© Victor Tan 2022

Spring Core Workshop
Lab 1

1 Lab setup

Make sure you have the following items installed

- Latest version of JDK 11 (note: labs are tested with JDK 11 but should work with higher versions with no or minimal changes)
- Eclipse Enterprise Edition for Java (or a suitable alternative IDE for Enterprise Java)

- Latest version of Maven
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`. The `changes` subfolder holds the source code and other related files for the lab, while the `final` subfolder holds the complete Eclipse project starting from its project root folder. We will use the code from the `changes` subfolder to build up our applications from scratch and you can always fall back on the complete Eclipse project if you encounter any errors while building up the application.

2 Demonstrating IoC and DI

The source code for this lab is found in `Basic-Concepts/changes` folder.

Switch to Java Perspective.

Create a new Java project: `BasicConcepts`

Create a new package: `com.workshop.original`

Place these files from the same package name in `changes` into `src`:

```
SwimmingExercise.java
JoggingExercise.java
Student.java
```

In `Student.java`, right click and select Run as -> Java Application

What happens if SwimmingExercise wants to change its implementation of doSwimming ?

Make the following changes to these files in `src` from the same package name in `changes`:

```
SwimmingExercise-v2.java
Student-v2.java
```

What happens if Student wants to do Jogging instead?

Make the following changes to these files in `src` from the same package name in `changes`:

```
Student-v3.java
```

Create a new package `com.workshop.useinterface`

Place these files from the same package name in `changes` into `src`:

```
Exercise.java
JoggingExercise.java
Student.java
SwimmingExercise.java
```

What happens if SwimmingExercise wants to change its implementation of doSwimming ?

Make the following changes to these files in `src` from the same package name in `changes`:

`SwimmingExercise-v2.java`

What happens if Student wants to do Jogging instead?

Make the following changes to these files in `src` from the same package name in `changes`:

`Student-v2.java`

Demonstrating Inversion of control (IoC)

Make the following changes to these files in `src` from the same package name in `changes`:

`Student-v3.java`

3 XML-based configuration basics

The primary issue with creating an initial Spring project is to ensure that the relevant Spring module JARs are on the build path of our application. There are two main ways to ensure this:

1) Create a basic Java project. Download and include relevant Spring module JARs on the build path of our project

2) Create a basic Maven project. Specify dependencies for relevant Spring modules into POM.xml

The source code for both approaches is found in `XML-Config-Basics/changes` folder.

We will start with the first approach.

3.1 Basic Java Project

Switch to Java SE perspective.

Create a new Java project: `XMLConfigWithJARs`

Place these files from `changes` into `src`:

`beansDefinition.xml`

Create a new package: `com.workshop.configxml`

In this package, create 5 classes:

```
SwimmingExercise.java
JoggingExercise.java
CyclingExercise.java
```

`Exercise.java`
`XMLConfigBasicMainApp.java`

Notice that there is a syntax error registered on `XMLConfigBasicMainApp` as the relevant Spring module classes are not on the build class path yet.

In the project, create a new folder `lib`.

Copy and paste the following JAR files from the `jars to use` folder into the `lib` folder of your project.

- `spring-aop-x.y.z.jar`
- `spring-beans-x.y.z.jar`
- `spring-context-x.y.z.jar`
- `spring-core-x.y.z.jar`
- `spring-expression-x.y.z.jar`
- `spring-jcl-x.y.z.jar`

On the project, select Properties -> Java Build Path
Select the Libraries Tab, then select Classpath. Click Add JARs. Select all the 6 JAR files that you just pasted into `lib`. Click Apply and Close.
Notice that you can expand any of these JAR files to see its contents in the Referenced Libraries entry in the Package Explorer.

The previously flagged syntax errors in `XMLConfigBasicMainApp` should now have disappeared. Right click on this file and select Run As -> Java Application. Verify that the correct bean is created and its console log displayed in the Console view.

