Spring Core Workshop
Lab 3

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 Java-based configuration

The source code for this lab is found in `Java-Config-DI/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: `JavaConfigDI`

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.

2.1 Basic configuration setup with @Configuration and @ComponentScan

Create a new package `com.workshop.javaconfig` in `src/main/java`:
Copy the following files from `changes` into `com.workshop.javaconfig` in `src/main/java`:

```
Exercise.java
JavaConfigDIMainApp.java
MainConfig.java
SwimmingExercise.java
```

The @ComponentScan here is the functional equivalent of <context:component-scan> element in the XML configuration file in Annotation-based Java configuration. It specifies the package hierarchy to scan for bean classes that are marked with @Component so that they can be initialized and added to the container
@ComponentScan can only be provided on a class annotated with @Configuration. The @Bean can be used to annotate methods in that class, but this is entirely optional as @ComponentScan can be used independently of @Bean
If @ComponentScan is provided on its own without the specification of a package, then component scanning starts from the package that the @ComponentScan class is in, and scans all subpackages
To scan multiple packages, use this form:

```
@ComponentScan( {"com.workshop.javaconfig",
"com.workshop.anotherpackage"})
```

The @Configuration is a type of @Component and hence the class annotated with this is also registered as a bean in the container

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Verify that the correct bean has been created and its output logged to the console correctly.

Notice that mainConfig (the class annotated with @Configuration) is also registered as a bean in the container.

2.2 Defining beans with @Bean and retrieving them

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

```
CollegeStudent.java
JoggingExercise.java
Student.java
```

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

```
JavaConfigDIMainApp-v2.java
MainConfig-v2.java
SwimmingExercise-v2.java
```

We can mark methods in the @Configuration class with @Bean (they can only be declared in a class like this). The primary purpose of the @Bean method is to provide a way for us to explicitly create and configure beans which are registered with the container when it calls the @Bean method to obtain the bean. This is contrasted with classes marked with @Component, which are automatically initialized by the container without any intervention on part of the developer.

Therefore, the classes that are explicitly instantiated inside the @Bean method DO NOT need to have the @Component annotation applied to their definitions. Notice that the 3 classes: SwimmingExercise, JoggingExercise and CollegeStudent DO NOT have the @Component annotation.

Beans are produced from a @Bean method and initialized and registered in the container. If we wish to obtain them via the getBean method, we will use either the explicit name given to the @Bean method or the default name (if no explicit name is given). The default name of the bean is the method name itself (and NOT the class that the bean is instantiated from).

If a bean produced in a @Bean method has a dependency, the code producing the bean will inject the dependency itself (through a constructor or setter) by calling another @Bean method that satisfies the dependency.

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Verify that the correct bean has been created and its output logged to the console correctly.

Notice the names of the registered beans in the container (either the explicit name or default name)

2.3 Using @Primary to give higher preference to a @Bean method

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

```
CyclingExercise.java
```

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

```
JavaConfigDIMainApp-v3.java
MainConfig-v3.java
```

Here, we have renamed the @Bean methods to their more conventional form: the same as the class that the bean it produces is instantiated from, except that it starts with lower case. This is conceptually identical to the way explicit and default component names work

We use @Primary here to mark a specific @Bean method as the one to produce the bean of choice when there is more than one possible candidate bean.

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Verify that the correct bean has been created and its output logged to the console correctly.

Remove the `@Primary` on `public Exercise cyclingExercise()` in `MainConfig` and run the application again. Notice this time an exception is thrown as Spring is unable to determine which implementation for the dependency to use.

2.4 Injecting values using @PropertySource and @Value

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

```
HighSchoolStudent.java
```

Copy the following files from `changes` into `src/main/resources`

```
highschool.properties
```

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

```
MainConfig-v4.java
JavaConfigDIMainApp-v4.java
```

The @Value annotation is applied to the specific properties in HighSchoolStudent whereby their values will be obtained from the properties file specified by @PropertySource in MainConfig.

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Notice that the correct values are read for HighSchoolStudent's fields.

