Setter-based & Constructor-based Dependency Injection (using XML Configuration)

Setter-based DI is accomplished by the **container** calling **setter** methods on your beans after invoking a no-argument constructor.

A) Literal Dependency Injection

Literal dependency injection involves injecting simple, literal values such as strings, numbers, or boolean values into a bean.

Let's create a `Student` class with `firstName` and `lastName` properties and then configure dependency injection using XML with setter-based injection.

Step 1: Define the Student class

```
package com.itsc.ioc.springioc;

public class Student {
	private String firstName;
	private String lastName;

	//setter & getter
	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}
	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	public String getFullName() {
		return firstName + " " + lastName;
	}

}
```

Step 2: Configure Dependency Injection using XML

beansConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ….>
```

```
    <!-- bean configuration goes here -->
    <bean id="student" class="com.itsc.ioc.springioc.Student">
        <property name="firstName" value="John" />
        <property name="lastName" value="Doe" />
    </bean>

</beans>
```

In this XML configuration:

- We define a bean named "student" of type `Student`.
- We use `<property>` tags to set the values of the `firstName` and `lastName` properties.

Step 3: Retrieve and Use the Bean

```
package com.itsc.ioc.springioc;

public class Main {
    public static void main(String[] args) {
        //load the Spring XML configuration
        ApplicationContext context = new
            ClassPathXmlApplicationContext("beansConfig.xml");

        // retrieve the Student bean from the context
        Student student = context.getBean("student",
                                          Student.class);

        // use the student objecct
        Sysout("Full Name: " + student.getFullName());
    }
}
```

Explanation:

- **XML Configuration**: In the XML file, we define a bean named "student" of type `Student`. We set the `firstName` and `lastName` properties using `<property>` tags.

- **Setter Injection**: The `Student` class defines setter methods (`setFirstName` and `setLastName`) for setting the `firstName` and `lastName` properties. Spring uses these setter methods to inject the dependencies.
- **Retrieving Bean**: In the main class, we load the Spring XML configuration using `ClassPathXmlApplicationContext` and retrieve the `Student` bean from the context using its ID ("student").
- **Using the Bean**: We use the `student` object to retrieve the full name of the student and print it.

Here, we have two private fields `firstName` and `lastName`, along with their respective setter methods (`setFirstName` and `setLastName`).

When Spring creates an instance of the `Student` class and injects the dependencies, it does so by calling these setter methods and passing the values specified in the XML configuration file.

For each `<property>` element, Spring identifies the corresponding setter method (`setFirstName` for `firstName` and `setLastName` for `lastName`). It then invokes these setter methods with the values specified in the configuration file (`John` and `Doe`, respectively).

B) Object Dependency Injection

Object dependency injection involves injecting other beans or objects into a bean. This allows for collaboration between different components of the application.

Let's modify the example to demonstrate object injection instead of literal value injection. We'll introduce a new class `Address` and inject it into the `Student` class.

Step 1: Define the Address class

```
package com.itsc.ioc.springioc;

public class Address {
     private String city;
```

```
	private String country;


	//getter & setter methods here methods
}
```

Step 2: Modify the Student class to accept Address object

```
package com.itsc.ioc.springioc;

public class Student {
	private String firstName;
	private String lastName;
	private Address address;

	//setters here for all

	public void setAddress(Address address) {
		this.address = address;
	}

	public String getFullName() {
		return firstName + " " + lastName;
	}

	public void getDetails() {
		printf("%s lives in %s city in %s", getFullName(),
				address.getCity(), address.getCountry());
	}


}
```

Step 3: Update the XML Configuration for Object Injection

```
<!-- bean configuration goes here -->
<bean id = "stdAddr" class = "com.itsc.ioc.springioc.Address">
	<property name="city" value="Addis" />
	<property name="country" value="Ethiopia" />
</bean>
<bean id="student" class="com.itsc.ioc.springioc.Student">
	<property name="firstName" value="John" />
	<property name="lastName" value="Doe" />
```

```
     <property name="address" ref=" stdAddr " />
</bean>


public class Main {
     public static void main(String[] args) {
          //load the Spring XML configuration
          ApplicationContext context = new
ClassPathXmlApplicationContext("beansConfig.xml");

          // retrieve the Student bean from the context
          Student student = context.getBean("student",
Student.class);

          // use the student objecct
          student.getDetails();
     }
}
```

In Spring XML configuration, the `ref` attribute is used to reference other beans defined within the same context file. It allows you to inject dependencies by specifying the ID of another bean that should be used as the value for the property.

In the context of our example, the `ref` attribute is used to inject an instance of the `Address` bean into the `Student` bean.

By using **ref="studentAddress"**, we're instructing Spring to inject the **Address** object defined with the ID **"studentAddress"** into the **address** property of the **Student** bean. This establishes a dependency relationship between the **Student** and **Address** beans, where the **Student** bean relies on the **Address** bean to provide address information.

C) Collections Dependency Injection

Let's modify the example to demonstrate how to inject collections using setter-based injection. We'll introduce a collection of subjects for each student.

Step 1: Define the Subject class

