Unit IV

Spring: Spring Core Basics-Spring Dependency Injection concepts, Introduction to Design patterns, Factory Design Pattern, Strategy Design pattern, Spring Inversion of Control, AOP, Bean Scopes- Singleton, Prototype, Request, Session, Application, WebSocket, Auto wiring, Annotations, Life Cycle Call backs, Bean Configuration styles.

Java applications (a loose term which runs the gamut from constrained applets to full-fledged n-tier server-side enterprise applications) typically are composed of several objects that collaborate with one another to form the application proper. The objects in an application can thus be said to have dependencies between themselves.

Prior to the advent of Enterprise Java Beans (EJB), Java developers needed to use JavaBeans to create Web applications. Although JavaBeans helped in the development of user interface (UI) components, they were not able to provide services, such as transaction management and security, which were required for developing robust and secure enterprise applications. The Spring framework(which is commonly known as Spring) has emerged as a solution to all these complications This framework uses various new techniques such as Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), and dependency injection (DI), to develop enterprise applications, thereby removing the complexities involved. This framework mainly focuses on providing various ways to help you manage your business objects. It made the development of Web applications much easier as compared to classic Java frameworks and Application Programming Interfaces (APIs), such as Java database connectivity(JDBC), JavaServer Pages(JSP), and Java Servlet.

What is Spring?

- It is a lightweight, loosely coupled and integrated framework for developing enterprise applications in java.
- Spring Framework is built on top of two design concepts – Dependency Injection and Aspect Oriented Programming.

Some of the features of spring framework are:

The features of the Spring framework such as IoC, AOP, and transaction management, make it unique among the list of frameworks.

- Lightweight and very little overhead of using framework for our development.
- Dependency Injection or Inversion of Control to write components that are independent of each other, spring container takes care of wiring them together to achieve our work.
- Spring IoC container manages Spring Bean life cycle and project specific configurations such as JNDI lookup.

- Spring MVC framework can be used to create web applications as well as restful web services capable of returning XML as well as JSON response.
- Support for transaction management, JDBC operations, File uploading, Exception Handling etc with very little configurations, either by using annotations or by spring bean configuration file.

What is IOC?
Inversion of control(IoC) is a programming technique in which *object coupling is bound at run time* by an assembler object and is typically not known at compile time using static analysis.

What is Dependency Injection?
Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by Spring, must adhere to the standard definition of Java-Bean. Dependency Injection in Spring also *ensures loose-coupling* between the classes.
Need for Dependency Injection:

Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be dependent on class Two. Now though it might appear okay to depend a module on the other but, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided.

Spring IOC resolves such dependencies with Dependency Injection, which makes the code easier to test and reuse. Loose coupling between classes can be possible by defining interfaces for common functionality and the injector will instantiate the objects of required implementation. The task of instantiating objects is done by the container according to the configurations specified by the developer.

Types of Spring Dependency Injection:
There are two types of Spring Dependency Injection. They are:

1. **Setter Dependency Injection (SDI)**: This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the **bean-configuration file** For this, the property to be set with the SDI is declared under the **<property>** tag in the bean-config file.

2. Constructor Dependency Injection (CDI): In this, the DI will be injected with the help of constructors. Now to set the DI as CDI in bean, it is done through the **bean-configuration file** For this, the property to be set with the CDI is declared under the **<constructor-arg>** tag in the bean-config file.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this −

```
public class TextEditor {

  public TextEditor() {
    Spellchecker spellChecker = new SpellChecker();
  }
}
```

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this −

```
public class TextEditor {
  private SpellChecker spellChecker;

  public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }
}
```

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus the flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

The second method of injecting dependency is through **Setter Methods** of the TextEditor class where we will create a SpellChecker instance. This instance will be used to call setter methods to initialize TextEditor's properties.

Note: *You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.*

Differentiating with dependency injection

Inversion of control is a design paradigm with the goal of giving more control to the targeted components of your application, the ones getting the work done. Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality. The two concepts work together in this way to allow for much more flexible, reusable, and encapsulated code to be written. As such, they are important concepts in designing object-oriented solutions.

