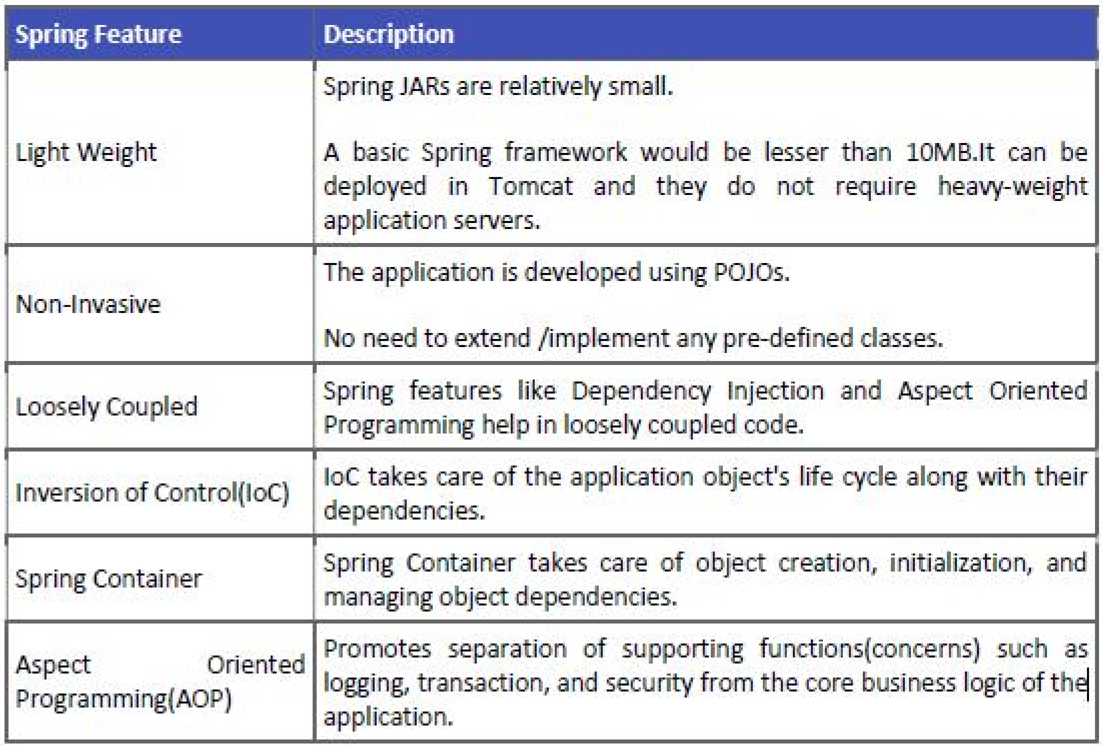
Spring 5 Basics

Spring framework helps in developing a loosely coupled application which is simple, easily testable, reusable and maintainable.

Inversion of control :

- Inversion of control(IoC) is the approach to handle application object life cycle by a third party such as a framework, there are different implementations for IoC and Spring Framework provides IoC implementation using Dependency Injection(DI).
- Spring's Dependency Injection features make an application loosely coupled by moving object dependencies away from the application to the Spring container. Now the container manages objects and its dependencies allowing the developer to focus more on application code.

Note: Inversion of Control (IoC) represents the inversion of application responsibility of the object's creation, initialization, dependency, and destruction from the application to the third party.

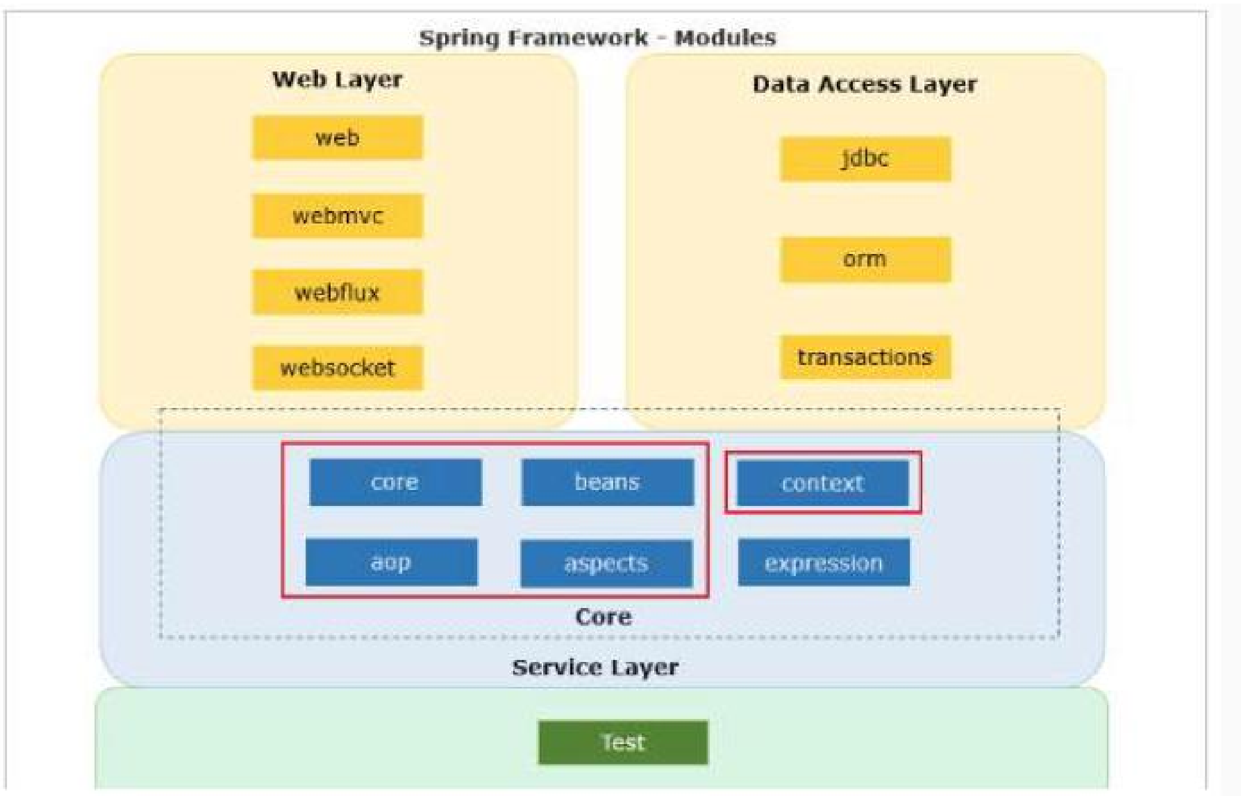


Spring Feature	Description
Light Weight	Spring JARs are relatively small. A basic Spring framework would be lesser than 10MB.It can be deployed in Tomcat and they do not require heavy-weight application servers.
Non-Invasive	The application is developed using POJOs. No need to extend /implement any pre-defined classes.
Loosely Coupled	Spring features like Dependency Injection and Aspect Oriented Programming help in loosely coupled code.
Inversion of Control(IoC)	IoC takes care of the application object's life cycle along with their dependencies.
Spring Container	Spring Container takes care of object creation, initialization, and managing object dependencies.
Aspect Oriented Programming(AOP)	Promotes separation of supporting functions(concerns) such as logging, transaction, and security from the core business logic of the application.

Spring Framework 5.x has the following key module groups:

- Core Container: These are core modules that provide key features of the Spring framework.
- Data Access/Integration: These modules support JDBC and ORM data access approaches in Spring applications.
- Web: These modules provide support to implement web applications.

- Others: Spring also provides few other modules such as the Test for testing Spring applications.



Core container has the following modules:

- Core: This is the key module of Spring Framework which provides fundamental support on which all other modules of the framework are dependent.
- Bean: This module provides a basic Spring container called BeanFactory.
- Context: This module provides one more Spring container called ApplicationContext which inherits the basic features of the BeanFactory container and also provides additional features to support enterprise application development.
- Spring Expression Language (SpEL): This module is used for querying/manipulating object value.
- AOP (Aspect Oriented Programming) and aspects: These modules help in isolating crosscutting functionality from business logic.

The following modules support Data Access/Integration:

- Java Database Connectivity (JDBC): It provides an abstract layer to support JDBC calls to relational databases.
- Object Relational Mapping (ORM): It provides integration support for popular ORM(Object-Relational Mapping) solutions such as Hibernate, JPA, etc.
- Transactions: It provides simple transaction API which abstracts the complexity of underlying repository specific transaction API's from the application.

Spring Framework Web module provides basic support for web application development. Web module has a web application context that is built on the application context of the core container. Web module provides complete Model-View-Controller(MVC) implementation to develop a presentation tier of the application and also support a simpler way to implement RESTful web services.

Spring Framework provides the following modules to support web application development:

- Web: This module has a container called web application context which inherits basic features from ApplicationContext container and adds features to develop web based applications.
- Webmvc: It provides the implementation of the MVC(model-view-controller) pattern to implement the server side presentation layer and also support features to implement RESTful Web Services.
- WebFlux: Spring 5.0 introduced a reactive stack with a web framework called Spring WebFlux to support Reactive programming in Spring's web layer and runs on containers such as Netty, Undertow, and Servlet 3.1+.
- WebSocket: It is used for 2 way communication between client and server in WebSocket based web applications.

Spring Framework has few additional modules, test module is one of the most commonly used one for testing Spring applications.

Test: This module provides required support to test Spring applications using TestNG or JUnit.

Spring Inversion of Control (IoC):

Spring container managed application objects are called beans in Spring. We need not create objects in dependency injection instead describe how objects should be created through configuration.

DI is a software design pattern that provides better software design to facilitate loose coupling, reuse and ease of testing.

Benefits of Dependency Injection(DI):

- Helps to create loosely coupled application architecture facilitating re-usability and easy testing.
- Separation of responsibility by keeping code and configuration separately. Hence dependencies can be easily modified using configuration without changing the code.
- Allows to replace actual objects with mock objects for testing, this improves testability by writing simple JUnit tests that use mock objects.

The core container module of Spring framework provide IoC using Dependency Injection.

The Spring container knows which objects to create and when to create through the additional details that we provide in our application called Configuration Metadata.

1. Application logic is provided through POJO classes.
2. Configuration metadata consists of bean definitions that the container must manage.

3. IoC container produces objects required by the application using POJO classes and configuration metadata. IoC container is of two types – BeanFactory and ApplicationContext.

Spring provides two types of containers

BeanFactory:

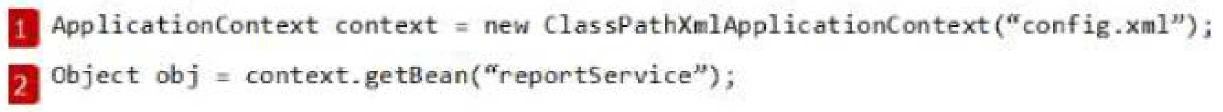
- It is the basic Spring container with features to instantiate, configure and manage the beans.
- org.springframework.beans.factory.BeanFactory is the main interface representing a BeanFactory container.

ApplicationContext:

- ApplicationContext is one more Spring container that is more commonly used in Spring applications.
- org.springframework.context.ApplicationContext is the main Interface representing an ApplicationContext container.
- It inherits the BeanFactory features and provides added features to support enterprise services such as internationalization, validation, etc.

ApplicationContext is the preferred container for Spring application development.

org.springframework.context.support.ClassPathXmlApplicationContext is one of the most commonly used implementation of ApplicationContext.

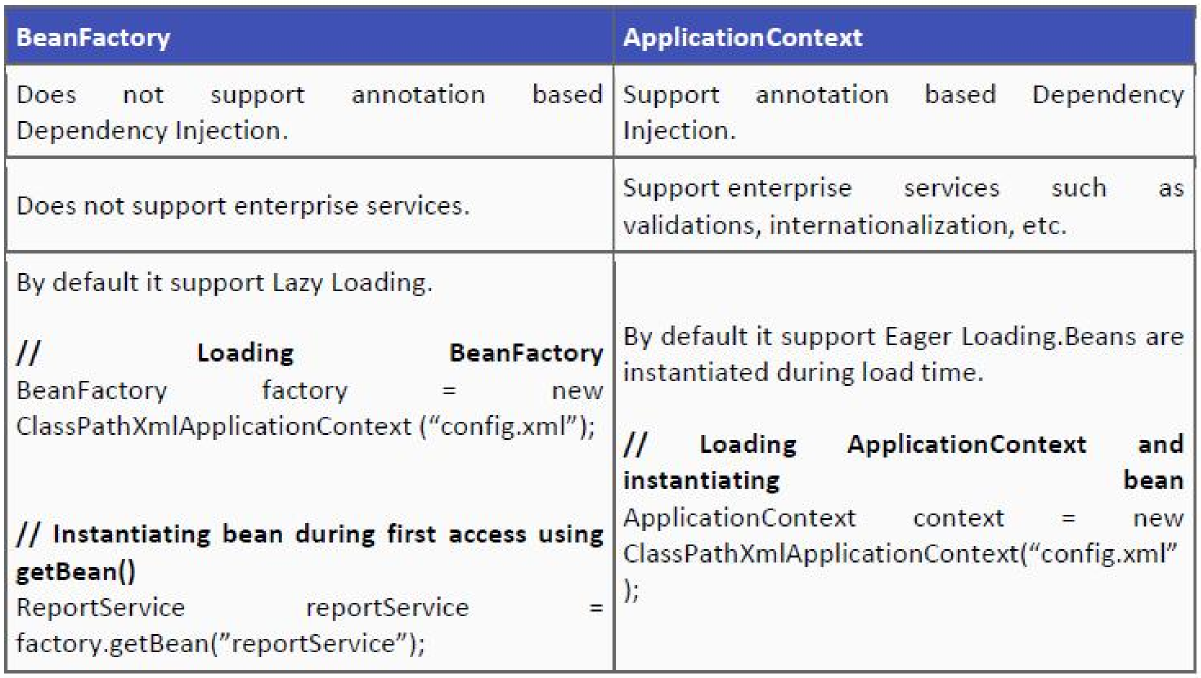


```
1 ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");
2 Object obj = context.getBean("reportService");
```

ApplicationContext container is instantiated by loading the configuration from the config.xml file available in CLASSPATH of the application.

Accessing the bean with id "reportService" from the container.

Let us now understand the differences between BeanFactory and ApplicationContext containers.



