Chapter 13: Generics Part II

Yepang LIU

liuyp1@sustech.edu.cn

Objectives

- A deeper analysis of generic methods
- A deeper analysis of generic classes

Recall the generic method Example

```java
public static void main(String[] args) {
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
    System.out.print("integerArray contains: ");
    printArray(integerArray);
    System.out.print("doubleArray contains: ");
    printArray(doubleArray);
    System.out.print("characterArray contains: ");
    printArray(characterArray);
}
```

```java
public static <T> void printArray(T[] array) {
    for (T element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

How does compiler work here?

```
public class GenericMethodExample {
  public static void main(String[] args) {
    Integer[] integerArray = ...;
    Double[] doubleArray = ...;
    Character[] characterArray = ...;
    printArray(integerArray);
    printArray(doubleArray);
    printArray(characterArray);
  }
  public static <T> void printArray(T[] array) {
    for (T element : array)
      System.out.printf("%s ", element);
    System.out.println();
  }
}
```

A high-level view

Determine `integerArray`'s type is `Integer[]`

↓

Locate a method named `printArray` with a single parameter of `Integer[]` type. Not such method.

↓

Determine whether there is a generic method named `printArray` with a single parameter of array type and uses a type parameter to represent the array element type. Yes, found the method.

↓

Check whether the operations in the method can be applied to the type of elements stored in the actual array argument. Yes, all objects have a `toString` method (implicit call here). **The code compiles!**

Under the hood: Erasure (消除)

- When the compiler translates generic method `printArray` into Java bytecodes, it **removes the type-parameter section** and **replaces the type parameters with actual types**. This process is known as **erasure**.
- By default, all generic types are replaced with type **`Object`**
- The compiled version of `printArray` is shown below (we show source code instead of bytecodes)

```
public static void printArray(Object[] array) {
    for (Object element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

Benefits of generic methods

In the earlier example, seems that using generic methods is the same as using `Object` array as parameter of `printArray` (like the code below). Why using generics then?

```
public static void main(String[] args) {
    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
    System.out.print("integerArray contains: ");
    printArray(integerArray);
    System.out.print("doubleArray contains: ");
    printArray(doubleArray);
    System.out.print("characterArray contains: ");
    printArray(characterArray);
}
public static void printArray(Object[] array) {
    for (Object element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

Benefits of generic methods

```
public static Object simplyReturn(Object o) {
    return o;
}
public static void main(String[] args) {
    String s = simplyReturn("hello");
}
```

The compiler sees that the method return type is `Object`, assigning a reference of `Object` to a `String` variable is illegal, so a compilation error will occur.

Programmers need to perform explicit type cast: `(String) simplyReturn("hello")`, which may generate `ClassCastExceptions` if the cast fails.

Benefits of generic methods

```
public static <T> T simplyReturn(T o) {
    return o;
}
public static void main(String[] args) {
    String s = simplyReturn("hello");
}
```

With the generic method, the compiler will perform careful type checking and infer the return type is `String` when the actual argument's type is `String` and inserts type cast automatically (such cast will never throw `ClassCastException`, **guaranteed to be safe**).

Therefore, the code can be successfully compiled and is more type safe (类型安全).

The benefits become obvious when the return type is also parameterized.

Bounded Type Parameter

```
public static <T> T simplyReturn(T o) {
    return o;
}
```

- In generic methods like the above, all reference types up to `Object` can be are passed to the type parameter (we say `Object` is an **upper bound**).
- There are times when you want to **restrict the types that are allowed to be passed to a type parameter**, e.g., a method that operates on numbers might only want to accept instances of `Number` or its subclasses
- **Bounded type parameters** are useful in such cases.

Bounded Type Parameter

- To declare a bounded type parameter, simply list the type parameter's name followed by the `extends` keyword and an upper bound
  - Here, `extends` is used in a general sense to mean either "`extends`" as in classes or "`implements`" as in interfaces.

```
public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
    T max = x;
    if (y.compareTo(max) > 0) max = y;
    if (z.compareTo(max) > 0) max = z;
    return max;
}
```

`T` can be any type that implements the `Comparable` interface

Example

```
Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8
Maximum of pear, apple and orange is pear
```

```java
public static void main(String[] args) {
    System.out.printf("Maximum of %d, %d and %d is %d\n", 3, 4, 5,
            maximum(3, 4, 5));
    System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f\n", 6.6,
            8.8, 7.7, maximum(6.6, 8.8, 7.7));
    System.out.printf("Maximum of %s, %s and %s is %s\n", "pear",
            "apple", "orange", maximum("pear", "apple", "orange"));
}
```

`Integer`, `Double` and `String` all implement the `Comparable` interface, so can be passed to the type parameter

Compiler's view

```
// Erasure: replacing the type parameter T with the upper bound Comparable
public static Comparable maximum(Comparable x, Comparable y, Comparable z) {
    Comparable max = x;
    if (y.compareTo(max) > 0) max = y;
    if (z.compareTo(max) > 0) max = z;
    return max;
}
```

When encountering method calls, infer the return type and insert explicit casts (the compiler guarantees that the cast will never throw `ClassCastException`):