Create Spring Maven Project:

Software required:

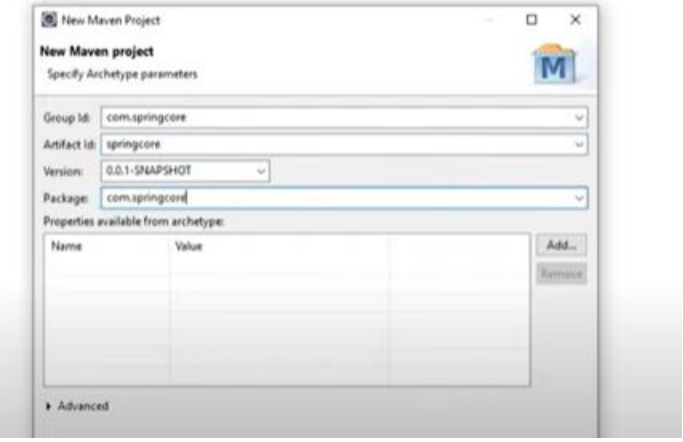
1. Eclipse/ Intellij/Netbeans

2. Tomcat Server

Steps to create project

1. **Create maven project**
2. **Adding dependencies -> Spring Core, Spring Context**
3. **Creating beans -> java POJO**
4. **Creating configuration file -> config.xml**
5. **Setting injection**
6. **Main class: which can pull the object and use**

Steps 1: in eclipse-> new-> create maven project->next-> window opens



& Finish

2. in pom.xml add dependencies

```
  <dependencies>
  <!-- https://mvnrepository.com/artifact/org.springframework/spring-core
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.2.3.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.3.RELEASE</version>
</dependency>

    <dependency>
```

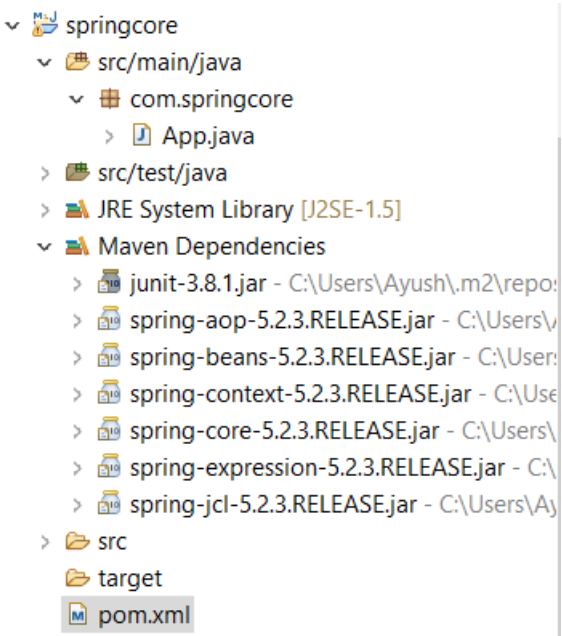
You find from here:

Spring context: https://mvnrepository.com/artifact/org.springframework/spring-context/5.2.3.RELEASE

Sprig core: https://mvnrepository.com/artifact/org.springframework/spring-core/5.2.3.RELEASE

I am using 5.2.3, version should be same for context & core. You can use any.

Now, Maven repository should look like this:



If error occurs, right click project-> Maven->update project->select your project->finish.

3. to create bean, create new class in your package name as Student.java

App.java | springcore/pom.xml | Student.java

```java
package com.springcore;

public class Student {

    private int studentId;
    private String studentName;
    private String studentAddress;
    //Constructors
    public Student() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Student(int studentId, String studentName, String studentAddress) {
        super();
        this.studentId = studentId;
        this.studentName = studentName;
        this.studentAddress = studentAddress;
    }


    //Getter & Setter Methods
    public int getStudentId() {
        return studentId;
    }
    public void setStudentId(int studentId) {
        this.studentId = studentId;
    }
    public String getStudentName() {
        return studentName;
    }
    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
    public String getStudentAddress() {
        return studentAddress;
    }
    public void setStudentAddress(String studentAddress) {
        this.studentAddress = studentAddress;
    }
    @Override
    public String toString() {
        return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentAddress="
                + "]";
    }

}
```

4. to call Student from app.java create config.xml

Reference spring-reference.pdf

App.java | springcore/pom.xml | Student.java | config.xml

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

<!-- This is our Bean -->
<bean name="student1" class="com.springcore.Student">
<!-- collaborators and configuration for this bean go here -->
<property name="studentId">
    <value> 22335</value>
</property>
<property name="studentName">
    <value> Ravi</value>
</property>
<property name="studentAddress">
    <value> Ghaziabad</value>
</property>
</bean>
</beans>
```

5. Add.java

App.java | springcore/pom.xml | Student.java | config.xml

