di-fall-2021

assignment5 of ooad course in 2021 fall semester

Classes

1.BeanFactory

Being used to inject instance according to the property files.

```
public interface BeanFactory {
    void loadInjectProperties(File file);

    void loadValueProperties(File file);

    <T> T createInstance(Class<T> clazz);
}
```

- `void loadInjectProperties(File file);`
  Load all inject data from `file` , which is a standard Java Properes file
- `void loadValueProperties(File file);`
  Load all inject data from `file` , which is a standard Java Properes file
- `<T> T createInstance(Class<T> clazz);`
  Create an instance which type is T.

The actual implementation class of `clazz` may be defined in `inject properties`.
If it is not defined in the properties, `clazz` itself will be the implementation class.

We ensure that in test cases all `abstract class` or `interface` that are passed as `clazz`` are declared in the inject property file.

2.Inject

Definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface Inject {
}
```

How to use it?

- **On fields**: `ElementType.FIELD`
  If `@Inject` is marked on field, only the **user defined classes** that could be annotated by `@Inject` annotation, which means, in `<T> T createInstance(Class<T> clazz);` method,

we not only needs to create an instance for current class, but also create instance for all fields that identified by `@Inject`.

- **On methods**: `ElementType.METHOD`
  If `@Inject` is marked on method, we can assume that the method is setter method. It takes only one parameter and set a field. We need to call all methods identified by `@Inject` to inject values.

```
public class Example1 {
    @Inject
    private A a;

    private B b;
    @Inject
    public void setB(B b) {
        this.b = b;
    }
}
```

- **On Constructors**: `ElementType.CONSTRUCTOR`
  If `@Inject` is marked on constructor, **only one constructor** in each class could be annotated by `@Inject` annotation.
  In the `<T> T createInstance(Class<T> clazz);` method, we only use the constructor that identified by `@Inject` to create an instance.
  Other than that, **we can ensure that classes in test cases have only one constructor identified by `@Inject` , or the test class only has the default constructor** , which means in `createInstance` , the constructor is either annotated by `@Inject` or the constructor is the default constructor.

```
public class ImplClz implements Clz {
    private A a;
    private B b;

    @Inject
    public ImplClz(A a, B b) {
        this.a = a;
        this.b = b;
    }
}
```

3.Value

Only primitive types or String will be annotated by `@Value`

```
byte, short, int, long, float, double, boolean, char, String
```

Definition:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
public @interface Value {
    String value();

    String delimiter() default ",";

    int min() default Integer.MIN_VALUE;

    int max() default Integer.MAX_VALUE;
}
```

How to use it?

- **On fields**: `ElementType.FIELD`
  If `@Value` is marked on field, in `<T> T createInstance(Class<T> clazz);` method, we not only needs to create an instance for current class, but also need to give all fields that identified by `@Value` a specified value

```
public class Example2 {
    @Value(value = "int-value")
    private int number;
    @Value(value = "name-value")
    private String name;
    @Inject
    private Course course; //combine @Value and @Inject
}
```

- **On parameter**: `ElementType.PARAMETER`
  If `@Value` is marked on parameters in constructor or method, when call the constructor or method, a specific value should be given to corresponding parameters.

We ensure that, in our test cases, all **parameters** in the constructor or method that annotated by `@Inject` are either **injected** or **annotated by** `@Value`

```
public class Example3 {
    private int number;
    private String name;
    private Course course;

    @Inject
    public Example3(Course course,
                    @Value("name-value") String name,
                    @Value("int-value") int number) {
        this.course = course;
        this.name = name;
        this.number = number;
    }
```

```
}
```

How to inject value?

We can ensure that in the mapping relations of the parameter `value` in `@Value` are existed in the property file `value properties`. More specifically, the values "int-value" and "name-value" are all appeared in `value properties` during our judging process. So that the inject value of the fields annotated by `@Value` are according to the mapping value in `value properties`.

`value properties` may contains multiple values for a key. For example, it may contain `ints=1-2-3-4-5` or `name=sustech,southern university of science and technology`. You should split the values by `delimiter()`,then find and inject the **first value** that satisfied the following condition or inject default value when no satisfied value found.

- For number, the condition is that the value of the number in range `[min(), max()]`, default value is `0`
- For `String`, the condition is that the length of the string in range `[min(),max()]`, default value is `"default value"`

We ensure that only `byte, short, int, long, String` may have multiple values.

For example

```
@Value(value = "prime", min = 10, max = 20, delimiter = "-")
int prime;
@Value(value = "name", max = 15)
int name;
```

```
prime=2-3-5-7-11-13-17-19-23-29
name=sustech,southern university of science and technology
```

In this case, you should assign `11` to `prime` and `"sustech"` to `name`

Properties

1. inject properties

```
testclass.E=testclass.EImpl
testclass.F=testclass.FEnhanced
testclass.J=testclass.JImpl
```

In our test cases, we ensure that the left side will only be `Abstract Class` , `Class` or `Interface` , while the right side is the implement class of the left side.

2. value properes

```
d.val=10
j.integers=1-4-8-34-14-6
j.strings=all values are non-empty
l.val=true
```

The left side are the key name of parameter `value` in `@Value` , while the right side are the specific value of the key that needs to be injected into parameter.

We ensure that in the mapping relations of the parameter `value` in `@Value` all exist in the property file `value properties`

Requirement

You should complete the class named `dependency_injection.BeanFactoryImpl` which implements the interface `BeanFactory` , and upload the file `BeanFactoryImpl.java` to Sakai.

You will GET A ZERO if one of the following happens:

- File name, class name, package name is not identical to the requirement
- Compilation fail
- Plagiarism