2.5 Injecting collections using @PropertySource, @Value and SPEL

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

```
CollegeStudent-v2.java
MainConfig-v5.java
JavaConfigDIMainApp-v5.java
```

Copy the following files from `changes` into `src/main/resources`

`collegestudent.properties`

Here we use SPEL expressions with @Value annotations to inject collections (a list and a map) that are hardcoded in a properties file into dependencies within collegeStudent.
We could have placed all the relevant properties into a single properties file. All the properties defined in all properties files specified by @PropertySource are accessible for injection into all @Value annotated properties in all beans defined in the @Bean methods.

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Notice that the correct values are read for CollegeStudent's collection fields.

2.6 Using @Component and @Autowired with @Bean

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

```
CollegeStudent-v3.java
HighSchoolStudent-v2.java
JavaConfigDIMainApp-v6.java
MainConfig-v6.java
```

We can use the @Component and @Autowired annotations from the annotation-based configuration approach here where they retain their original purpose and meanings.
When you apply @Configuration and @ComponentScan to a class X, you now have the option to initialize and register beans in two ways:

- via the @Bean methods defined in class X
- via the @Component classes defined in the package and subpackages specified in @ComponentScan

Subsequently, these registered beans can then be used to perform DI for the @Autowired dependencies in any of the @Component classes.

We have made the following changes here:

- highSchoolStudent and collegeStudent are no longer initialized and registered as beans via the @Bean methods. Instead, they are simply marked with @Component
- Both HighSchoolStudent and CollegeStudent now used @Autowired field injection for their myExercise property (as opposed to previously where we initialized this property explicitly in their constructors in the @Bean method logic).
- Since @Autowired is by type, and there are 3 possible classes that implement Exercise, the container uses the bean marked with @Primary (in this case cyclingExercise) to initialize this property in both HighSchoolStudent and CollegeStudent

Notice that the specification of all relevant properties files (`highschool.properties` and `collegestudent.properties`) are still centralized in MainConfig, and the values in these files

are accessible to any property marked with @Value in any bean (regardless of whether they are produced via @Bean method or from a @Component class).

Open and right click on `JavaConfigDIMainApp` and select Run As -> Java Application. Verify that the output from the retrieved beans is as expected.

At this point we are using @Primary annotation to decide between the 3 possible candidate bean classes of type Exercise to use for @Autowired DI in both HighSchoolStudent and CollegeStudent

We have seen before that we can also use @Qualifier for this purpose. Lets make a minor change to demonstrate this:

Remove the `@Primary` from cyclingExercise() in MainConfig.

Add `@Qualifier("swimmingExercise")` to the `myExercise` dependency in CollegeStudent

Add `@Qualifier("joggingExercise")` to the `myExercise` dependency in HighSchoolStudent

2.7 Case use for @Bean vs @Component

For simple cases, using @Bean to initialize a bean appears to be unnecessarily verbose compared to simply marking a class with @Component, whereby the container automatically initializes a bean from that class for us in the background. However, @Bean can cater for specific situations which cannot be handled adequately with @Component.

- You need to have a customized creation and configuration of beans that depends on certain conditions
- You wish to use components (classes) from 3rd party libraries as @Autowired dependencies in your classes. However, which you do not have access to the source code of the library, and therefore cannot annotate those classes with @Component

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

`MathStudent.java`

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

`JavaConfigDIMainApp-v7.java`

Assume that we want to generate a random number. This can be easily done using the `java.util.Random` class that is part of the Java standard library. In MathStudent, we create an @Autowired dependency of type Random, which we subsequently use in the dailyActivity method.

Run the app. Notice we have a NoSuchBeanDefinitionException due to the inability to locate a bean of type `java.util.Random`. Although this class exists in the Java library, it is not annotated with @Component, so it is not registered as a bean, and therefore we cannot directly use it to instantiate an @Autowired dependency. However, cannot apply the @Component directly to the `java.util.Random` class because we don't have access to the source code of this class.

To accomplish the construction and initialization of myRandomGenerator, we would have to remove the @Autowired annotation on it and instead directly initialize it in a no-arg constructor for MathStudent, which will be called automatically by the container when it creates and initializes the bean:

```
private Random myRandomGenerator;

public MathStudent() {
  myRandomGenerator = new Random();
  myRandomGenerator.setSeed(888L);
}
```

Now the app should run fine, and will produce the same result each time because we seed the generator with the same number before using it.