BeanFactory	ApplicationContext
Does not support annotation based Dependency Injection.	Support annotation based Dependency Injection.
Does not support enterprise services.	Support enterprise services such as validations, internationalization, etc.
By default it support Lazy Loading. **// Loading BeanFactory** BeanFactory factory = new ClassPathXmlApplicationContext (“config.xml”); **// Instantiating bean during first access using getBean()** ReportService reportService = factory.getBean(”reportService”);	By default it support Eager Loading.Beans are instantiated during load time. **// Loading ApplicationContext and instantiating bean** ApplicationContext context = new ClassPathXmlApplicationContext(“config.xml”);

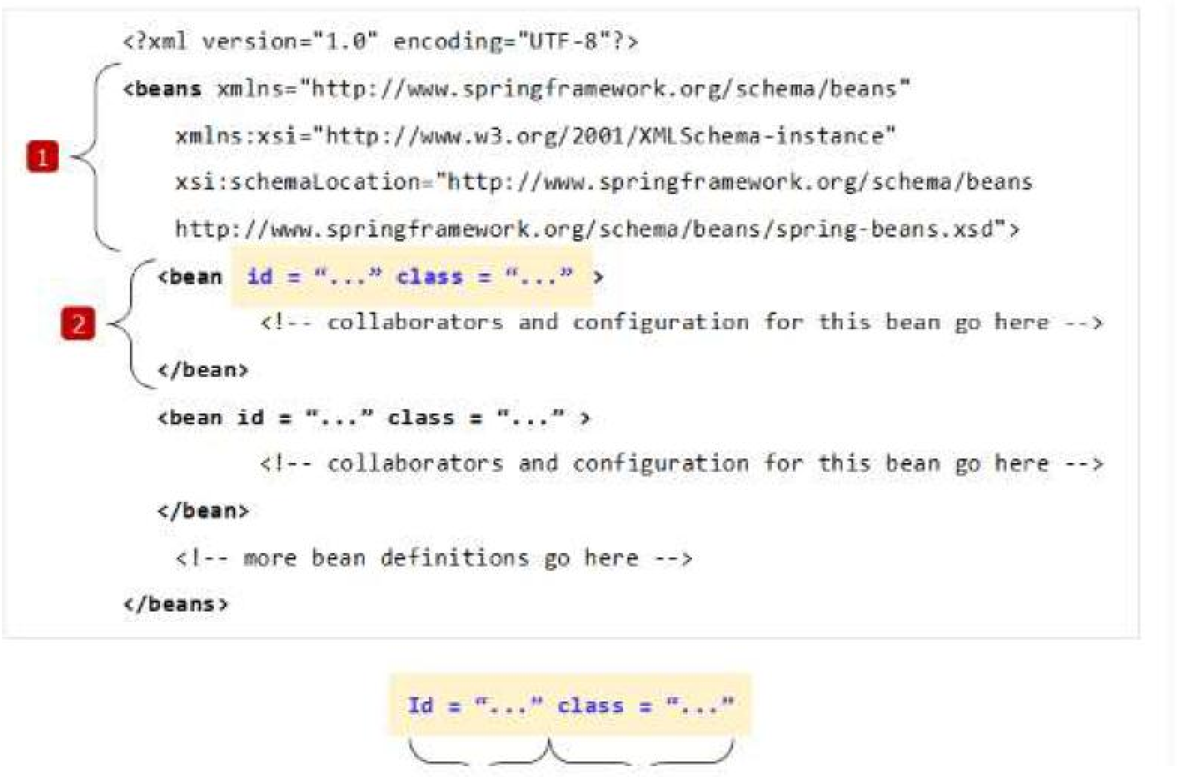
The Spring configuration metadata consists of definitions of the beans that the container must manage.

Spring allows providing the configuration metadata using :

- XML configuration
- Annotation based configuration
- Java based configuration

In XML configuration, the configuration metadata is provided as a simple XML file describing bean definitions that the Spring container has to manage.

Basic XML based configuration structure is as follows:


```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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean  id = "..." class = "..." >
          <!-- collaborators and configuration for this bean go here -->
   </bean>
   <bean id = "..." class = "..." >
          <!-- collaborators and configuration for this bean go here -->
   </bean>
    <!-- more bean definitions go here -->
</beans>
```

1. <beans> is the root element & also includes namespace declarations
2. Bean definition
3. id attribute represents a unique bean identifier
4. class attribute represents a fully qualified class name

Demo1 : Spring IoC

There are different ways to access bean in Spring

1. Traditional way of accessing bean based on bean id with explicit type cast

   ReportService reportService = (ReportService)context.getBean("reportService");

2. Accessing bean based on class type to avoid type cast if there is unique bean of type in the container

   ReportService reportService = context.getBean(ReportService.class);

3. Accessing bean through bean id and also type to avoid explicit type cast

```
ReportService service = (ReportService)
context.getBean("reportService2",ReportService.class);
```

Introduction to Dependency Injection:

How do we initialize bean with some specific values in Spring?

This can be achieved through Dependency Injection in Spring.

Inversion of Control pattern is achieved through Dependency Injection (DI) in Spring. In Dependency Injection, developer need not create the objects but specify how they should be created through configuration.

Spring container uses one of these two ways to initialize the properties:

- Constructor Injection: This is achieved when the container invokes parameterized constructor to initialize the properties of a class
- Setter Injection: This is achieved when the container invokes setter methods of a class to initialize the properties after invoking a default construct

What is mandatory for constructor injection?

- Parameterized constructor is required in the ReportService class as we are injecting the values through the constructor argument
- constructor-arg tag in the bean definition

Demo2 : Demo2_DependencyIngection_ConstructorInjection

Setter Injection

What is mandatory to implement setter injection?

- Default constructor and setter methods of respective dependent properties are required in the ReportService class. For setter injection, Spring internally uses the default constructor to create a bean and then invokes setter method of the respective property based on name attribute in order to initialize the values.
- property tag in the bean definition

Demo 3: Demo3_DependencyIngection_SetterInjection

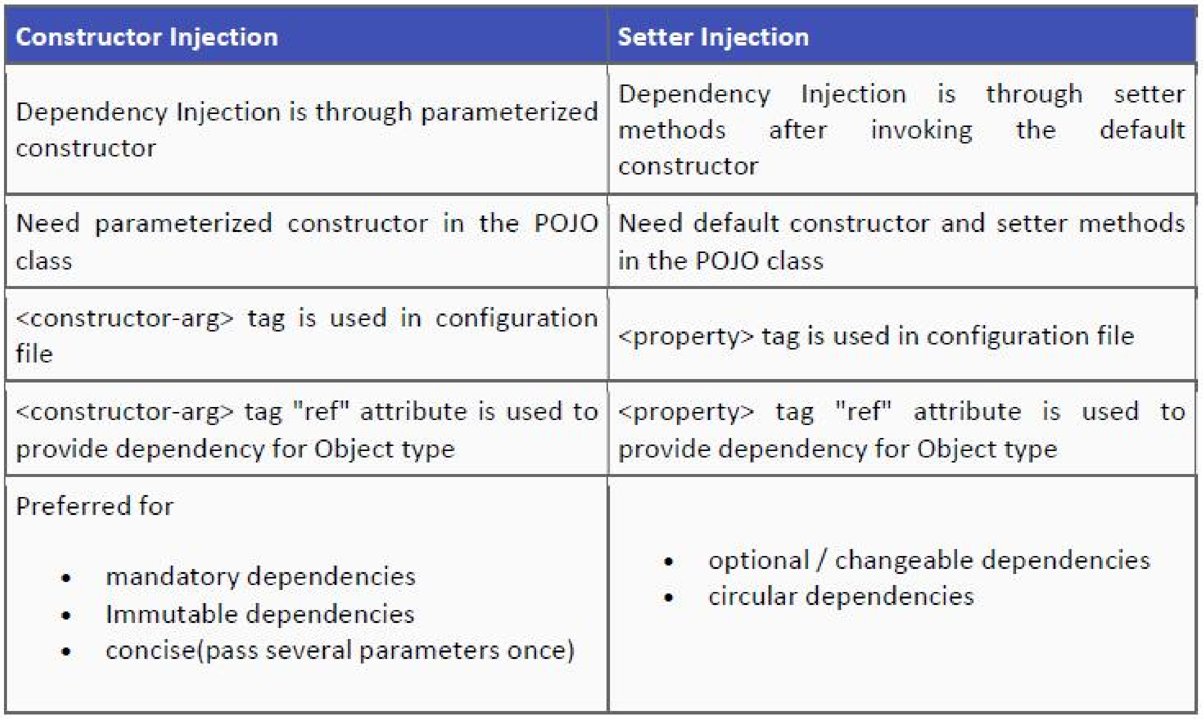
Setter Injection in collections:

Spring supports passing multiple values like Java Collection types such as List, Set, Map and Properties.

Spring provides four types of collection configuration elements as given below:

- <list> helps in injecting a list of values allowing duplicates
- <set> helps in injecting a set of values without any duplicates
- <map> helps in injecting a collection of key-value pairs where key and value can be of any type
- <props> helps in injecting a collection of key-value pairs where key and value are both
- Strings

Comparison Constructor Injection and Setter Injection :



Constructor Injection	Setter Injection
Dependency Injection is through parameterized constructor	Dependency Injection is through setter methods after invoking the default constructor
Need parameterized constructor in the POJO class	Need default constructor and setter methods in the POJO class
<constructor-arg> tag is used in configuration file	<property> tag is used in configuration file
<constructor-arg> tag "ref" attribute is used to provide dependency for Object type	<property> tag "ref" attribute is used to provide dependency for Object type
Preferred for • mandatory dependencies • Immutable dependencies • concise(pass several parameters once)	• optional / changeable dependencies • circular dependencies

Introduction to Autowiring:

It is not required to specify all the properties values while defining bean in the configuration.

For non-primitive properties we can use the Autowiring feature.

If Autowiring is enabled,Spring container automatically initializes the dependencies of that bean.

There is no need to provide explicit <property > or <constructor-arg> with ref tags in the bean definition for object dependencies.

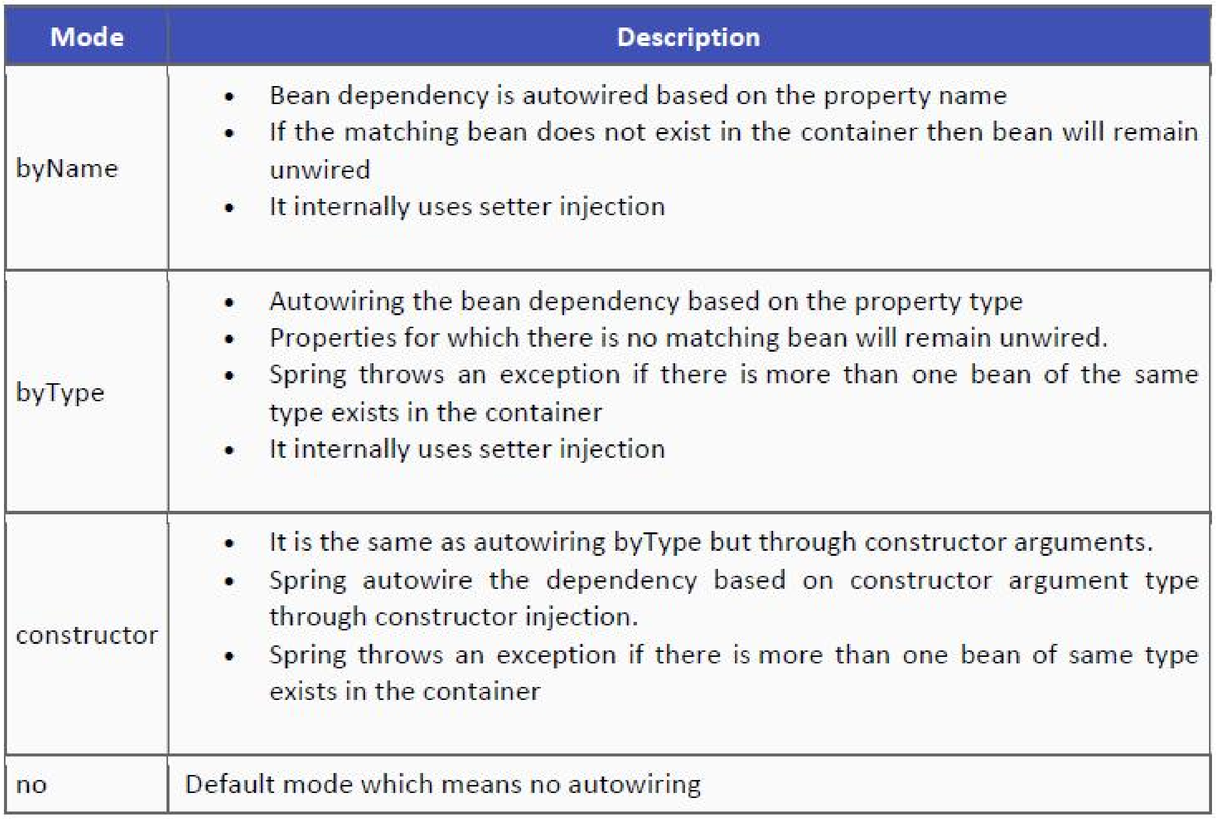
Autowiring cannot be used to inject primitive and string values.

Spring automatically inject the bean dependency. The autowiring has four modes.

- no (no autowiring)
- byName
- byType
- Constructor

In XML configuration, autowire mode is specified in the bean definition using the autowire attribute.

Autowiring modes:



Mode	Description
byName	• Bean dependency is autowired based on the property name • If the matching bean does not exist in the container then bean will remain unwired • It internally uses setter injection
byType	• Autowiring the bean dependency based on the property type • Properties for which there is no matching bean will remain unwired. • Spring throws an exception if there is more than one bean of the same type exists in the container • It internally uses setter injection
constructor	• It is the same as autowiring byType but through constructor arguments. • Spring autowire the dependency based on constructor argument type through constructor injection. • Spring throws an exception if there is more than one bean of same type exists in the container
no	Default mode which means no autowiring

Autowiring using byName mode:

In byName mode, Spring looks for a bean in the container with id same as property name to autowire the dependency.

In byName mode, it is mandated to have bean id same as the property name and also the respective setter methods as Spring inject the bean dependency internally using setter injection.

Demo : Demo4_Autowiring_ByName_ByType

Autowiring using byType mode:

In byType mode, Spring looks for the bean in the container based on type of bean to autowire the dependency.

In byType mode, there is no mandate to have bean id same as property name as the dependency now is autowired based on type of property instead of the name. byType also requires the respective setter methods in the class as Spring inject the bean dependency internally using setter injection.

Autowiring using constructor mode:

In this mode, Spring does autowiring of beans internally byType similar to that of byType mode however here the dependency is injected using constructor injection instead of setter injection. Hence it is required to have parameterized constructor in the respective class.

