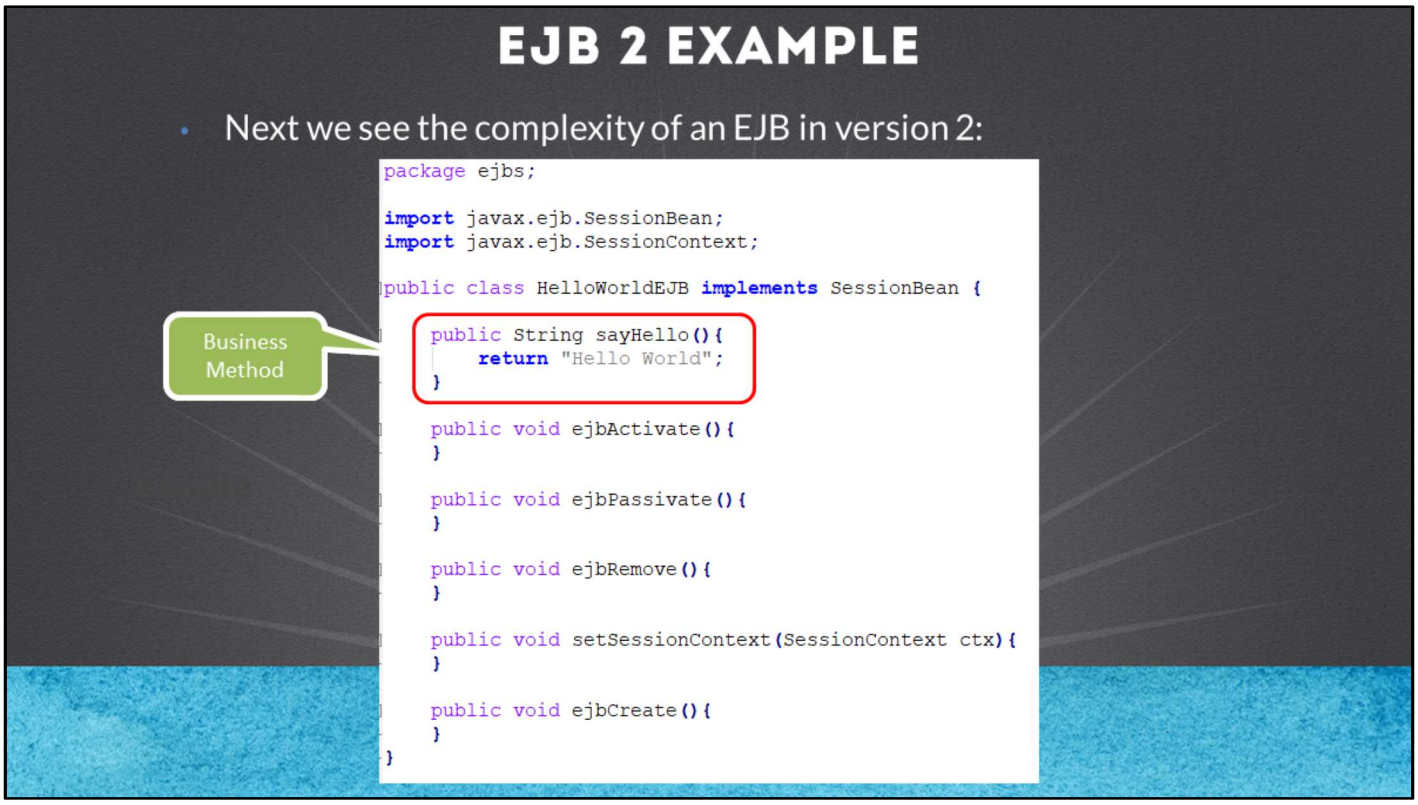
Hello, Ubaldo Acosta greets you again. I hope you're ready to start with this lesson.

We are going to study we go basics of Spring Framework.

Are you ready? Come on!