However, the whole point of using @Autowired is to remove the need for us to explicitly create the object ourselves and initialize it in the manner above. We can still use @Autowired on our dependency here, but delegate the creation of a bean of type Random to a @Bean method.

Remove the no-arg constructor in MathStudent and add the @Autowired back to the myRandomGenerator.

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

`MainConfig-v7.java`

Here we "wrap" the creation and initialization of the Random object inside a @Bean method. And with this approach, the same Random object is accessible for autowiring to any dependency in any bean class of this type.

Run the app again. This time you should get the desired result.

3 Additional topics

3.1 Using overloaded versions of getBean()

The various containers that are typically used (AnnotationConfigApplicationContext and ClassPathXmlApplicationContext) both inherit from AbstractApplicationContext which offers several overloaded versions of getBean to retrieve a bean from the container:

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/support/AbstractApplicationContext.html

So far, we have only used one or two of these methods. Let's examine some of the various other options open to us:

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

`JavaConfigDIMainApp-v8.java`

Run this app and examine all the different ways of retrieval. We have a NoUniqueBeanDefinitionException for the last bean retrieval because currently there are 3 bean classes that are of type Exercise (SwimmingExercise, CyclingExercise and JoggingExercise), so the container is unable to decide which bean to return.

We have already seen how to resolve this issue. Select any one of the @Bean methods for any of these 3 classes in MainConfig and annotate it with @Primary.
Run the app again and this time all beans are retrieved successfully.

So far, we have used programming to interfaces when working with creating and retrieving beans.
For e.g. SwimmingExercise, CyclingExercise and JoggingExercise all implement the Exercise interface.

We can also use inheritance as well. For e.g. by having Exercise being a normal class and all the 3 other classes inheriting from it instead.

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

```
CyclingExercise-v2.java
Exercise-v2.java
JoggingExercise-v2.java
SwimmingExercise-v3.java
```

Run the app again and verify that everything works properly.

You can also just implement standalone classes to be used in defining and retrieving beans WITHOUT the need for any kind of implementation or inheritance approach. For e.g.

```
@Component
public class StandAloneClass
…
…
…

StandAloneClass          sac          =          (StandAloneClass)
context.getBean("standAloneClass");
```

3.2 Specifying bean scope using @Scope

The scope of a bean defines the life cycle and visibility of that bean in the contexts we use it. There are 6 bean scopes overall. The first 2 (singleton, prototype) are applicable to all Spring applications while the remaining 4 (request, session, application and websocket) are applicable only in a web application.

When we define a bean with the singleton scope, the container creates a single instance of that bean; all requests for that bean name will return a reference to that single object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

A bean with the prototype scope will return a different instance every time it is requested from the container.

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

```
SimpleCounter
```

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

```
JavaConfigDIMainApp-v9.java
```

Run the app. Notice that there is only one single instance of SimpleCounter created and all the references returned from the getBean method are pointing to this single instance.

Now add this definition below the @Component in SimpleCounter to change the scope to prototype:

```
@Scope("prototype")
```

Run the app. Notice now that each call to getBean returns a reference to a new instance of SimpleCounter

Now comment out the @Scope annotation from SimpleCounter and save.

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

```
CyclingExercise-v3.java
JoggingExercise-v3.java
JavaConfigDIMainApp-v10.java
```

When we have @Autowired dependencies in different classes, the container will inject all these dependencies with a reference to the same instance of SimpleCounter

Run the app and verify the results.

