Theory: Generic programming

17 minutes 0 / 5 problems solved

Skip this topic Start practicing

4636 users solved this topic. Latest completion was less than a minute ago.

§1. Generic programming

There are situations when methods and classes do not depend on the data types on which they operate. For example, the algorithm to find an element in an array can process arrays of strings, integers or custom classes. It does not matter what the array stores: the algorithm is always the same. Yet we cannot write it as a single method, because it requires different arguments (`int[]`, `String[]`, etc).

Since version 5, Java has supported generic programming that introduces abstraction over types. Parameterized types allow us to declare a method or a class that handles different types in the same general way. A concrete type is determined only when a developer creates an object of the class or invokes the method. This approach enables us to write more abstract code and develop reusable software libraries. Let us consider it step by step using examples written in Java.

§2. Type parameters

A generic type is a generic class (or an interface) that is parameterized over types. To declare a generic class, we should declare a class with the type parameter section delimited by angle brackets following the class name.

In the following example, the class `GenericType` has a single type parameter named `T`. We assume that the type `T` is "some type" and write the class body regardless of the concrete type.

Current topic:

Generic programming

Topic depends on:

- ✓ Constructor (Stage 6)
- ✓ Instance methods (Stage 6)
- ✓ Boxing and unboxing

Topic is required for:

- Generics and Object
- Generic methods
- Lambda expressions
- What are collections
- Optional

Table of contents:

```
class GenericType<T> {

    /**
     * A field of "some type"
     */
    private T t;

    /**
     * Takes a value of "some type" and set it to the field
     */
    public GenericType(T t) {
        this.t = t;
    }


    /**
     * Returns a value of "some type"
     */
    public T get() {
        return t;
    }


    /**
     * Takes a value of "some type", assigns it to a field and then returns it
     */
    public T set(T t) {
        this.t = t;
        return this.t;
    }
}
```

After being declared type parameter can be used inside the class body as an ordinary type, for example:

- a type of a *field*
- *constructor* argument type
- *instance method* argument type and return type

The behavior of both instance methods does not depend on the concrete type of `T`; it can take/return a string or a number in the same way.

A class can have any number of type parameters. For example, the following class has three.

```
class Three<T, U, V> {
    T t;
    U u;
    V v;
}
```

But, most of the generic classes have one or two type parameters.

§3. The naming convention for type parameters

There is a naming convention that restricts type parameter name choices to single uppercase letters. Without this, it would be difficult to tell the difference between a type variable and an ordinary class name.

The most commonly used type parameter names are:

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

§4. Creating objects of generic classes

To create an object of a generic class (standard or custom), we should specify the type argument following the type name.

```
GenericType<Integer> obj1 = new GenericType<Integer>(10);

GenericType<String> obj2 = new GenericType<String>("abc");
```

It is important, that a type argument must be a reference type. It is impossible to use a primitive type like `int` or `double` as a type argument.

Since Java 7, it has been possible to replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments, as long as the compiler can ***infer*** the type arguments from the context.

```
GenericType<Integer> obj1 = new GenericType<>(10);

GenericType<String> obj2 = new GenericType<>("abc");
```

We will do this in all further examples.

The pair of angle brackets `<>` is informally called the *diamond operator.*

Sometimes, declaring a variable with a generic type can be too long and poorly readable. Fortunately, you can write `var` instead of a specific type to force automatic type inference based on the type of assigned value.

```
var obj3 = new GenericType<>("abc");
```

It looks very modern, doesn't it?

After we have created an object with a specified type argument, we can invoke methods of the class that take or return the type parameter:

```
Integer number = obj1.get(); // 10
String string = obj2.get();  // "abc"

System.out.println(obj1.set(20));    // prints the number 20
System.out.println(obj2.set("def")); // prints the string "def"
```

If a class has multiple type parameters, we should specify all of them when creating instances in the following format:

```
GenericType<Type1, Type2, ..., TypeN> obj = new GenericType<>(...);
```

§5. Custom generic array

As a more complicated example, let us consider the following class, representing a generic immutable array. It has one field to store items of the type `T`, a constructor to set items, a method to get an item by the index, and another method to get the length of the internal array. The class is immutable because it does not provide methods to modify the items array.

```
public class ImmutableArray<T> {

    private T[] items;

    public ImmutableArray(T[] items) {
        this.items = items;
    }

    public T get(int index) {

        return items[index];

    }


    public int length() {

        return items.length;

    }

}
```

The class above shows that a generic class can have methods (like length) that do not use the parameter type at all.

Let's create an instance of `ImmutableArray` to store three strings and then output the items to the standard output.

```
var stringArray = new ImmutableArray<>(new String[] {"item1", "item2", "item3"});

for (int i = 0; i < stringArray.length(); i++) {
    System.out.print(stringArray.get(i) + " ");
}
```

It is possible to parameterize `ImmutableArray` with any reference type, including arrays, standard classes, or your own classes.

```
var doubleArray = new ImmutableArray<>(new Double[] {1.03, 2.04});

MyClass obj1 = ..., obj2 = ...; // suppose, you have two objects of your custom class

var array = new ImmutableArray<>(new MyClass[] {obj1, obj2});
```

We used `var` in the examples above, since otherwise they take too much space. You can explicitly specify the type if you want, e.g. `ImmutableArray<String> stringArray = ...;` and so on.

§6. Conclusion

A class can declare one or more type parameters and use them inside the class body as types for fields, method arguments, return values, and local variables. In this case, the class does not know the concrete type on which it operates. The concrete type should be specified when creating instances of the class. This approach allows you to write classes and methods that can process many different types in the same way without writing code to process each concrete type.

It is important that only a reference type (an array, a standard class, a custom class) can be used as a concrete type for generics. Instead of primitive types, you can use wrapper classes such as `Integer`, `Double`, `Boolean`, and so on.

Report a typo

504 users liked this theory. 14 didn't like it. What about you?

Start practicing

Comments (17) Hints (0) Useful links (0) Show discussion