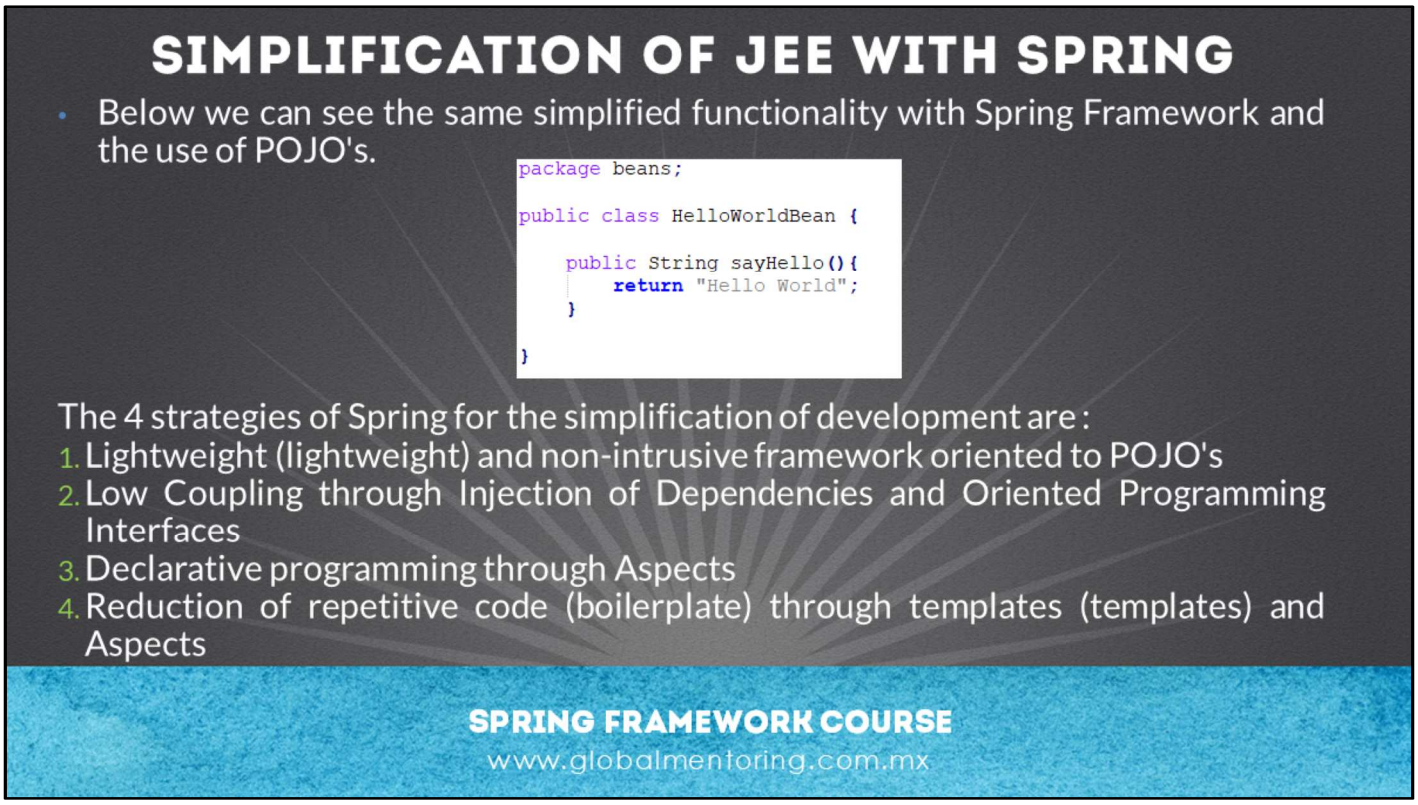
Spring is an open source project, originally created by Rod Johnson and described in his book Expert One-on-One: J2EE Design and Development. Spring was created with the aim of simplifying the complexity of J2EE business applications, allowing the use of POJOs or JavaBeans when using it, with the aim of adding functionality that was only possible with EJB's.

As we can see in the figure, when defining an EJB in version 2, we were OBLIGED to implement an interface and some other classes, and if that were not enough, we had to add several methods to comply with the EJB interface contract.

This type of solution is known as Intrusive code, which means that we must adapt to the required code, instead of focusing on our business.

We can see that the only business method is sayHello() method. All other methods have nothing to do with our business, however to be able to comply with the specification of the EJB 2.x, all the other methods shown must be added.

This was just one of many reasons why Spring was created, among others was to simplify the way to test, promote the development oriented to Interfaces, use of pure classes of Java (POJO's), Injection of Dependencies, among several others.



Spring allows you to simplify not only some layer in a Java architecture, but helps Simplify Development by programming with Java in its entirety.

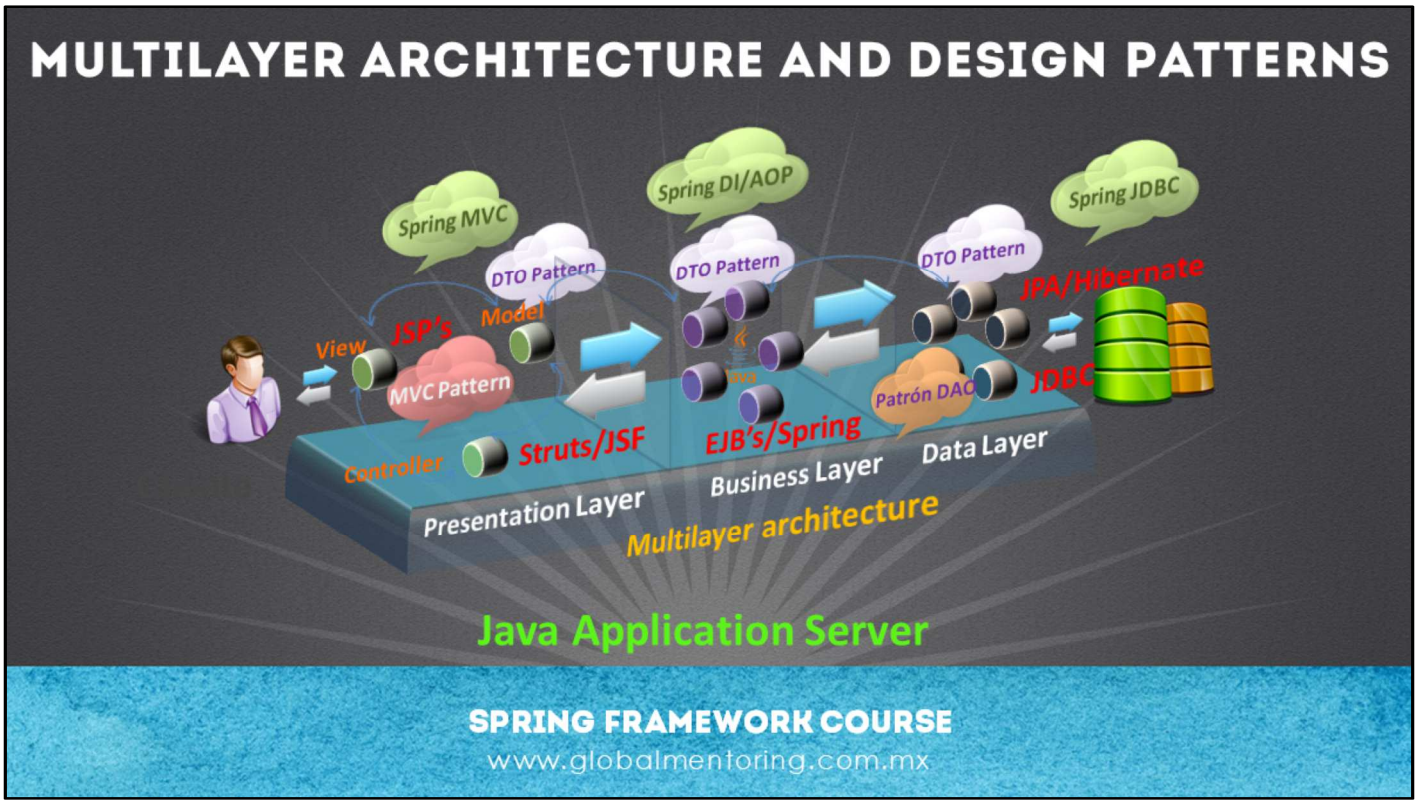
However, Spring is not limited to business applications, but to any standard Java application.