Now add this definition below the @Component in SimpleCounter to change the scope to prototype:

```
@Scope("prototype")
```

Run the app. Notice now that each @Autowired dependency gets a reference to a new instance of SimpleCounter

Some additional points to keep in mind:

- Although both prototype and singleton beans can carry state, generally prototype beans will carry state while singleton beans are stateless.
- A common use for singleton beans is to implement some service functionality, where only one instance needs to be injected into dependencies of other classes. Service functionality such as creating a new database or opening a database connection should only be done once, usually at the time of instantiation of that singleton service bean. Subsequently, that single bean instance can be shared between multiple objects that need to read/write to that database. In this scenario, the singleton service bean is typically stateless because whatever state they maintain will not be client-specific.
- Other examples where the default scope of singleton is with @Controller classes in Spring MVC apps, for which only one instance needs to be created to handle incoming requests (rather than creating a new instance of a class to handle every single incoming request).

Since classes with prototype scope will have different bean instances instantiated from them every time they are retrieved via a getBean call, we can actually pass arguments to their constructors when retrieving them to create beans with different internal states.

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

```
JavaConfigDIMainApp-v11.java
```

Here, we are using another version of the getBean method which allows us to explicitly pass one or more arguments to the constructor of the bean class.

Make sure that the @Component in SimpleCounter has prototype scope:

```
@Scope("prototype")
```

Run the app. Notice how we are able to call the single-arg constructor of SimpleCounter, passing it a different value each time via the getBean method.

Now comment out the prototype scope in Simple Counter and save.

Run the app. Notice this time that we are no longer able to call the single-arg constructor of SimpleCounter, instead only the no-arg constructor is called.

3.3 Eager vs lazy loading

By default, the container (application context) will attempt to instantiate all beans with singleton scope immediately during the startup / bootstrapping of the container, even if they are not explicitly retrieved and utilized in the application code. This is known as eager loading / initialization. The primary reason behind this is simple: to avoid and detect all possible errors immediately rather than at runtime.

However, sometimes some beans may incur a heavy overhead during initialization and if we have many beans of this nature, the initial container bootstrap may take a long time. To avoid this, we can configure the container so that beans are only initialize when they are required or requested.

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

```
CollegeStudent-v4.java
CyclingExercise-v4.java
HighSchoolStudent-v3.java
JavaConfigDIMainApp-v12.java
JoggingExercise-v4.java
MainConfig-v8.java
MathStudent-v2.java
SimpleCounter-v2.java
SwimmingExercise-v4.java
```

Here we have added no-arg constructors with a console output statement to all the classes that can be potentially initialized as beans. This allows us to track which classes have been created and initialized.

Run the app. Notice that all the beans are initialized and registered immediately after container startup, even if we are not requesting any of the beans registered via getBean.

Now add this annotation

```
@Lazy
```

below @Component in HighSchoolStudent and CollegeStudent and save.

Run the app. This time notice that HighSchoolStudent and CollegeStudent are no longer initialized. On the other hand, both beans are still shown as being registered in the container.

This is because Spring will scan all the packages specified by @ComponentScan or <context:component-scan> and then register all the classes marked with @Component EVEN if it has not yet initialized them. This will show up in the etBeanDefinitionNames call to the container.

Although both HighSchoolStudent and CollegeStudent are no longer initialized, their specific dependencies (joggingExercise and swimmingExercise) are still being initialized.

In MainConfig, add the @Lazy annotation again to the list of annotations for swimmingExercise() and joggingExercise().

Run the app. This time notice that swimmingExercise and joggingExercise are no longer initialized. On the other hand, both beans are still shown as being registered in the container.

We can make all beans created via @Bean methods in a @Configuration class to be lazily initialized.

Remove the @Lazy annotation for swimmingExercise() and joggingExercise() in MainConfig

Add @Lazy to the list of annotations for MainConfig and save.

Run the app. Again notice that none of the beans created via @Bean methods in MainConfig are initialized EXCEPT for getRandomGenerator. This is because this is an @Autowired dependency of MathStudent which is eagerly initialized.
In other words, even if a bean class marked with @Component or produced via @Bean method is marked with @Lazy, it will still be initialized if it is used for injection for an @Autowired dependency of any other bean class that is initialized.

Add @Lazy to MathStudent and save.

Run the app. This time, the only bean that is eagerly initialized is SimpleCounter.

With lazy initialization, beans are only initialized when they are requested (for e.g. via getBean in XML configuration or when DI needs to be performed for @Autowired dependencies in Java annotation).

