Day 27- Facilitation Guide

Index

For (1.5 hrs) ILT

During the previous session about the IOC Container,Dependency Injection, we covered a variety of important concepts:

IoC Container in Spring:

The Inversion of Control (IoC) container in Spring is a core component responsible for managing the creation, configuration, and lifecycle of objects (beans) in a Spring application. It follows the principle of Inversion of Control, where control over object instantiation and management is delegated to the container, rather than being controlled by application code. The IoC container eliminates tight coupling between components, promotes reusability, and simplifies the configuration and management of application components.

Dependency Injection in Spring:

Dependency Injection (DI) is a key feature in the Spring framework that allows you to inject dependencies (collaborating objects) into a class rather than having the class create or manage its dependencies. Spring provides several methods for achieving DI, including constructor injection, setter injection, and field injection. By using DI, Spring enhances the flexibility and maintainability of your application, making it easier to change components, manage configurations, and write testable code. This design pattern promotes loose coupling between classes and facilitates the separation of concerns, improving the overall quality of software design in Spring applications.

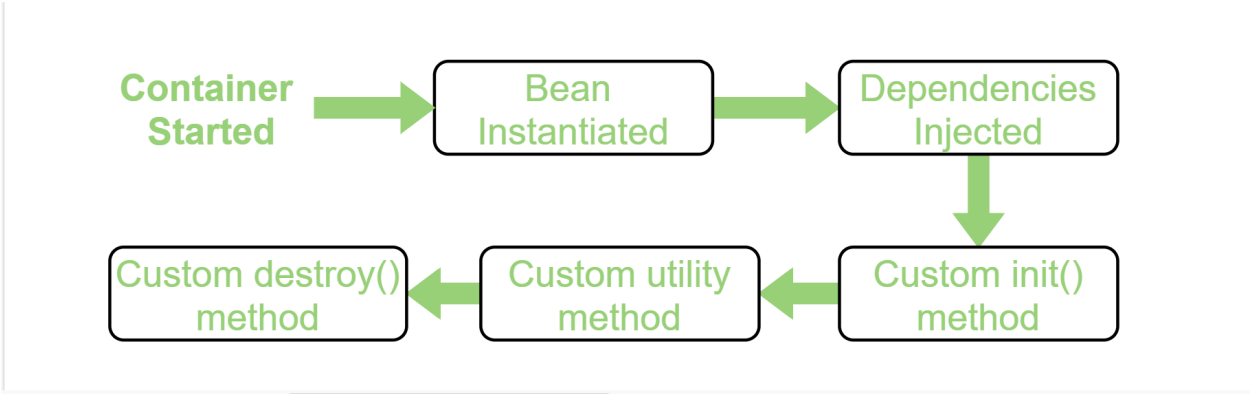
In this session, we'll investigate Bean Lifecycle, Bean Scope,Injecting bean using xml configuration using setter and constructor injection and along with the Autowiring beans using Annotation.

I. Bean lifecycle

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method and the destroy() method.

The following image shows the process flow of the bean life cycle.



Note: We can choose a custom method name instead of init() and destroy(). Here, we will use the init() method to execute all its code as the spring container starts up and the bean is instantiated, and destroy() method to execute all its code on closing the container.

Ways to implement the life cycle of a bean

Spring provides three ways to implement the life cycle of a bean. In order to understand these three ways, let's take an example.

In this example, we will write and activate init() and destroy() method for our bean (HelloWorld.java) to print some messages on start and close of the Spring container. Therefore, the three ways to implement this are:

1. By XML: In this approach, in order to avail custom init() and destroy() methods for a bean we have to register these two methods inside the Spring XML configuration file while defining a bean. Therefore, the following steps are followed:

Step 1: Firstly, we need to create a bean **HelloWorld.java** in this case and write the init() and destroy() methods in the class.

```
package bean_lifecycle;

public class HelloWorld {
	// This method executes
	// automatically as the bean
	// is instantiated
	public void init() throws Exception {
		System.out.println("Bean HelloWorld has been " + "instantiated and I'm " +
"the init() method");
	}

	// This method executes
	// when the spring container
	// is closed
	public void destroy() throws Exception {
		System.out.println("Container has been closed " + "and I'm the destroy()
method");
	}
}
```

