**SPRING CORE**

* **Spring** : spring framework is a **Dependency Injection framework** to make java application loosely coupled. Spring was **develoved by Rod Johnson in 2003** and it is the most used and popular framework of java for **J2EE** or application development.

Spring provides alot of modules such as **Spring MVC, Spring Security, Spring Core**. With the help of these our application development becomes more easy.

1. **What is loosely coupled?**

**Loosely coupled** means We can make changes easily in our application.

1. **Why spring is called as dependency injection framework?**

Because it **injects dependencies** or objects itself.

Spring provides **IOC** with the help of which we perform **dependency injection**.

1. **What is dependency injection?**

It is the main functionality provided by Spring **IOC**. Dependency injection is a **design pattern** and a **core part of IOC**, by following which we can develop applications.

**Dependency** means one class is dependent on another class to do the work. In java we create object using new keyword. And if we do this then our application will become **tightly coupled**.

So what does Spring do is, the object we were creating using **new** keyword. Now it will be done by **dependency injection**. It will automatically create the object at **runtime** and will **inject** that object in another class.

**Dependency Injection** is a specific implementation of the IoC principle. It refers to the technique of passing **(injecting) dependencies** (objects) into a class at runtime rather than the class creating them itself. This can be done through **constructor injection, setter injection, or method injection**.

**Advantages:**

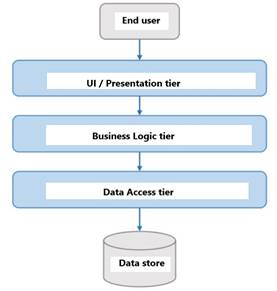
1. **Clean code**
2. **Decoupling** is more effective.
3. Classes become **easier to test**.
4. **What is IOC?**

**Inversion of control** is a design principle in which the control of object creation and management is transferred from the **application code to a container** or framework.

1. **Where this design pattern (Dependency Injection) required?**

It is particularly useful in **scenarios where decoupling components, managing configurations, and improving code quality are priorities**. By leveraging DI, developers can create more **flexible** and **robust applications** that are easier to manage and evolve over time.

1. **Design pattern / N-tier pattern / Layered architecture pattern**

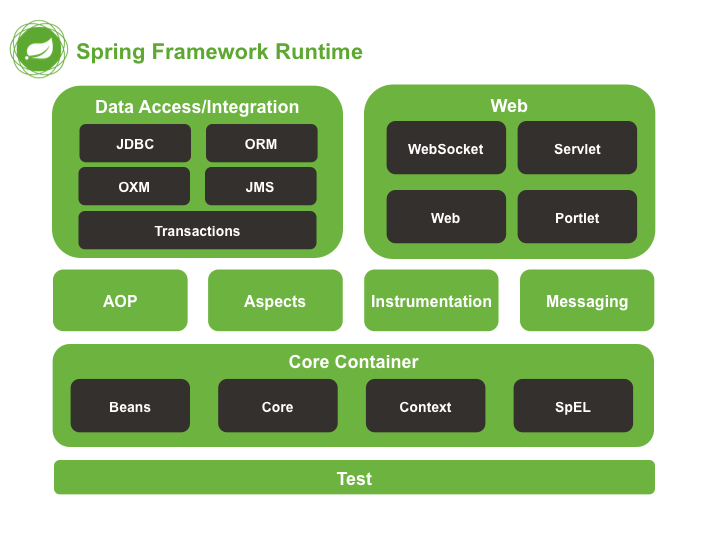


we always follow design patterns while developing a application. This process happens on server side.

1. **UI Layer (ProductController) :** It is a simple java **class**. This class needs to access some services or some business logics. It has the capabilities to accept requests. It will simply use the services of other class (**ProductService).** We do not write logic in this class.
2. **Business/Service Layer (Product Service) :** Business logics are written there and the **ProductController** will simply use these logics. Business layer does not directly communicate with database. It only provides business services.
3. **Data Access Layer (ProductDao) :** It communicates with database and send that to service layer.

**Spring Dependency Injection** will create the object of **ProductDao** and will inject it to **ProductService**.And will create the object of **ProductService** and will inject it to **ProductController**. This process will take place by I**OC container.**

* **Spring Modules :**



1. **Core container :** The core container consists of the Core, Beans, Context, and Expression Language modules.

* The **Core and Beans** modules **provide** the **fundamental parts of the framework**, including the IoC and Dependency Injection features. The **BeanFactory** is the advanced version of the factory pattern. As it removes the need of singletons and allows us to decouple our code. We can set up (or configure) how our objects should be created and what they depend on (like what tools they need to work) separately from the main part of our program.
* The **Context** module **inherits the features from bean module** and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting.
* The **spEL** (**Spring Expression Language**) module provides a powerful expression language for querying and manipulating an object graph at runtime.

1. **Data Access/Integration :** The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

* The **JDBC** module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding(Repetitive JDBC code) and parsing of database-vendor specific error codes(understanding and handling error codes from different databases).
* The [**ORM**](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/orm.html#orm-introduction) module provides integration layers for popular object-relational mapping APIs, including [JPA](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/orm.html#orm-jpa), [JDO](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/orm.html#orm-jdo), and [Hibernate](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/orm.html#orm-hibernate). Using the ORM package we can use all of these O/R-mapping frameworks.
* The [**OXM**](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/oxm.html) module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
* The **Java Messaging Service (**[**JMS**](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/jms.html)**)** module contains features for producing and consuming messages.
* The [**Transaction**](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/transaction.html) module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs (plain old Java objects).