In constructor mode of autowiring, again there is no mandate to have bean id same as property name as the dependency now is autowired based on type of property instead of the name. However it is mandatory to have parametarized constructor in the ReportService class because the dependency is now injected through the constructor.

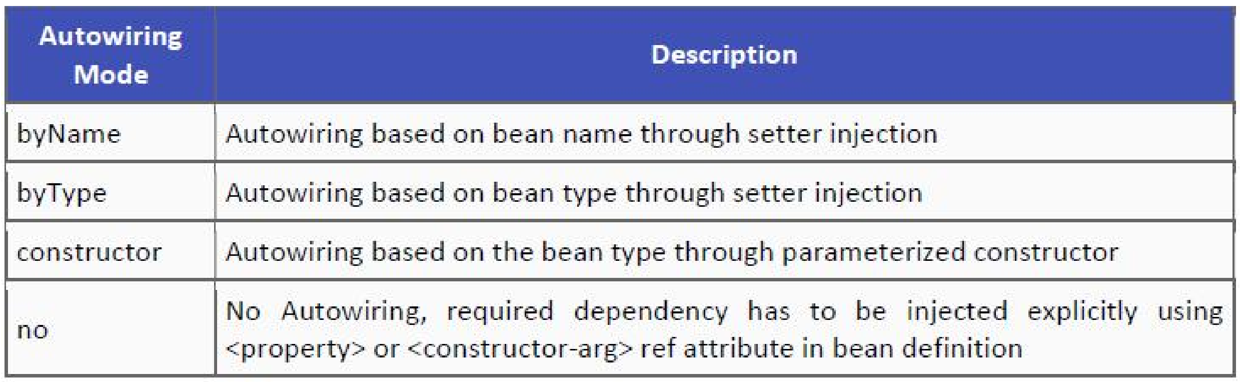
Demo : Demo5_Autowiring_ConstructorMode

Autowiring using no mode :

Another mode of autowiring is "no", which means no autowiring. Dependency is explicitly wired using <property name="propertyName" ref="beanId"> in setter injection or <constructor-arg ref="beanId"> in constructor injection configuration in the configuration.

"no" mode of autowiring is the default one in case of autowiring attribute is not present explicitly in the bean definition.

Autowiring Modes:



Autowiring Mode	Description
byName	Autowiring based on bean name through setter injection
byType	Autowiring based on bean type through setter injection
constructor	Autowiring based on the bean type through parameterized constructor
no	No Autowiring, required dependency has to be injected explicitly using <property> or <constructor-arg> ref attribute in bean definition

Bean Scope :

The lifetime of a bean depends on its scope. Bean's scope can be defined while declaring it in the configuration metadata file.

A bean can be in singleton or prototype scope. A bean with "singleton" scope is initialized during the container starts up and the same bean instance is provided for every bean request from the application. However in case of "prototype" scope, a new bean instance is created for every bean request from the application.

There will be a single instance of "singleton" scope bean in the container and the same bean is given for very request from the application.

In XML configuration, bean scope is specified using the "scope" attribute as shown in the below example.

Here, reportService bean is defined as singleton scope.

<bean id="reportService" class="com.infosys.demo.ReportService" scope="singleton"/>

This is the default mode, if not specified explicitly in the bean definition. In the below example, reportService bean has a singleton scope.

Demo : Demo6_BeanScope_Singleton

For "prototype" bean, there will be a new bean created for every request from the application.

In the below given example, reportService bean is defined with prototype scope. There will be a new reportService bean created for every bean request from the application.

```
<bean id="reportService" class="com.infosys.demo.ReportService" scope="prototype"/>
```

You can use prototype scope in scenarios where you need stateful beans and use singleton scope for stateless beans.

Demo : Demo7_BeanScope_Prototype

Introduction to Spring Annotation Based Configuration :

Spring Annotation Based Configuration:

Till now we have been using XML for writing Spring configurations. Spring also supports using Annotations for configuration. And the best part is that, both of them can also co-exist in the same project.

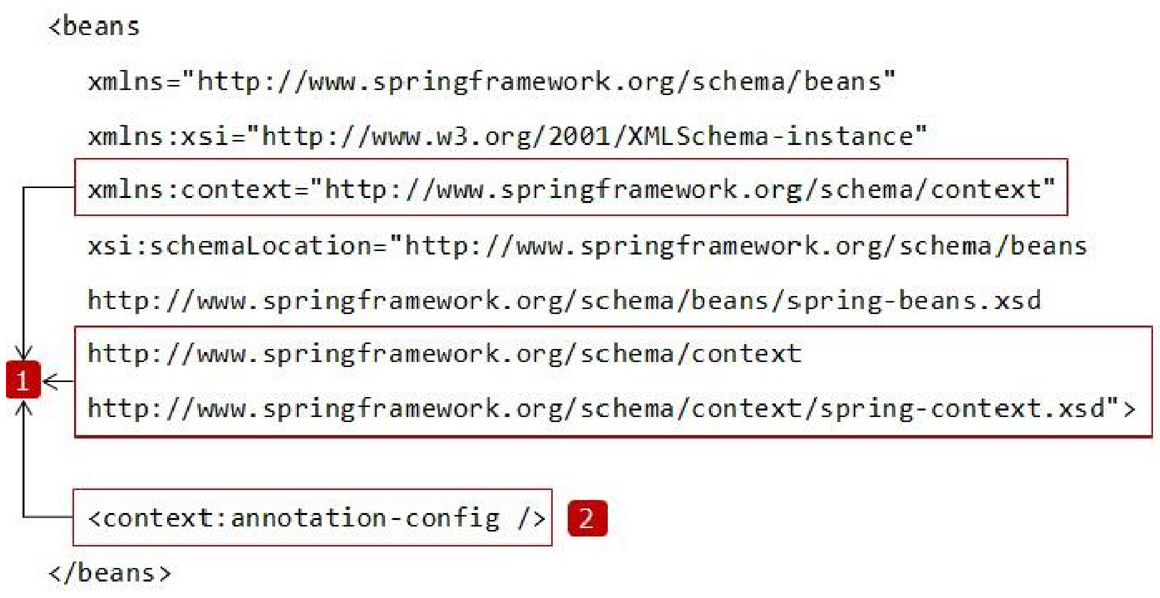
In Annotation based configuration, configuration is provided by using annotations on the relevant property, method or the class.

Annotation based configuration reduces the amount of configuration required in the XML file by using

- Autowiring: This supports autowiring dependencies using annotations such as @Autowired and @Qualifier.
- Auto Scanning: This enable Spring to auto create beans of the required classes and register with container for usage by eliminating the explicit <bean> definitions in the XML file.

By default autowiring using annotations is not turned on, <context:annotation-config> tag has to be specified in XML-based configuration to enable annotations support.

The XML configuration looks like as follows:



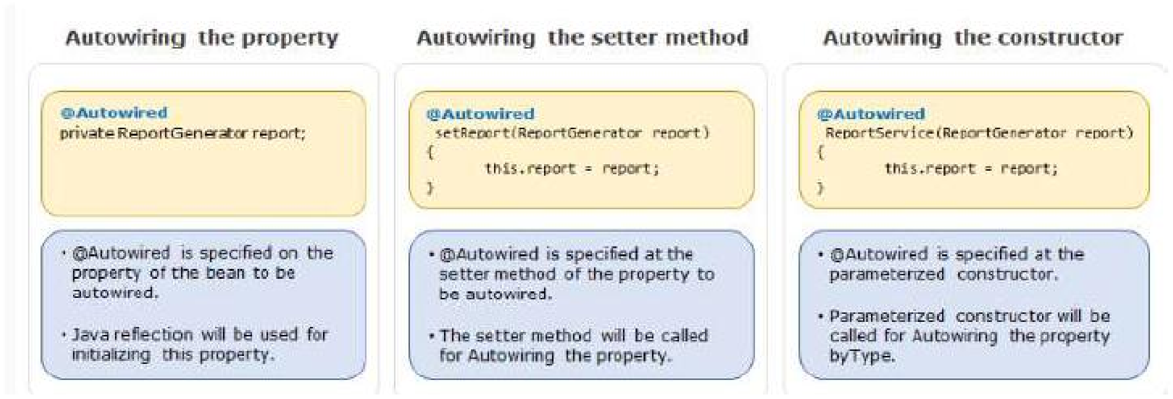
```
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd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

   <context:annotation-config />
</beans>
```

1. spring "context" namespace has "annotation-config" tag and hence context namespace has to be included in the declaration
2. This will enable support for annotations

@Autowired annotation perform byType Autowiring i.e. dependency is injected based on bean type

@Autowired cannot be used for primitive values.

This annotation can be used at property, setter method or constructor level



Autowiring the property

```
@Autowired
private ReportGenerator report;
```

- @Autowired is specified on the property of the bean to be autowired.
- Java reflection will be used for initializing this property.

Autowiring the setter method

```
@Autowired
setReport(ReportGenerator report)
{
        this.report = report;
}
```

- @Autowired is specified at the setter method of the property to be autowired.
- The setter method will be called for Autowiring the property.

Autowiring the constructor

```
@Autowired
ReportService(ReportGenerator report)
{
        this.report = report;
}
```

- @Autowired is specified at the parameterized constructor.
- Parameterized constructor will be called for Autowiring the property byType.

Demo8_Autowired_OnProperty

Demo9_Autowired_OnConstructor

Demo10_Autowired_OnSetterMethod

Introduction to Auto wired annotations:

Introduction to @Qualifier Annotation :

@Autowired annotation wire the dependency based on type of bean and in byType there has to be only one bean of specified type in the configuraiton but if there are two beans(htmlReportGenerator and pdfReportGenerator) of type ReportGenerator and hence there is an ambiguity on which bean to wire causing the application to throw an exception "UnsatisfiedDependencyException" for ReportGenerator dependency.

How do we resolve this exception to make one of the bean to be wired for the dependency?

@Qualifier("beanId") annotation can be used along with @Autowired annotation to achieve this requirement.

Demo11_Autowired_Qualifier

Introduction to Auto Scanning:

Annotation based configuration using Autowiring reduces the size of XML configuration file by replacing the tags like <constructor-arg> and <property> using @Autowired annotation.

But still, the bean definitions using <bean> tag need to be defined with Spring configuration file.

Auto scanning helps to remove explicit bean definition using <bean> tag from the XML file.

Spring provides a way to automatically detect the beans to be injected and avoid even the bean definitions within the Spring configuration file through Auto Scanning.

In Auto Scanning, Spring Framework automatically scans, detects and instantiates the beans from the specified base package, if there is no declaration for the beans in the XML file.

Auto Scanning can be switched on by using <context:component-scan>.

<context:component-scan> can also do whatever <context:annotation-config> does in addition to auto scanning.

Automatic discovery of beans can be achieved by doing the following:

- Use @Component annotation at POJO class level
- Include the following statement in the configuration file

<context:component–scan base–package = "packageName"/>

<context:component-scan /> tag looks for classes with the following annotations and creates beans for such classes automatically.

- @Component: It indicates the class(POJO class) as a Spring component.
- @Controller: It indicates Controller class(POJO class in Spring MVC) in the presentation layer.
- @Repository: It indicates Repository class(POJO class in Spring DATA) in the persistence layer.
- @Service: It indicates Service class(POJO class) in the business layer.

@Component is a generic annotation and other three are more specific annotations. For classes annotated with any of these annotations, Spring creates a bean with a default name which is same as the class name after making its first letter lowercase.

In the Report Generation application, following is the configuration to provide required bean definitions.

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

   <bean id="reportService" class="com.infosys.demo.ReportService" >
            <property name="master" ref= "htmlGenerator"/>
            <property name="recordsPerPage" value= "500"/>
   </bean>
   <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />
   <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />

</beans>

```

Let us now understand how Auto Scanning feature of Spring helps to remove explicit bean definition(using <bean> tag) from the XML file using simple configuration as shown below:

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

        <!--  Enable auto scanning feature in Spring -->
        <context:component-scan base-package="com.infosys" />

</beans>

```

Use of component-scan tag from context namespace enable auto scanning feature

- Allows Spring to scan package "com.infosys" for the classes annotatated with any of these annotations @Component/@Service/@Repository/@Controller
- Helps in auto creation of beans for such classes with bean name same as class name lowering the first letter

Auto scanning helps programmer to provide simple configuration for the applications by eliminating lot of coding effort.

Demo12_AutoScanning

Introduction to Spring Java Based Configuration:

Annotation based configuration needs the existence of a XML file with required declarations for enabling auto scan feature of Spring.

We can eliminate the need for a XML file completely and provide the same configurations in a Java class file.

Configurations in a XML file and the configurations in a Java class can co-exist as well.

In Java based configuration, we need to use @Configuration and @Bean annotations in the Java class to provide bean definitions.

@Configuration

It indicates Spring container that the class contains the bean definitions.

This is a pure Java approach to configure Spring container.

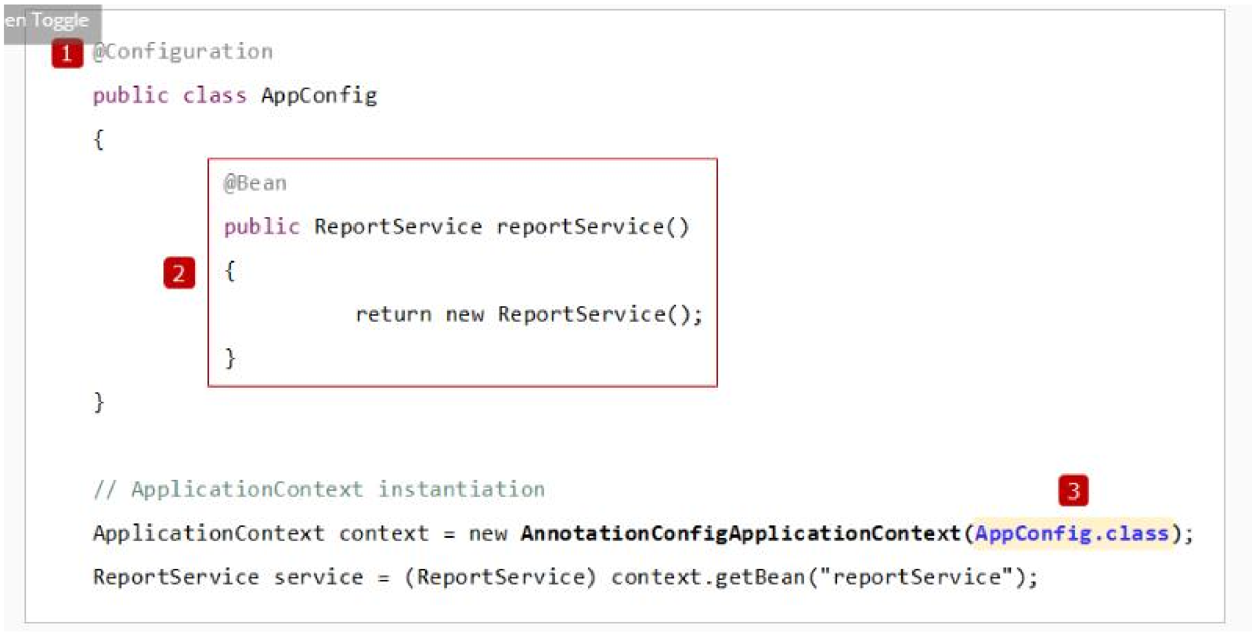
@Bean

@Bean is used for defining a bean in Java based configuration.