As we can see in the figure, our Java classes, unlike the EJB, only has the business method that really interests us.

Spring, through its main characteristics as DI (Dependency Injection) and AOP (Aspect Oriented Programming) allows adding functionality that only an EJB could have previously, that is, transactionality, security, logging, among many other features, this only with the use of POJO's (Plain Old Java Objects) or what is the same, pure classes of Java.

The 4 strategies of Spring for the simplification of development are:

1. Lightweight (lightweight) and non-intrusive framework oriented to POJO's
2. Low Coupling through Injection of Dependencies and Oriented Programming Interfaces
3. Declarative Programming through aspects
4. Reduction of repetitive code (boilerplate) through templates (templates) and aspects



Java developments are usually medium to large systems, so it is necessary to divide the responsibilities because the teams of programmers is also usually large.

A Java architecture is usually divided into 3 layers: Presentation Layer, Business Layer and Data Layer, there may be more, but these are the most representative. By dividing our application into layers, it is possible to delegate responsibilities to a team of programmers, provide better maintenance, integrate changes more quickly, among many other benefits.

On the other hand, a design pattern is a guide, which in turn allows us to solve a problem that occurs repetitively. When we talk about the layers of a JEE architecture (Java Enterprise Edition), each layer can have several design patterns as we can see in the figure.

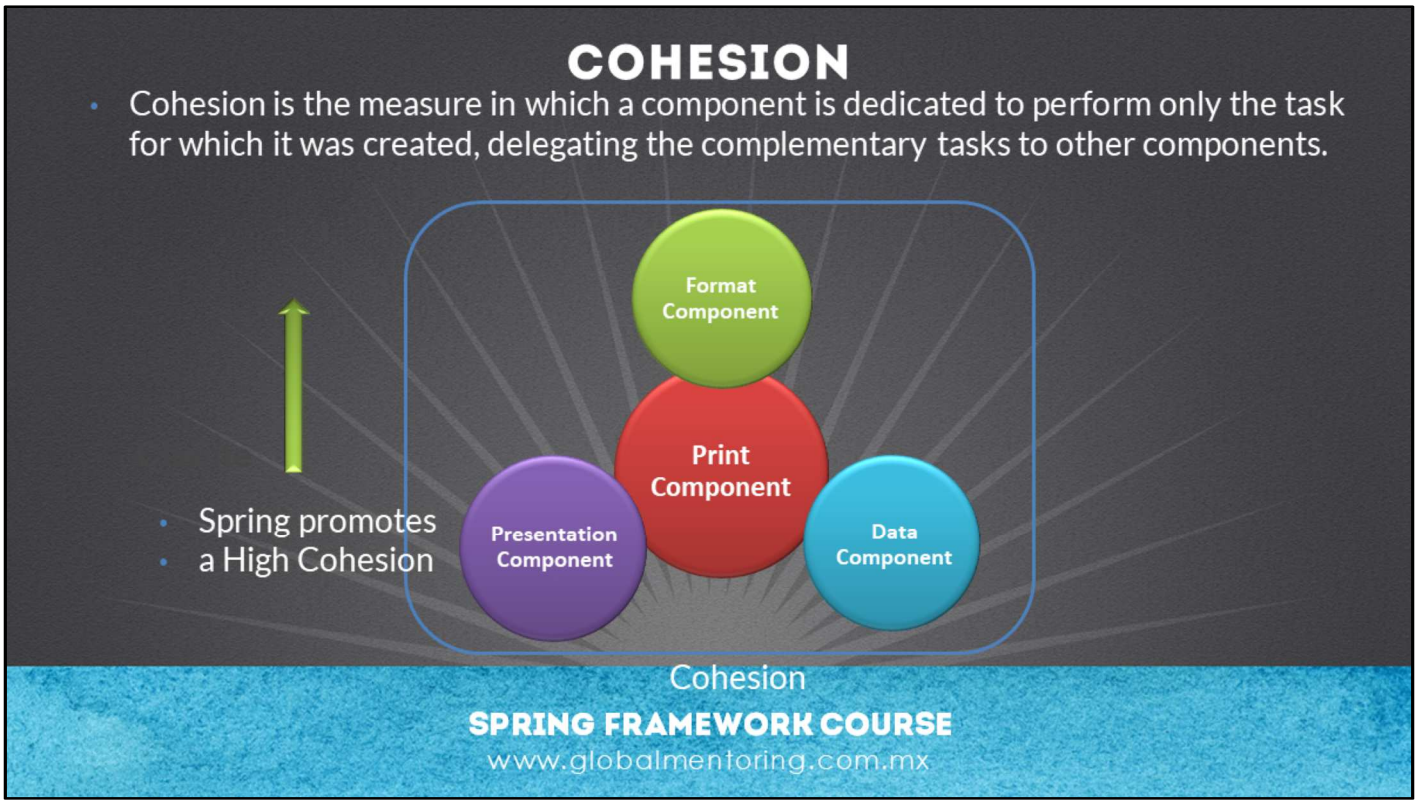
In the presentation layer we can observe the MVC (Model View Controller) pattern, and its objective is to divide the responsibilities in these 3 areas.