Step 2: Now, we need to configure the spring XML file config.xml under src/main/java folder and need to register the init() and destroy() methods in it.

```
<?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-3.0.xsd
```

```
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

        <bean id="hw" class="bean_lifecycle.HelloWorld" init-method="init"
                destroy-method="destroy" />
</beans>
```

Step 3: Finally, we need to create a driver class to run this bean.

```
package bean_lifecycle;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

        public static void main(String[] args) {
                // Loading the Spring XML configuration
      // file into the spring container and
      // it will create the instance of
      // the bean as it loads into container

                ConfigurableApplicationContext context
        = new ClassPathXmlApplicationContext("config.xml");

      // It will close the spring container
      // and as a result invokes the
      // destroy() method
      context.close();
        }

}
```

2. By Programmatic Approach: To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to implement our bean with two interfaces namely InitializingBean, DisposableBean and will have to override afterPropertiesSet() and destroy() method. afterPropertiesSet() method is invoked as the container starts and the bean is instantiated whereas, the destroy() method is invoked just after the container is closed.

Note: To invoke the destroy method we have to call a close() method of ConfigurableApplicationContext.

Therefore, the following steps are followed:
Step 1: Firstly, we need to create a bean HelloWorld.java in this case by implementing InitializingBean, DisposableBean, and overriding afterPropertiesSet() and destroy() method.

```
package bean_lifecycle;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class HelloWorld implements InitializingBean, DisposableBean {

	// This method is invoked
	// just after the container
	// is closed
	@Override
	public void destroy() throws Exception {
		System.out.println("Container has been closed " + "and I'm the destroy()
method");

	}

	// It is the init() method
	// of our bean and it gets
	// invoked on bean instantiation
	@Override
	public void afterPropertiesSet() throws Exception {
		System.out.println("Bean HelloWorld has been " + "instantiated and I'm
the " + "init() method");

	}

}
```

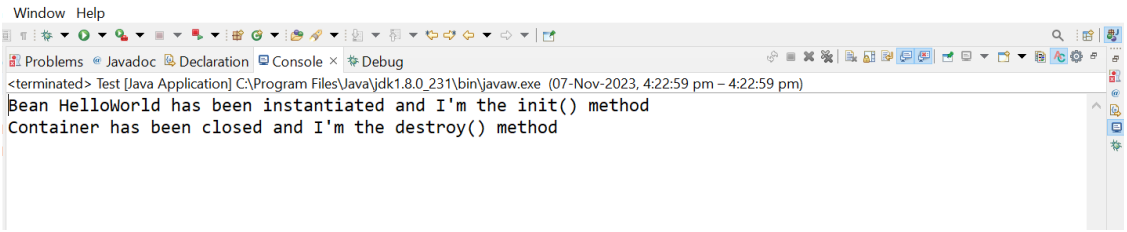
Step 2: Now, we need to configure the spring XML file config.xml and define the bean.

```
<?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-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">

 <bean id="hw" class="bean_lifecycle.HelloWorld"/>
</beans>
```

Step 3: Finally, we need to create a driver class to run this bean.

```
package bean_lifecycle;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test {

        public static void main(String[] args) {
             // Loading the Spring XML configuration
      // file into the spring container and
      // it will create the instance of the bean
      // as it loads into container
      ConfigurableApplicationContext context
          = new ClassPathXmlApplicationContext(
              "config.xml");

      // It will close the spring container
      // and as a result invokes the
      // destroy() method
      context.close();
        }

}
```

Output:



3. Using Annotation:

To provide the facility to the created bean to invoke custom init() method on the startup of a spring container and to invoke the custom destroy() method on closing the container, we need to annotate init() method by @PostConstruct annotation and destroy() method by @PreDestroy annotation.

Note: To invoke the destroy() method we have to call the close() method of ConfigurableApplicationContext.

Question: Can a bean be both initialized and destroyed?

II. Bean scope

When you create a bean definition what you are actually creating is a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to choose the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware ApplicationContext).

The Spring Framework supports the following five scopes, three of which are available only if you use a web-aware ApplicationContext.

- **Singleton:** This scopes the bean definition to a single instance per Spring IoC container (default).