In @Configuration annotated Java class, methods annotated with @Bean provides the bean definition.

Spring processes @Bean annotated method and register the object returned by this method with Spring container.

By default, bean name is the same as the method name.



```
@Configuration
public class AppConfig
{
        @Bean
        public ReportService reportService()
        {
                return new ReportService();
        }
}

// ApplicationContext instantiation
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
ReportService service = (ReportService) context.getBean("reportService");
```

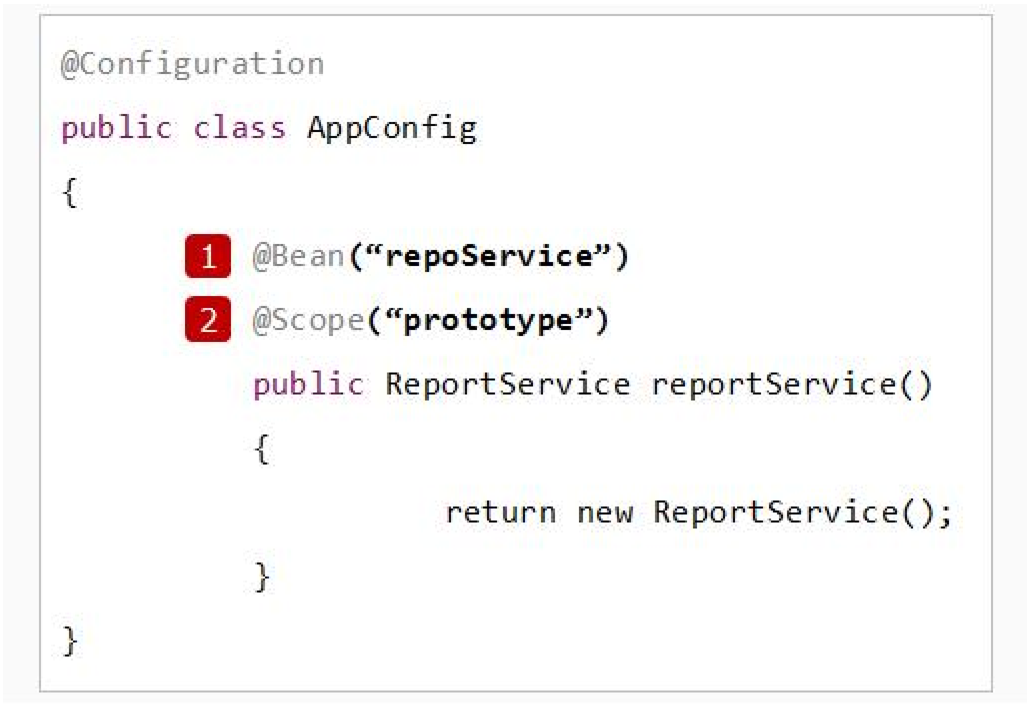
1.@Configuration indicates Spring that this class has bean definitions

2.ReportService class bean definition. In this example, the bean id is "reportService" which is the default name (same as the method name)

3.A class annotated with @Configuration is used as input for installing an ApplicationContext through AnnotationConfigApplicationContext class

Bean id can be specified explicitly in @Bean annotation using the "name" attribute. Explicit bean name overrides default name.

The bean scope also can be defined for a bean using @Scope annotation in Java configuration.

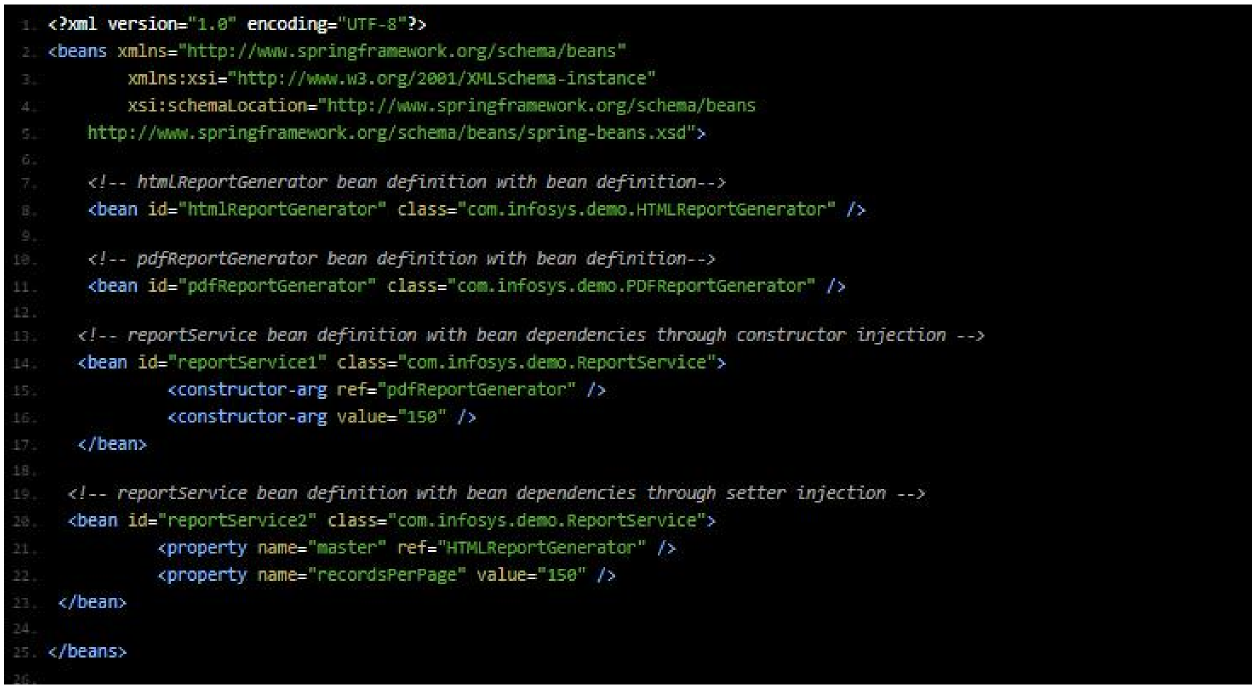


```
@Configuration
public class AppConfig
{
        1 @Bean("repoService")
        2 @Scope("prototype")
          public ReportService reportService()
          {
                  return new ReportService();
          }
}
```

4. Bean id is explicitly specified as "repoService", which overrides the default name
5. Bean scope is specified as "prototype"

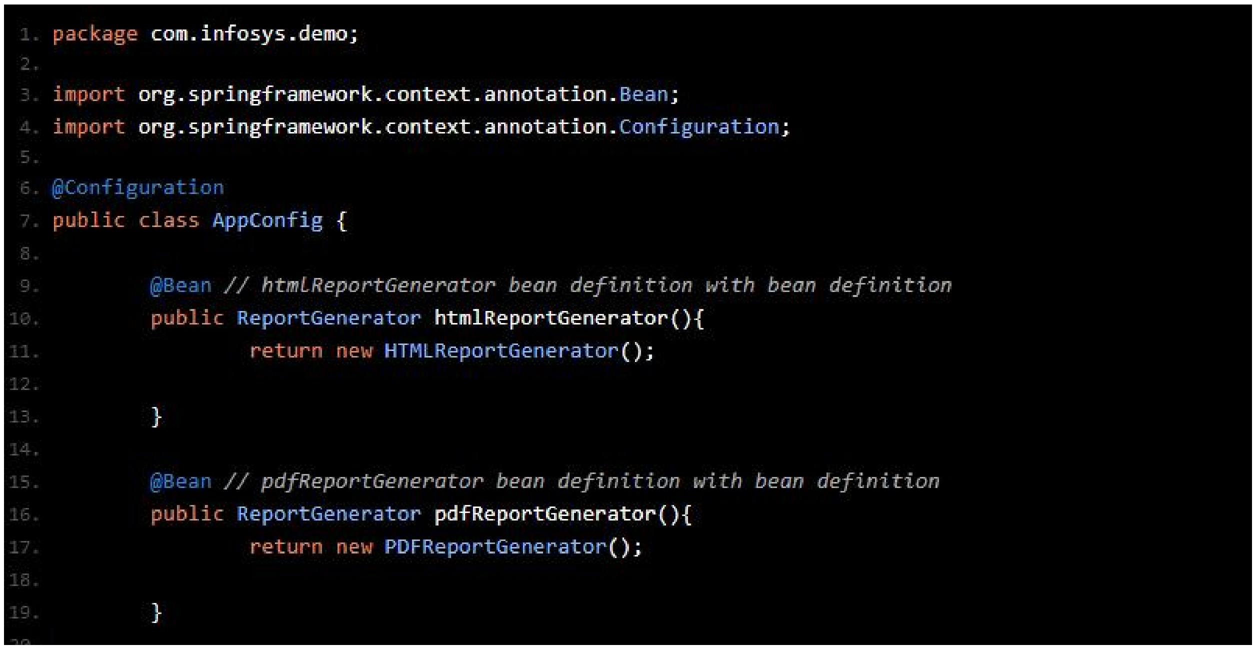
Let us now understand how the configuration metadata can be provided through Java class and eliminate the need for an XML file completely.

Consider the Report Generation application Spring configuration provided through XML configuration as shown below

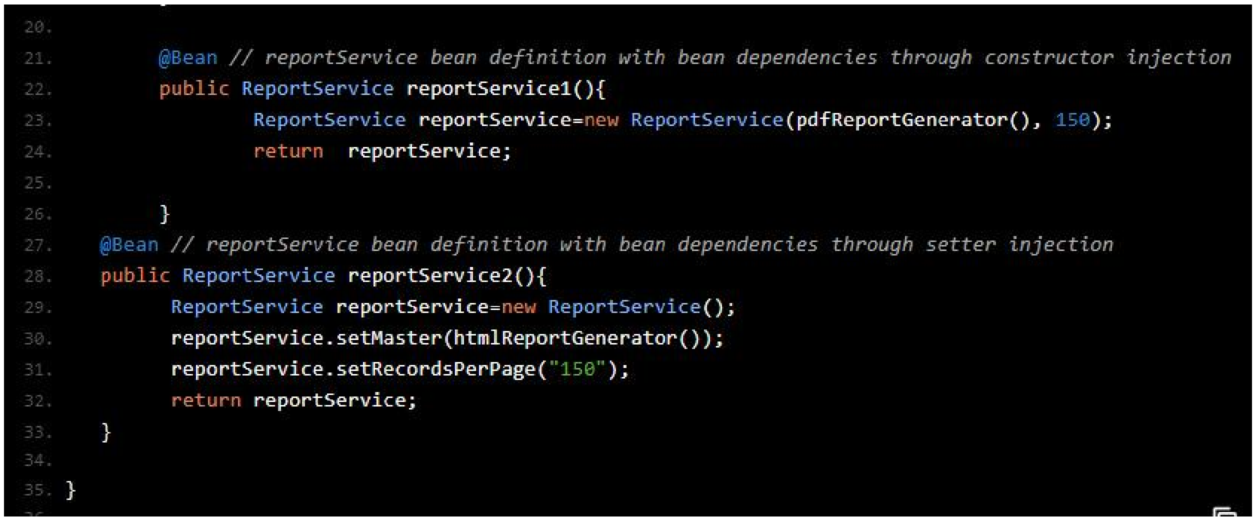


```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- htmlReportGenerator bean definition with bean definition-->
    <bean id="htmlReportGenerator" class="com.infosys.demo.HTMLReportGenerator" />

    <!-- pdfReportGenerator bean definition with bean definition-->
    <bean id="pdfReportGenerator" class="com.infosys.demo.PDFReportGenerator" />

   <!-- reportService bean definition with bean dependencies through constructor injection -->
   <bean id="reportService1" class="com.infosys.demo.ReportService">
            <constructor-arg ref="pdfReportGenerator" />
            <constructor-arg value="150" />
   </bean>

  <!-- reportService bean definition with bean dependencies through setter injection -->
  <bean id="reportService2" class="com.infosys.demo.ReportService">
           <property name="master" ref="HTMLReportGenerator" />
           <property name="recordsPerPage" value="150" />
 </bean>

</beans>
```

Let us now replace above XML configuration with Spring Java based configuration as shown below through a JAVA class.