```java
package com.springcore;

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

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
      ApplicationContext context=  new ClassPathXmlApplicationContext("config.xml");
      Student student1=(Student)context.getBean("student1");
      System.out.println( student1 );
    }
}
```

6. run

For Reference: https://github.com/Divya-021/Web-Technology-KCA-021/tree/main/Spring/springcore

Design Patterns in Java

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time. According to Gang of Four(GoF), authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Types of Design Patterns

there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

Design Pattern - Factory Pattern

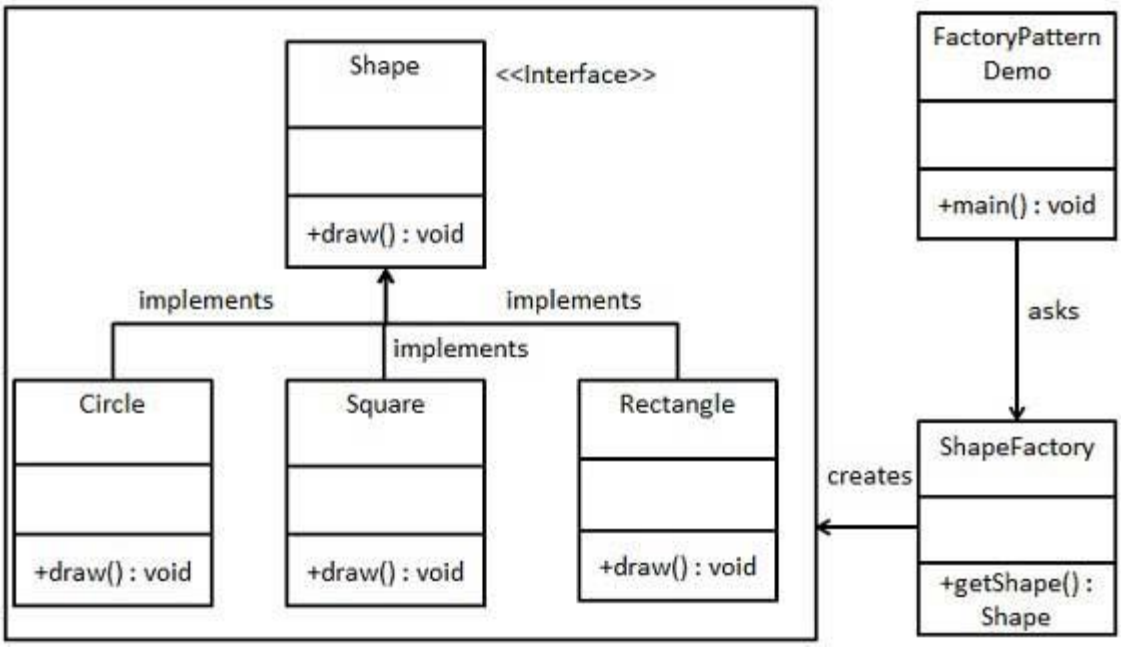
Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under **creational pattern** as this pattern provides one of the best ways to create an object.

In Factory pattern, **we create object without exposing the creation logic to the client** and refer to newly created object using a common interface.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Refer Program FactoryDesign

For Reference: https://github.com/Divya-021/Web-Technology-KCA-021/tree/main/Spring/FactoryDesign

Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
```

```
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

  public static void main(String[] args) {
    ShapeFactory shapeFactory = new ShapeFactory();

    //get an object of Circle and call its draw method.
    Shape shape1 = shapeFactory.getShape("CIRCLE");

    //call draw method of Circle
    shape1.draw();

    //get an object of Rectangle and call its draw method.
    Shape shape2 = shapeFactory.getShape("RECTANGLE");

    //call draw method of Rectangle
    shape2.draw();

    //get an object of Square and call its draw method.
    Shape shape3 = shapeFactory.getShape("SQUARE");

    //call draw method of square
    shape3.draw();
  }
}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Design Patterns - Strategy Pattern

In Strategy pattern, a class behavior or its **algorithm can be changed at run time**. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

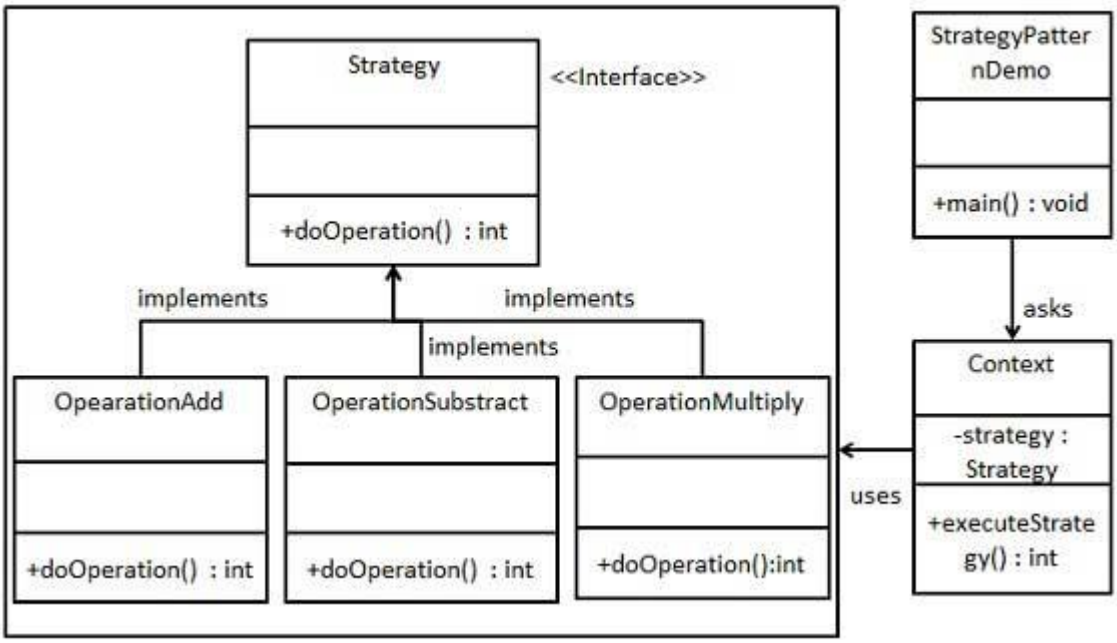
Refer Program StrategyDesignPattern

For Reference: https://github.com/Divya-021/Web-Technology-KCA-021/tree/main/Spring/StrategyDesignPattern

Implementation

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.

StrategyPatternDemo, our demo class, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.



Step 1

Create an interface.

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Step 2

Create concrete classes implementing the same interface.

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Step 3

Create *Context* Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Step 4

Use the *Context* to see change in behavior when it changes its *Strategy*.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
```

```
    System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
  }
}
```