The DTO (Data Transfer Object) pattern represents an object in the domain of the problem. This pattern appears in the 3 layers because it is used to transfer an entity or a list of entities of a certain type between the different layers of the application.

In the data layer we have the DAO pattern, which helps us extract and store information in the database, using the DTO and / or entity objects.

In the Service or Business layer, we have the pattern as Business Delegate which handles the details in the call to some service method. At the same time we have the Service Locator pattern, which is used by the Business Delegate pattern to locate the services when the Java JNDI directory is used, however with Frameworks such as Spring and the Dependency Injection, these types of patterns are no longer used.

There is an extensive catalog of design patterns for Java, and throughout the course we will apply the ones mentioned in the figure.



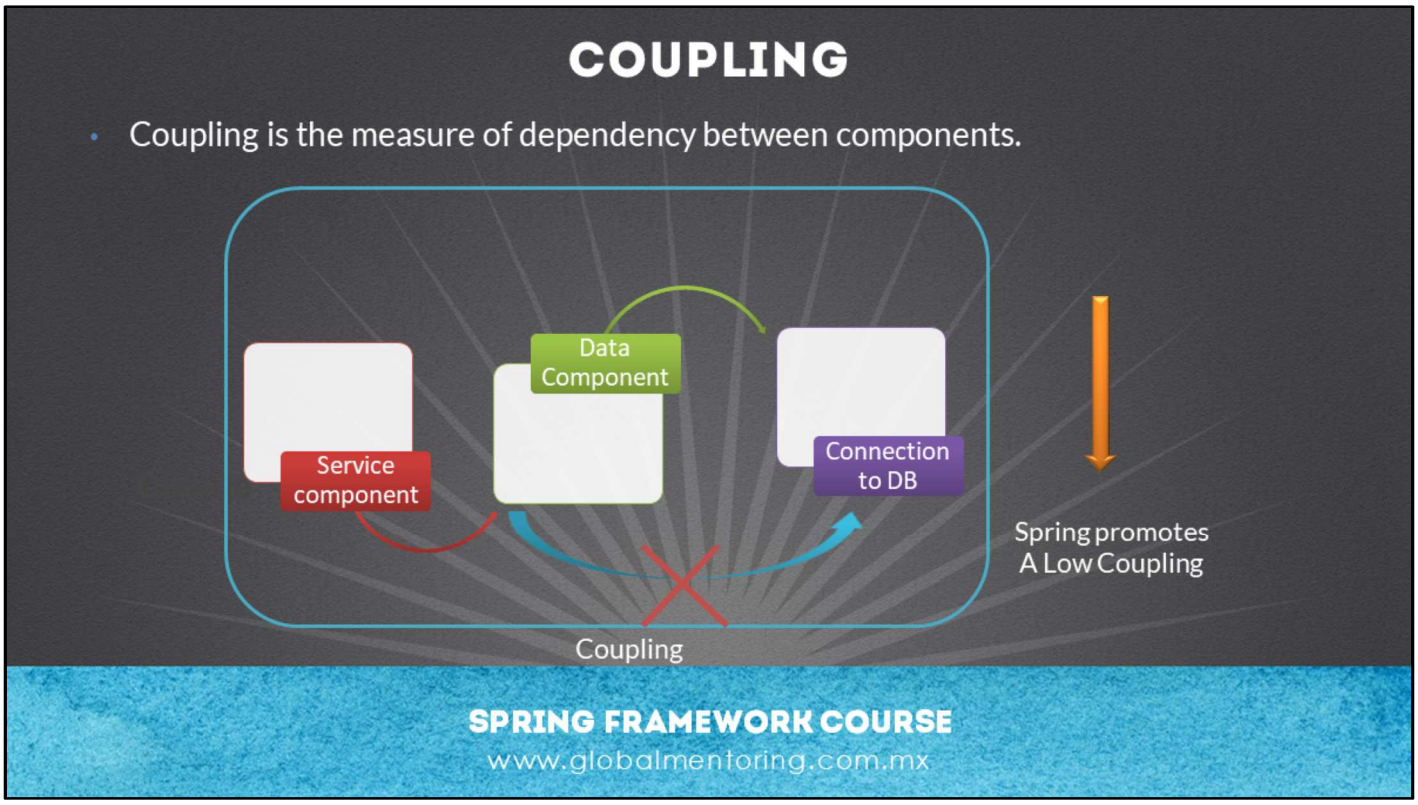
Cohesion and coupling play a central role in software design. Subsequent changes will be required when designing our modules, so the design and architecture of the software can directly impact the time and cost associated with making those changes.