- **Prototype:** This scopes a single bean definition to have any number of object instances.

- **Request:** This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

- **Session: T**his scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

- **Global-session:** This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

The singleton scope

If a scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

The default scope is always singleton. However, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file, as shown in the following code snippet −

```
<!-- A bean definition with singleton scope -->
<bean id = "..." class = "..." scope = "singleton">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

Let's take a look at the example:

Here is the content of **HelloWorld.java** file −

```
package bean_scope;

public class HelloWorld {
        private String message;

        public String getMessage() {
                return message;
        }
```

```
	public void setMessage(String message) {
		this.message = message;
	}


}
```

Following is the content of the MainApp.java file −

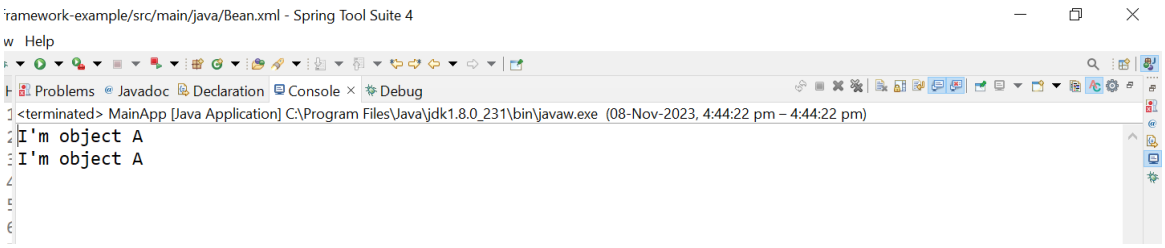
```
package bean_scope;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("Bean.xml");
		HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

		objA.setMessage("I'm object A");
		System.out.println(objA.getMessage());

		HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
		System.out.println(objB.getMessage());
	}

}
```

Following is the configuration file **Beans.xml** required for singleton scope:

```
<?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-3.0.xsd
```

```
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<bean id = "helloWorld" class = "bean_scope.HelloWorld" scope = "singleton">
</bean>
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −



The prototype scope

If the scope is set to prototype, the Spring IoC container creates a new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the **scope** property to **prototype** in the bean configuration file, as shown in the following code snippet −

```
<!-- A bean definition with prototype scope -->
<bean id = "..." class = "..." scope = "prototype">
   <!-- collaborators and configuration for this bean go here -->
</bean>
```

Let's take a look at the example:
Here is the content of **HelloWorld.java** file −

```
package bean_scope;

public class HelloWorld {
   private String message;

   public String getMessage() {
      return message;
   }
```

```
	public void setMessage(String message) {
		this.message = message;
	}


}
```

Following is the content of the MainApp.java file −

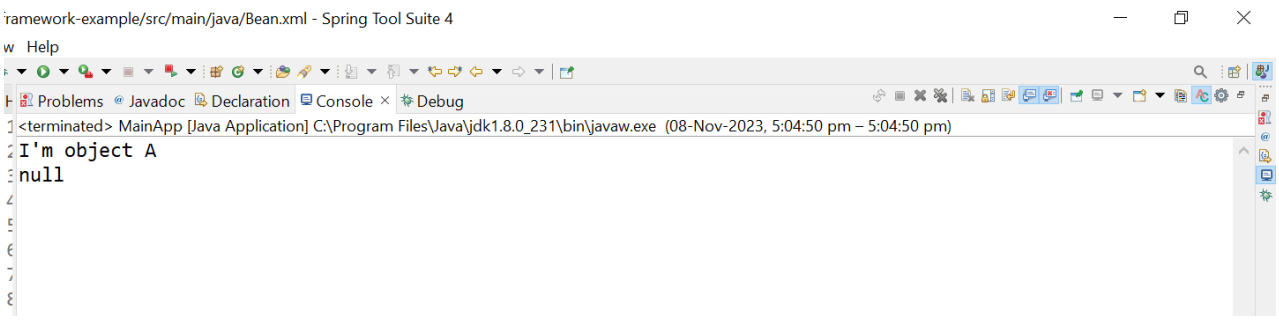
```
package bean_scope;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {

	public static void main(String[] args) {
		ApplicationContext context = new
ClassPathXmlApplicationContext("Bean.xml");
		HelloWorld objA = (HelloWorld) context.getBean("helloWorld");

		objA.setMessage("I'm object A");
		System.out.println(objA.getMessage());

		HelloWorld objB = (HelloWorld) context.getBean("helloWorld");
		System.out.println(objB.getMessage());
	}

}
```