Step 5

Verify the output.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Aspect Oriented Programming (AOP)

One of the key components of Spring is the **Aspect Oriented Programming (AOP)** framework. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects including logging, declarative transactions, security, caching, etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. DI helps you decouple your application objects from each other, while AOP helps you decouple cross-cutting concerns from the objects that they affect.

The AOP module of Spring Framework provides an aspect-oriented programming implementation allowing you to *define method-interceptors and pointcuts to cleanly decouple code* that implements functionality that should be separated.

```
class A{
public void a1(){...}
public void a2(){...}
public void a3(){...}
public void a4(){...}
public void a5(){...}
public void b1(){...}
public void b2(){...}
public void c1(){...}
public void c2(){...}
public void c3(){...}
}
```

You can see that there are 5 methods that start from a, 2 methods that start from b and 3 methods that starts from c.

First, let's understand Scenario- Here, I have to maintain a log and send notification after calling methods that start from m. So what is the problem without AOP? Here, We can call methods (that maintains a log and sends notification) from the methods starting with a. In such a scenario, we need to write the code in all the 5 methods. But, in case if a client says in future, I don't have to send a notification, you need to change all the methods. It leads to a maintenance problem. So with AOP, we have below solution.

The solution with AOP– With AOP, we don't have to call methods from the method. We can simply define the additional concern like maintaining a log, sending notification etc. in the method of a class. Its entry is given in the XML file. Suppose in future, if a client says to remove the notifier functionality, we need to change only in the XML file. So, maintenance is easy in AOP.

Spring - IoC Containers

The Spring container is at the core of the Spring Framework. The container will *create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.* The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans.