Cohesion is the measure in which a component is dedicated to carrying out only the task for which it was created, delegating the complementary tasks to other components.

As we can see, the "Component of Printing" needs other components to be able to perform its task adequately, since if it performed all the tasks itself, any change or modification would be more complicated to perform, taking more time and cost maintenance .

In the development of software, it is sought that the components have a High Cohesion, because if the components perform a single task, it will be easier to integrate them into new systems and / or reuse their code, minimizing the time and cost of maintaining said component.

Spring by default promotes the development of systems with a high cohesion, applying the concepts of DI (Dependency Injection) and AOP (Aspect Oriented Programming), which we will study in detail in later lessons.



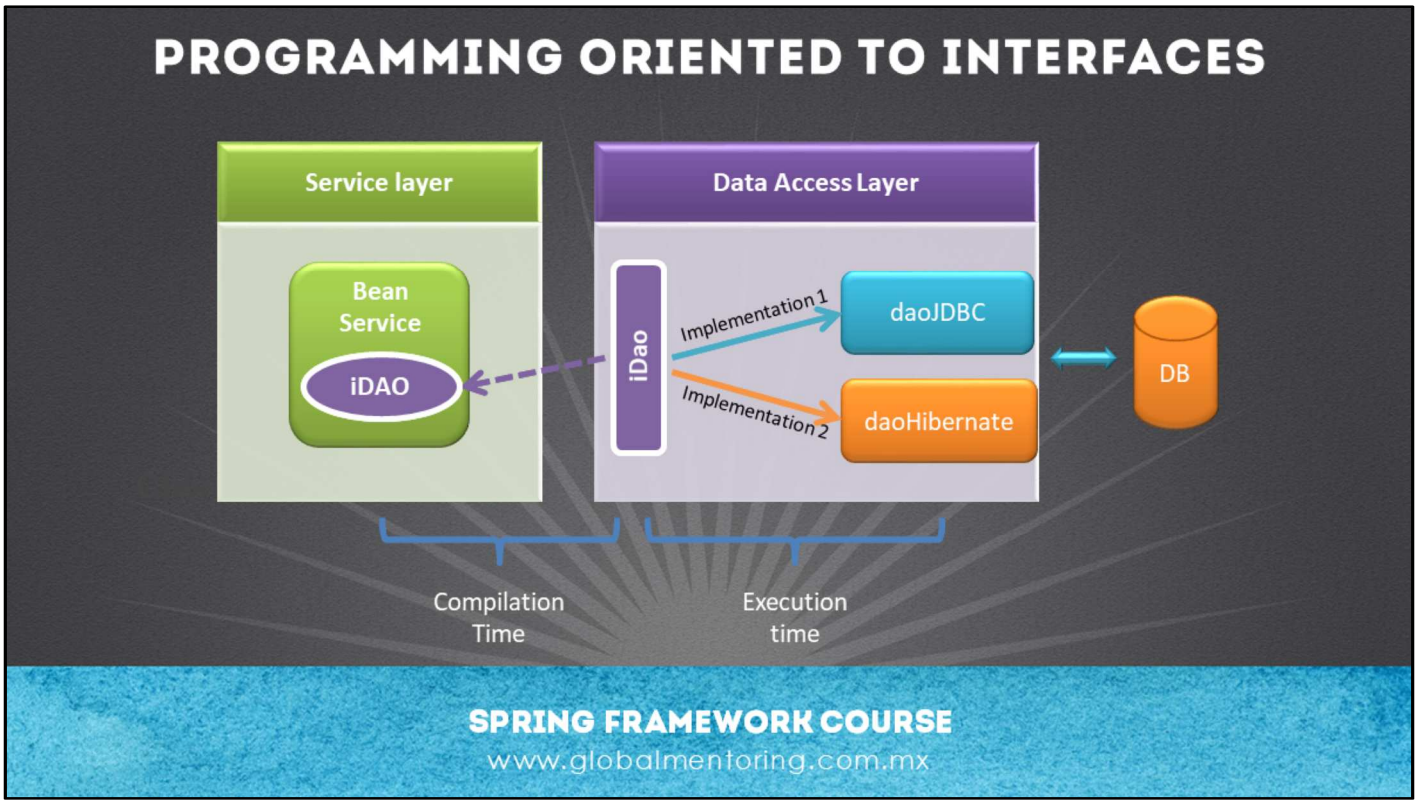
The coupling measures the degree of dependence between two or more elements. These elements can be modules, classes or any software component.