ClassPathXmlApplicationContext is the Spring IoC container that is initialized and bootstrapped in `XMLConfigBasicMainApp`
Once it starts up, it will read all the bean definitions in the XML configuration, whereupon these beans are said to be registered with the container. We then subsequently retrieve the registered beans using the `getBean` method of ClassPathXmlApplicationContext

Change the contents of `beansDefinition.xml` to reflect different classes for `favoriteExercise`, for e.g.

```
<bean id="favouriteExercise"
    class="com.workshop.configxml.CyclingExercise">
</bean>
```

```
<bean id="favouriteExercise"
    class="com.workshop.configxml.JoggingExercise">
</bean>
```

And again run `XMLConfigBasicMainApp` to verify that the correct bean is instantiated based on the console output to the screen.

3.1.1 XML schema explanation

Each main module of the Spring framework has a set of elements with their own respective schemas

https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#xsd-schemas
For e.g. in Spring core, these elements have their own schema

- aop
- context
- beans

and so on.

The list of all schemas online is browsable at:
http://www.springframework.org/schema/

If you want a schema related to a particular older version of Spring, specify the schema version explicitly. If you do not specify a version, Spring will use the latest version for that particular schema.

For Spring Core, typically you will be working with the Spring <beans> and <context> elements, so you will need to include the schema for both of this in the XML configuration file:
https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#xsd-schemas-context

```
<?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">

    <!-- Define your beans here -->

</beans>
```

Keep in mind that every element defined in the XML configuration file must have 2 parts to it in the schema definition for it:

- A `xmlns` binding. For e.g. here we have a binding for the:

<beans> namespace (`beans xmlns=` *http://www.springframework.org/schema/beans*)
<context> namespace (`xmlns:context =` *http://www.springframework.org/schema/context*)

- A `xsi:schemaLocation` specification (consisting of 2 parts for each element) : the element namespace identified earlier in the binding and the actual location of the schema for that namespace

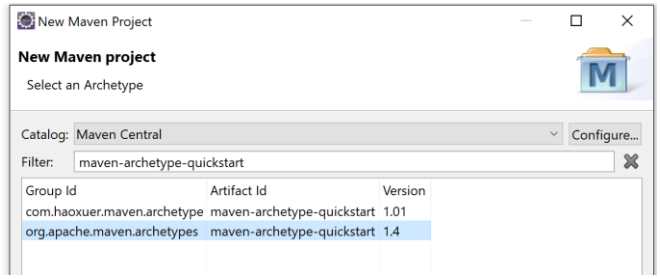
For e.g. for <beans>, this is
http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd
and for <context> this is
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd

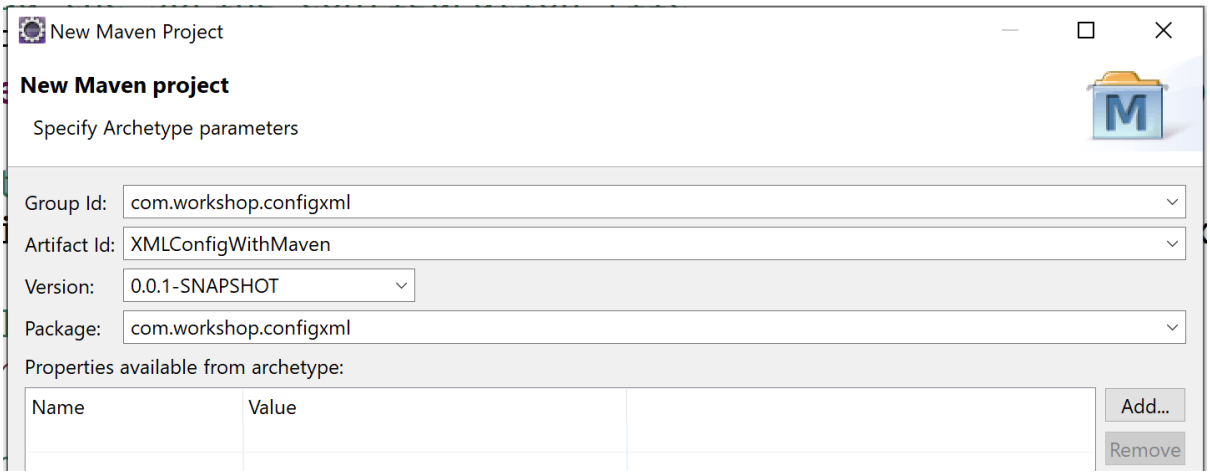
3.2 Basic Maven Project

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter in the following details and click Finish.



Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`.

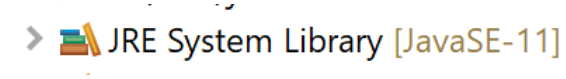
We have made the following changes to the standard autogenerated POM:

- Update the Java version
- Added in the `spring-context` dependency at the latest version:

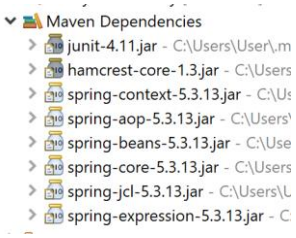
The latest version of Spring-Context can be found at:
https://mvnrepository.com/artifact/org.springframework/spring-context

Typically this should align with the core Spring Framework version shown at:
https://spring.io/projects/spring-framework

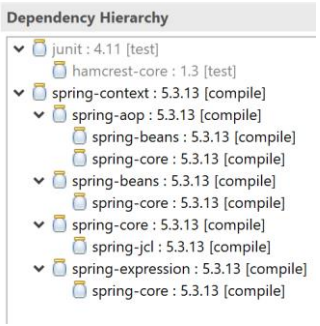
Right click on the project, select Maven -> Update Project, and click OK. You should see the JRE system library entry in the project list update to the new version.



You should also the following JARs automatically added as Maven dependencies

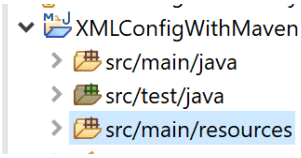


Notice that the single `spring-context` direct dependency itself has a number of transitive dependencies, which themselves in turn have transitive dependencies. Also notice that the multiple instances of transitive dependencies (such as spring-core) all have identical versions, so there is no need for Maven to perform dependency mediation and also no issue of the application code not working properly.



Right click on project and select New -> Folder. Create a new folder named `src/main/resources`

Right click on the new folder, select Build Path -> Use as Source Folder. This will add the contents of this folder to the build path. You should now see this folder being registered in the entries below the project name (all these entries indicate folders which are on the application build path).



In `src/main/java`, there should already be a package: `com.workshop.configxml`
You can delete the autogenerated `App.java` in here.

Copy all the previous 5 classes from the previous project `XMLConfigWithJars/src` and place them in the same package `com.workshop.configxml` in this Maven project.

```
SwimmingExercise.java
JoggingExercise.java
CyclingExercise.java
Exercise.java
XMLConfigBasicMainApp.java
```

Copy `beansDefinition.xml` from the previous project `XMLConfigWithJars/src` and paste it into `src/main/resources` in this Maven project

Open and right click on `XMLConfigBasicMainApp` and select Run As -> Java Application. Verify that the correct bean is created and its console log displayed in the Console view.

Change the contents of `beansDefinition.xml` to reflect different classes for `favoriteExercise`, for e.g.

```
<bean id="favouriteExercise"
    class="com.workshop.configxml.CyclingExercise">
</bean>
```

```
<bean id="favouriteExercise"
    class="com.workshop.configxml.JoggingExercise">
</bean>
```

And again run `XMLConfigBasicMainApp` to verify that the correct bean is instantiated based on the console output to the screen.

3.2.1 Reading from multiple XML configuration files

We can configure the IoC container `ClassPathXmlApplicationContext` to read from more than one XML configuration file at once in order to obtain bean definitions. Let's demonstrate this:

Make the following changes:

- Add `backupDefinition.xml` to `src/main/resources`
- Make the change `XMLConfigBasicMainApp-v2.java`

Run `XMLConfigBasicMainApp` and verify that the output is as expected

3.2.2 Using the default (no-arg) constructor to initialize beans

When the ClassPathXmlApplicationContext container attempts to instantiate the beans defined in the XML configuration at startup, it will call the default (no-arg) constructor of the bean class. Currently, we do not have any constructors in the 3 classes (CyclingExercise, JoggingExercise, SwimmingExercise),

therefore Java will provide a default (no-arg) constructor implicitly. However, if we were to explicitly specify a constructor in the bean classes, this will no longer be the case.

Make the changes to the following files in `src/main/java` from `changes`:

`SwimmingExercise-v2`

Here we have added in a single field and constructor to initialize that field.