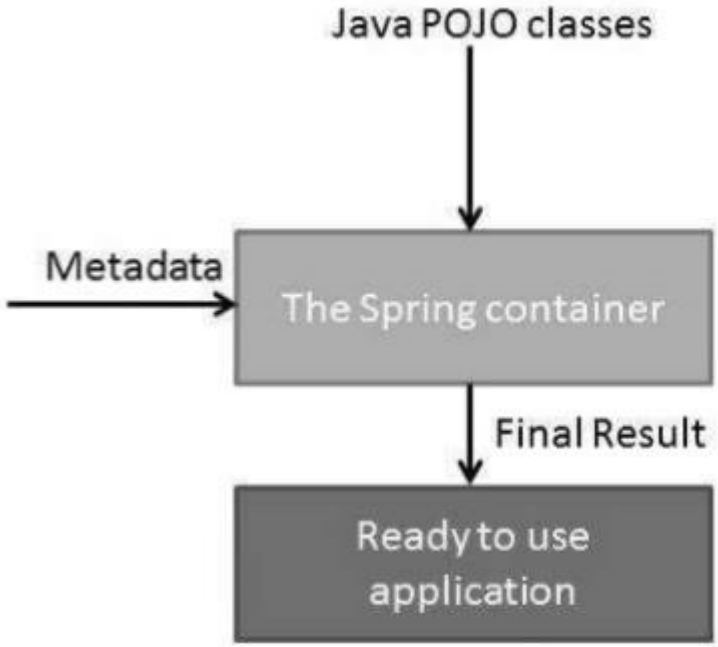
The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.

How do you provide configuration metadata to the Spring Container?

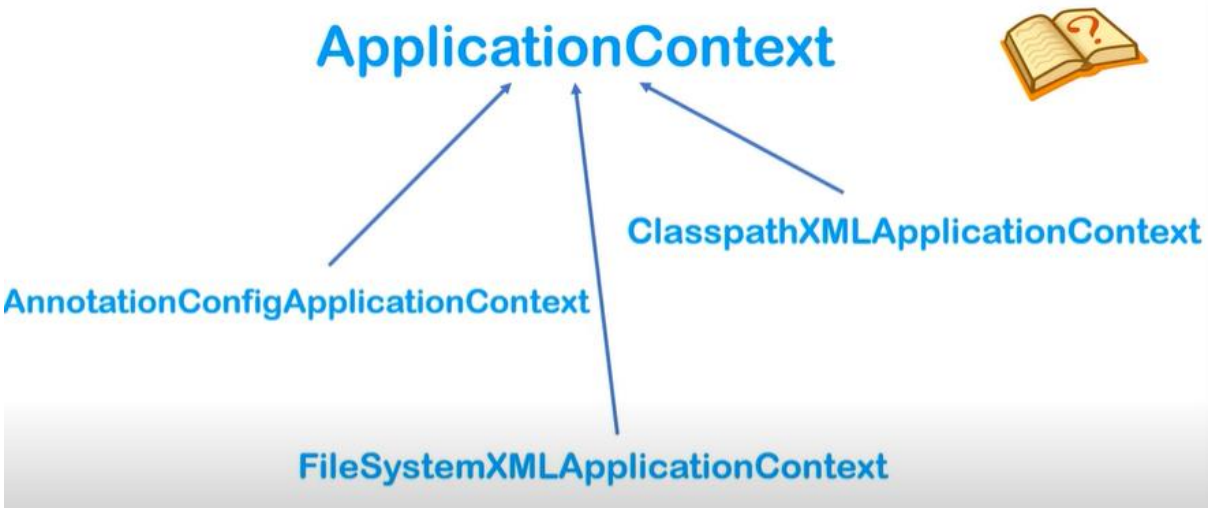
There are following three important methods to provide configuration metadata to the Spring Container:

- XML based configuration file.
- Annotation-based configuration
- Java-based configuration

The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



ApplicationContext is an interface represents an IoC Container implemented by these three classes.



What are Spring beans?

The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

What does a bean definition contain?

The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

What are different scopes of Spring Bean?
There are six scopes defined for Spring Beans.

1. singleton: Only one instance of the bean will be created for each container. This is the default scope for the spring beans. While using this scope, make sure spring bean doesn't have shared instance variables otherwise it might lead to data inconsistency issues because it's not thread safe.

2. prototype: A new instance will be created every time the bean is requested.

3. request: This is same as prototype scope; however it's meant to be used for web applications. A new instance of the bean will be created for each HTTP request.

4. session: A new bean will be created for each HTTP session by the container.

5. Application: It creates the bean instance for the lifecycle of a *ServletContext.* the same instance of the bean is shared across multiple servlet-based applications running in the same *ServletContext.*
6. Websocket: beans are stored in the *WebSocket* session attributes. The same instance of the bean is then returned whenever that bean is accessed during the entire *WebSocket* session.