Following is the configuration file **Beans.xml** required for singleton scope:

```
<?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-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<bean id = "helloWorld" class = "bean_scope.HelloWorld" scope = "prototype">
</bean>
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message −



Knowledge check…

Now that we've covered the basic understanding of Bean lifecycle and Bean Scope, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1. **What is the first phase in the Spring bean lifecycle?**

A) Construction
B) Initialization
C) Destruction
D) Population

2. **In Spring, what is the default scope of a bean if no scope is specified explicitly?**

A) Prototype

B) Singleton
C) Request
D) Session

3. **Which of the following scopes is typically used in a web-based Spring application?**

A) Singleton
B) Prototype
C) Request
D) Session

At the end of the quiz,the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.

III. Injecting Bean object using XML configuration

Injecting Objects by Setter Injection

Let's learn how we are going to use Spring to inject our dependencies into our object values by Setter Injection. Object is a basic unit of Object-Oriented Programming and represents real-life entities. So generally in Java, we create objects of a class using the new keyword.

```
Test t = new Test();
// creating object of class Test
// t is the object
```

Implementation: Suppose we have a class named “Score” and inside the class, we want to inject the object of another class named “Student” and “Course”.

Student Bean:

```
package com.demo;

import java.util.Date;
```

```
public class Student {

	private String studentId;
	private String firstName;
	private String lastName;
	private Date dateOfBirth;
	private String gender;
	private String email;
	private String phone;

//defining Setter and Getter Method

	public String getStudentId() {
		return studentId;
	}

	public void setStudentId(String studentId) {
		this.studentId = studentId;
	}

	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public Date getDateOfBirth() {
		return dateOfBirth;
	}
```

```
	public void setDateOfBirth(Date dateOfBirth) {
		this.dateOfBirth = dateOfBirth;
	}

	public String getGender() {
		return gender;
	}

	public void setGender(String gender) {
		this.gender = gender;
	}

	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	public String getPhone() {
		return phone;
	}

	public void setPhone(String phone) {
		this.phone = phone;
	}

	//defining toString()
		@Override
		public String toString() {
			return "Student [studentId=" + studentId + ", firstName=" +
firstName + ", lastName=" + lastName
					+ ", dateOfBirth=" + dateOfBirth + ", gender=" +
gender + ", email=" + email + ", phone=" + phone + "]";
		}

}
```

Course Bean:

```
package com.demo;

public class Course {

	private int courseId;
	private String courseTitle;
	private int credits;

	public int getCourseId() {
		return courseId;
	}

//defining Setter and Getter method
	public void setCourseId(int courseId) {
		this.courseId = courseId;
	}

	public String getCourseTitle() {
		return courseTitle;
	}

	public void setCourseTitle(String courseTitle) {
		this.courseTitle = courseTitle;
	}

	public int getCredits() {
		return credits;
	}

	public void setCredits(int credits) {
		this.credits = credits;
	}

//definning toString() method
	@Override
	public String toString() {
```

```
		return "Course [courseId=" + courseId + ", courseTitle=" + courseTitle + ",
credits=" + credits + "]";
	}

}
```

Now let's create a Score class and inject Student and Course bean into it using Setter Injection.

Score Bean:

```
package com.demo;

import java.util.Date;

public class Score {

	private int scoreId;
	private String creaditObtained;
	private Date dateOfExam;
	private Student student;
	private Course course;

	public int getScoreId() {
		return scoreId;
	}

	public void setScoreId(int scoreId) {
		this.scoreId = scoreId;
	}

	public String getCreaditObtained() {
		return creaditObtained;
	}

	public void setCreaditObtained(String creaditObtained) {
		this.creaditObtained = creaditObtained;
	}

	public Date getDateOfExam() {
```

```
		return dateOfExam;
	}

	public void setDateOfExam(Date dateOfExam) {
		this.dateOfExam = dateOfExam;
	}

	public Student getStudent() {
		return student;
	}

	public void setStudent(Student student) {
		this.student = student;
	}

	public Course getCourse() {
		return course;
	}

	public void setCourse(Course course) {
		this.course = course;
	}

	@Override
	public String toString() {
		return "Score [scoreId=" + scoreId + ", creaditObtained=" +
creaditObtained + ", dateOfExam=" + dateOfExam
				+ ", student=" + student + ", course=" + course + "]";
	}

