

DI-OOAD-2022 Fall

Contributor List:

Yitong WANG: Design, Document, Solution

Hengcheng YUAN: JUnit, Judger, Explanation

NewbieOrange and Gogo: 2020-2021 DI Assignment Design

:octocat: For this assignment, you can view all materials from this repo [DI-OOAD-22Fall-public](#).
And if you have any question about this assignment, you are welcomed to submit an issue [here](#)!

Background story

YeeTone is working hard on OOAD project of `Code Management Platform` topic. 🤖

In his group, he is responsible for the backend using Java language, but he is not quite clear about the principle of DI (dependency injection) on Spring framework. 😓 If you could write some code passing all testcases about DI, he will be easily understand your idea and do a satisfactory work in the final presentation! 💪

Before starting this DI assignment:

Here are some suggestions from the document author:

- ★ Please be patient and read **EVERY** sentence in the document carefully
- ★ Please listen to Mrs.ZHU or Mr.YUAN's explanation in class clearly
- ★ This DI assignment will be very interesting, and YeeTone hopes you could enjoy it
- 😊 If you feel difficult about the assignment and need help, please watch this [video](#)

I. Classes

1. BeanFactory

This class is an interface, which is used to inject instance according to the property files.

Here is the definition:

```
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 Properties](#) file.

- `void loadValueProperties(File file);`

Load all value data from file , which is a standard [Java Properties](#) file.

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

Create an instance which type is T.

👉 Notice:

1. The actual implementation on class of `clazz` may be defined in `inject.properties`. If it is not defined in the properties, `clazz` itself will be the implementation on class.
2. We ensure that in the JUnit test cases, **ALL** abstract class or interface that are passed as `clazz` are declared in the inject property file.

2. Inject

This class is an annotation. Here is the definition:

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

Please read the following requirement from 2.1 to 2.2.

2.1 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 the current class, but also create instance for all fields that identified by `@Inject`.

For example,

```
public class AB1 {
    @Inject
    private A aDep;

    @Inject
    private B bDep;
}
```

👉 When creating the object of class `AB1`, you also need to create A,B instances for the fields of it(i.e. aDep and bDep) as well.

2.2 On Constructor: `ElementType.CONSTRUCTOR`

1. If `@Inject` is marked on constructor, we ensure that **ONLY ONE** constructor in each class could be annotated by `@Inject` annotation, and **ONLY** use the constructor that identified by `@Inject` to create an instance.
2. Other than that, we 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 `<T> T createInstance(Class<T> clazz)` method, the constructor is either annotated by `@Inject` or the constructor is the default constructor.

For example,

```
public class AB2 {  
    private A a;  
    private B b;  
  
    @Inject  
    public AB2(A a, B b){  
        this.a = a;  
        this.b = b;  
    }  
}
```

👉 When creating the object of class `AB2`, you need to invoke the **ONLY** constructor with parameters A and B. The constructor is annotated with `@Inject`, so you also need to create instances for the constructor parameters instead of `null`.

3. Value

This class is an annotation as well. Here is the definition:

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

Only the following types will be annotated by `@Value`:

```
boolean, int, String  
boolean[], int[], String[]  
List<?>, Set<?>, Map<?, ?>
```

! [IMPORTANT]Explanation:

i. The `values()` of annotation `@Value` may contain multiple values for the annotated objects. For the simple types(i.e. `boolean`, `int` and `String`), You should **ONLY** select the first one fitting the type requirement as the value; In particular, for the `boolean` type, if a String can be equal to `"true"` or `"false"` regardless of case, it meets the requirement.

Example 1 for i:

```
public class Example1 {
    @Value(value = "homo+1145141919810+114514", delimiter = "\\+")
    private int homo; // homo will be 114514
    //homo is not a decimal number and 1145141919810 exceeds INTEGER.MAX_VALUE
    // + is a special character in regex expression, so we use \\ to escape it.

    @Value(value = "deadbeef deadc0de TrUe false", delimiter = " ")
    private boolean magic; // magic will be true
    //deadbeef and deadc0de are not boolean Strings, and TrUe is equal to
    boolean value true regardless of cases.

    @Value(value = "never gonna give you up")
    private String lyric; // lyric will be "never gonna give you up"
    //default delimiter is "," so lyric is "never gonna give you up" directly
}
```

ii. For the array type, you should **ONLY** select those values fit the type requirements into the array;

Example 2 for ii:

```
public class Example2 {
    @Value(value = "[10086,10010,10000,baadfood]")
    private int[] mobiles; // mobiles will be an array of length 3, whose values
    are 10086, 10010 and 10000.
    // baadfood is not a decimal number, so we ignore it.