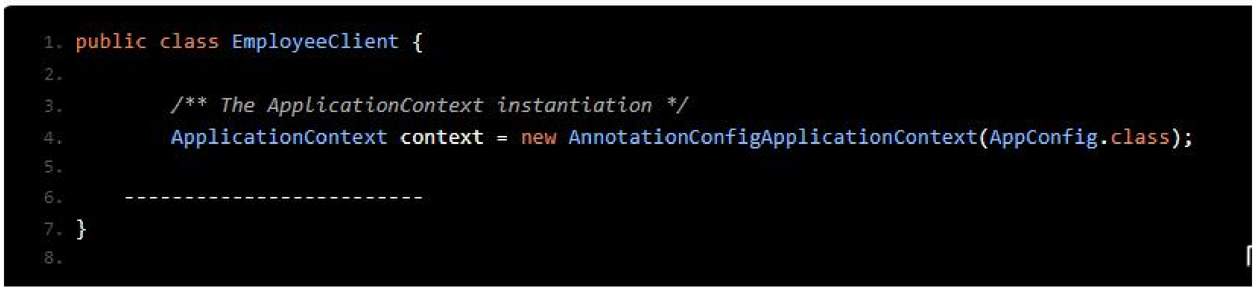


```java
package com.infosys.demo;

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

@Configuration
public class AppConfig {

        @Bean // htmlReportGenerator bean definition with bean definition
        public ReportGenerator htmlReportGenerator(){
                return new HTMLReportGenerator();

        }

        @Bean // pdfReportGenerator bean definition with bean definition
        public ReportGenerator pdfReportGenerator(){
                return new PDFReportGenerator();

        }
```



```
        @Bean // reportService bean definition with bean dependencies through constructor injection
        public ReportService reportService1(){
                ReportService reportService=new ReportService(pdfReportGenerator(), 150);
                return  reportService;

        }
    @Bean // reportService bean definition with bean dependencies through setter injection
    public ReportService reportService2(){
         ReportService reportService=new ReportService();
         reportService.setMaster(htmlReportGenerator());
         reportService.setRecordsPerPage("150");
         return reportService;
    }

}
```

Now because the configuration is provided through Java class, ApplicationContext in client code is instantiated using AnnotationConfigApplicationContext implementation by providing configuration class as argument.



```
public class EmployeeClient {

        /** The ApplicationContext instantiation */
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

    -------------------------
}
```

Demo13_JavaBasedSpringConfig

@ComponentScan and @PropertySource in Spring Java Based Configuration :

Benefits of Java based configuration:

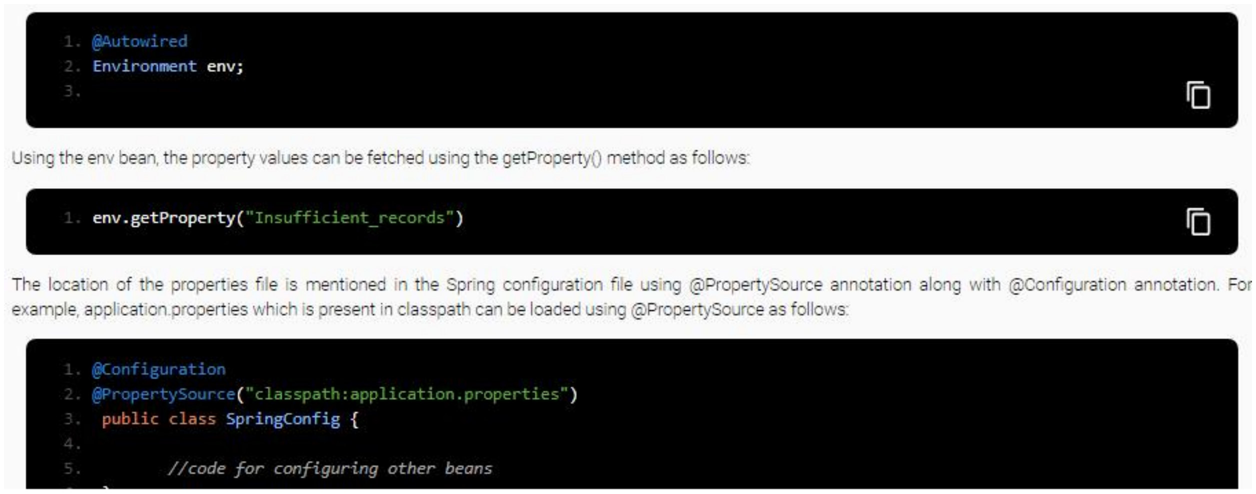
- XML parser is not required as the entire configuration is now provided through Java code
- It does compile time check which helps in debugging the code easily
- This support easy refactoring of the code
- It simplify test automation

As part of the annotation-based configuration, we learned how to eliminate explicit bean definitions by using auto scan feature of Spring.

Does Java-based configuration support auto scan feature of Spring?

Yes, it supports auto scan feature of Spring using the annotation @ComponentScan.

Spring provides Environment class to directly access the values in the properties file. In order to access the properties, the Environment class' bean can be autowired as follows:



```
@Autowired
Environment env;

```

Using the env bean, the property values can be fetched using the getProperty() method as follows:

```
env.getProperty("Insufficient_records")
```

The location of the properties file is mentioned in the Spring configuration file using @PropertySource annotation along with @Configuration annotation. For example, application.properties which is present in classpath can be loaded using @PropertySource as follows:

```
@Configuration
@PropertySource("classpath:application.properties")
 public class SpringConfig {

        //code for configuring other beans
```

Demo14_JavaBasedSpringConfig_PropertyScan

Introduction to Spring AOP :

AOP (Aspect Oriented Programming) is used for applying common behaviors like transaction, security, logging etc. to the application.

These common behaviors generally needs to be called from multiple locations in an application. Hence, they are also called as cross cutting concerns in AOP.

Spring AOP provides the solution to cross cutting concerns in a modularized and loosely coupled way.

Advantages

- AOP ensures that cross cutting concerns are kept separate from the core business logic.
- Based on the configurations provided, the Spring apply cross cutting concerns appropriately during the program execution.
- This allows to create a more loosely coupled application wherein you can change the cross cutting concerns code without affecting the business code.

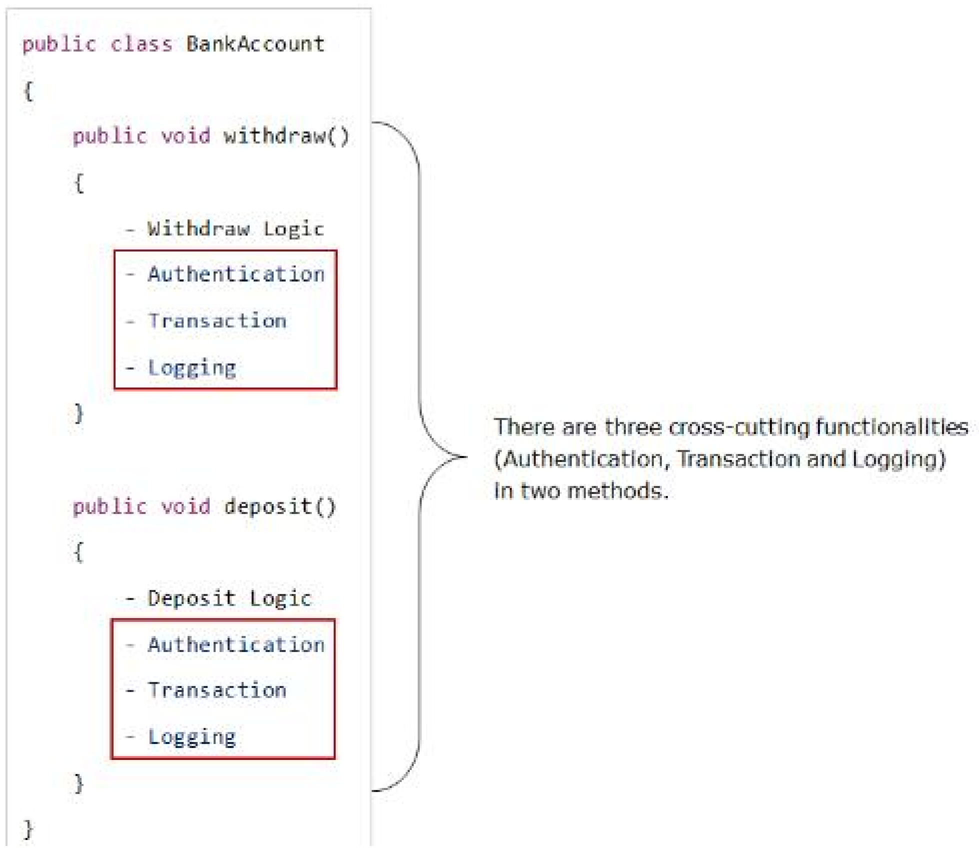
In Object Oriented Programming(OOP), the key unit of modularity is class. But in AOP the key unit of modularity is an Aspect.

What is an Aspect?

Aspects are the cross-cutting concerns that cut across multiple classes.

Examples : Transaction Management, Logging, Security etc.

Spring AOP: For better understanding of Aspect Oriented Programming(AOP) concepts let us consider a Banking scenario comprising of BankAccount class with withdraw and deposit functionalities as shown below.

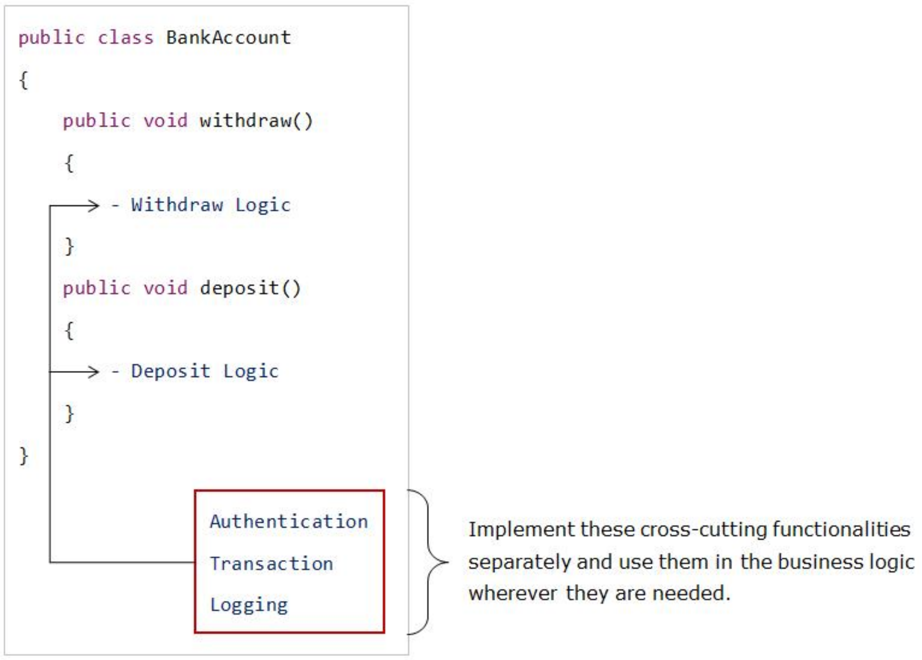


```
public class BankAccount
{
    public void withdraw()
    {
        - Withdraw Logic
        - Authentication
        - Transaction
        - Logging
    }

    public void deposit()
    {
        - Deposit Logic
        - Authentication
        - Transaction
        - Logging
    }
}
```

There are three cross-cutting functionalities (Authentication, Transaction and Logging) in two methods.

In a single class, cross-cutting functionalities are repeated twice. Think of a bigger scenario with many classes. You might need to repeat the cross-cutting concerns many times.

In Spring AOP, we can add these cross-cutting concerns at run time by separating the cross-cutting concerns from the client logic as shown below.


```
public class BankAccount
{
    public void withdraw()
    {
        - Withdraw Logic
    }
    public void deposit()
    {
        - Deposit Logic
    }
}
```
Authentication
Transaction
Logging
Implement these cross-cutting functionalities separately and use them in the business logic wherever they are needed.

In Spring AOP, we can add the cross-cutting functionalities at run time by separating the system services (cross-cutting functionalities) from the client logic.

In Spring AOP, we need to modularize and define each of the cross cutting concern in a single class called Aspect.

Each method of the Aspect which provides the implementation for the cross cutting concern is called an Advice.

The business methods of the program before or after which the advice can be called is known as a Joinpoint. The advice does not get inserted at every Joinpoint in the program. An advice gets applied only to the Joinpoints that satisfy the Pointcut defined for the advice.

Pointcut represents an expression that evaluates to the business method name before or after which the advice needs to be called.

Following table describes the various Spring AOP terminologies.

Terminologies	Definition
Aspects	These are the cross-cutting concerns that cut across multiple classes. Example: Logging, transition, security, etc.
Join point	The possible program execution points where the Aspects are applied. In Spring it is always the method invocation
Advice	This represents code to be executed at the selected Joinpoint(s).
Pointcut	This identifies on what Joinpoints that advice needs to be applied. AspectJ Pointcut Expression Language(EL) is used for this identification
Target Object, AOP Proxy and Weaving	At runtime, proxies are created for the business objects(called target objects) with the Aspects linked using JDk Dynamic/CGLIB proxy. This process is called Weaving. The object created is called AOP Proxy.

Spring AOP can be implemented using different approaches.

- Classic Spring Proxy based AOP: This is the earlier approach to support AOP in Spring applications using lower-level Spring AOP APIs
- XML Schema-based configuration
- @AspectJ annotation (Explained in this course)

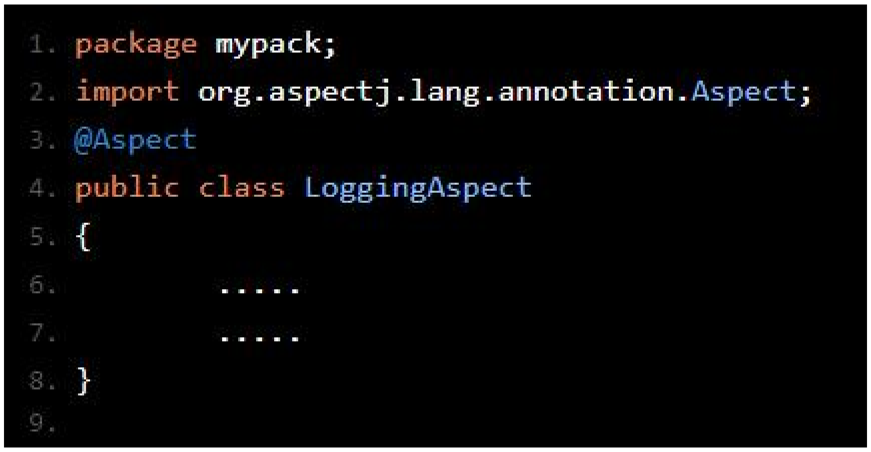
Spring introduced XML Schema-based and @AspectJ annotation approaches to write custom Aspects and Pointcut expressions in a simpler and more powerful way. It is recommended to use any of these approaches for new applications.

In @AspectJ annotation approach

- Aspects are defined as a POJO using annotations supported by AspectJ.
- AspectJ is the de-facto standard for AOP to support AOP capabilities.

Note: In XML Schema-based approach, Spring offers support for defining Aspects using the "aop" namespace tags in an XML based format using same Pointcut expressions and advice types of @AspectJ annotation approach.

Any class in the Spring context with @Aspect annotation is an Aspect



```
package mypack;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class LoggingAspect
{
        .....
        .....
}
```

Aspect class is a Java class with

- Methods and fields
- Pointcut declarations

Spring AOP - Advice types

Advice represents code to be executed at the selected Joinpoint/s. There are five advice types.