From the point of view of software design, we look for a low coupling, because the fewer relationships exist between components, the easier it will be to maintain and / or reuse those components in other systems.

The more relationships there are between the components, the harder it will be to separate them, and therefore their dependence will be very strong.

It must be taken into account that a high cohesion can cause a high coupling, because a component that is small (high cohesion) needs more elements to complete a task, so that the number of relationships among the components increases (high coupling), it is therefore necessary to introduce the concept of balance, which allows a balance between the concepts of cohesion and coupling.

Spring by default promotes the development of systems with low coupling and high cohesion, applying the concepts of DI (Dependency Injection) and AOP (Aspect Oriented Programming).



Programming oriented to Interfaces means that we can change the implementation of some kind programmatically or declaratively at run time.

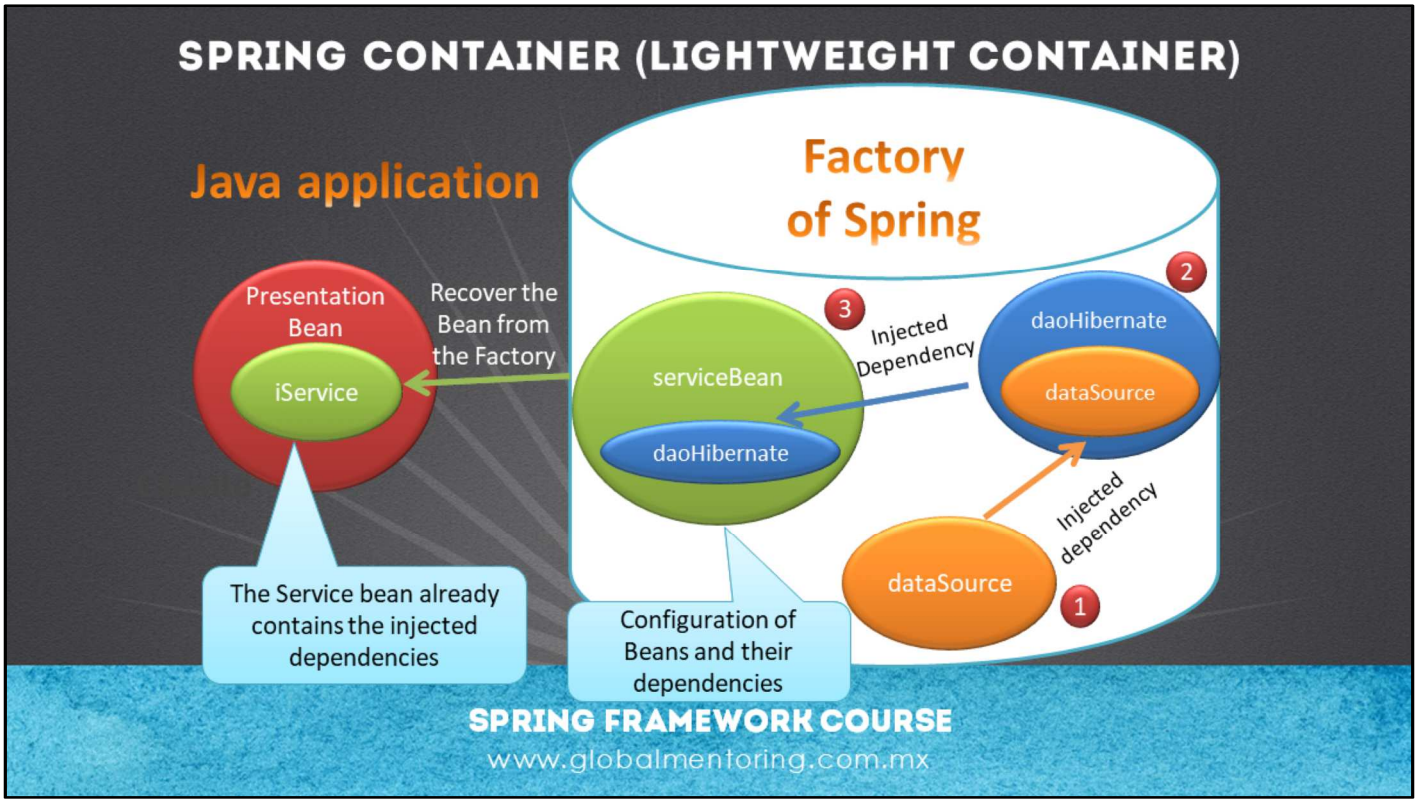
In the figure shown we can see an example involving the service layer and the data layer. In this example, the service layer uses a type of the iDAO interface within the Service Bean.

Subsequently, and in execution time, an implementation of this interface must be injected. In the figure we visualize that an implementation of the DAO can be injected using either JDBC or Hibernate, although in reality it can be any technology that implements this interface.

The benefits of using Interface types instead of specific classes are several:

- The service bean is not affected if the implementation is changed.
- The implementation can use real or test data sources.
- It allows to perform unit tests more easily, in this case on the Service Bean.
- It allows to reduce the dependency between the classes, in addition it is not required to know the detail of the implementation, among several other benefits.

