Tutorial of Dependency Injection

> The exercise is refered to the assignment of ooad course in 2021 fall semester and 2022 fall semester.
>
> Designed by ZHU Yueming in Dec 18th 2023.

Dependency Injection Introduce:

Dependency injection is to create an instance object for specific class. We need to implement and design a general method to create an instance of **any type**, rather than using the **new** keyword to create an instance of a **specified type**. In this case, we need to use reflection.

1. Introduce of Reflection and Annotation:

If we have three classes `AA`, `BB` and `CC`:

```java
public class BB {}
public class CC {}
public class AA {
    @Value(value = "n1")
    private int field;

    private boolean isOk;

    private BB bb;
    private CC cc;

    public AA(BB bb, CC cc) {
        this.bb = bb;
        this.cc = cc;
    }

    @Inject
    public AA(BB bb, CC cc, @Value(value = "falseValue") boolean isOk) {
        this.bb = bb;
        this.cc = cc;
        this.isOk = isOk;
    }

    public AA(int field) {
    }

    public int getField() {
        return field;
    }
    @Override
    public String toString() {
        return String.format("AA{field=%d,isOK=%s,bb=%s,cc=%s}", this.field,
this.isOk, this.bb, this.cc);
    }
}
```

We have two annotation:

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

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
public @interface Value {
    String value();
    int min() default Integer.MIN_VALUE;
    int max() default Integer.MAX_VALUE;
}
```

Before your start, you should load the property files

```
public static Properties loadProp(String path) {
        Properties p = new Properties();
        InputStream in = null;
        try {
            in = new BufferedInputStream(new FileInputStream(path));
            p.load(in);
            return p;
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

Using the method above to create two property instances:

```
Properties injectProp = loadProp("properties/private-inject.properties");
Properties valueProp = loadProp("properties/private-value.properties");
```

1.1 How to create instance by reflection?

Only for non-parameter constructor:

```
BB bObject = BB.class.getDeclaredConstructor().newInstance();
CC cObject = CC.class.getDeclaredConstructor().newInstance();
```

For constructor which has two objects:

If we use `public AA(BB bb, CC cc)` to create an instance of AA, we will:

```
//create objects array for the parameters of public AA(BB bb, CC cc)
Object[] objects = new Object[]{bObject, cObject};
//get constructor AA(BB bb, CC cc)
Constructor<?> AConstructor = AA.class.getDeclaredConstructor(BB.class,
CC.class);
//create instance
AA aa = (AA)AConstructor.newInstance(objects);
```

1.2 Get all parameter types of Constructor:

Normally, we don't understand the signature of constructor we will use, which means we don't know AA class has a constructor which signature is `AA(BB bb, CC cc)`. In this case, we need to find the constructor with Annotation firstly, then get the signature of this constructor.

How to get constructor only with @Inject

```
Constructor<?> constructor = null;
for (Constructor<?> c : AA.class.getDeclaredConstructors()) {
            if (c.getAnnotation(Inject.class) != null) {
                constructor = c;
                break;
            }
}
```

How to get parameters of Constructor

```
Parameter[] parameters = constructor.getParameters();
//Test all parameters' type
for (Parameter p:parameters) {
            System.out.printf("Type = %s\n",p.getType());
}
```

1.3 Create an instance only by a constructor.

We have got the parameter types above, then we will do following steps:

- **Create an Object array for parameters**, so that the length of the object array equals to the length of parameter array.

```
Object[] objects = new Object[parameters.length];
```

- **Check whether a Parameter type p has an `@Value` annotation:**

```
if(p.getAnnotation(Value.class) != null) // p is a parameter type
```

- **How to get `@Value` Annotation instance:**

```
Value valueAnnotation = p.getAnnotation(Value.class); // p is a parameter
type
```

  - **How to get the `value()` , `min()` and `max()` of in `@Value` Annotation:**

```
valueAnnotation.value();//valueAnnotation is a @Value type. and value() is
defined in @Value annotation class.
valueAnnotation.min();
valueAnnotation.max();
```

Example:

```
 if (p.getAnnotation(Value.class) != null) {
           System.out.println("The type of parameter:" +
p.getType().getName());
           Value valueAnnotation = p.getAnnotation(Value.class);
           System.out.println("Annotation Name = " +
valueAnnotation.value());
           System.out.println("Annotation Value = " +
valueProp.getProperty(valueAnnotation.value()));
           if (p.getType() == boolean.class) {
                parameterObject =
Boolean.parseBoolean(valueProp.getProperty(valueAnnotation.value()));
           }
           if (p.getType() == int.class) {
               parameterObject =
Integer.parseInt(valueProp.getProperty(valueAnnotation.value()));
           }
           if (p.getType() == double.class) {
               parameterObject =
Double.parseDouble(valueProp.getProperty(valueAnnotation.value()));
          }
```

```
            if (p.getType() == String.class) {
                 parameterObject =
valueProp.getProperty(valueAnnotation.value());
          }
}
```

