Spring Dependency Injection

Exercise SDI.1 – Spring Hello World

The Setup:

The main objective of this exercise is to test Spring by creating a simple Spring application that prints *"Hello World"* to the console.

Download the W1D2-Empty_Maven_Project from Sakai. Once extracted change the name of the directory to: W1D2-Dependency_Injection-1 and modify the pom.xml to do the following

1. Change the artifact name to: **W1D2-Dependency_Injection-1**

2. Add the following dependencies to the dependencies section:

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.14</version>
</dependency>
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.17.1</version>
</dependency>
```

Part A of the Exercise:
   a) We'll first do this exercise with Java Configuration.

Create a file called **Greeting.java** inside the cs544 directory with the following contents:

```java
package edu.miu.cs544;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Greeting {
    @Value("Hello World")
    private String greeting;

    public void sayHello() {
        System.out.println(greeting);
    }
}
```

```java
    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}
```

Next we want to create a spring configuration file. Create a file called **Config.java** inside the cs544 directory with the following content:

```java
package edu.miu.cs544;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("edu.miu.cs544")
public class Config {
}
```

Then update App.java to create a Spring context, retrieve the Greeting bean from it, and call the .sayHello() method. In other words, your main should contain the following:

```java
ApplicationContext context = new
AnnotationConfigApplicationContext(Config.class);
Greeting greeting = context.getBean("greeting", Greeting.class);
greeting.sayHello();
```

In Visual Studio Code you can run your main by clicking on the word run just above the main method, see screenshot below.

```java
1    package edu.mum.cs544;
2
3    import org.springframework.context.ApplicationContext;
4    import org.springframework.context.annotation.AnnotationConfigApplicationContext;
5
6    public class App {
        Run | Debug
7        public static void main(String[] args) {
8            ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
9            Greeting greeting = context.getBean("greeting", Greeting.class);
10           greeting.sayHello();
11       }
12   }
```

The output should appear on the Debug Console (you may have to click on it to see the output)

PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL

ERROR StatusLogger No Log4j 2 configuration file found. Using default con
o show Log4j 2 internal initialization logging. See https://logging.apach
Hello World

You may no longer get this error (newer versions of Log4J don't seem to complain). But if you do you can fix it by providing log4j with a configuration file – create a file called **log4j2.xml** inside the resources directory with the following contents:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n" />
    </Console>
  </Appenders>
  <Loggers>
    <Root level="warn">
      <AppenderRef ref="Console" />
    </Root>
    <Logger name="org.springframework" level="warn">
      <AppenderRef ref="Console" />
    </Logger>
  </Loggers>
</Configuration>
```

If you run App again your output should no longer have the error.

Part B of the Exercise:

b) Now lets make it work with XML and the already existing annotations.

Create a file called **springconfig.xml** inside the resources directory with the following contents:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">
```

```
    <context:component-scan base-package="edu.miu.cs544" />
</beans>
```

Then update your App.java to use this springconfig.xml file instead of Config.java by changing the first line inside main to be:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("springconfig.xml");
```

Running your application again should create the exact same output as before.

Part C of the Exercise:

    c) Lastly lets configure everything with XML (no annotations).

Update your **springconfig.xml** to the following:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="greeting" class="edu.miu.cs544.Greeting">
             <property name="greeting" value="Hello World from XML" />
       </bean>
</beans>
```

This time when you run your application you should see:

```
Hello World from XML
```

Submit on Sakai:

You can already submit your report for this exercise on Sakai. Please also attach your solution code as a zip file. Be sure to delete the target directory from your Maven project, and then zip the whole exercise directory when uploading to Sakai.

## Exercise SDI.2 – Dependency Injection

The Setup:

The main objective of this exercise is to convert an existing application to use Spring. In the process of which you will practice creating configuration for Spring and doing Dependency Injection.

Start this project downloading the W1D2-Dependency_Injection_2 maven project from the Sakai assignment, and opening it in your IDE.

The Application:

The application provided is a very simple application that has a ProductService class, which implements the IProductService interface and contains a list of Products. App.java calls the

getProduct() method on ProductService.
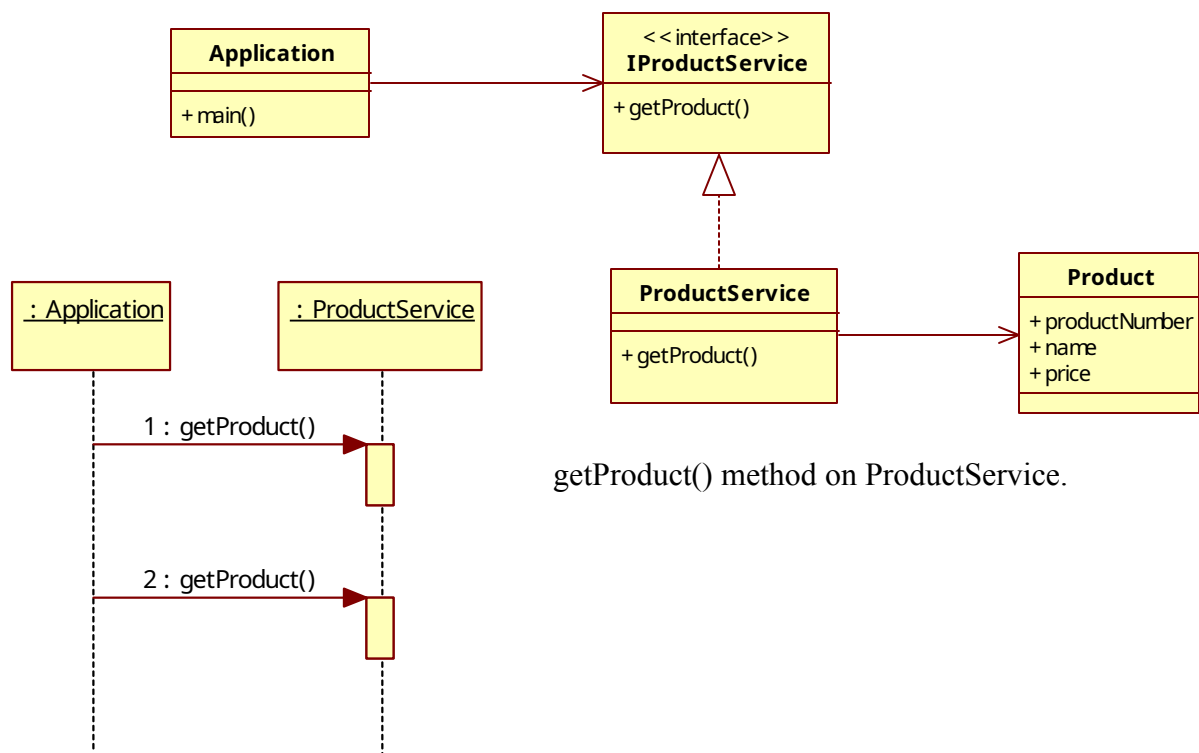
You can run the file **App.java** (for Visual Studio Code we described how to do this in the previous exercise) and should see the following output:

```
productnumber=423  ,name=Plasma TV ,price=992.55
productnumber=239  ,name=DVD player ,price=315.0.
```

Part A of the Exercise:

a) Change the application in such way that **App.java** no longer instantiates **ProductService** but instead retrieves this object from the Spring context. In other words, change the following code:

```
IProductService productService = new ProductService();
```

to the following lines of code:

```
ApplicationContext context = new
    AnnotationConfigApplicationContext(Config.class);
IProductService productService =
    context.getBean("productService", IProductService.class);
```

Of course, this will only work if we add the Spring Dependencies in the pom.xml, and then configure the ProductService to be a bean and create a Config.java file that knows about it.

Since we demonstrated how to this both in the lecture and in the previous exercise I will leave the implementation of these steps up to you.

Part B of the Exercise:

b) Now we want to add an **InventoryService** object, and inject this InventoryService into the ProductService using Spring dependency injection.
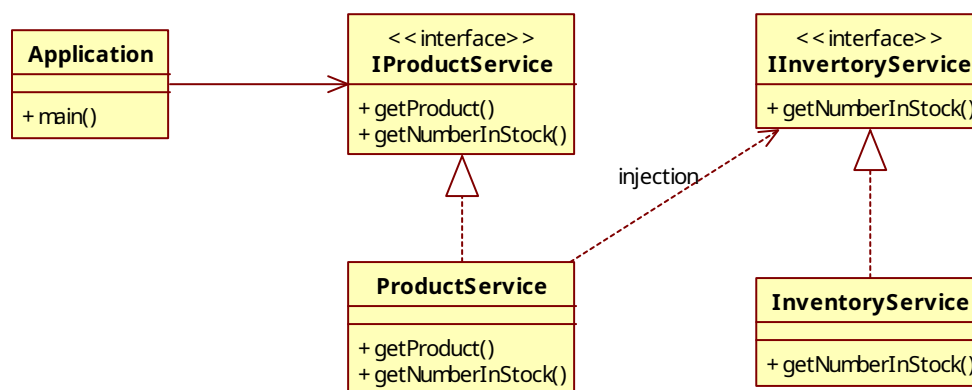
We will also want to add a **getNumberInStock()** method to the ProductService that calls the getNumberInStock() method of InventoryService.
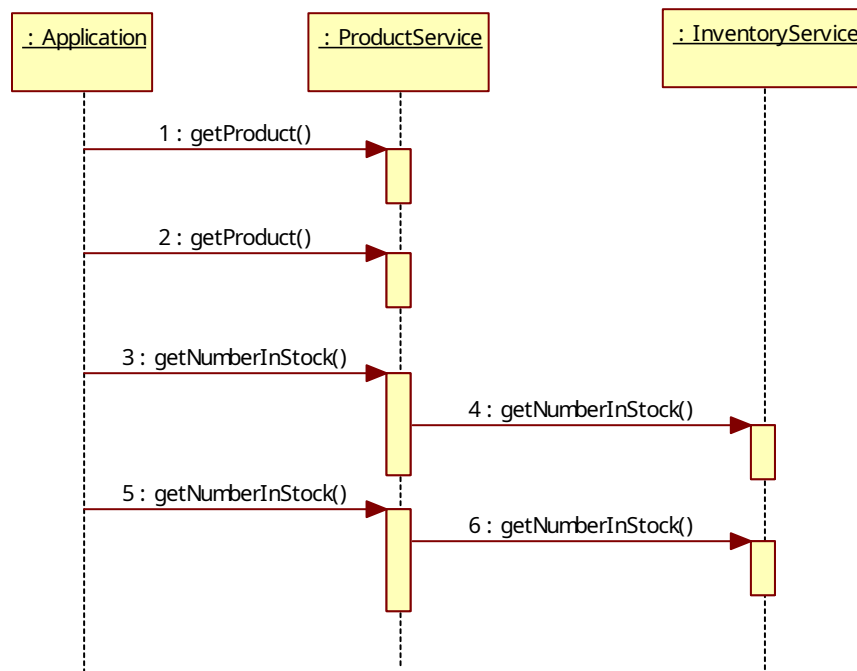
First, create a new interface called IInventoryService and then a class called InventoryService using the code below:

```java
public interface IInventoryService {
    public int getNumberInStock(int productNumber);
}


public class InventoryService implements IInventoryService{

    public int getNumberInStock(int productNumber) {
        return productNumber-200;
    }

}
```

Then, update IProductService and ProductService as shown in the code below.

```java
public interface IProductService {
    public Product getProduct(int productNumber);
    public int getNumberInStock(int productNumber);
}
```

```java
public class ProductService implements IProductService{
    private IInventoryService inventoryService;
    private Collection productList= new ArrayList();

    public ProductService(){
        productList.add(new Product(234,"LCD TV", 895.50));
        productList.add(new Product(239,"DVD player", 315.00));
        productList.add(new Product(423,"Plasma TV", 992.55));
    }
    public Product getProduct(int productNumber) {
        for (Product product : productList) {
            if (product.getProductNumber() == productNumber)
                return product;
        }
        return null;
    }
    public int getNumberInStock(int productNumber) {
        return inventoryService.getNumberInStock(productNumber);
    }
    public void setInventoryService(IInventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }
}

}
```

Then configure the Inventory Service to be a spring bean, and make sure that it is properly injected into the ProductService.

To test if this works, add the following two lines to **App.java**:

```java
System.out.println("we have " + productService.getNumberInStock(423)
        + " product(s) with productNumber 423 in stock");
System.out.println("we have " + productService.getNumberInStock(239)
        + " product(s) with productNumber 239 in stock");
```

When you run **App.java** it should give the following output:

```
productnumber=423  ,name=Plasma TV ,price=992.55
productnumber=239  ,name=DVD player ,price=315.0
we have 223 product(s) with productNumber 423 in stock
we have 39 product(s) with productNumber 239 in stock
```

Submit on Sakai:

Update your report on Sakai to include your comments for this exercise. Again also include your solution code attached as a zip file (delete the target directory before creating the zip).
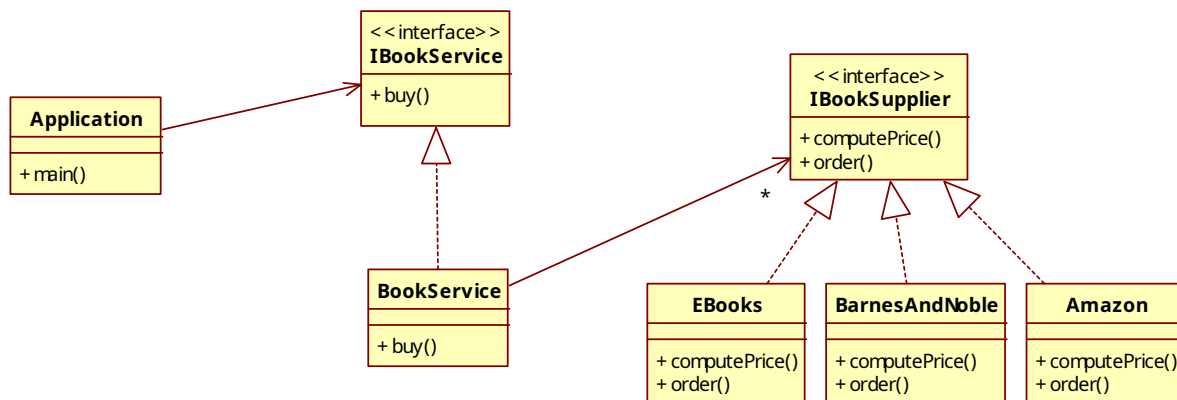
## Exercise SDI.3 – Dependency Injection using Lists

The Setup:

The purpose of this exercise is for you to use the more advanced list configuration feature of dependency injection.

Start by downloading the **W1D2_Dependency_Injection_3** project from Sakai and adding the spring dependencies to the **pom.xml** file.

The Application:

Looking through the code, you will see that the application buys 3 books through the IBookService implemented by BookService. In the buy method, the BookService checks each of its IBookSuppliers, finding the cheapest one and ordering the book from there.
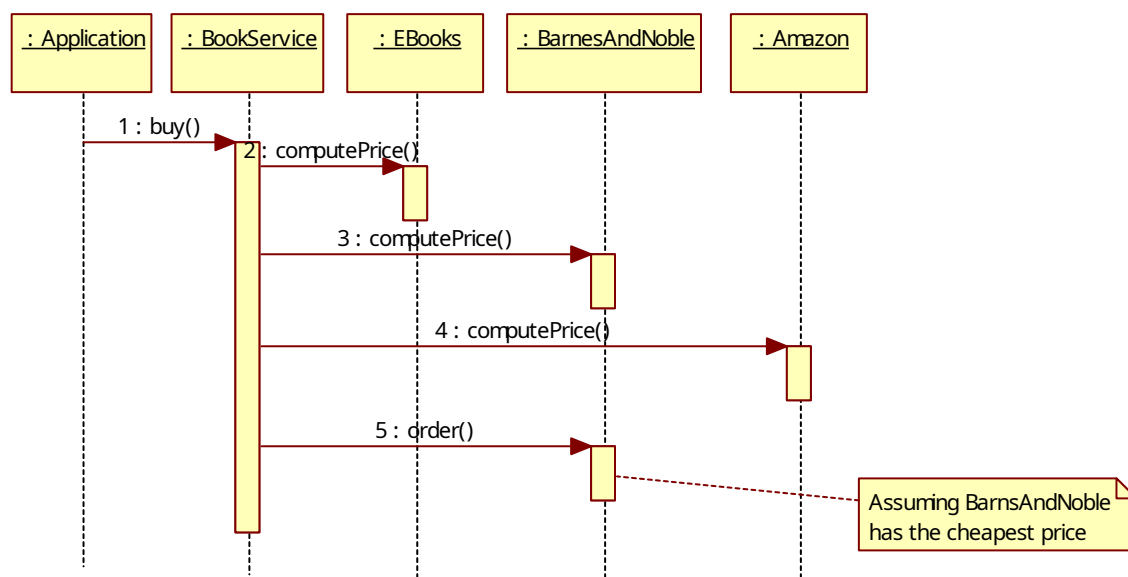
Running Application.java should produce the following (randomized) output:

```
Amazon charges $30.693970696674995 for book with isbn 123433267
Barnes&Noble charges $6.554986457356226 for book with isbn 123433267
EBooks charges $41.282876907999295 for book with isbn 123433267
Book with isbn = 123433267 is ordered from Barnes&Noble
Amazon charges $42.430314310592614 for book with isbn 888832678
Barnes&Noble charges $21.83991052230513 for book with isbn 888832678
EBooks charges $15.977769931887757 for book with isbn 888832678
Book with isbn = 888832678 is ordered from EBooks
Amazon charges $35.20968199800302 for book with isbn 999923156
Barnes&Noble charges $0.5630258200828281 for book with isbn 999923156
EBooks charges $10.151810464638245 for book with isbn 999923156
Book with isbn = 999923156 is ordered from Barnes&Noble
```

The downside of this application is that the BookService is hardcoded with Amazon, EBooks and Barnes & Noble as IBookSuppliers. If we should want to add another book supplier, we would have to go in and change the code.

The Exercise:

a) Change the application in such a way that the **List<IBookSupplier>** is injected into the BookService (already filled with IbookSuppliers). In other words, remove the code that instantiates them with *new* . You will also have to update **App.java** to retrieve the BookService from Spring.
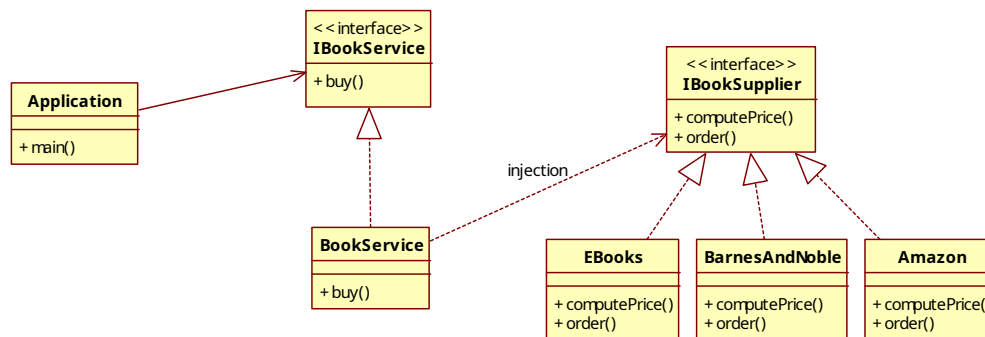
b) Once this is done, add the **Borders** BookSupplier to the list while only changing configuration (no business related Java code).
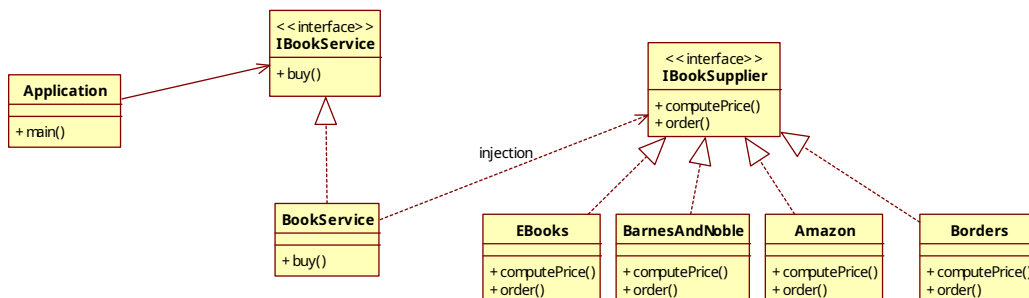
Submit on Sakai:

The last step is of course deleting the target directory in your project, zipping it up and attaching it on Sakai and writing your report in the textbox.

Please let me know how everything went, even if one or more of the exercises did not work for you I need to know (so that I can still give you some points for the homework).