



# **Advance Java**

-Professor Anamika Dhawan



# Spring Framework

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as [Struts](#), [Hibernate](#), Tapestry, [EJB](#), [JSF](#), etc.

The framework, in broader sense, can be defined as a structure where we find solution of the various technical problems.

The Spring framework comprises several modules such as IOC, AOP, DAO, Context, ORM, WEB MVC etc



## Spring Framework provide:

1. Application context and dependency injection.
2. AOP
3. Data Access(JDBC,ORM,OXM,JMS,etc)
4. Spring MVC

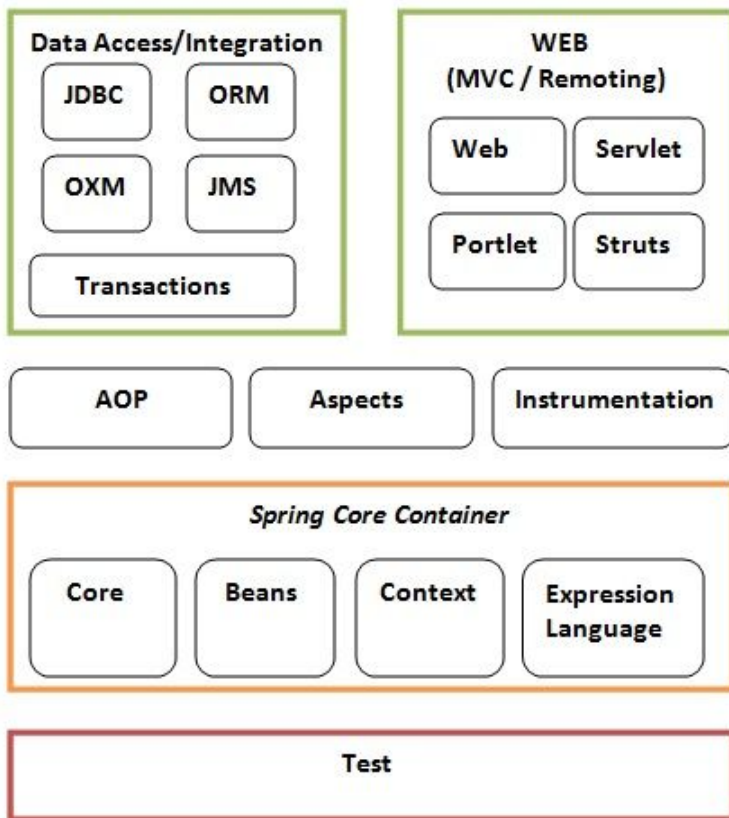


## Spring Module

The Spring framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc.

These modules are grouped into Test, Core Container, AOP, Aspects, Instrumentation, Data Access / Integration, Web (MVC / Remoting) as displayed in the following diagram.

## *Spring Framework Runtime*





# Spring Framework Architecture

Refer:

<https://www.geeksforgeeks.org/spring-framework-architecture/>



# EJB

Enterprise JavaBeans, the distributed business component model of the J2EE platform.

- 1. Application needs Remote Access.** In other words, it is distributed.
- 2. Application needs to be scalable.** EJB applications supports load balancing, clustering and fail-over.
- 3. Application needs encapsulated business logic.** EJB application is separated from presentation and persists



## **J2EE**

Java 2 Enterprise Edition, an umbrella specification that brings several different technologies together and forms the enterprise Java environment. Java Enterprise Edition (Java EE) is its newer name after Java release 5.





# POJO

One of the new features of added by Sun Microsystems in EJB 3.0 is POJO (Plain Old Java Object).

Plain Old Java Objects, a term devised to infer Java classes that don't depend on any environment-specific classes or interfaces, and don't need any special environment to run in.

Therefore, all normal Java objects are POJO's only.

The interface class in POJO model is a Plain Old Java Interface (POJI).



## POJO Rules

1. Class must be public
2. Properties/Variables must be private
3. Must have public default constructor
4. Can have argument constructor
5. Every property should have public getter and setter methods



## Advantage of POJO

POJO programming model is that coding application classes is very fast and simple. This is because classes don't need to depend on any particular API, implement any special interface, or extend from a particular framework class. You do not have to create any special callback methods until you really need them.

POJO-based classes don't depend on any particular API or framework code, they can easily be transferred over the network and used between layers. Therefore, you don't need to create separate data transfer object classes in order to carry data over the network.



## Advantage of POJO

You don't need to deploy your classes into any container or wait for long deployment cycles so that you can run and test them. You can easily test your classes within your favorite IDE using JUnit.

The POJO programming model lets you code with an object-oriented perspective instead of a procedural style. It becomes possible to reflect the problem domain exactly to the solution domain. Business logic can be handled over a more fine-grained model, which is also richer in terms of behavioral aspects.