}
```

Now let's create a Score Bean in the beans.xml file and inside the bean, you have to add your property's name and its corresponding values inside the <property> tag, But here we are injecting objects, not literal values.Let's have a look:

```
<?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"
```

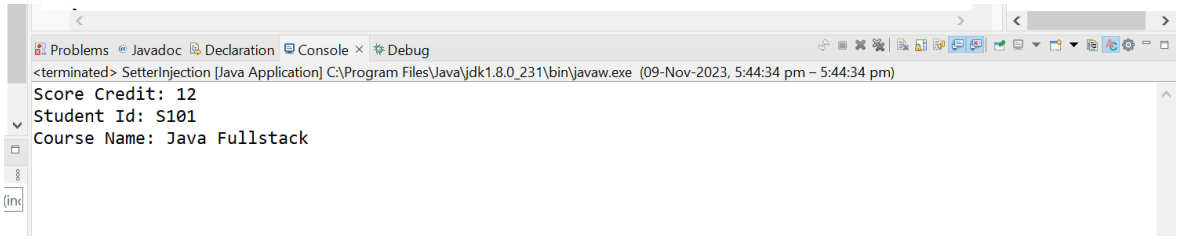
```
 xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- First create Student Bean -->
 <!-- Bean Configuration -->
<bean id="dateFormater" class="java.text.SimpleDateFormat">
    <constructor-arg value="dd-MM-yyyy" />
 </bean>

<bean class="com.demo.Student" name="student1">

<property name="studentId" value="S101"/>
<property name="firstName" value="John"/>
<property name="lastName" value="Doe"/>

  <property name="dateOfBirth">
       <bean factory-bean="dateFormater" factory-method="parse">
        <constructor-arg value="1996-08-12" />
       </bean>
    </property>
<property name="gender" value="M"/>
<property name="email" value="john@gamil.com"/>
<property name="phone" value="8890678456"/>
</bean>

 <!-- Second create Course Bean -->
  <bean class="com.demo.Course" name="course1">
<property name="courseId" value="111"/>
<property name="courseTitle" value="Java Fullstack"/>
 <property name="credits" value="11"/>
</bean>

<!-- Now Configure Score Bean -->

<bean class="com.demo.Score" name="score1">
<property name="scoreId" value="1001"/>
<property name="creaditObtained" value="12"/>
 <property name="dateOfExam">
```

```
        <bean factory-bean="dateFormater" factory-method="parse">
         <constructor-arg value="2023-08-12" />
        </bean>
        </property>

        <!-- Here we are injecting Student and Course Bean in Score Bean -->
        <property name="student" ref="student1"/>
        <property name="course" ref="course1"/>

  </bean>
  </beans>
```

So for testing this stuff let's create a main method and call the methods inside the Score class. Below is the code for the Main.java file.

```
package com.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SetterInjection {

        public static void main(String[] args) {
                // Load the Spring configuration
        ApplicationContext context=new
ClassPathXmlApplicationContext("score-bean.xml");

        // Get the Score bean from the Spring container
        Score score=(Score)context.getBean("score1");

        // Display the Score Credit,Student id and Course title
        System.out.println("Score Credit: "+score.getCreaditObtained());
        System.out.println("Student Id: "+score.getStudent().getStudentId());
        System.out.println("Course name: "+score.getCourse().getCourseTitle());
        }

}
```

Output:



Food for Explore..

The trainer can ask the students to explore How can we perform constructor injection to inject one bean object into another bean in Spring?

IV. @Autowire @Component and @Bean Annotation

In the Spring Framework, @Autowired, @Component, and @Bean are annotations commonly used for dependency injection and bean management. Let's go through each of them with examples:]

1. @Autowired:

@Autowired is used for automatic dependency injection. It can be applied to fields, constructors, or methods.