Now run `XMLConfigBasicMainApp` again. The console output now shows that a BeanCreationException is thrown due to a BeanInstantiationException due to the inability to locate a default constructor.

Make the changes to the following files in `src/main/java` from `changes`:

`SwimmingExercise-v3`

We now add in a default constructor to initialize the field with a hardcoded constant.

Now run `XMLConfigBasicMainApp` again. This time the output appears as expected as the bean can be initialized successfully.

3.2.3 Listing the defined beans registered in the container

It is often useful to be able to see all the beans registered in the container after it has been bootstrapped.

Make the changes to the following files in `src/main/java` from `changes`:

`XMLConfigBasicMainApp-v3`

Here we retrieve the bean names using getBeanDefinitionNames and print them out.

Now run `XMLConfigBasicMainApp` again. Notice that the name of the beans are given by the `id` attribute in the XML configuration, and not the actual class name of the bean.

Introduce an error into the FQN (fully-qualified name of the class -> package name + class name) of any of the bean definitions in the XML configuration file. For e.g.

```
<bean id="favouriteExercise"
    class="com.dumbo.xyz.SwimmingExercise">
</bean>
```

Now run `XMLConfigBasicMainApp` again.

This time, bean initialization fails with CannotLoadBeanClassException.

Revert the value of the class attribute back to the correct FQN, and run `XMLConfigBasicMainApp` again to verify that it can work as usual.

3.2.4 Multiple aliases for a bean

We can provide multiple names / aliases for bean using the `name` attribute.

Make the changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v2.xml`

Now change your code in `XMLConfigBasicMainApp` to attempt to retrieve the bean using any one of the new aliases for `favouriteExercise`, for e.g.

```
Exercise firstExercise = context.getBean("exercise1", Exercise.class);
```

Notice that the bean can be retrieved successfully, but the listing of defined bean names in the container still shows the value for the `id` attribute.

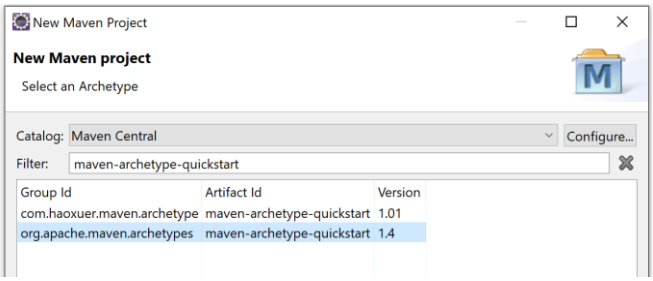
Try using an invalid name instead when retrieving the bean (i.e. a value that does not exist in the `id` or `name` attribute of any bean defined in the XML configuration files). You will obtain a NoSuchBeanDefinitionException.

4 XML-based constructor DI

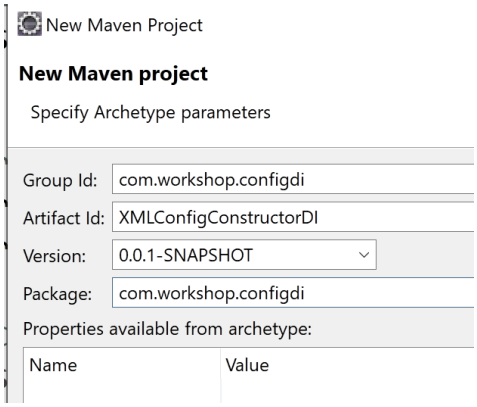
The source code for this lab is found in `XML-Config-Constructor-DI/changes` folder.

Switch to Java EE perspective.

Start with File -> New -> Maven Project. Select Next and type `maven-archetype-quickstart` in the filter. Eclipse may freeze temporarily while it attempts to filter through all the archetypes available at Maven Central. Select the entry with group id: `org.apache.maven.archetypes` and click Next



Enter in the following details and click Finish.



Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

Right click on project and select New -> Folder. Create a new folder named `src/main/resources` Right click on the new folder, select Build Path -> Use as Source Folder. This will add the contents of this folder to the build path.