Before - Code specified with Before advice type is executed before a Joinpoint execution and then will continue with Joinpoint execution

AfterReturning - Code specified with AfterReturning advice type is executed after a Joinpoint executes and returns successfully without exceptions

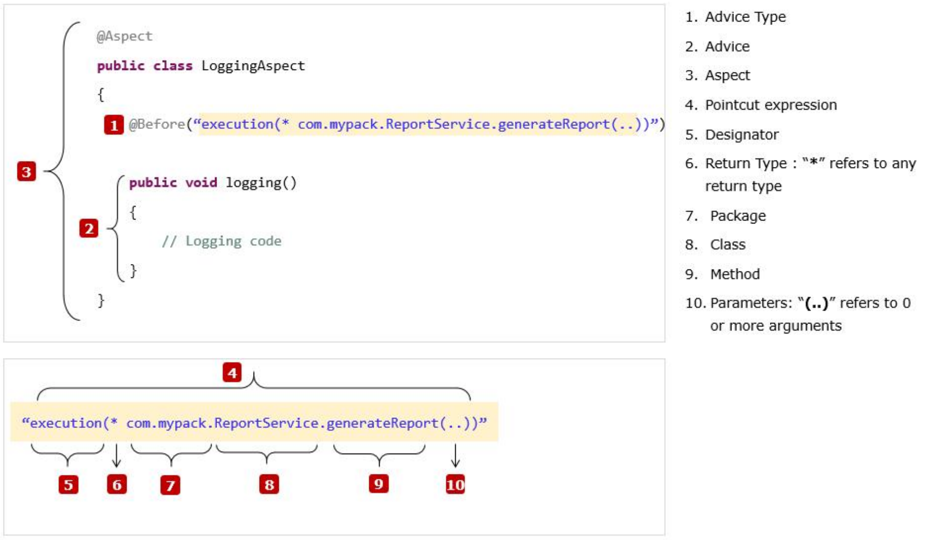
AfterThrowing - Code specified with AfterThrowing is executed only when a Joinpoint exits by throwing exception

After - Similar to finally block in exception handling. Code specified with After advice type will be executed after the Joinpoint returns with or without exception

Around - Code specified with Around advice type has some logic which gets executed before Joinpoint invocation and some logic which gets executed after the Joinpoint returns successfully

Spring AOP - Pointcut declaration:

Consider an Aspect "LoggingAspect" as shown below to understand different Spring AOP terminologies. LoggingAspect is defined as Spring AOP Aspect by using @Aspect annotation.



```
@Aspect
public class LoggingAspect
{
 @Before("execution(* com.mypack.ReportService.generateReport(..))")

  public void logging()
  {
      // Logging code
  }
}
```

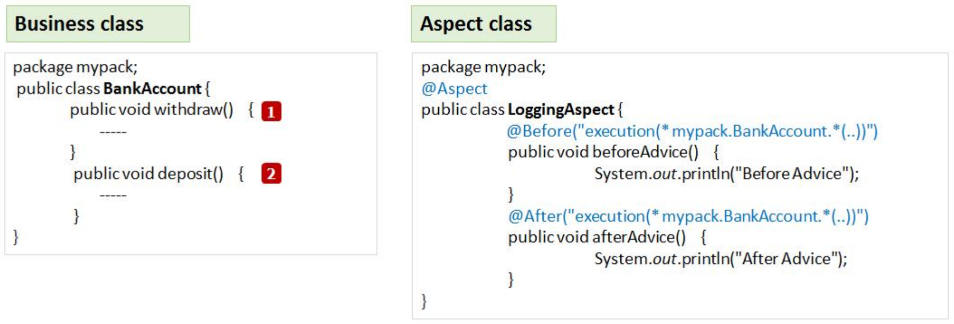
1. Advice Type
2. Advice
3. Aspect
4. Pointcut expression
5. Designator
6. Return Type : "*" refers to any return type
7. Package
8. Class
9. Method
10. Parameters: "(..)" refers to 0 or more arguments

The logging method of the LoggingAspect will be called before a business method (Joinpoint) that satisfies the Pointcut expression. In this case, logging method will be called before the generateReport() method of the class ReportService belonging to the package com.mypack having any number of arguments and any return type as indicated by the Pointcut expression.

Spring AOP – Joinpoints:

Joinpoints are the possible program execution points where the Aspects are applied. In Spring it is always the method invocation.

Let us understand Joinpoints and how to declare advice types in Aspect class using annotations



Business class

```
package mypack;
public class BankAccount {
        public void withdraw()  {  1
            -----
        }
        public void deposit()  {  2
            -----
        }
}
```

Aspect class

```
package mypack;
@Aspect
public class LoggingAspect {
        @Before("execution(* mypack.BankAccount.*(..))")
        public void beforeAdvice()  {
                System.out.println("Before Advice");
        }
        @After("execution(* mypack.BankAccount.*(..))")
        public void afterAdvice()  {
                System.out.println("After Advice");
        }
}
```

Spring AOP and AspectJ:

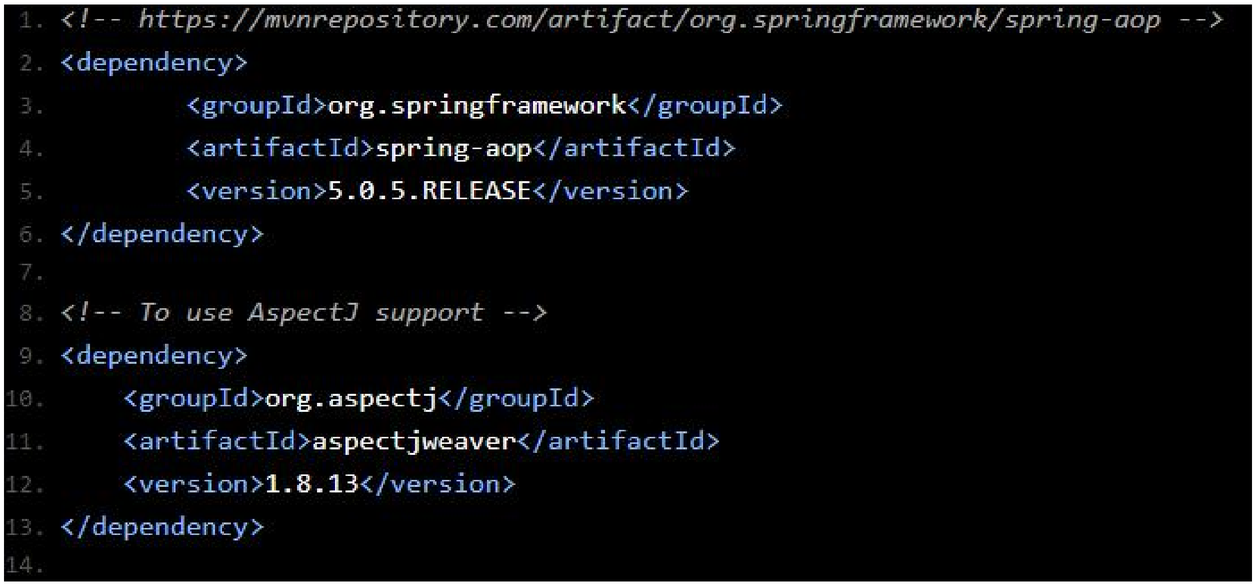
AspectJ annotations refer to the style of declaring regular Java classes as Aspects using Java 5 annotations.

The below configuration is required for using the AspectJ support in Spring.

6. Add following jars to the classpath
- spring-aop-5.0.5.RELEASE.jar
- aspectjweaver.jar (version 1.6.8 or later)

OR

Add below dependency along with Spring core dependency in Maven pom.xml file



```xml
<!-- https://mvnrepository.com/artifact/org.springframework/spring-aop -->
<dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>5.0.5.RELEASE</version>
</dependency>

<!-- To use AspectJ support -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>

```

7. Enable AspectJ annotation support
- XML based configuration: Include <aop:aspectj-autoproxy/> tag in configuration file
- Java Based Configuration: Annotate configuration class with @EnableAspectJAutoProxy

After Advice:

Let us now understand how the cross cutting concerns are implemented separately and applied at run time to business logic in Spring AOP using After Advice type.

After advice type - Code specified with After advice type will be executed after a Joinpoint returns with or without exception.

In Report Generation application, whenever a report is generated there is a requirement to log details such as the date and time of report generated along with Joinpoint signature for audit purpose. As of now details need to be logged on the console.

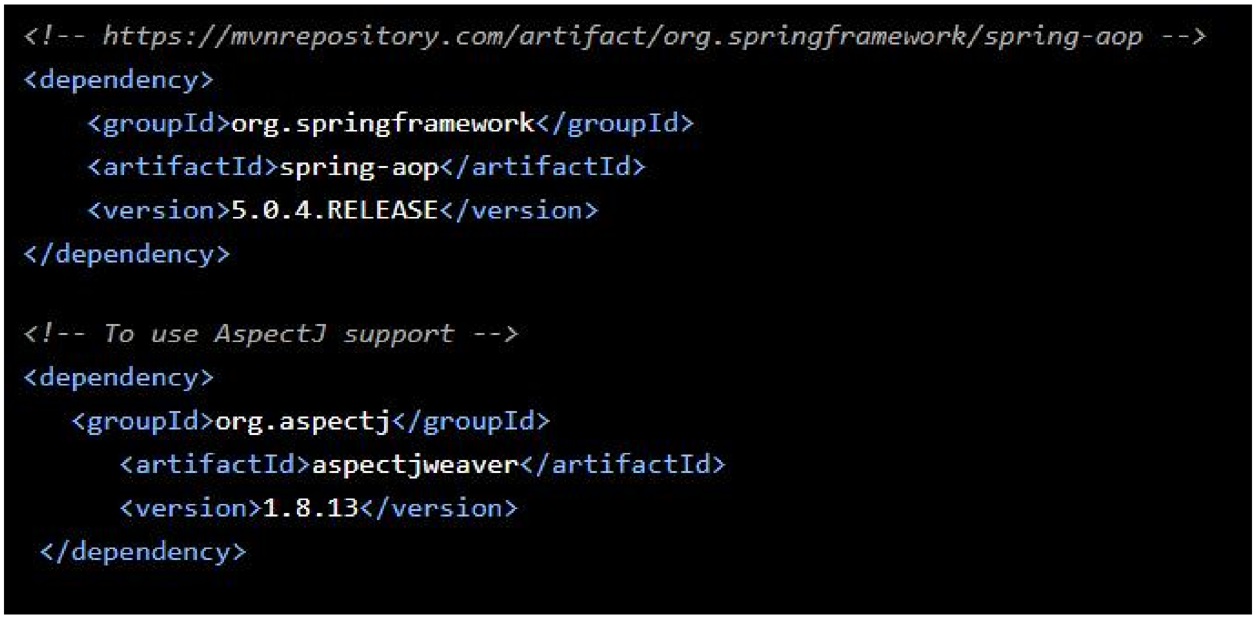
Following are the steps to be followed.

1. Add below given jars along with Spring core jars

- spring-aop-5.0.4.jar
- aspectjweaver.jar-1.8.13.jar

OR

Add below dependency along with Spring core dependency in Maven pom.xml file



```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-aop -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.0.4.RELEASE</version>
</dependency>

<!-- To use AspectJ support -->
<dependency>
   <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.13</version>
 </dependency>
```

2. Following configuration is required to implement aspects using AspectJ annotation support.

- Enable AspectJ annotations in XML Configuration

1.Include aop namespace and relevant xsd in the declaration

2.Use <aop:aspectj-autoproxy /> to enable AspectJ annotation support

3.Auto creation of aspect bean using component-scan

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

        <!-- Enable AOP support with aspectJ annotations -->
        <aop:aspectj-autoproxy />

        <!-- To enable Auto Scanning feature -->
        <context:component-scan base-package="com.infosys" />

</beans>
```

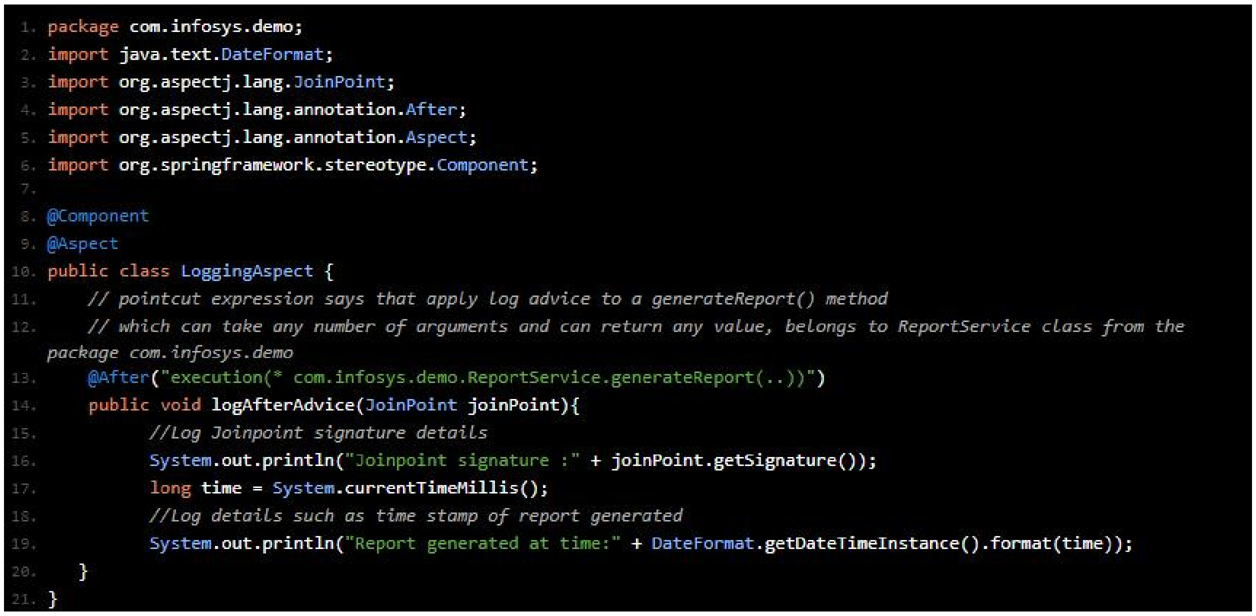
Enable AspectJ annotations in Java Configuration

- Annotate configuration class with @EnableAspectJAutoProxy to enable
- Enable auto creation of aspect bean using @ComponentScan

```java
package com.infosys.demo;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy // To enable AspectJ annotation support for AOP implementation
@ComponentScan("com.infosys.demo") // To enable auto creation of aspect bean and other application beans
public class AppConfig {

}

```

3. We need a logging advice which has to be invoked after the generateReport() method execution. Define an aspect with After advice type as below to log the details.

We need a bean of LoggingAspect and hence @Component has been used for enabling auto creation of bean by Spring.