```
package com.demo;

import org.springframework.beans.factory.annotation.Autowired;

public class Car {
	private Engine engine;

	@Autowired
	public Car(Engine engine) {
		this.engine = engine;
	}
	// Other methods and properties

	public Engine getEngine() {
		return engine;
	}
```

```
	public void setEngine(Engine engine) {
		this.engine = engine;
	}

}
```

In this example, the Car class has a dependency on the Engine class, and Spring will automatically inject an instance of Engine when creating a Car object.

2. @Component:
@Component is used to declare a class as a Spring bean. Spring will automatically detect and register the bean during the component scanning process.

```
package com.demo;

import org.springframework.stereotype.Component;

@Component
public class Engine {

	private String engineType;

	public String getEngineType() {
		return engineType;
	}

	public void setEngineType(String engineType) {
		this.engineType = engineType;
	}


}
```

In this example, the Engine class is annotated with @Component, and Spring will automatically register it as a bean.

3. @Bean:

@Bean is used to define a bean explicitly in a Java configuration class. It is commonly used with @Configuration annotated classes.

The @Configuration annotation in Spring is used to indicate that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

When you annotate a class with @Configuration, it essentially signals to Spring that this class should be considered as a source of bean definitions. This is often used in conjunction with @Bean annotated methods, where each method serves as a bean definition. These methods may contain the logic to instantiate, configure, and return a bean.

```
package com.demo;

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

@Configuration
public class AppConfig {

        @Bean
    public Engine engine() {
        return new Engine();
    }

    @Bean
    public Car car() {
        return new Car(engine());
    }
}
```

In this example, the AppConfig class is annotated with @Configuration, and the engine() and car() methods are annotated with @Bean. These methods define and configure the beans explicitly.

Remember, to enable component scanning and annotation-based configuration, you need to add the following to your Spring configuration:

@Configuration

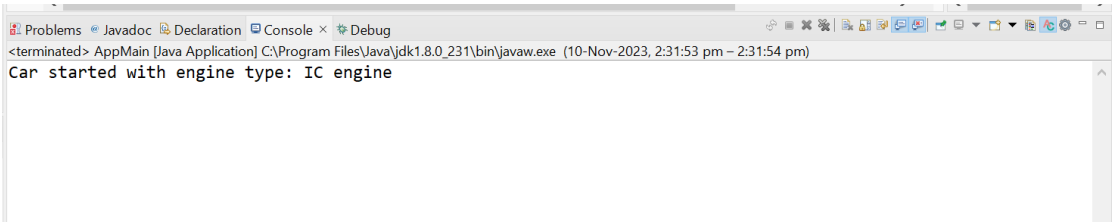
```
@ComponentScan(basePackages = "com.demo")
public class AppConfig {
}
```

Here, basePackages should be the package where your components are located. These annotations and configurations help Spring manage dependencies and beans effectively.

Here's an example App.java file that demonstrates how to use the beans defined in the AppConfig class:

```
package com.demo;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AppMain {

	public static void main(String[] args) {
		// Load the Spring configuration class
	AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

	// Retrieve beans from the Spring container
	Engine engine = context.getBean(Engine.class);
	Car car = context.getBean(Car.class);

	//set the Engine Type
	engine.setEngineType("IC engine");

	// Use the beans
	System.out.println("Car started with engine type: " +
car.getEngine().getEngineType());

	// Close the context
	context.close();
	}

}
```

Output:



In this example:

- The AnnotationConfigApplicationContext is used to load the Spring configuration from the AppConfig class.
- Beans (Engine and Car) are retrieved from the Spring container using getBean().
- The retrieved beans are then used in the application logic. In this case, it prints out the type of engine the car has.
- Finally, the AnnotationConfigApplicationContext is closed to release resources.

This MainApp.java file is a simple entry point for your Spring application that utilizes the beans configured in the AppConfig class.

Exercise:

Use ChatGPT to explore implementing the life cycle of a bean using Annotation:

Put the below problem statement in the message box and see what ChatGPT says.

I am currently studying the lifecycle of Spring beans and would like to delve into implementing bean lifecycle using annotations. Could you provide a tutorial along with code snippets on this topic?