Custom annotations and types of annotations

Java annotations provide meta information to source code.

Defining a new annotation

Custom annotations have to be defined in @interface files.
Let's create a special annotation that represents class or method description with its author and version number. First, we define the @interface:

```
public @interface Description {}
```

Retention policies

The first is annotation **Retention**.
It specifies at which level your annotation will be applied. It is specified by @Retention annotation on your custom annotation class, like in the following snippet:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Description {}
```

There are 3 available retention types in Java:

SOURCE

These annotations are used by the compiler during the compilation process.
For example, @Override annotation has this type of retention:

```
@Retention(RetentionPolicy.SOURCE)
public @interface Override { }
```

CLASS

It is the default retention.
These annotations are recorded in the class file on compilation, but then they are not available during run time.

It is basically used to perform byte code modifications.
It can be used in code obfuscation or code generation libraries (e.g. Lombok).

For example, @NotNull annotation has this type of retention:

```
@Retention(RetentionPolicy.CLASS)
public @interface NotNull { }
```

RUNTIME

Runtime annotations are also recorded in the class file, and then they can be read at run time.

The @Deprecated annotation and @Retention itself has a runtime retention policy.

This one is generally used for custom annotations because it is the only policy that can be processed manually during program execution.

Our custom @Description annotation will have it as well (check the construction):

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Description {}
```

Target types

@Target tells where the annotation can be placed: methods, packages, annotations themselves, etc.
If you don't set up its value, the annotation will be applicable to all elements.

For our annotation we are going to use **METHOD** value:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Description {}
```

The ElementType.TYPE_PARAMETER constant is one of the values of the java.lang.annotation.ElementType enum. It represents a declaration of a type parameter. This ElementType constant is used in Java annotations to specify where an annotation type is legal.
When you annotate an element with an annotation that specifies ElementType.TYPE_PARAMETER, it means that the annotation is applicable to the declaration of a type parameter. Type parameters are declared in generic type declarations, such as classes, interfaces, methods, and constructors, using angle brackets (<>).

code

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
@Target(ElementType.TYPE_PARAMETER)
@interface MyTypeAnnotation
{ // Annotation members here }
```

```
class MyClass<@MyTypeAnnotation T>
{ // Class implementation }
```

In this example:

- @MyTypeAnnotation is a custom annotation that is targeted at type parameters.
- It's applied to the type parameter T in the declaration of the MyClass generic class.
- Annotations targeted at type parameters can provide additional metadata or constraints related to the type parameter declaration.

Annotations on type parameters are typically used to provide additional information or constraints on generic types, such as specifying bounds (extends and super clauses) or other restrictions on the types that can be used as type arguments.

The following table shows all available target types.

ElementType.ANNOTATION_TYPE	Annotations
ElementType.CONSTRUCTOR	Constructors
ElementType.FIELD	Fields
ElementType.LOCAL_VARIABLE	Local variables
ElementType.METHOD	Methods
ElementType.MODULE	Modules
ElementType.PACKAGE	Packages
ElementType.PARAMETER	Parameters
ElementType.RECORD_COMPONENT	Record components
ElementType.TYPE	Class, interface, enum, record or annotation
ElementType.TYPE_PARAMETER	Type parameter declaration
ElementType.TYPE_USE	Use of a type

Annotations parameters

Let's add some parameters to our annotation that should specify different parts of our description. These parameters have restrictions on the type:

- primitives;
- String;
- Class;
- Enum;
- annotation;
- an array of these types.

Beware that the default value cannot be null. Now we can define our @Description annotation parameters:

- description of the method itself;
- author name;
- version number.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Description {
    String author();
    String description();
    int version() default 0;
}
```

ELEMENT.TYPE_USE:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE_USE)
@interface MyTypeAnnotation {
    // Annotation members here
}

public class Example {
    public static void main(String[] args) {
        @MyTypeAnnotation String str = "Hello"; // Annotation on variable
declaration

        // Annotation on type cast
        Object obj = "Test";
        String casted = (@MyTypeAnnotation String) obj;

        // Annotation on type parameter
        Example.<@MyTypeAnnotation String>genericMethod("Generic");
    }

    public static <T> void genericMethod(@MyTypeAnnotation T param) { //
Annotation on type parameter
        // Method implementation
    }
}
```

The ElementType.TYPE_USE constant is one of the values of the java.lang.annotation.ElementType enum. It represents any use of a type. This ElementType constant is used in Java annotations to specify where an annotation type is legal.