1. **WEB :** The Web layer consists of the Web, Web-Servlet, WebSocket and Web-Portlet modules.

* Spring’s **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring’s remoting support.
* The **Web-Servlet** module contains Spring’s model-view-controller ([MVC](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/mvc.html#mvc-introduction)) implementation for web applications. Spring’s MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.
* The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

1. **AOP and Instrumentation :**

* Spring’s **AOP aspect-oriented programming** implementation allows us to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
* The **Instrumentation**module provides class instrumentation support and classloader implementations to be used in certain application servers.
* The **messaging** **application** serves a foundation for messaging based application. There are so many annotations in this to map messages with methods.

1. **Test** : the **Test** module provides support for unit testing and integration testing with Junit and TestNG. It also provides mock objects that you can use to test your code in isolation (seperately).

* **Spring IOC Container :**

**IOC** is also known as DI. It is a process whereby objects define their dependencies. 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 inverse (in traditional programming, the app. code calls liberaries or frameworks but with **IOC**, the framework calls the app. code). hence it is called **Inversion of control**. It is responsible for some work such as Object creation, holding that objectin memory, and injectingthe object in another object. It maintains the overall lifecycle of an object(from creation to destruction). The control of object is transferred from the application code to container.

for this, we have to provide two things to **IOC container**:

1. **Beans** : the beans or java classes that the container has to manage.
2. **Config** : (config files or XML configuration) it specifies how these beans are related and how their dependencies should be injected.

**Spring container** will use this configurations and will create the objects and will perform the injection. And then our app. code will use these objects.

* **Application context : AC** is a sub-interface of **Bean factory.** It adds easier integration with Spring’s AOP features such as WebApplicationContext for use in web applications.

this context basically extends the Bean factory**.** It also has some additional properties but as AC is an interface we can not create its object. But it is common to create an instance of

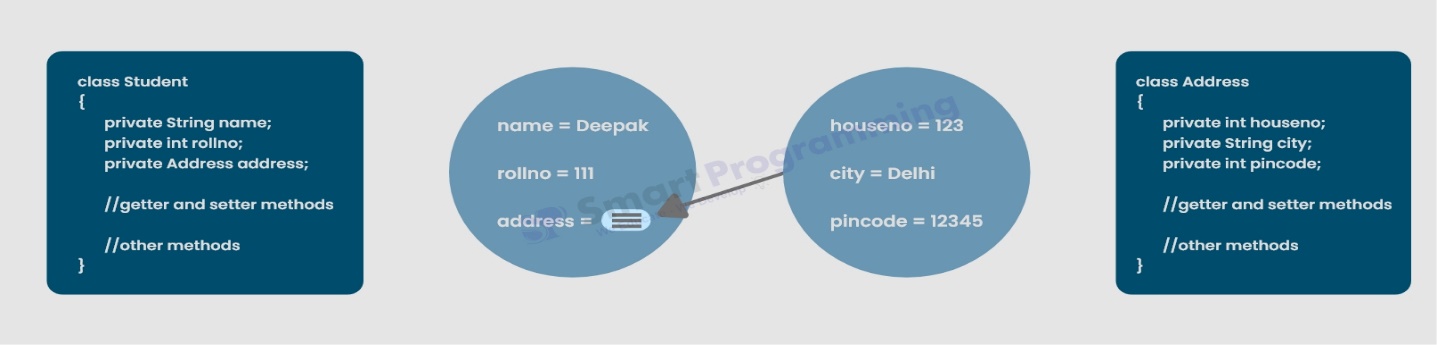
* **ClasspathXMLApplicationContext,**
* **AnnotationConfigApplicationContext,** and
* **FileSystemXMLApplicationContext.**

So that we will be able to get values/object from container.

1. **ClasspathXMLApplicationContext** searches XML configuration from java classpath.
2. **AnnotationConfigApplicationContext** searches beans on which we have used annotations. When using annotations we use this context.
3. **FileSystemXMLApplicationContext** searches configuration files from file system. We searching any config file from file system we use this context.

but we mostly use ClasspathXMLApplicationContext because we write XML and keep them in classpath.

* **How the DI is perfomed by IOC Container?**



**Injection:** inserting/injecting a obj in another object.

The student class needs the object of Address class. It means we can say that, Student class is totally dependent on Address class. So we will have to perform DI with the help of IOC container.

Steps:

1. The container will create an obj of Address class and will set the values of all the variables automatically.
2. Then it will create the obj of Student as well and will set the values of all the variables automatically.
3. Then it will inject the obj of Address class in Student class.
4. When it will inject all the dependencies at runtime. then we can ask for the obj of Student class from the container for use.
5. All the dependency management will be performed automatically at runtime by IOC container. and the obj of Student class is ready to use.

* **Ways of Injecting Dependencies :**

The DI can be done in two ways by IOC container :

1. Using **Setter Injection(**property Injection**)**
2. Using **Constructor Injection**

To set any value IOC will use only these two ways.