```
maximum(3, 4, 5) → (Integer) maximum(3, 4, 5)
maximum(6.6, 8.8, 7.7) → (Double) maximum(6.6, 8.8, 7.7)
maximum("pear", "apple", "orange") → (String) maximum("pear", "apple", "orange")
```

Recall the generic Stack class

```
public class Stack<T> {
    private ArrayList<T> elements; // use an ArrayList to implement the stack
    public Stack() {  this(10);  }
    public Stack(int capacity) {
        int initCapacity = capacity > 0 ? capacity : 10;
        elements = new ArrayList<T>(initCapacity);
    }
    public void push(T value) {
        elements.add(value);
    }
    public T pop() {
        if(elements.isEmpty())
            throw new EmptyStackException("Stack is empty, cannot pop");
        return elements.remove(elements.size() - 1);
    }
}
```

Note: `EmptyStackException` is a self-defined exception type

Use the generic Stack class

```
public static void main(String[] args) {

    Stack<Double> doubleStack = new Stack<Double>(5);
    Stack<Integer> integerStack = new Stack<Integer>();

    doubleStack.push(1.2);
    Double value = doubleStack.pop();
    System.out.println(value);

    integerStack.push(1);
    integerStack.push(2);

    while(true) {
        Integer i = integerStack.pop();
        System.out.println(i);
    }
}
```

```
1.2

2

1

Exception...
```

Compiler’s view

Erasure (similar to generic methods): Replacing all type parameters with `Object` or their bounds if the type parameters are bounded

The produced bytecodes contain only ordinary classes, interfaces, and methods, i.e., **no generics at the bytecode level**

```
public class Stack {
    private ArrayList<Object> elements;
    public Stack() {  this(10);  }
    public Stack(int capacity) {...
        elements = new ArrayList<Object>(initCapacity);
    }
    public void push(Object value) { ... }
    public Object pop() { ... }
}
```

Compiler's view

The compiler will insert type casts if necessary to preserve type safety

```
Stack<Double> doubleStack = new Stack<Double>(5);

doubleStack.push(1.2);

Double value = doubleStack.pop();
```

```
Stack doubleStack = new Stack(5);

doubleStack.push(1.2);

Double value = (Double) doubleStack.pop();
```

Let’s test our understanding

- **Q1:** Will the compiler successfully compile the following code?

```
String s = "hello world";
Object obj = s;
```

- It is **safe** to assign `s` (of type `String`) to `obj` (of type `Object`) because an instance of a subclass (subtype) is also an instance of a superclass (supertype).
- **“Safe”** means any operations that can be done via the reference `obj` are also allowed to be done via the reference `s`

A more difficult question

- **Q2:** Will the compiler successfully compile the following code?

```
ArrayList<String> ls = new ArrayList<String>();
List<String> ls2 = ls;
```

- It is safe to assign `ls` to `ls2` because an `ArrayList` of `String` is also a `List` of `String`.

- Any operations that can be done via the reference `ls2` can also be done via the reference `ls`

The “hardest” question about generics

- **Q3:** Will the compiler successfully compile the following code?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- This boils down to the question: *is a `List` of `String` a `List` of `Object`?* (Most people will instinctively answer “yes”…)

- **What if we ask the safety question:** *is it true that any operations that can be done via the reference `lo` can also be done via `ls`?*

Let’s do some analysis

- As a reference of type `List<Object>`, `lo` can be used for the following operation:

```
lo.add(new Double());
```

- However, we cannot perform the same operation via the reference `ls` because it is of type `List<String>`

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // type mismatch
```


Further analysis from compiler's view

- If the compiler allows assigning `ls` to `lo`, then the code

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
lo.add(new Double(0.0));
String s = ls.get(0);
```

- will be compiled into the following form:

```
List ls = new ArrayList();
List lo = ls;
lo.add(new Double(0.0));
String s = (String) ls.get(0);
```

ClassCastException

Generic classes are designed to provide type safety, such exceptions are awkward

General Rule

- If `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is not the case that `G<Foo>` is a subtype of `G<Bar>`.

- This is probably the hardest thing one needs to learn about generics, because it goes against our intuitions…

Suppose we want to write a method to handle all kinds of collections

```
public static void printCollection(Collection<Object> c) {
    for(Object e : c) System.out.println(c);
}

Collection<String> strs = new ArrayList<String>();

printCollection(strs); // is this call ok?
```

Apply the rule, and we will know that `Collection<String>` is not a subtype of `Collection<Object>`, so the code cannot compile.

What is the supertype of all kinds of collections then?

Wildcards

- `Collection<?>` (pronounced **"collection of unknown"**), that is, a collection whose element type matches anything

```
// We can call this method with any kind of collection
public static void printCollection(Collection<?> c) {
    for(Object e : c) System.out.println(c);
}
```

- Note that we can still read elements from `c` and give them the type "`Object`", since whatever the actual type of the collection element is, it is a subtype of `Object`.

Wildcards

- Java allows the following code:

```
Collection<?> c = new ArrayList<String>();
```

- However, it does not allow you to add arbitrary objects to c, the "collection of unknown"

```
c.add(new String());
c.add(new Object());
```

- In the declaration of Collection<E>, the add method takes arguments of type E, the type of the collection's element. Here, the actual type parameter is ? (unknown type), so anything that we pass to add must be the type of the unknown type. That means we cannot pass anything in except null.