Spring Framework

Dependency Injection Workshop

Objectives

In this workshop, we are going to test the implementation of an application which will cause coupling.
We will try to solve the problem by decoupling by applying:

- Program to Interfaces
- Dependency Injection

Coupling Example project

Java version- IDE

It is recommended that you use Java 17 for this project.

You can use Eclipse or Intellij.

Download the project

You can download the code *1-beginof-coupling.zip*,
The Coupling Example project in this folder.

Download the project Coupling Example and add it to your IDE.

The Project

- In this project there is a class named *Invoice*. This is an ordinary Java class. (POJO)

- We have another class *InvoicingDAO.java* which deals with the database connection and saving into the database.
- We are using the JDBC library to implement our sql database.
- And we are going to use *hsqldb* for the database. It creates a database on the fly in the memory and at the end of the program the database is written to a file. We use this

database because we do not need to think about installing, configuring the database etc. We can just concentrate on Spring. But you are free to use another database.

- Our design does not want us to couple the client directly to the dao, therefore we have a service class in the middle. This is because if we want to change the data access strategy later, for example using hibernate, we do need to change the client.

Now run the client code. Refresh the project. You see that there are two files that have been added in the project:

- *invoicedb.dat.script*
- *invoicedb.dat.properties.*

We can open the script file and see the context. There is an insert statement in the file.

The problems with this implementation

This implementation has two problems.
The application is going to become bigger with more classes that we add overtime and we will encounter two problems:

1. Changes will be difficult to make when the application becomes bigger.
2. Difficult to unit test any of our service classes.

These two problems actually lead to the same problem: coupling.

So we should try reducing coupling as much as we can.
We will solve the problem by decoupling by applying:

- Program to Interfaces
- Dependency Injection
- Centralize configuration (we will apply this in the next workshop)

Program to Interfaces

Interface in java allows us to define just the specifications of the class. No implementation.
For example, we can have an interface:

- InvoicingDAO which has three methods:
    - save,
    - delete and
    - update.

So we need to:

- change the name of the class InvoicingDAO to *InvoicingDAOJdbcImplementation*.
- And declare that this class implements InvoicingDAO.

Create the Interface

```
public interface InvoicingDAO {
   public void save(Invoice newInvoice);
   public void delete(Invoice oldInvoice);
   public void update(Invoice invoiceToCancel);
}
```

Create the implementation

For example: HibernateDAO class.

```
public class HibernateDAO implements InvoicingDAO {
	@Override
	public void save(Invoice newInvoice) {
		System.out.println("Saving an invoice using Hibernate");
	}

	@Override
	public void delete(Invoice oldInvoice) {
		// TODO Auto-generated method stub

	}

	@Override
	public void update(Invoice invoiceToCancel) {
		// TODO Auto-generated method stub

	}
}
```

Changes in InvoiceService class (the principle of programming to an interface)

In *InvoiceService* class, we follow the principle of programming to an interface:

So we change the implementation to interface:

```
InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
```

This will be our first step of loosening the coupling. But still we have coupling. The word “new” means that we still have some coupling. This means that without recompiling our code, there is no way to change the implementation class. But we have done step 1 and it is one step toward reducing coupling.

Dependency Injections

Now we need to do step 2: Dependency Injections
Dependency injection means to pass in the collaborators as parameters rather than hardcoding them.
It means that we stop creating objects with ‘new’ and instead we use a method to set the property.

- First we make ‘*dao’* as an attribute of the class.

Still in *InvoiceService* class, we make some changes in our class: we make ‘dao’ as a private property and we delete the creation of dao in the method *raiseInvoice*l:

```
private InvoicingDAO dao;
```

- Now we create a method to create the object dao:

```
public void setDao (InvoicingDAO dao){
            this.dao = dao;
}
```

You should remove the line:

```
InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
```

in the raiseIvoice method, otherwise the code will not work.
The raiseInvoice method should only contain:

```
dao.save(newInvoice);
```

The thing that we have done above is called *dependency injection*. By this, we have got much more flexibility. Now we can pass either the jdbc implementation or the hibernate implementation or a testing one to this class.

- In the client class, we can create an instance of the ‘*dao*’ and instantiate it as a jdbc implementation and tie it to the InvoicingService class:

```
InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
```

```
invoices.setDao(dao);
```

Change the data for new Invoice to:
Invoice newInvoice = new Invoice("11076", "Susan McLarane");

So the Client class should look like this:

```
public class Client {
	public static void main(String[] args){
		InvoiceService invoices = new InvoiceService();
		Invoice newInvoice = new Invoice("11075", "Bill McLarane");

		InvoicingDAO dao = new InvoicingDAOJdbcImplementation();
		invoices.setDao(dao);
		invoices.raiseInvoice(newInvoice);
	}
}
```

Run the code.
Now check the invoicedb.dat.script file. You will see that there is another insert statement in it:
INSERT INTO INVOICES VALUES('11076','Susan McLarane')

Next Step

If we want to change the jdbc to hibernate, it would be a minor change in the client code and only switch the implementation. Now we have got some flexibility but still we have coupling in our client code. Now we should the next step which is centralise configuration and here spring comes in.
In the next chapter, we will look at the containers and Spring specifically.