3.4 Generating an executable JAR with maven-assembly-plugin

So far, we have not yet run any Maven build on this project. We will generate a JAR from this project.

Right click on the project, select Run As -> 4 Maven Build, and select for the goals: `clean package`

Refresh the project and confirm that a JAR (`JavaConfigDI-0.0.1-SNAPSHOT.jar`) has been generated in the `target` folder.

Copy this JAR to any suitable folder and examine its content. Notice that it only contains the package hierarchy and bytecode classes for the current project. None of the other project dependencies from the Spring libraries are included. This means we won't be able to execute the JAR as a standalone JAR. You can try this now from the command prompt to confirm for yourself:

```
java -jar JavaConfigDI-0.0.1-SNAPSHOT.jar
```

There are three ways to construct a uber / fat jar that includes all its project dependences in a single JAR so that it can be executed as a standalone JAR.

a) Unshaded JAR - the dependencies from other JARs are extracted out into their respective package structure and classes and included in the root of the generated JAR.

b) Shaded JAR - same as unshaded, except that we can change the name of the package structure that the dependencies are extracted into so as to avoid interference with other projects that may use conflicting versions of that dependency

c) JAR of JARs - This placed the dependency JARs directly into the generated JAR without extracting them into a package structure first. It requires the implementation of a custom class loader as the conventional Java class loader is designed to read classes from package structure

within a JAR, and not from other JARs within a JAR (as is the case here). Typically, we will do this using Spring Boot's package facility.

We will configure maven-assembly-plugin for the first approach. This is one of the standard plugins officially supported by the Maven project (look under the Tools section) in:
https://maven.apache.org/plugins/index.html#supported-by-the-maven-project

Copy `pom-v2.xml` from changes to overwrite `pom.xml` in the current project.

Notice that we have configured the maven-assembly-plugin in a <plugins> section outside of the <pluginManagement> section but inside the main <build> section. This is important because <pluginManagement> only provides for configuration of the plugins (such as version numbers) but they do not activate the plugin for use in the current project. For that, the plugin MUST be declared and configured (if necessary) in a separate <plugins> section

The configuration of this plugin involves 2 key elements:

- <mainClass> element - refers to the class containing the public static void main method to be executed when the JAR is executed
- <descriptorRef> relement - refers to the extension that will be appended to the generated JAR name to indicate that is a uber / fat JAR.

For more information on configuring this plugin:
https://maven.apache.org/plugins/maven-assembly-plugin/usage.html

Now we can execute appropriate goals from this plugin.

Right click on the project, select Run As -> 4 Maven Build, and select for the goals:

```
clean compile assembly:single
```

Note here that `clean` and `compile` are phases within the clean and default life cycle, while `assembly:single` is executing the single goal within the assembly plugin (maven-assembly-plugin)

Refresh the project and confirm that a JAR (`JavaConfigDI-0.0.1-SNAPSHOT-jar-with-dependencies.jar`) has been generated in the `target` folder.

Copy this JAR to any suitable folder and examine its content. Notice that it now contains the package hierarchy and bytecode classes for the current project AS WELL AS all the other project dependencies from the Spring libraries, which are extracted into their respective package hierarchies as well. This means we can now execute this JAR as a standalone JAR.

You can try this now from the command prompt to confirm for yourself:

```
java -jar JavaConfigDI-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```

Copy `pom-v3.xml` from changes to overwrite `pom.xml` in the current project.

We can now add further configuration to bind the `assembly:single` goal to the `package` phase. This makes is slightly easier for us to generate the uber JAR by reusing the standard command for generating a JAR.

Right click on the project, select Run As -> 3 Maven Build, and select the previous goals:
`clean package`

Refresh the project again and confirm that a JAR with the same name as before (`JavaConfigDI-0.0.1-SNAPSHOT-jar-with-dependencies.jar`) has been generated in the `target` folder.

Copy this JAR to any suitable folder and examine its content and verify that it is exactly the same as the previously generated JAR. Test the execution of this JAR as a standalone JAR.

```
java -jar JavaConfigDI-0.0.1-SNAPSHOT-jar-with-dependencies.jar
```