When you annotate an element with an annotation that specifies ElementType.TYPE_USE, it means that the annotation is applicable to any use of a type. This includes variable declarations, type casts, type parameter declarations, and more.
Here's an example illustrating the use of ElementType.TYPE_USE:

In this example:

- @MyTypeAnnotation is a custom annotation that is targeted at type uses.
- It's applied to various type uses, including variable declarations (@MyTypeAnnotation String str), type casts ((@MyTypeAnnotation String) obj), and type parameter declarations (<@MyTypeAnnotation String>genericMethod).
- Annotations targeted at type uses can provide additional metadata, constraints, or behavior associated with the usage of a type, offering enhanced expressiveness and control over the types used in the program.

Additional meta-annotations

The annotation can be marked as @Repeatable and then it can be used multiple times at the same place. You should provide a container class name: the class with an array collecting repeatable annotations:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Descriptions {
   Description[] value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(Descriptions.class)
public @interface Description {...}
```

Now the annotation can be applied more than one time:

```
@Description(author = "John Doe", description = "first description")
@Description(author = "Richard Roe", description = "second description")
public void testMethod() {
```

@Documented takes no parameters and includes an annotation in the Javadoc documentation.
If the annotation is marked as @Inherited, then it will be applied to all subclasses of the annotated class.

Processing annotations

We will write a special class to process the annotation and construct the message to print method in it. As we made it @Repeatable, we need to retrieve container annotation @Descriptions first and then iterate over its @Description annotations:

```
public class DescriptionProcessor {

    public void printDescription(Object o) {
        // Get processing object class
        Class<?> processingClass = o.getClass();
        for (Method m : processingClass.getDeclaredMethods()) {
            // Check if method has container @Descriptions annotation
            if (m.isAnnotationPresent(Descriptions.class)) {
                // Get container annotation
                Descriptions descriptions = m.getAnnotation(Descriptions.class);
                StringBuilder sb = new StringBuilder();
                // Iterate over exact @Description annotations
                for (Description d : descriptions.value()) {
                    sb.append("Description: ")
                            .append(d.description())
                            .append(" Author : " )
                            .append(d.author())
                            .append(" Version : ")
                            .append(d.version())
                            .append("\n");
                }
                // Print result
                System.out.println(m.getName() + " Descriptions: ");
                System.out.println(sb.toString());
            }
        }
    }
}
```

Now it's testing time. Let's create a test method and add an annotated method there:

```
public class TestClass {
    @Description(
            author = "John Doe",
            description = "Testing method"
    )
    @Description(
            author = "Richard Roe",
            description = "Repeatable description",
            version = 1
    )
    public void testMethod() {
        System.out.println("The method to test the @Description annotation");
    }
}
```
And now we can test it in Main method:
```
public class Main {
    public static void main(String[] args) {
        // Creating processor object
        DescriptionProcessor processor = new DescriptionProcessor();
        // Creating test object with annotated method
        TestClass test = new TestClass();
        // Call processing method
        processor.printDescription(test);
    }
}
```
And look at console output:
```
testMethod Descriptions:
Description: Testing method Author : John Doe Version : 0
Description: Repeatable description Author : Richard Roe Version : 1
```

———————More examples———————
The @SuppressWarnings annotation in Java can accept an array of values for the types of warnings to suppress. This feature allows developers to suppress multiple types of warnings using a single annotation.
Here's the definition of the @SuppressWarnings annotation:
java

code

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings
{ String[] value(); }
```

As you can see, the value() method of the @SuppressWarnings annotation returns an array of String, which represents the types of warnings to be suppressed.
Here's an example of how you can use @SuppressWarnings with multiple values:
java

code

```
@SuppressWarnings({"unchecked", "rawtypes"})
public void myMethod() { // Code here }
```

In this example, the @SuppressWarnings annotation suppresses two types of

warnings: "unchecked" and "rawtypes". These warnings are related to unchecked operations and the use of raw types, respectively.
By providing an array of values to @SuppressWarnings, you can suppress multiple types of warnings at once, making the code more concise and readable.