```java
package com.infosys.demo;
import java.text.DateFormat;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {
    // pointcut expression says that apply log advice to a generateReport() method
    // which can take any number of arguments and can return any value, belongs to ReportService class from the
package com.infosys.demo
    @After("execution(* com.infosys.demo.ReportService.generateReport(..))")
    public void logAfterAdvice(JoinPoint joinPoint){
            //Log Joinpoint signature details
            System.out.println("Joinpoint signature :" + joinPoint.getSignature());
            long time = System.currentTimeMillis();
            //Log details such as time stamp of report generated
            System.out.println("Report generated at time:" + DateFormat.getDateTimeInstance().format(time));
    }
}
```

Demo15_SpringAOP_AfterAdvice

After Returning Advice:

After" advice type will be executed after the Joinpoint generateReport() executes successfully or throws an exception.

What if I want to log the details only for success scenarios and not for exceptional scenarios?

This can be achieved using AfterReturning advice type.

AfterReturning advice type - Code specified with AfterReturning advice type is executed only after a Joinpoint executes and returns successfully without exception.

Consider Report Generation application, now the requirement is to log only for success scenario i.e report generated successfully and there should not be any log in case of any exception in generating the report through generateReport() method.

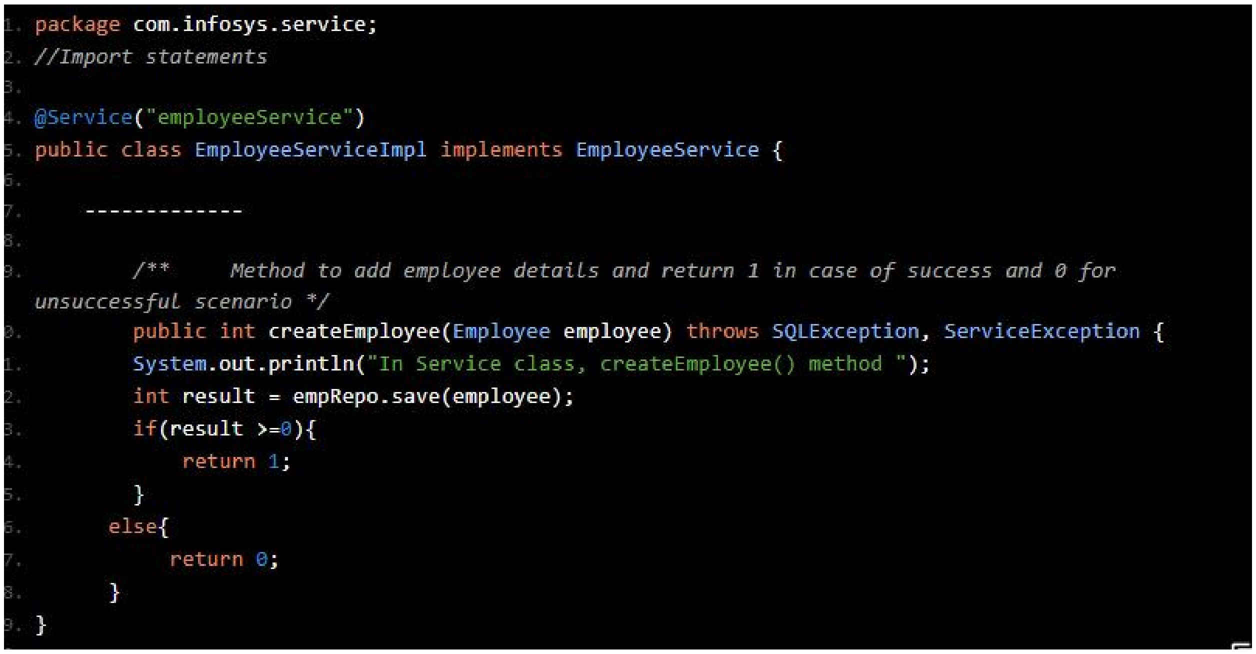
Consider the previously discussed After advice type code and now replace @After by @AfterReturning annotation in the LoggingAspect.java.

Demo16_SpringAOP_AfterReturningAdvice

Access the return value using AfterReturning Advice:

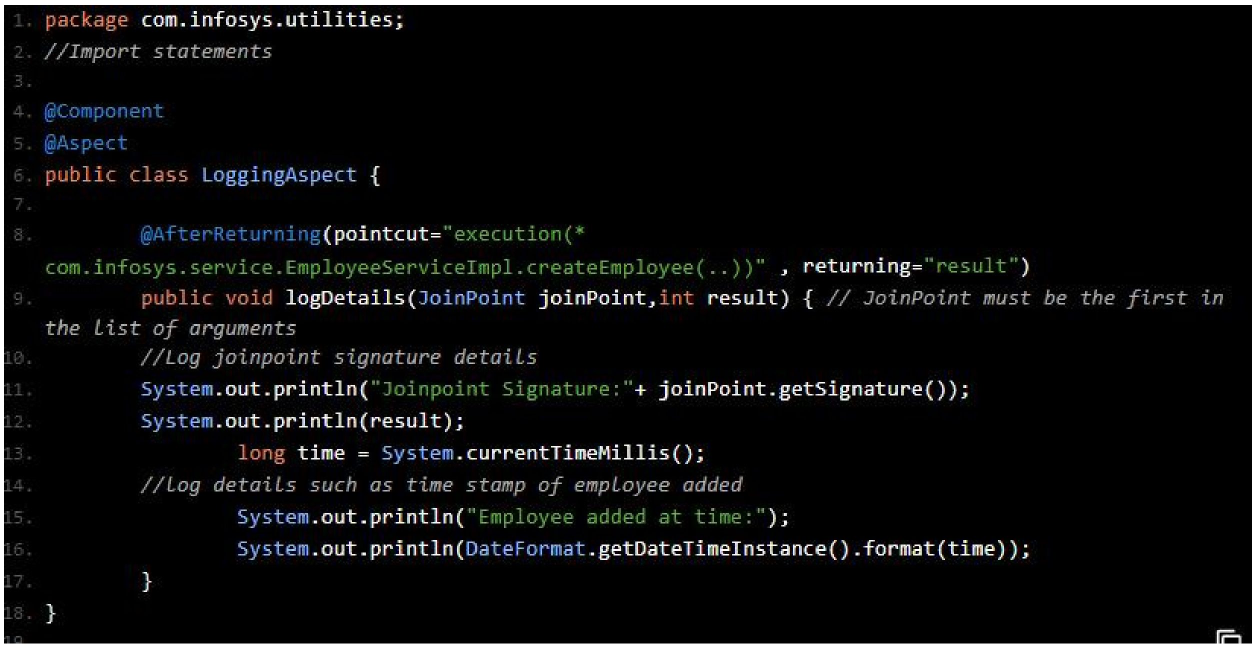
AfterReturning advice can access the return value from Joinpoint but it cannot return modified value to client code i.e value returned to the client will be the one returned from the Joinpoint.

Consider the following the Employee application with createEmployee(Employee employee) method of EmployeeServiceImpl to understand the concept better.



```java
package com.infosys.service;
//Import statements

@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {

    -------------

        /**     Method to add employee details and return 1 in case of success and 0 for
unsuccessful scenario */
        public int createEmployee(Employee employee) throws SQLException, ServiceException {
        System.out.println("In Service class, createEmployee() method ");
        int result = empRepo.save(employee);
        if(result >=0){
            return 1;
        }
      else{
           return 0;
      }
}
```

LoggingAspect with AfterReturning advice capturing the return value of Joinpoint (i.e return value of createEmployee() method).



```java
package com.infosys.utilities;
//Import statements

@Component
@Aspect
public class LoggingAspect {

        @AfterReturning(pointcut="execution(*
com.infosys.service.EmployeeServiceImpl.createEmployee(..))" , returning="result")
        public void logDetails(JoinPoint joinPoint,int result) { // JoinPoint must be the first in
the list of arguments
        //Log joinpoint signature details
        System.out.println("Joinpoint Signature:"+ joinPoint.getSignature());
        System.out.println(result);
                long time = System.currentTimeMillis();
        //Log details such as time stamp of employee added
                System.out.println("Employee added at time:");
                System.out.println(DateFormat.getDateTimeInstance().format(time));
        }
}
```

In the above code, return value from Joinpoint is automatically captured in the Pointcut expression attribute `returning` with name as "result" (it is a user defined name) as shown above.

In order to access captured value in the advice code logDetails, corresponding argument has to be specified after the JoinPoint argument.

Now in the logged details on console, you can observe logged return value for success scenario.

Success scenario : Add a new employee with required details to the respective table in the database (will log details)

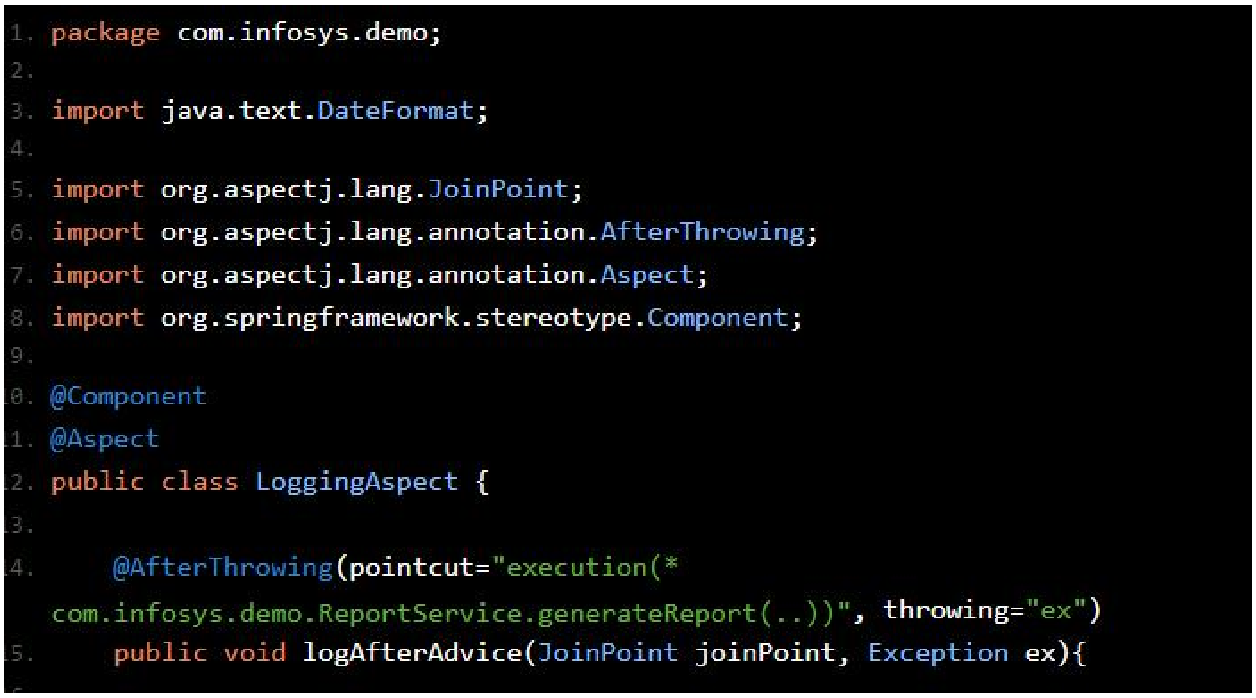
Exception scenario: Try to add an employee after removing/renaming the corresponding table in the database (will not log details)

AfterThrowing Advice :

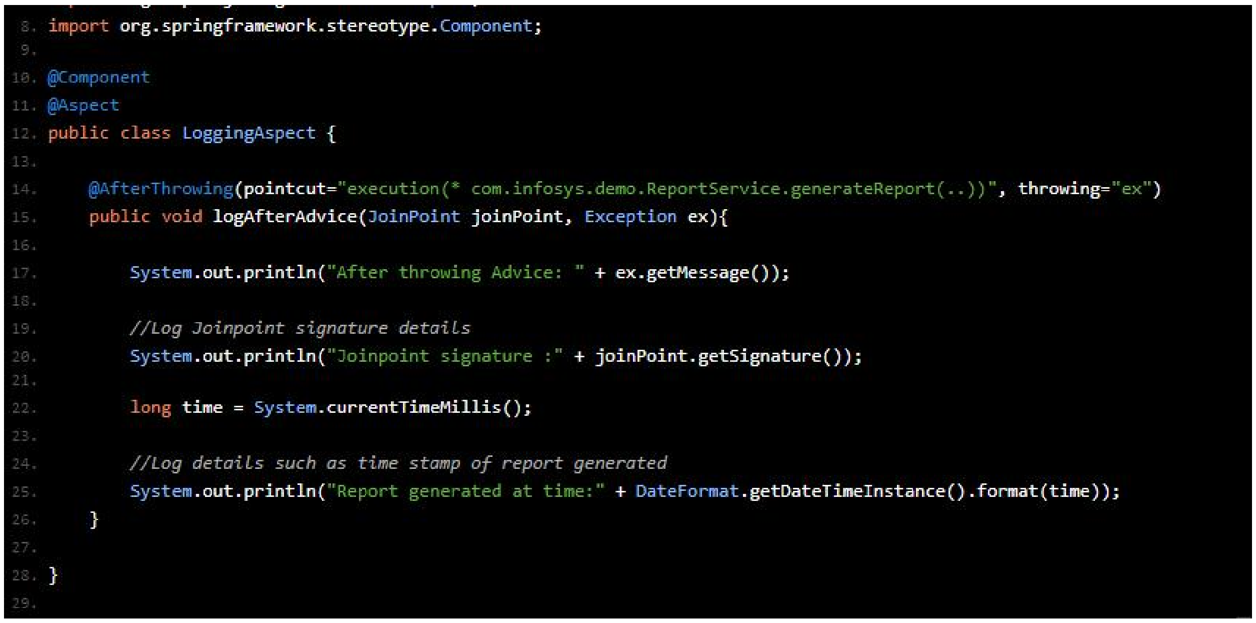
Now we need the opposite of AfterReturning i.e advice code has to be applied only when the Joinpoint throws an exception.

We can use AfterThrowing advice type for this requirement.

AfterThrowing advice type - Code specified with AfterThrowing is executed only when a Joinpoint exits by throwing an exception.