The framework of Spring promotes the use of programming oriented to Interfaces and thus NOT directly use specific classes. This can be achieved either declaratively through the Spring xml descriptor or through annotations, which we will study later.



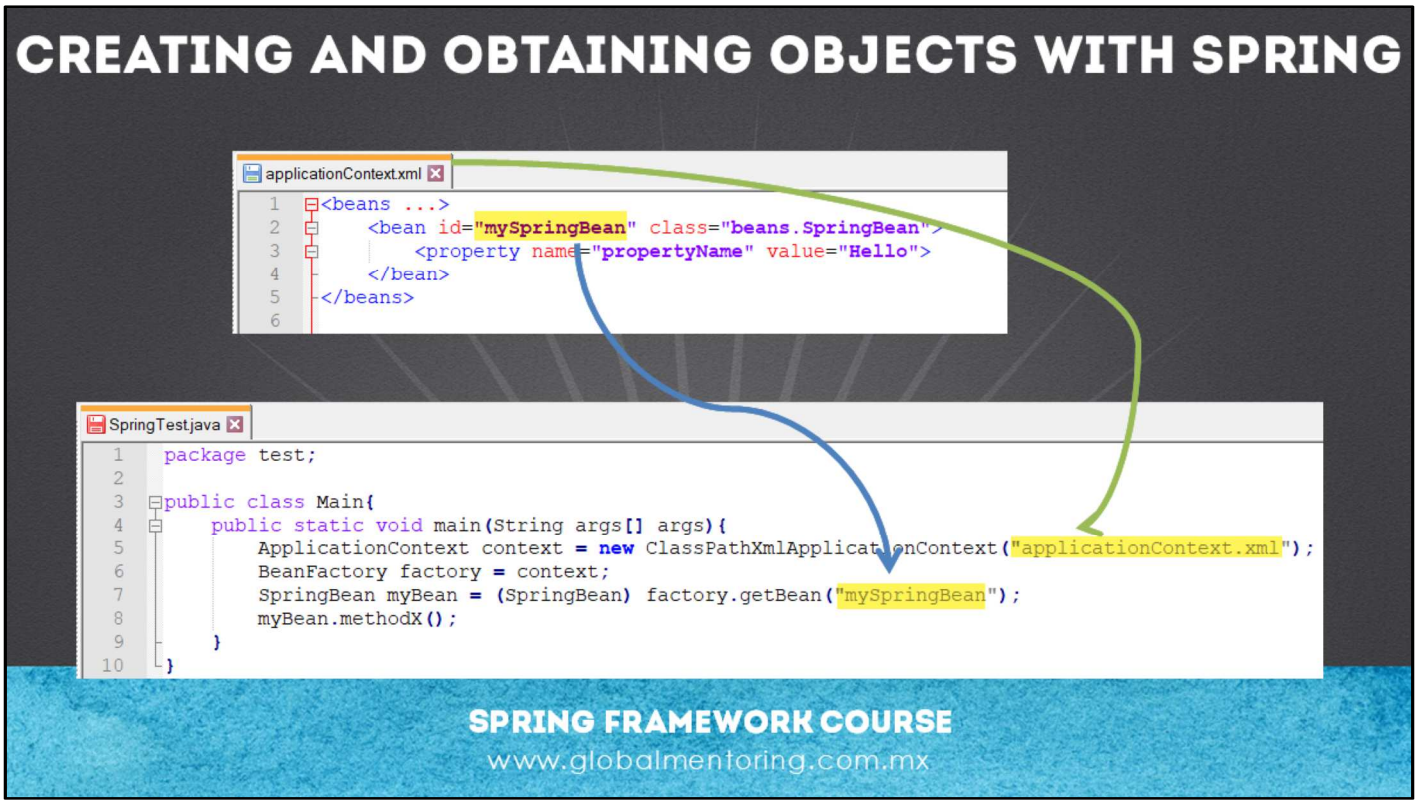
The Spring Container is an object factory. The main idea is to manage the objects and their dependencies by applying the concept of Dependency Injection. In addition, applying the AOP concept can modify a POJO in a much more robust class, for example, in a transactional POJO, and all this happens inside the container of Spring

As we can see in the figure, Spring through its factory is responsible for instantiating the objects that are necessary, in such a way that an object tree is created.

The configuration of these objects is done either in the Spring descriptor file, commonly called applicationContext.xml or through Spring annotations.

In the example we can see that a Bean of the Presentation layer needs to use a service from the Spring Factory. In turn, the factory reviews the dependencies of the requested object and instantiates each of the objects from least to most dependent (first dataSource and then daoHibernate).

In case the factory already has these objects, and depending on how it has been configured, it may no longer be instantiated, since by default the Singleton design pattern is applied, which means that there is only one instance of the Bean In Memory. We will review the Beans configuration in more detail.



Spring configuration can be done through the xml descriptor, usually known as applicationContext.xml.