    @Value(value = "[FaLSe-0-1-Yes-No-tRuE]", delimiter = "-")
    private boolean[] yesOrNo; // yesOrNo will be an array of length 2, whose
    values are false and true.
    // 0, 1, Yes, No are not boolean Strings, so they are omitted.
    // FaLSe and tRuE are equal to boolean values false and true regardless of
    cases.

    @Value(value = "[never gonna give you up]", delimiter = " ")
    private String[] swindle; // swindle will be an array of length 5
    // Its values are "never", "gonna", "give", "you", "up"
}
```

iii. For the generic type, you should **ONLY** select those values fit the type requirements into the container and consider the characteristic of the container type(i.e. Set should **NOT** contain duplicate values, and Map should **NOT** contain duplicate keys).

Example 3 for iii:

```
public class Example3 {
    @Value(value = "[2345,360,kingsoft,]")
    private List<String> rogue3Software; // rogue3Software will be a list of
    size 3
    // Its values are "2345", "360", "kingsoft".

    @Value(value = "{hatsune-miku-39-39-39c5bb}", delimiter = "-")
    private Set<Integer> mikuSet; // mikuSet will be a set of size 1, which
    only contains 39.
    // hatsune, miku, 39c5bb are not decimal numbers, so they are not allowed
    to be injected into the set.

    @Value(value = "{never-gonna:give-you-up,never-gonna:let-you-down}",
    delimiter = ",")
    private Map<String, String> swindle; //swindle will be a map with 1 entry
    // The entry's key is "never-gonna" and value is "let-you-down".
    // The value "give-you-up" is replaced by "let-you-down"

    @Value(value = "{tRUe:955,fALsE:icu,Yes:955,No:996}", delimiter = ",")
    private Map<Boolean, Integer> work; //work will be a map with 1 entry
    // The entry's key is true and value is 955.
    // icu is not an integer number, so its entry is ignored
    // Yes and No are not boolean String, so their entries are ignored as well
}
```

iv. For the array types(i.e. `boolean[]`, `int[]` and `String[]`) and the generic container types(i.e. `List<?>`, `Set<?>` and `Map<?, ?>`), the `values()` will be the String with brackets after being replaced; And for the entries of `Map<?, ?>`, the key and values are connected via `:` so the `delimiter()` of Map type will **NEVER** be `":"`, and the key or value of entries will **NEVER** contain `":"` as well. You need to use `delimiter()` to split the entry elements.

To simplify the problem, we ensure that:

1. We will not contain the `delimiter()` String for the entry-key element in the testcases for `Map<?, ?>` type injection.
2. You do not need to consider some special expression character in regex expression, such as `'+'`, `'*'` and `'?'`. If it appears, we will use `\\` to escape (转义), so you do not need to worry about the usage of `String::split` method.

Example 4 for iv:

```
public class Example4 {
    // @Value(value = "{this-will:never-appear-in-the-testcases:of-JUnit}",
    delimiter = "-")
    // You do not need to consider this kind of testcase, since delimiter()
    appears in the key-entry
    // and we do not know how to split it.
    // private Map<String, String> neverAppear;
}
```

v. Notice that `?` (Unbounded Generic, 无界通配符) in the type list can **ONLY** be one of the following types:

```
Boolean, Integer, String
```

This could be easily understood, so we omitted the example 5.

vi. If no value fit the requirement, then there are some default values for all types except `String`:

```
boolean: false
int: 0
array: an array of length 0 instead of null
generic container: a container of size 0 instead of null
```

Example 6 for vi:

```
public class Example6 {
    @Value(value = "R U OK", delimiter = " ")
    private boolean ruok; // ruok will be false

    @Value(value = "[]")
    private String[] emptyStrs; // emptyStrs will be an array of length 0

    @Value(value = "[q,w,e,r]")
    private List<Integer> qwer; // qwer will be an empty list
}
```

3.1 On Fields: ElementType.FIELD

👉 If `@Value` is marked on field, in `<T> T createInstance(Class<T> clazz)` method, we not only needs create an instance for current class, but also given those fields an specified value that identified by `@Value`. If the `value()` String appears in the `value.properties`, then first replace it as defined in the properties file; Otherwise, use itself as `value()`.

For example, consider the following code:

```
public class FieldValueSample {
    @Value(value = "china-railway")
```

```

private int chinaRailway;

@Value(value = "bool-array", delimiter = "-")
private boolean[] boolArray;

@Value(value = "work-load")
private Set<Integer> workLoad;
@Value(value = "bool-list", delimiter = "-")
private List<Boolean> boolList;

@Inject
private ObjectWithFields owf; //combine @Value and @Inject
}

```

And display a part of `value.properties` file:

```

china-railway=12306
bool-array=[True-OK-false-true-Yes-No]
work-load={9,9,6,0,0,7}
bool-list=[false-true-unknown]


```

After injected, the fields are:

```

int chinaRailway = 12306
boolean[] boolArray = [true, false, true]
Set<Integer> workLoad = {9,6,0,7}
List<Boolean> boolList= [false, true]

```

 Hint: You may need to think about how to get the generic type information from the object of `java.lang.reflect.Field`.

3.2 On Parameters: ElementType.PARAMETER

👉 If `@Value` is marked on parameters in constructor, when execute the constructor, an specific value should be given to corresponding parameters. We ensure that, in our test cases, all parameters in the constructor that annotated by `@Inject` are either injected or annotated by `@Value`. If the `value()` String appears in the `value.properties`, then first replace it as defined in the properties file; Otherwise, use itself as `value()`.

For example,

```

public class ConstructorValueSample {
    private int credits;
    private String name;
    private Course course;

    @Inject
    public ConstructorValueSample(Course course,

```

```

        @Value(value = "name-value", delimiter = "-") String name,
        @Value("int-value") int credits) {
    this.course = course;
    this.name = name;
    this.credits = credits;
}
}

```

and the `value.properties`:

```

name-value=CS309-OOAD
int-value=3

```

After injected, some fields' values are:

```

int credits = 3
String name = "CS309"

```

💡 Hint: You may need to think about how to get the generic type information from the object of `java.lang.reflect.Parameter` of the constructor.

4. How to inject value?

👉 If the `value()` String appears in the `value.properties`, then first replace it as defined in the properties file; Otherwise, use itself as `value()`. So that the inject value of the fields annotated by `@Value` are according to the mapping value in `value.properties`.

👉 `value.properties` may contain multiple values for a key. For example, it may contain `bili=bilibili-22-33-2233` or `rick=[never,gonna,give,you,up]`. You should read the previous 6 explanations to inject values for it.

For example:

Some field code of a class:

```

@Value(value = "bili", delimiter = "-")
int bili;
@Value(value = "hello,world")
String hello;
@Value(value = "rick")
String[] rick;

```

properties file:

```

bili=bilibili-22-33-2233
rick=[never,gonna,give,you,up]

```


After injected:

```
int bili = 22
String hello = "hello"
String[] rick = ["never", "gonna", "give", "you", "up"]
```

Explanation:

`bili` and `rick` can be found in the properties file, so first replace the `value()` as defined in the properties file.

For field `bili`, `bilibili` is not decimal number, so we ignore it; 22 is the first number meeting the requirement, so we select 22 as the result of `bili` variable.

For field `rick`, `"never,gonna,give,you,up"` is split by the default `delimiter()` and injects an array of values `{"never", "gonna", "give", "you", "up"}` into array `rick`.

For field `hello`, we cannot find `"hello,world"` in the properties file, so we use `"hello,world"` itself as `value()`. Variable `hello` is injected as String `"hello"` since `"hello"` is the first String meeting the requirement.

II. Properties

1. inject properties

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.

For example,

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

2. value properties

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. If the `value()` String appears in the `value.properties`, then first replace it as defined in the properties file; Otherwise, use itself as `value()`.

For example,

```
niconiconi=2-5-2-5-2
swindle=[Never,Gonna,Give,You,Up]
watermelon={2-yuan:1-jin}
ahhhhh={Chicken-soup-is-coming}
```

III. Requirement

🤖 You should create a public class named `BeanFactoryImpl` which implements the interface `BeanFactory`, and upload the file `BeanFactoryImpl.java` to Sakai.

There are localjudge JUnits and officialjudge JUnits for you. If you can pass all localjudge JUnits, then you will get at least **60** points.

⚠️ You will get a **ZERO** if one of the following happens:

- File name does not meet the requirement
- Compilation failure
- Plagiarism(We **ENSURE** we will launch duplication checking program for your submission!)