```java
package com.infosys.demo;

import java.text.DateFormat;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {

    @AfterThrowing(pointcut="execution(*
com.infosys.demo.ReportService.generateReport(..))", throwing="ex")
    public void logAfterAdvice(JoinPoint joinPoint, Exception ex){
```



```
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {

    @AfterThrowing(pointcut="execution(* com.infosys.demo.ReportService.generateReport(..))", throwing="ex")
    public void logAfterAdvice(JoinPoint joinPoint, Exception ex){

        System.out.println("After throwing Advice: " + ex.getMessage());

        //Log Joinpoint signature details
        System.out.println("Joinpoint signature :" + joinPoint.getSignature());

        long time = System.currentTimeMillis();

        //Log details such as time stamp of report generated
        System.out.println("Report generated at time:" + DateFormat.getDateTimeInstance().format(time));
    }

}
```

Consider the Report Generation application code discussed for AfterReturning advice type and now replace @AfterReturning annotation with @AfterThrowing annotation in LoggingAspect.java.

AfterThrowing advice can access the exception thrown by Joinpoint but it will not handle the exception.

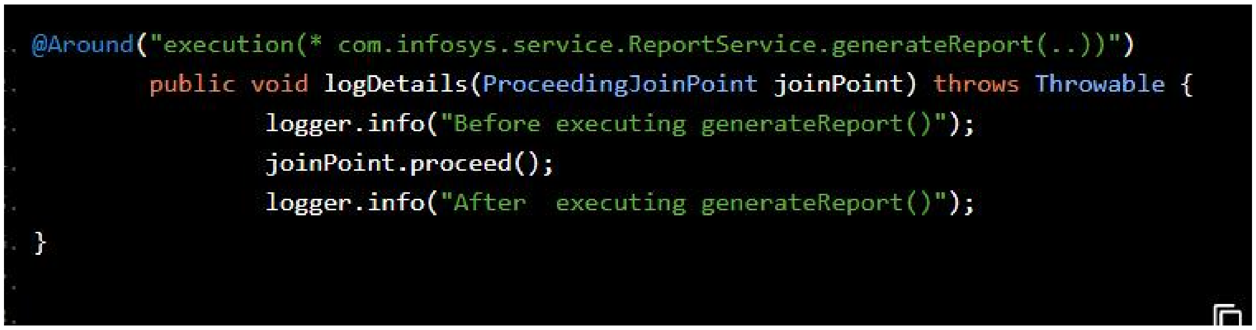
Around Advice:

We can choose Around advice type in scenarios where we need some customized behavior to be performed before as well as after the Joinpoint execution.

Around advice also decides on whether to continue to the Joinpoint or return its own value or throw an exception.

Around advice type - Method annotated with Around advice type has some logic that gets executed before Joinpoint and some logic which gets executed after returning(successfully or throws an exception) from the Joinpoint

Let us understand how Around advice works.



```
@Around("execution(* com.infosys.service.ReportService.generateReport(..))")
        public void logDetails(ProceedingJoinPoint joinPoint) throws Throwable {
                logger.info("Before executing generateReport()");
                joinPoint.proceed();
                logger.info("After  executing generateReport()");
}
```

In the above code, we are applying around advice for generateReport() method:

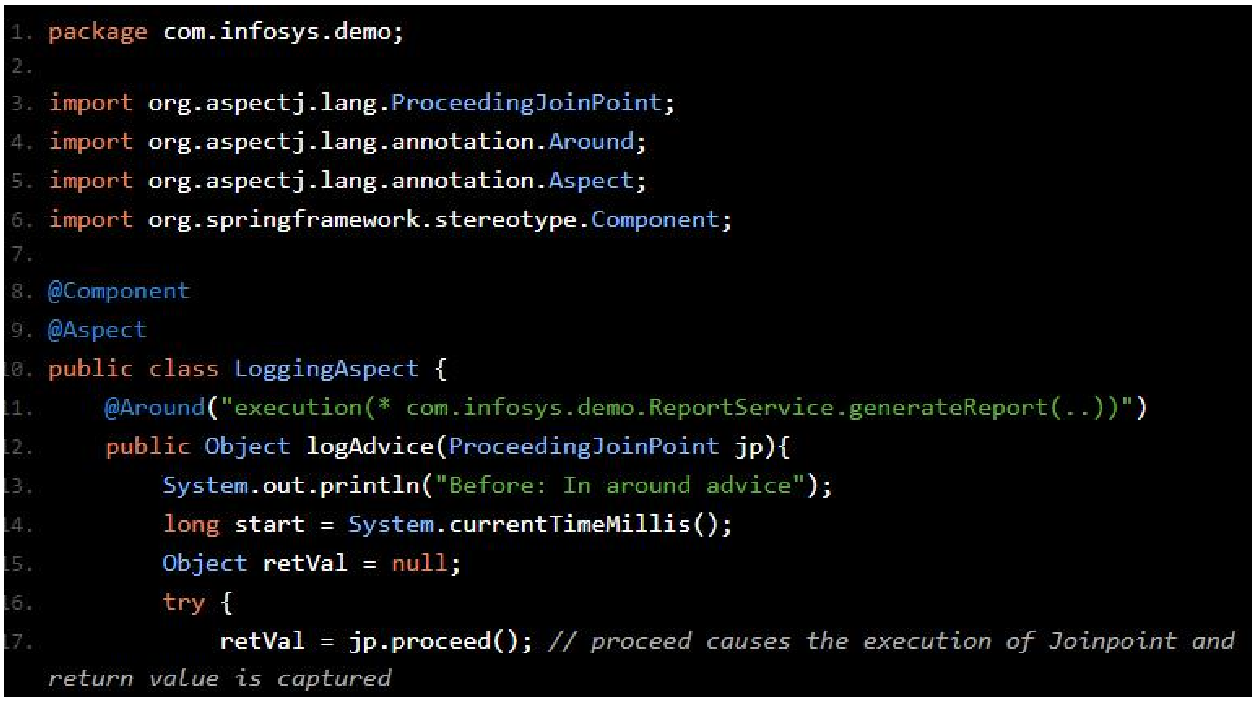
- When generateReport() method of ReportService class is invoked in the application, first the control goes to the around advice and the code till the proceed() will be executed.

- proceed() makes the control move to Joinpoint and hence Joinpoint generateReport() gets executed and then control comes back to the advice.
- The remaining code after the proceed() will get executed.

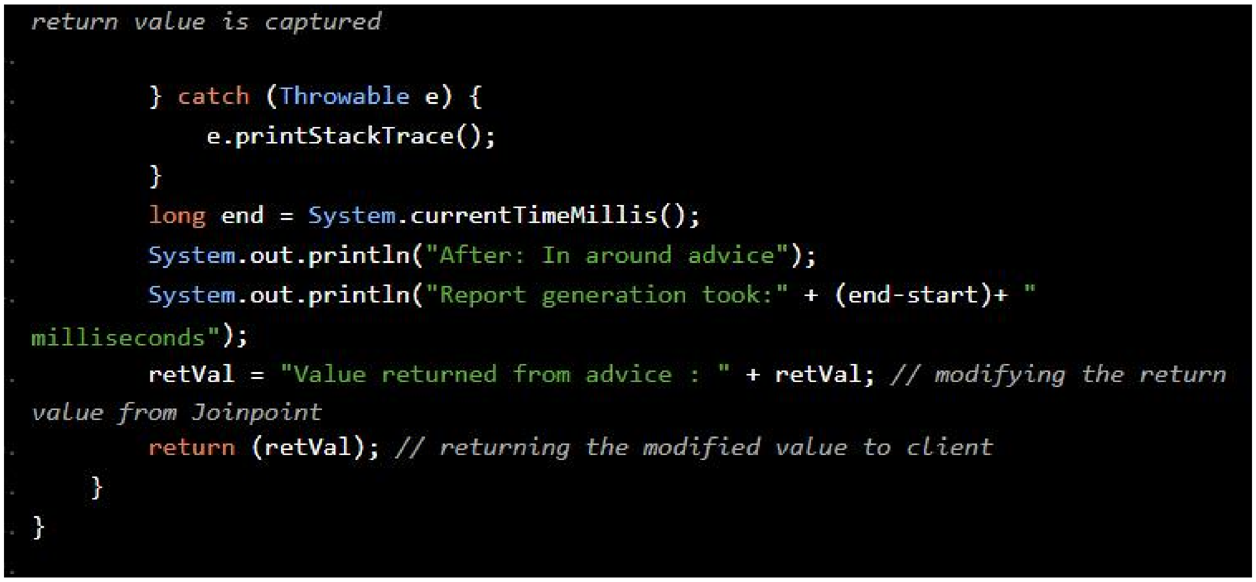
Around advice is a powerful one compare to other advices due to the following:

- It can execute custom behavior both before and after the Joinpoint.
- Around advice access can access the return value of Joinpoint, can modify the value and return the modified value back to the client.
- It can access arguments of Joinpoint and can modify and pass the modified arguments to the Joinpoint.
- Around advice is capable to decide on whether to proceed to the Joinpoint or return its own value or an exception to the client.

Let us look the Around advice for Report Generation Application, LoggingAspect class with Around advice is defined as below:



```java
package com.infosys.demo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {
    @Around("execution(* com.infosys.demo.ReportService.generateReport(..))")
    public Object logAdvice(ProceedingJoinPoint jp){
        System.out.println("Before: In around advice");
        long start = System.currentTimeMillis();
        Object retVal = null;
        try {
            retVal = jp.proceed(); // proceed causes the execution of Joinpoint and return value is captured
```



```
return value is captured

        } catch (Throwable e) {
            e.printStackTrace();
        }
        long end = System.currentTimeMillis();
        System.out.println("After: In around advice");
        System.out.println("Report generation took:" + (end-start)+ "
milliseconds");
        retVal = "Value returned from advice : " + retVal; // modifying the return
value from Joinpoint
        return (retVal); // returning the modified value to client
    }
}
```

In the above code, you can observe that Around advice is capable of capturing the return value from Joinpoint, can modify the return value and send the modified value to the client. Client always gets the value returned from the Around advice, not from the Joinpoint.

Following is the expected result:

```
Before: In around advice
In Report Service generate method
In generate Report method of HTMLReportGenerator
Value return by report: Generated HTML Report with 100 records
After: In around advice
Report generation took:21 milliseconds
Value returned from advice : Generated HTML Report with 100 records
```

In the log details, the first line is from advice, the next 3 lines are from generateReport() method, next 2 lines are again from the advice and the last line is from client code.

Demo17_SpringAOP_AoundAdvice

Access the JoinPoint arguments using Around Advice:

In the previous example, we learned how to capture the return value of Joinpoint and then return the modified return value to client code.

How can Around advice access the Joinpoint arguments?

Around advice can access the Joinpoint arguments using getArgs() method of ProceedingJoinpoint.

Let us understand with simple scenario based on Report Generation application.

Consider the below given modified generateReport() method of service class. Here recordsPerPage value is passed as method argument.

```java
public void generateReport(int recordsPerPage) {
        System.out.println("recordsPerPage value in Report Service method : " + recordsPerPage);
        --------------------
        --------------------
}
```

Below given is the Aspect defined with around advice to accept the recordsPerPage argument of generateReport() method (Joinpoint) using getArgs() method.

```java
@Component
@Aspect
public class MyAspect {
    @Around("execution(* com.infosys.demo.ReportService.generateReport(..))")
    public void myAdvice(ProceedingJoinPoint jp){
             Object[] args = pjp.getArgs(); // getArgs() returns generateReport(int recordsPerPage) method argument. In this example, it is recordsPerPage value.
             System.out.println("Accessed value in advice :" + ((Integer)args[0]).intValue() );
             args[0] = args[0] + 10;  // Modifying accessed value of recordsPerPage
             proceed(recordsPerPage); // passing modified value of recordsPerPage to the Joinpoint.
     }
}
```

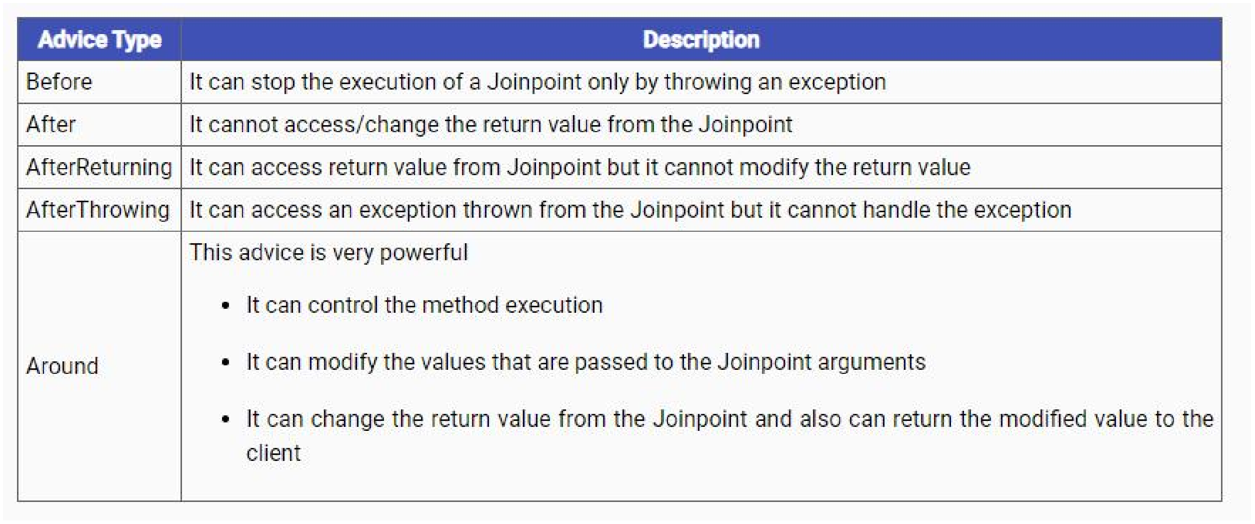
In the above code, assume you are passing value of recordsPerPage as 100 from client. You can observe below response on console.

```
Accessed value in advice : 100

recordsPerPage value in Report Service method : 110
```

It is based on requirement in your application, you can use around advice

- To access Joinpoint arguments in advice
- Modify accessed value
- Also pass the modified value to the Joinpoint argument

Below mentioned are some of the important points about various advice types.



Advice Type	Description
Before	It can stop the execution of a Joinpoint only by throwing an exception
After	It cannot access/change the return value from the Joinpoint
AfterReturning	It can access return value from Joinpoint but it cannot modify the return value
AfterThrowing	It can access an exception thrown from the Joinpoint but it cannot handle the exception
Around	This advice is very powerful • It can control the method execution • It can modify the values that are passed to the Joinpoint arguments • It can change the return value from the Joinpoint and also can return the modified value to the client

Maven based Spring application:

Maven is an Apache project, created with the main objective to make developer understand the state of total development effort in a short time.

8. Maven is an intelligent, comprehensive, project management and build tool (developed in Java) to build deployable artifacts from the source code and has a superset of features available
9. The core of Maven project is the POM file that is an XML file holding project configuration information. It contains meta data of/references to project resources like source code, test code, dependencies (external jar files) etc
10. There will be a POM file with required dependency details located in the root directory of the Maven project.

Key Advantages of Maven

11. Simple setup with best practices
12. Easy dependency management with central repository for multiple projects being worked upon at the same time

Demo18_MavenProject