# Spring Container

IoC is also known as *dependency injection* (DI).

It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.



# Spring Container

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container.

The [BeanFactory](#) interface provides an advanced configuration mechanism capable of managing any type of object. [ApplicationContext](#) is a sub-interface of `BeanFactory`.

The `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality.

The `ApplicationContext` is a complete superset of the `BeanFactory`



## Spring Container

In Spring, 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.

Otherwise, a bean is simply one of many objects in your application. Beans, and the *dependencies* among them, are reflected in the *configuration metadata* used by a container.



## Container Overview

The interface `org.springframework.context.ApplicationContext` represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans.

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

The configuration metadata is represented in XML, Java annotations, or Java code.

It allows you to express the objects that compose your application and the rich interdependencies between such objects.

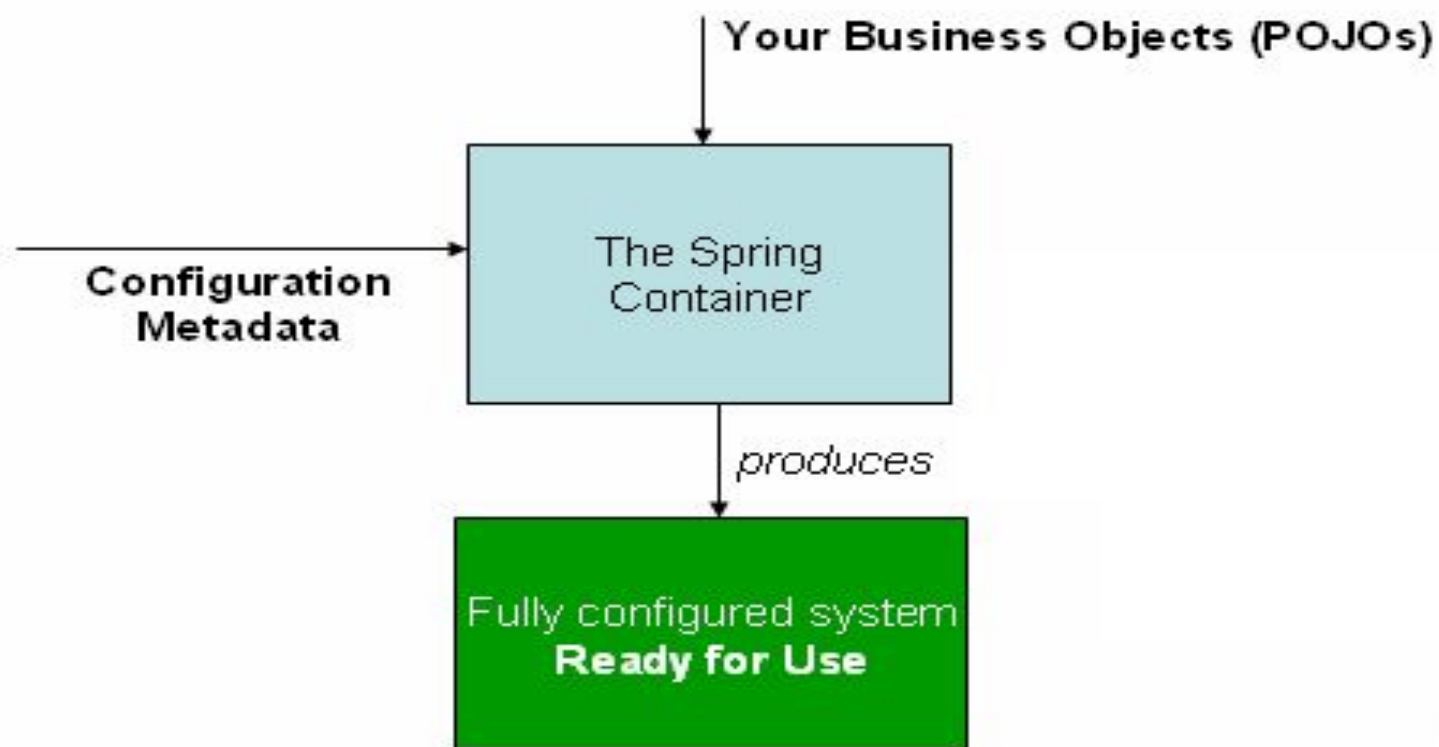




## Container Overview

Several implementations of the `ApplicationContext` interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of [`ClassPathXmlApplicationContext`](#) or [`FileSystemXmlApplicationContext`](#).

While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.





## Configuration Metadata

The Spring IoC container consumes a form of *configuration metadata*; this configuration metadata represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format.



## Configuration Types

[Annotation-based configuration](#): Spring 2.5 introduced support for annotation-based configuration metadata.