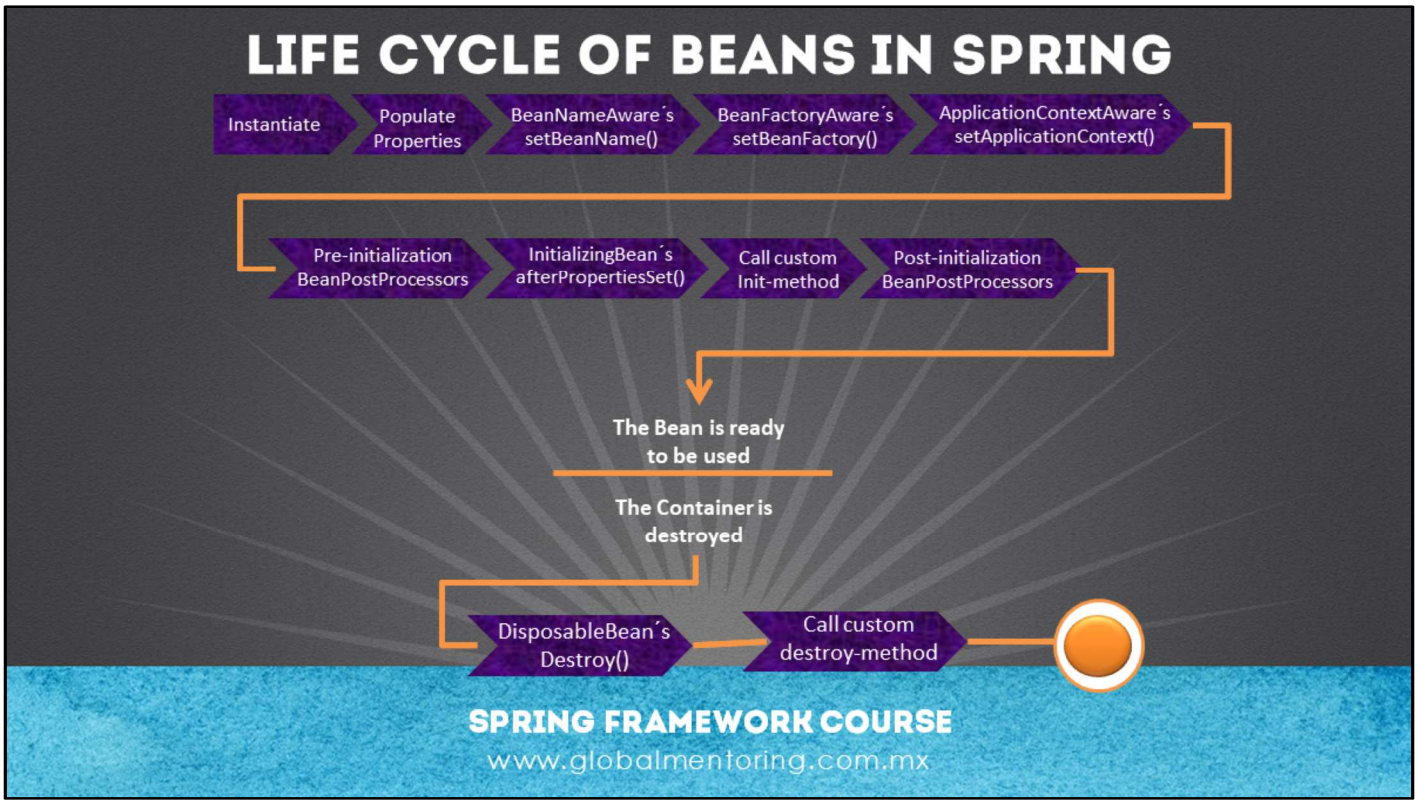
One of the advantages derived from using the Spring container is that it is no longer necessary to create the objects on our own, that is, we will no longer observe the new operator SpringBean() in our code.

The above allows to create cleaner code and delegate the configuration of the dependencies to Spring.

By creating and configuring the classes and dependencies using Spring allows writing modular code, easy to test, integrating dependencies very easily and promoting the use of POJOs, being able to strengthen them through AOP.

In the figure we can see an example of configuring a Spring Bean inside the applicationContext.xml file and, on the other hand, obtaining it within a Java class.

The above is just an example of what we will be doing in our configuration exercises with Spring, since there are many ways to configure it and obtain the objects from the Spring container.



The Life Cycle of a Bean in Spring follows a series of well-defined steps. We can observe in figure this series of steps.

The beginning of a bean in Spring is at the moment in which the instance of it is created and loaded in the Spring Application Context (Spring Factory). As you can see, the Spring factory executes several steps before a bean is ready to be used. The general steps are:

1. Spring instantiates the Bean.
2. Spring injects the values and references of the beans into the properties of the beans.
3. If the bean implements the BeanNameAware interface, Spring passes the bean ID to the setBeanName () method.
4. If the bean implements the BeanFactoryAware interface, Spring calls the setBeanFactory () method, passing the bean factory itself.
5. If the bean implements the ApplicationContextAware interface, Spring calls the setApplicationContext () method, passing a reference from the Application Context.
6. If the bean implements the BeanPostProcessor interface, Spring calls the postProcessBeforeInitialization () method.
7. If the bean implements the InitializingBean interface, Spring calls the afterPropertiesSet () method. Similarly, if the method contains the declaration of an init-method, then the declared method is called.
8. If the bean implements the BeanPostProcessor interface, Spring calls the postProcessAfterInitialization () method
9. At this point, the bean is ready to be used, and the bean remains in the factory until the application Context is destroyed.
10. If the bean implements the DisposableBean interface, Spring calls the destroy () method. Likewise, if the bean contains a destroy-method declaration, then the specified method is called.