There are 2 approaches to specify constructor injection with a bean:

- Nest the child bean within the parent bean as a child element of the of the parent bean using the **<constructor-arg>** element
- Define the child bean separately and then reference it via the **ref** attribute of the **<constructor-arg>** element

4.1 Constructor injection with nested child bean

Copy `beansDefinition.xml` from `changes` into `src/main/resources`

Copy the following files from `changes` into `com.workshop.configdi in src/main/java`:

```
CollegeStudent.java
CyclingExercise.java
DemoCollectionsUse.java
Exercise.java
HighSchoolStudent.java
JoggingExercise.java
SwimmingExercise.java
Student.java
XMLConfigConstructorDIMainApp.java
```

Notice that since the nested child bean is defined in its entirety inside CollegeStudent and will not be used or referenced anywhere else, we can define it without an `id` or `name` attribute.

Open and right click on `XMLConfigConstructorDIMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

4.2 Constructor injection with referenced child bean

Make changes to the following files in `com.workshop.configdi` in `src/main/java` from `changes`:

`XMLConfigConstructorDIMainApp-v2.java`

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v2.xml`

Right click on `XMLConfigConstructorDIMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

4.3 Constructor injection with literal values

Make changes to the following files in `com.workshop.configdi` in `src/main/java` from `changes`:

`CyclingExercise-v2.java`
`XMLConfigConstructorDIMainApp-v3.java`

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v3.xml`

Notice that the <constructor-arg> element values are passed based on matching the ordering of these elements and the order of the parameters in the constructor. All the values for the `value` attribute must be Strings: Spring will auto-convert them to the appropriate field type when calling the constructor.

Right click on `XMLConfigConstructorDIMainApp` and select Run As -> Java Application. Verify that the 3 fields are initialized correctly via constructor injection and output in the console accordingly.

4.4 Constructor injection with collections

Make changes to the following files in `com.workshop.configdi` in `src/main/java` from `changes`:

`SwimmingExercise-v2.java`
`XMLConfigConstructorDIMainApp-v4.java`

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v4.xml`

SwimmingExercise now includes two commonly-used Java collections: a List and a Map, which are initialized through a constructor. Note that we need to additionally define a no-arg constructor for SwimmingExercise because it is instantiated as an inner bean to be used in CollegeStudent.

Notice how these two collections are injected with literal values in the XML configuration.

Open and right click on `DemoCollectionsUse` and select Run As -> Java Application for a basic example of storing and iterating over a List and a Map.

Right click on `XMLConfigConstructorDIMainApp` and select Run As -> Java Application. Verify that these 2 collections are initialized correctly via constructor injection and output in the console accordingly.

5 XML-based setter DI

The source code for this lab is found in `XML-Config-Setter-DI/changes` folder.

We can create a Maven project from scratch, or we can make a copy from an existing one.
To facilitate the lab progression, we will make a copy of the previous lab project: `XMLConfigConstructorDI`

In the Project Explorer, right click on `XMLConfigConstructorDI`, select Copy and then right click in any empty space in the Explorer and select Paste.

In the Copy Project dialog box, for the new project name, type: `XMLConfigSetterDI`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

Delete all the packages and files in `src/main/java` and `src/main/resources`. We will start populating the project from scratch.

There are 2 approaches to specify setter injection

- Nest the child bean within the parent bean as a child element of the of the parent bean using the **<property>** element
- Define the child bean separately and then reference it via the **ref** attribute of the **<property>** element

5.1 Setter injection with nested child bean

Place `beansDefinition.xml` into `src/main/resources`

Create the package `com.workshop.setterdi in src/main/java` and copy these classes from `changes` into it:

```
CollegeStudent.java
CyclingExercise.java
Exercise.java
HighSchoolStudent.java
JoggingExercise.java
Student.java
SwimmingExercise.java
```

`XMLConfigSetterDIMainApp.java`

Open and right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

5.2 Setter injection with referenced child bean

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v2.xml`

Make changes to the following files in `com.workshop.setterdi` in `src/main/java` from `changes`:

`XMLConfigSetterDIMainApp-v2.java`

Right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

5.3 Setter injection with literal values

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v3.xml`

Make changes to the following files in `com.workshop.setterdi` in `src/main/java` from `changes`:

`XMLConfigSetterDIMainApp-v3.java`
`CollegeStudent-v2.java`

Notice here that we have a mixture of literal values and bean references for setter injection

Right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the output logged to the console is as expected.