[Java-based configuration](#): Starting with Spring 3.0, many features provided by the [Spring JavaConfig project](#) became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import` and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata shows these beans configured as `<bean/>` elements inside a top-level `<beans/>` element.

```
<?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>
```



## Instantiating a Container

The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

`ApplicationContext context =`

```
new ClassPathXmlApplicationContext("services.xml");
```



## Composing XML-based configuration metadata

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load bean definitions from another file or files.

  
`<beans>` `<import resource="services.xml"/>` `<import resource="resources/messageSource.xml"/>` `<import resource="/resources/themeSource.xml"/>` `<bean id="bean1" class="..."/>` `<bean id="bean2" class="..."/>``</beans>`





## Using the Container

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. Using the method `T getBean(String name, Class<T> requiredType)` you can retrieve instances of your beans.

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.class);

// use configured instance
List userList service.getUsernameList();
```



## Dependency Injection

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application.

*Dependency injection* (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then *injects* those dependencies when it creates the bean.



## Dependency Injection

This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the *Service Locator* pattern.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies.

The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.



# Dependency Injection

DI exists in two major variants, [Constructor-based dependency injection](#) and [Setter-based dependency injection](#).



# Spring Annotation

<https://javaconceptoftheday.com/spring-annotation-based-configuration/>

Deprecated annotations:

<https://data-flair.training/blogs/spring-annotation/>

# Spring Annotation



The `<context:component-scan />` tag in the Spring framework is a powerful feature used to detect and register beans automatically within the Spring container.

## Detects Bean Annotations:

- It scans the specified package(s) for classes annotated with the following annotations:
  - **@Component**: Generic stereotype for any Spring-managed component.
  - **@Repository**: Specialization of **@Component** for DAO (Data Access Object) components.
  - **@Service**: Specialization of **@Component** for service layer components.
  - **@Controller**: Specialization of **@Component** for MVC controllers.
  - **@RestController**: Combination of **@Controller** and **@ResponseBody** for REST APIs.
  - **@Configuration**: Indicates that the class is a source of Spring bean definitions.



# Spring Annotation

It processes dependency injection annotations like:

- **@Autowired**: Automatically injects a bean by type.
- **@Qualifier**: Specifies which bean to inject when multiple candidates exist.
- **@Inject** (from JSR-330): Similar to **@Autowired**.
- **@Value**: Injects values from properties or environment variables.

# Spring Bean Autowiring



Bean wiring corresponds to providing the dependencies a bean might need to complete its job. In Spring, beans can be wired together in two ways : **Manually** and **Autowiring**.

**Manual wiring** : using `ref` attribute in `<property>` or `<constructor-arg>` tag

In this approach, we use the 'ref' attribute to refer to exact bean we want to be wired. This is the cleanest approach, and easy to understand

```
<bean id="driver" class="com.websystique.spring.domain.Driver">
  <property name="license" ref="license"/>
</bean>
<bean id="license" class="com.websystique.spring.domain.License" >
  <property name="number" value="123456ABCD"/>
</bean>
```





## Spring Bean Autowiring

**Autowiring** : using `autowire` attribute in `<bean>` tag.

```
<bean id="application" class="com.websystique.spring.domain.Application"  
autowire="byName"/>
```

In this approach, beans can be automatically wired using Spring autowire feature. There are 4 supported options for autowiring.

1. `autowire="byName"`
2. `autowire="byType"`
3. `autowire="constructor"`
4. `autowire="no"`



## Spring Bean Autowiring


1. `autowire="byName"` : Autowiring using property name. If a bean found with same name as the property of other bean, this bean will be wired into other beans property
2. `autowire="byType"` : If a bean found with same type as the type of property of other bean, this bean will be wired into other beans property
3. `autowire="constructor"` : If a bean found with same type as the constructor argument of other bean, this bean will be wired into other bean constructor
4. `autowire="no"` : No Autowiring. Same as explicitly specifying bean using 'ref' attribute



[https://www.tutorialspoint.com/spring/spring\\_utowiring\\_byname.htm](https://www.tutorialspoint.com/spring/spring_utowiring_byname.htm)

[https://www.tutorialspoint.com/spring/spring\\_utowiring\\_bytype.htm](https://www.tutorialspoint.com/spring/spring_utowiring_bytype.htm)

[https://www.tutorialspoint.com/spring/spring\\_utowiring\\_byconstructor.htm](https://www.tutorialspoint.com/spring/spring_utowiring_byconstructor.htm)



Explain spring bean lifecycle and write a program using declarative approach.(Content beyond Syllabus)

<https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/>

Explain types of dependency injection with an example for each.

- a. setter
- b. Constructor

Explain spring core architecture

[https://www.tutorialspoint.com/spring/spring\\_architecture.htm](https://www.tutorialspoint.com/spring/spring_architecture.htm)

Explain the concept of POJO programming.