Generics

By

Anand Kulkarni

Anand.pune38@gmail.com

Contents

Module	Topic
Module 1	Why Generics?
Module 2	Generic Types
Module 3	Single type parameter example
Module 4	Multiple type parameter example
Module 5	Bounded type parameters
Module 6	Wildcards

Why Generics?

See the code below:

```
List<String> list = new ArrayList<String>();

list.add("hello");

list.add(new Integer(5)); //compiler error

String s = list.get(0);   // no cast
```

- Stronger type checks at compile time.
- Elimination of casts.
- Enabling programmers to implement generic algorithms.

Generic types

Here are the possible generic types:

- **E** - Element (used extensively by Collections Framework)
- **K** - Key
- **N** - Number
- **T** - Type
- **V** - Value
- **S,U,V** etc. - 2nd, 3rd, 4th types

Single type parameter example

```
public class Box<T> {
   // T stands for "Type"
   private T t;
   public void set(T t) { this.t = t; }
   public T get() { return t; }
}

Box<Integer> box = new Box<Integer>();
box.set(12);
Int x = box.get();
```

Multiple type parameter example

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
                this.key = key;
                this.value = value;
    }
    public K getKey()     { return key; }
    public V getValue() { return value; }
}
```

Multiple type parameter example continue…

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

Bounded Type Parameters

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

Bounded Type Parameters continue…

```
public <U extends Number> void inspect(U u){
      // code
}

Box<Integer> integerBox = new Box<Integer>();

integerBox.set(new Integer(10));

integerBox.inspect(35); // ok

integerBox.inspect("some text"); // error: this is a String!
```

Wildcards

- In generic code, the question mark (?), called the wildcard, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type.
- There are three ways to use wildcards:
  - Unbounded wildcards
  - Upper bounded wildcards
  - Lower bounded wildcards

Unbounded wildcards

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.println(elem + " ");
}
```

The above printList() method can print the list of any type. Hence it is called as unbounded wildcard.

Upper bounded wildcards

```
public void process(List<? extends Number> list) {
        //code
}
```

The above process() method can process list of generic types those extend class Number. For example Number, Integer, Float etc.

```
List<Integer> listOfInt = new ArrayList<Integer>();
List<String> listOfStr = new ArrayList<String>();
process(listOfInt); //ok
process(listOfStr); // Error
```

Lower bounded wildcards

- A lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>

```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

- The addNumbers() method will accept only List<Integer>, List<Number> & List<Object> as an argument.