```
package com.itsc.ioc.springioc;

public class Subject {
```

```
	private String subjectName;

	public String getSubjectName() {
		return subjectName;
	}

	public void setSubjectName(String subjectName) {
		this.subjectName = subjectName;
	}

}
```

Step 2: Modify the Student class to accept a collection of subjects

```
package com.itsc.ioc.springioc;

import java.util.List;

public class Student {
	private String firstName,lastName;
	private List<Subject> subjects; // collection of subjects

	//setter & getter


	public void setSubjects(List<Subject> subjects) {
		this.subjects = subjects;
	}

	public List<Subject> getSubjects() {
		return subjects;
	}

	public String getFullName() {
		return firstName + " " + lastName;
	}

	public void getDetails() {
		System.out.println("Name: " + getFullName());
		System.out.println("Subjects: ");
		for (Subject subject : subjects) {
			System.out.println("- " +
subject.getSubjectName());
		}
```

```
    }
}
```

Step 2: Update the XML Configuration for Collection Injection

```
<!-- bean configuration goes here -->
<bean id="math" class="com.itsc.ioc.springioc.Subject">
     <property name="subjectName" value="Maths"></property>
</bean>
<bean id="history" class="com.itsc.ioc.springioc.Subject">
     <property name="subjectName" value="History"></property>
</bean>

<bean id="student" class="com.itsc.ioc.springioc.Student">
     <property name="firstName" value="John" />
     <property name="lastName" value="Doe" />
     <property name="subjects">
          <list>
               <ref bean="math"/>
               <ref bean="history"/>
          </list>
     </property>
</bean>
```

In Spring Framework XML configuration, the **`<list>`** element is used to define a list of elements that are injected into a property of a bean.

In the context of our example, we use the **`<list>`** element to define a list of **`Subject`** objects and inject them into the **`Student`** bean.

Here's a breakdown of the XML configuration:

```
<property name="subjects">
          <list>
               <ref bean="math"/>
               <ref bean="history"/>
          </list>
</property>
```

- **`<property name="subjects">`**: This specifies that we're injecting a property named **`subjects`** into the **`Student`** bean.

- `<list>`: This element defines a list of elements to be injected into the property. In our case, it's a list of `Subject` beans.
- `<ref bean="math"/>`, `<ref bean="science"/>`, `<ref bean="history"/>`: These elements reference other beans by their IDs (`math`, `science`, `history`). These beans are instances of the `Subject` class, and Spring injects them into the `Student` bean's `subjects` property as a list.

So, in summary, the `<list>` part of the XML configuration defines a list of beans to be injected into a property of another bean, providing a way to inject collections in Spring Framework XML configuration.

Constructor-based Injection

Example:

Student.java

```
package com.itsc.ioc.springioc;

import java.util.List;

public class Student {
	private String firstName;
	private String lastName;

	public Student(String firstName, String lastName) {
		this.firstName = firstName;
		this.lastName = lastName;
	}

	public String getFullName() {
		return firstName + " " + lastName;
	}
}
```

beansConfig.xml

```
<!-- bean configuration goes here -->
<bean id="student" class="com.itsc.ioc.springioc.Student">
	<constructor-arg name="firstName" value="John" />
	<constructor-arg name="lastName" value="Doe" />
</bean>
```

Main.java

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

// retrieve the Student bean from the context
Student student = context.getBean("student", Student.class);

// use the student objecct
System.out.println(student.getFullName());
```

When there are collections

Subject.java

```
package com.itsc.ioc.springioc;

public class Subject {
	private String subjectName;

	public Subject(String subjectName) {
		this.subjectName = subjectName;
	}

	public String getSubjectName() {
		return subjectName;
	}

}
```

Student.java

```
package com.itsc.ioc.springioc;

import java.util.List;

public class Student {
	private String firstName;
	private String lastName;
	private List<Subject> subjects; // collection of subjects

	public Student(String firstName, String lastName,
List<Subject> subjects) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.subjects = subjects;
	}

	public String getFullName() {
		return firstName + " " + lastName;
	}

	public List<Subject> getSubjects() {
		return subjects;
```

```
    }

    public void getDetails() {
        System.out.println("Name: " + getFullName());
        System.out.println("Subjects: ");
        for (Subject subject : subjects) {
            System.out.println("- " +
subject.getSubjectName());
        }
    }
}
```

beansConfig.xml

```
        <!-- bean configuration goes here -->
    <bean id="math" class="com.itsc.ioc.springioc.Subject">
        <constructor-arg value="Maths" />
    </bean>
    <bean id="history" class="com.itsc.ioc.springioc.Subject">
        <constructor-arg value="History" />
    </bean>
    <bean id="logic" class="com.itsc.ioc.springioc.Subject">
        <constructor-arg value="Logic" />
    </bean>

    <bean id="student" class="com.itsc.ioc.springioc.Student">
        <constructor-arg name="firstName" value="John" />
        <constructor-arg name="lastName" value="Doe" />
        <constructor-arg>
            <list>
                <ref bean="math"/>
                <ref bean="history"/>
                <ref bean="logic"/>
            </list>
        </constructor-arg>
    </bean>
```

Music Player Application

Create an interface called **`MusicPlayer`** that represents the functionality of a music player.

