static <T extends Collection>T largerCollection(T collection1, T collection2) {
		return (collection1.size() > collection2.size())? collection1 : collection2;
	}
```

}
In the above code the syntax **<T extends Collection>**, informs the compiler that it should ensure that only subtypes of Collection can be passed as **type arguments** for the **type parameter T.**

Note that since the **type argument** passed for **T** will be a subtype of Collection, we are directly able to call the method size() present in references collection1 and collection2 which will be instances of some subtype of Collection class.

The extends in the code fragment <T **extends** Collection> is used for both classes and interfaces.

Extends can also be used to restrict the type to be a subtype of multiple types at the same time. For example:
<T extends A & B & C>
In the above code the **type argument** which will be passed for **T** has to be a subtype of A, B and also C. However, care must be taken that if one multiple types is a class, then the class must be placed before the interfaces in the list of types.

See and retype the below code.

```java
package q11395;
import java.util.*;
public class BoundedTypeExample {
    public static void main(String[] args) {
        List<String> namesList = new ArrayList<>();
        namesList.add("Ganga");
        namesList.add("Godavari");
        namesList.add("Krishna");
        Set<String> namesSet = new LinkedHashSet<>();
        namesSet.add("Ganga");
        namesSet.add("Godavari");
        namesSet.add("Krishna");
        namesSet.add("Yamuna");
        namesSet.add("Narmada");
        System.out.println("largerCollection : " + largerCollection(namesList, namesSet));
    }
    public static <T extends Collection> T largerCollection(T collection1, T collection2) {
        return (collection1.size() > collection2.size())? collection1 : collection2;
    }
}

```

Wildcards and Types

Q1. The question mark character ? is called the wildcard. It represents an unknown type. Its usage and meaning in different contexts are given below with examp les.

- **unbounded wildcard** - eg: **List<?>**, represents a **List** of unknown type. We use such code when we want to work only with the methods in **List** interface without the knowledge of the type of elements it stores.
- **upper bounded wildcard** - eg: **List<? extends A>**, represents a **List** whose elements are of type A or a subtype of A. (Note **extends** is used for both a class and an interface).
- **lower bounded wildcard** - eg: **List<? super A>**, represents a **List** whose elements are of type A, or a super type of A. (Note **super** is used for both a class and an interface).

Note the below thumb rules while choosing to use wildcards:

- **unbounded wildcard** - should be used when we want to access only the methods in Object class on the parameters passed.
- **upper bounded wildcard** - should be used as parameters when we want to send data to methods as parameters. It can be thought of as an **in** parameter. It serves a **read-only** copy of data which cannot be manipulated.
- **lower bounded wildcard** - should be used as parameters when we want to retrieve processed data from methods via parameters. It can be thought of as an **out** parameter. It allows for data manipulation.
- **no wildcard** - do not use a wildcard, instead use a specific type, when we want to use a type to be acting as both **in** and **out** parameter, meaning when we want a parameter to carry data and also be open for manipulation.

See and retype the below code.

```java
package q11396;
import java.util.*;
public class WildCardTypesDemo {
    public static void main(String[] args) {
        List<? extends Number> upperList = Arrays.asList(2, 3, 4);
        List<? super Number> lowerList = new ArrayList<>();
        List<Integer> noBoundsList = new ArrayList<>();
        upperBoundedMethod(upperList);
        lowerBoundedMethod(lowerList);
        noBoundedMethod(noBoundsList);
    }
    public static void upperBoundedMethod(List<? extends Number> list) {
        System.out.println("In upperBoundedMethod");
        for (Number number : list) {
            System.out.println(number);
        }
    }
    public static void lowerBoundedMethod(List<? super Integer> list) {
        System.out.println("In lowerBoundedMethod");
        list.add(2);
        list.add(3);
        list.add(4);
        System.out.println("list : " + list);
    }
    public static void noBoundedMethod(List<Integer> list) {
        System.out.println("In noBoundedMethod");
        list.add(new Integer(8));
        list.add(new Integer(7));
        System.out.println("list : " + list);
    }
}

```