Spring Framework is extendable, and we can create our own scopes too, however most of the times we are good with the scopes provided by the framework. To set spring bean scopes we can use "scope" attribute in bean element or @Scope annotation for annotation-based configurations.

Example: for XML based configuration

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

Example for Annotation based configuration

```
@Bean

@Scope("singleton")
```

```
public Person personSingleton() {
 return new Person();
}
```

Difference between Singleton and Prototype

Singleton	Prototype
Only one instance is created for a single bean definition per Spring IoC container	A new instance is created for a single bean definition every time a request is made for that bean.
Same object is shared for each request made for that bean. i.e. The same object is returned each time it is injected.	For each new request a new instance is created. i.e. A new object is created each time it is injected.
By default scope of a bean is singleton. So we don't need to declare a been as singleton explicitly.	By default scope is not prototype so you have to declare the scope of a been as prototype explicitly.
Singleton scope should be used for stateless beans.	While prototype scope is used for all beans that are stateful

Spring - Beans Auto-Wiring

You have learnt how to declare beans using the <bean> element and inject <bean> using <constructor-arg> and <property> elements in XML configuration file.

The Spring container can **autowire** relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

There are many autowiring modes:

No.	Mode	Description
1)	no bydefault.	It is the default autowiring mode. It means no autowiring

2)	byName	The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3)	byType	The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4)	constructor	The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.

Example:

```
<bean id="b" class="org.sssit.B"></bean>

<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

Spring - Annotation Based Configuration

Starting from Spring 2.5 it became possible to configure the dependency injection using **annotations**. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So consider the following configuration file in case you want to use any annotation in your Spring application.

```
<?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">

   <context:annotation-config/>
   <!-- bean definitions go here -->

</beans>
```

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us look at a few important annotations to understand how they work −

Sr.No.	Annotation & Description
1	@Required The @Required annotation applies to bean property setter methods.
2	@Autowired The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties.
3	@Qualifier The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
4	JSR-250 Annotations Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations.

Bean lifecycle and CallBack Methods in Spring

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.Spring lifetime callbacks

The Spring Framework provides several callback interfaces to change the behavior of your bean in the container; they include **InitializingBean and DisposableBean**.

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

To define setup and teardown for a bean, we simply declare the **<bean>** with **init-method** and/or **destroy-method** parameters. The **init-method** attribute specifies a method that is to be called on the bean immediately upon instantiation. Similarly, **destroy-method** specifies a method that is called just before a bean is removed from the container.

Initialization callbacks:

Implementing the org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The **InitializingBean interface** specifies exactly one method:

void afterPropertiesSet() throws Exception;

Generally, the use of the InitializingBean interface can be avoided and is discouraged since it unnecessarily couples the code to Spring. You have to use afterPropertiesSet(),you cannot change name of method. There is alternative for this i.e., XML-based configuration metadata. This is done using the 'init-method' attribute of <bean> tag. It provides flexibility of changing method name.

<bean id="countryBean" class="com.technicalstack.Country" init-method="init"/>

public class Country {
public void init () {
// do some initialization work
}
}

Destruction callbacks:

Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the container containing it is destroyed. The **DisposableBean interface** specifies a single method:

void destroy () throws Exception;

There is alternative for this i.e. XML-based configuration metadata. This is done using the 'destroy-method' attribute of <bean> tag.

<bean id=" countryBean" class= " com.technicalstack.Country" destroy-method= "destroy" />

public class Country {
public void destroy () {
// do some destruction work(like releasing pooled connections)
}
}

Example Code:

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

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;
```

```
   public void setMessage(String message){
      this.message = message;
   }
   public void getMessage(){
      System.out.println("Your Message: " + message);
   }
   public void init (){
      System.out.println("Bean is going through init.");
   }
   public void destroy () {
      System.out.println("Bean will destroy now.");
   }
}
```

Following is the content of the **MainApp.java** file:

```
package com.tutorialspoint;

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

public class MainApp {
   public static void main(String[] args) {
      AbstractApplicationContext context = new
ClassPathXmlApplicationContext("Beans.xml");

      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
      context.registerShutdownHook();
   }
}
```

Following is the configuration file **Beans.xml** required for init and destroy methods −

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

<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld" init-method = "init"
      destroy-method = "destroy">
      <property name = "message" value = "Hello World!"/>
   </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 −

```
Bean is going through init.
Your Message : Hello World!
Bean will destroy now.
```