For simplicity, let's assume it has two methods: **`play()`** and **`stop()`**.

```
package com.itsc.ioc.springioc;

public interface MusicPlayerInterface {
	void play();
	void stop();
}
```

Create two classes that implement the **`MusicPlayer`** interface: **`CdPlayer`** & **`Mp3Player`**. These classes will represent different types of music players.

```
package com.itsc.ioc.springioc;

public class CdPlayer implements MusicPlayerInterface {

	@Override
	public void play() {
		System.out.println("CD player playing");
	}

	@Override
	public void stop() {
		System.out.println("CD player stopped");
	}
}
```

```java
package com.itsc.ioc.springioc;

public class Mp3Player implements MusicPlayerInterface {

	@Override
	public void play() {
		System.out.println("MP3 player playing");
	}

	@Override
	public void stop() {
		System.out.println("MP3 player stopped");
	}

}
```

Create an XML configuration file named **beansConfig.xml**. This file will define the beans and their dependencies.

```xml
<bean id="cdPlayer" class="com.itsc.ioc.springioc.CdPlayer" />
<bean id="mp3Player" class="com.itsc.ioc.springioc.Mp3Player" />
```

Create a **MainApp** class that will serve as the entry point of our application. In this class, we'll use Spring's **ApplicationContext** to load the XML configuration and retrieve the appropriate **MusicPlayerInterface** bean.

```java
package com.itsc.ioc.springioc;

public class MainApp {

	public static void main(String[] args) {
		ApplicationContext ctx = new CPXAC("beansConfig.xml");

		MPI cdPlayer = (MPI) ctx.getBean("cdPlayer");
		cdPlayer.play();
		cdPlayer.stop();
```

```
	}
}
```

Let's enhance our music player application by adding a new feature: the ability to switch between different output devices, such as speakers and headphones.

Define the OutputDevice Interface

Create a new interface called **`OutputDevice`** that represents the functionality of an output device.

For simplicity, let's assume it has a single method called **`outputSound()`**.

```
package com.itsc.ioc.springioc;

public interface OutputDeviceInterface {
	void outputSound();
}
```

Implement the OutputDevice Components

Create two classes that implement the `OutputDevice` interface: **`Speakers`** and **`Headphones`**.

These classes will represent different types of output devices.

```
package com.itsc.ioc.springioc;

public class Speakers implements OutputDeviceInterface {

	@Override
	public void outputSound() {
		System.out.println("Speakers used");
	}
}
```

```java
package com.itsc.ioc.springioc;

public class Headphones implements OutputDeviceInterface {

	@Override
	public void outputSound() {
		System.out.println("Headphones used");
	}
}
```

Inject the OutputDevice Dependency into MusicPlayer

Update the **`MusicPlayer`** interface to include a new method called **`setOutputDevice()`**.

This method will be used to inject the **`OutputDevice`** dependency into the music player.

```java
package com.itsc.ioc.springioc;

public interface MusicPlayerInterface {
	void play();
	void stop();
	void setOutputDevice(OutputDeviceInterface outputDevice);
}
```

Update the implementations of `CdPlayer` and `Mp3Player` to include the `setOutputDevice()` method.

```java
package com.itsc.ioc.springioc;

public class CdPlayer implements MusicPlayerInterface {
	private OutputDeviceInterface outputDevice;
	@Override
```

```
	public void play() {
		System.out.println("CD player playing");
		outputDevice.outputSound();
	}

	@Override
	public void stop() {
		System.out.println("CD player stopped");
	}

	@Override
	public void setOutputDevice(OutputDeviceInterface
outputDevice) {
		this.outputDevice = outputDevice;
	}

}
```

```
package com.itsc.ioc.springioc;

public class Mp3Player implements MusicPlayerInterface {
	private OutputDeviceInterface outputDevice;
	@Override
	public void play() {
		System.out.println("MP3 player playing");
		outputDevice.outputSound();
	}

	@Override
	public void stop() {
		System.out.println("MP3 player stopped");
	}

	@Override
	public void setOutputDevice(OutputDeviceInterface
outputDevice) {
		this.outputDevice = outputDevice;

	}

}
```

Update the XML Configuration for MusicPlayer Beans

Modify the XML configuration file to inject the appropriate **`OutputDevice`** dependency into the **`CdPlayer`** and **`Mp3Player`** beans.

```
<!-- bean configuration goes here -->
<bean id="speakers" class="com.itsc.ioc.springioc.Speakers" />

<bean id="cdPlayer" class="com.itsc.ioc.springioc.CdPlayer">
     <property name="outputDevice" ref="speakers"></property>
</bean>
```