Theory: Reification

26 minutes 0 / 5 problems solved

Generics are known for their type safety, which is obviously a good thing. However, it has a flip side. As we've already discussed, type erasure makes some processes quite complicated. And now it's a time to discuss another generics-related notion — **reification**.

§1. What is reification

Though information about some types that is available at compile-time is no longer present at run-time, other types are not affected at all by this process and are fully presented in byte code. That makes two kinds of types: types that save information about itself during type erasure are called **reifiable**, while types whose information is erased are called **non-reifiable**. The term **reification** means that the type parameters are available at runtime as well as at compile-time.

§2. The two groups

Let's recall what types are replaced during type erasing and can be called non-reifiable. They are parameterized types like `<T>` which are replaced by `Object` and bounded generics or wildcards, for example `<T extends Number>` or `<? extends Number>` are replaced by `Number`.

Reifiable types group is more extensive. Obviously it includes primitive types like `int`, `double` and non-parameterized type like `String`, `Number` and others. There are more complicated reifiable types, which are technically equivalent to `Object`. The first is a raw type. It is a type that can be parameterized but is not. For instance, if class `Box<T>` is declared as `Box box = new Box()` then it's a raw type. The second is an unbounded wildcard type, for example `Box<?>`. It includes arrays whose component type is reifiable as well.

§3. Non-reifiable limitations

Non-reifiable types are pretty good at compile-time but the fact that they are not present at runtime leads to some problems. Let's consider them.

1) It is prohibited to create a non-reifiable typed instance.

It is extremely simple to create an instance of `Box<T>` with a specific type `Box<String> box = new Box<>()`. However generic constructor call causes a compilation error:

```
class Box<T> {
    private T instance;

    public void init() {
        this.instance = new T(); // compile-
time error: Type parameter T cannot be instantiated directly
    }
}
```

This limitation is reasonable since we have no way to guarantee that `T` will implement any particular constructor.

2) Another limitation for a non-reifiable type includes using `instanceof` operator.

```
class Box<T> {
    ...
    public boolean isIntegerSuperType() {
        return Integer.valueOf(0) instanceof T; // compile-
time error: Illegal generic type for instanceof
    }
}
```

Current topic:

Reification

Topic depends on:

× Type Erasure

Table of contents:

This operation is prohibited since the information on the exact type is unavailable at runtime for non-reifiable types. This makes it impossible to create an instance and check if the operation is safe to be run.

3) The type must be reifiable to extend `java.lang.Throwable`.

Suppose that there is a generic class which extends `Throwable`.

```
class MyException<T> extends Exception {}
```

Actually in that case compiler raises the message `Generic class may not extend java.lang.Throwable`. To illustrate the problem let's imagine that we hacked the compiler and this case is out of validation. Now look at the code below

```
try {
    ...
} catch (MyException<String> e) {
    System.out.println("String");
} catch (MyException<Long> e) {
    System.out.println("Long");
}
```

Both caught types should be translated into single parameterless `MyException` type after the type erasure. As a result, we have a dilemma on how to handle `MyException`. For that reason, any generic extensions of `Throwable` are prohibited.

4) The creation of an array instance requires a reifiable type. This limitation also relates to Varargs, because they are translated into an array.

Let's look at the signature of `<T> T[] toArray(T[] a)` method in `Collection` class. The main task of an array passed as an argument is to provide type information at the runtime.

Remember that due to type erasure the code:

```
Collection<Integer> col = new ArrayList<Integer>();
Integer[] array = col.toArray(new Integer[0]);
```

is equivalent to:

```
Collection col = new ArrayList();

// col has no type parameter information at runtime.

// Which array type should we create inside toArray() method without a parameter?
Integer[] array = (Integer[]) col.toArray();
```

So, it's perfectly fine to call this method in a way:

```
Collection<Integer> col = ... initializing of this Collection

// toArray will create array of appropriate size
Integer[] array = col.toArray(new Integer[0]);
```

In the example, we used `Integer` , in particular `Integer[] array = col.toArray(new Integer[0])`, to avoid erasure of all information about the type and to make the type of an array available at runtime.

5) Casting to non-reifiable type usually issues a warning to notify the programmer that it may not be safe and potentially leads to exceptions.

§4. Conclusion

Type erasure can cause certain mismatches in the code. Some types lose information about their parameterization. Such types are called non-reifiable, others are reifiable. It is very important to use the right variable type in your code because such a mistake can cause problems later, that were hidden at first. Non-reifiable types have limitations and some operations are

prohibited: creating instances and arrays, usage of `instanceof` operator, creation parameterized successors of `Throwable`. In addition, there is also the chance of losing type safety on casting to non-reifiable types.

Report a typo

19 users liked this theory. **7** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (1) Show discussion