- Create an instance by reflection

```
objects[0] = bObject;
objects[1] = cObject;
objects[2] = parameterObject;
AA aObject2 = (AA) constructor.newInstance(objects);
System.out.println(aObject2);
```

1.4 How to inject value into a private field?

```
Field field = aObject.getClass().getDeclaredField("field");
if (field.getAnnotation(Value.class) != null) {
      Value valueAnnotation = field.getAnnotation(Value.class);
      if (field.getType() == int.class) {
          field.setAccessible(true);
          field.set(aObject,
Integer.parseInt(valueProp.getProperty(valueAnnotation.value())));
          field.setAccessible(false);
     }
                 //todo: similary way to inject other type of field
}
System.out.println(aObject);
```

Exercise Question:

1.BeanFactory class

Being used to inject instance according to the property files.

```
public interface BeanFactory {
    void loadInjectProperties(String path);
    void loadValueProperties(String path);
    <T> T createInstance(Class<T> clazz);
}
```

- `void loadInjectProperties(String path);`
  Load all inject data from `path`
- `void loadValueProperties(String path);`
  Load all inject data from `path`

- `<T> T createInstance(Class<T> clazz);`
  Create an instance which type is T.

Notice:

- 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 Annotation

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`.

  ```
  public class Example{
      @Inject
      private A a;
      private B b;
  }
  ```

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

  ```
  public class Example {
      @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 Annotation

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

```
int, double, boolean, String
```

Definition:

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

    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 = "n1")
    private int number;
    @Value(value = "name1")
    private String name;
    @Inject
    private Course course; //combine @Value and @Inject
}
```

- **On parameter**: `ElementType.PARAMETER`
  If `@Value` is marked on parameters in constructor or member 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") String name,
                      @Value("n1") 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`.

Other than that, we set min and max to **int**, **double** and **String** value as the value check.

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

For example

```
@Value(value = "n1", min = 10, max = 20)
int number1;
@Value(value = "n2", min = 10, max = 20)
int number2;
@Value(value = "name", max = 15)
String name;
```

```
n1=25
n2=15
name=Liming
```

After injection:

```
number1 = 0, number2 = 15, name = Liming
```

4. Properties Files

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

```
n1=25
n2=15
n3=5
d1=10.33
d2=80.5
d3=0.95
name1=liMing
name2=hanMei
name3=helloWorld
trueValue=true
falseValue=false
```

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`

Hints and working flow:

Step 1: Find the implement class

You can design a implement class type like:

```
Class<?> ImplClz = null
```

Then check whether the name of `Class<T> clazz` is in `private-inject.properties`

```
if(injectProp.containsKey(clazz.getName()){

}else{

}
```

Step 2: Find Constructor with @Inject Annotation

Following code can return all declared contractors in ImplClz class, then design your code to find the constructor with @Inject Annotation.

```
ImplClz.getDeclaredConstructors();
```

Step 3: Build the parameters array of constructor

- Create a parameter array of constructor, the type of which is `Object[]`
- For each parameter, check if it has `@Value` annotation or not.
  - Has `@Value` annotation: Should build value object.
  - Doesn't have: It must be a class type, and the current parameter object is:

    ```
    createInstance(p.getType());//p is the current parameter
    ```

Step 4: Create instance object

After you have got the constructor and build all parameters array of constructor, you can create instance.

```
T instance = (T) constructor.newInstance(objects);
```

Step 5: Inject all fields of the object

- Get all fields in current class.

  ```
  Field[] fields = ImplClz.getDeclaredFields();
  ```

- **For each** fields, check whether has `@Value` or `@Inject` annotation
  - If has `@Value` : Should confirm a value and set the value to the current field of current object.
  - If has `@Inject` : Should create a new Instance for current field of current object.

  In this case, you can use:

  ```
  //f is the current field of current object
  f.setAccessible(true);
  f.set(instance, xxx);//xxx is a value or a instance
  f.setAccessible(false);
  ```

Step 6: Find methods with @Inject Annotation and invoke it

- Find method by:

  ```
   Method[] methods = ImplClz.getDeclaredMethods();
  ```

- **For each** methods, check whether has `@Inject` annotation.
  - If has. Similar way about the contructor. Build the **parameter array** of current method.
  - Invoke the method by:

    ```
    //m is the current method
    //instance is the current object
    //objectsParameters is the parameter array of current method
    m.invoke(instance, objectsParameters);
    ```

Requirement

You should complete the class named `factory.BeanFactoryImpl` which implements the interface `BeanFactory` , and only check the file `BeanFactoryImpl.java` in checking time.

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