5.4 Setter injection using properties file

Place `highSchool.properties` from `changes` into `src/main/resources`

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v4.xml`

Make changes to the following files in `com.workshop.setterdi` in `src/main/java` from `changes`:

`XMLConfigSetterDIMainApp-v4.java`
`HighSchoolStudent-v2.java`

Right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the output logged to the console is as expected.

5.5 Setter injection with collection of primitive values

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v5.xml`

Make changes to the following files in `com.workshop.setterdi` in `src/main/java` from `changes`:

`XMLConfigSetterDIMainApp-v5.java`
`CollegeStudent-v3.java`

Right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the output logged to the console is as expected.

5.6 Setter injection with collection of beans

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v6.xml`

Make changes to the following files in `com.workshop.setterdi` in `src/main/java` from `changes`:

`XMLConfigSetterDIMainApp-v6.java`
`HighSchoolStudent-v3.java`

Notice here how we combine bean references and nested bean definitions inside the list definition in the XML configuration.

Right click on `XMLConfigSetterDIMainApp` and select Run As -> Java Application. Verify that the output logged to the console is as expected.

6 XML-based autowiring

The source code for this lab is found in `XML-Config-Autowiring/changes` folder.

We can create a Maven project from scratch, or we can make a copy from any of the existing Maven projects.
Choose any previous Maven lab project to make a copy from, for e.g.: `XMLConfigConstructorDI`

In the Project Explorer, right click on `XMLConfigConstructorDI`, select Copy and then right click in any empty space in the Explorer and select Paste.

For the new project name, type: `XMLConfigAutowiring`

Replace the contents of the `pom.xml` in the project with `pom.xml` from `changes`. Right click on the project, select Maven -> Update Project, and click OK.

Delete all the packages and files in `src/main/java` and `src/main/resources`. We will start populating the project from scratch.

In the previous examples of constructor and setter injection, we explicitly specified the relationships (or dependencies) between beans - i.e. we explicitly wired the beans. One of key features of the Spring framework is automatic DI where it performs autowiring of beans, whereby it is capable of figuring out the relationship between the various beans in the application without requiring this relationship to be explicitly specified. It can then perform appropriate DI by deciding the correct beans to initialize and inject into the constructors or properties of other beans.

- For XML configuration, autowiring is achieved by adding the `autowire` attribute to the bean definition where it will be used
- For annotation based configuration, autowiring is achieved by applying the @Autowired annotation

There are 3 approaches to performing autowiring by XML configuration

- using byName
- using byType
- using constructor

6.1 Autowiring using byName

Place `beansDefinition.xml` into `src/main/resources`

Create the package `com.workshop.autowiring in src/main/java` and copy these classes from `changes` into it:

```
CollegeStudent.java
Exercise.java
Student.java
SwimmingExercise.java
XMLConfigAutowiringMainApp.java
```

Notice here that CollegeStudent has a single property of type Exercise as well as some setter and getter methods for it. However, the definition of the bean in the XML Configuration does not explicitly specify the bean that will be used to instantiate this property.

In this case, the container will have to figure out how to autowire (or link a suitable bean) to this property. For autowiring by name, it looks for a bean with the same name as the property that needs to be autowired (`myExercise`). Remember that the name of a bean defined in XML configuration is given by its `id` or `name` attribute.

Open and right click on `XMLConfigAutowiringMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

The container will use setter injection to initialize `myExercise` once it has located the correct bean to create, so you must ensure that there is a proper setter method for it. Comment out the setter method for `myExercise` in CollegeStudent and run it again.

Notice that although the container is able to startup and initialize the `collegeStudent` bean, a NullPointer exception occurs. This is because it is unable to find a setter method to initialize the `myExercise` property in CollegeStudent, resulting in it having a null value and hence the NullPointer exception when an attempt is made to invoke a method on it.

Uncomment the setter method for `myExercise` in CollegeStudent and run it again.

Now change the `id` of the bean created from SwimmingExercise to some random value:

```
<bean id="cat" class="com.workshop.autowiring.SwimmingExercise">
</bean>
```

Run `XMLConfigAutowiringMainApp` again. Again we get the same error, but this time it is because the container is unable to find a bean with a matching name of myExercise to wireup to this property in CollegeStudent, resulting it having a null value and the same issue arising again.

Change back to the correct value.

This points to one of the possible risks involving in using autowiring: we cannot be guaranteed that the container will be able to instantiate an appropriate bean to perform the autowiring operation properly. In that case, we must always ensure that we do a null check before attempting to use the property that is autowired.

6.2 Performing a null check to safeguard against autowire failure

Make changes to the following files in `com.workshop.autowiring` in `src/main/java` from `changes`:

`CollegeStudent-v2.java`

This introduces some simple code to do a null check.
Run `XMLConfigAutowiringMainApp` again and introduce any one of those autowiring errors to confirm that the null check works properly.

Make changes to the following files in `com.workshop.autowiring` in `src/main/java` from `changes`:

`CollegeStudent-v3.java`

Here we use the Optional class from Java 8 to perform a null check.

Run `XMLConfigAutowiringMainApp` again and introduce any one of those autowiring errors to confirm that the null check works properly.

6.3 Autowiring using byType

Make changes to the following files in `com.workshop.autowiring` in `src/main/java` from `changes`:

```
HighSchoolStudent.java
XMLConfigAutowiringMainApp-v2.java
```

Make changes to the following files in `src/main/resources` from `changes`:

```
beansDefinition-v2.xml
```

Matching by type is based on the following concept:
Class A is considered of type X, if class A implements interface X or inherits from parent class X

HighSchoolStudent has a property favouriteExercise of type Exercise. In the XML configuration, there is exactly one bean that is of this type: myExercise, which is based on SwimmingExercise that in turn implements Exercise. Notice that the name of the bean does not match the name of the property which is being autowired, but that is not an issue, since we are autowiring by type.

Open and right click on `XMLConfigAutowiringMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.

Comment out the `myExercise` bean definition in the XML configuration and run again. As expected, we obtain a null pointer exception similar to the previous case.

Uncomment the `myExercise` bean definition

6.4 Ambiguity in autowiring byType due to multiple same bean types

Make changes to the following files in `com.workshop.autowiring` in `src/main/java` from `changes`:

```
CyclingExercise.java
```

Make changes to the following files in `src/main/resources` from `changes`:

```
beansDefinition-v3.xml
```

We introduce another bean definition (coolExercise) which is also of the type Exercise into the XML Configuration. This means there are now two beans of the same type Exercise which could be autowired into favouriteExercise in HighSchoolStudent.

Run `XMLConfigAutowiringMainApp` again.

Now we have a NoUniqueBeanDefinitionException thrown because the container is unable to decide between these two candidate beans to initialize the dependency. In addition, a

UnsatisfiedDependencyException is also thrown, since the container is now no longer able to inject the dependency for the highSchoolStudent bean.

To resolve this, we must provide a way to only specify one single bean for autowiring.

One possible approach is to identify the primary candidate bean for autowiring:

Add the following attribute to the coolExercise bean and save:

```
    <bean id="coolExercise" class="com.workshop.autowiring.CyclingExercise"
primary="true">
    </bean>
```

Run `XMLConfigAutowiringMainApp` again. Verify that the coolExercise bean was actually used to instantiate the dependency for the highSchoolStudent bean.

The second approach is to specify which beans are excluded from being considered as candidates for autowiring:

```
    <bean id="coolExercise" class="com.workshop.autowiring.CyclingExercise"
autowire-candidate="false">
    </bean>
```

Run `XMLConfigAutowiringMainApp` again. Verify that now the myExercise bean was actually used to instantiate the dependency for the highSchoolStudent bean.

6.5 Autowiring using constructor

Make changes to the following files in `com.workshop.autowiring` in `src/main/java` from `changes`:

`HighSchoolStudent-v2.java`

Make changes to the following files in `src/main/resources` from `changes`:

`beansDefinition-v4.xml`

This is conceptually identical to autowiring using byType (where candidate beans are identified based on matching type), except that the bean dependency is now initialized via constructor injection instead (so a suitable constructor should be provided)

Open and right click on `XMLConfigAutowiringMainApp` and select Run As -> Java Application. Verify that the correct parent and child bean have been created and their output logged to the console correctly.