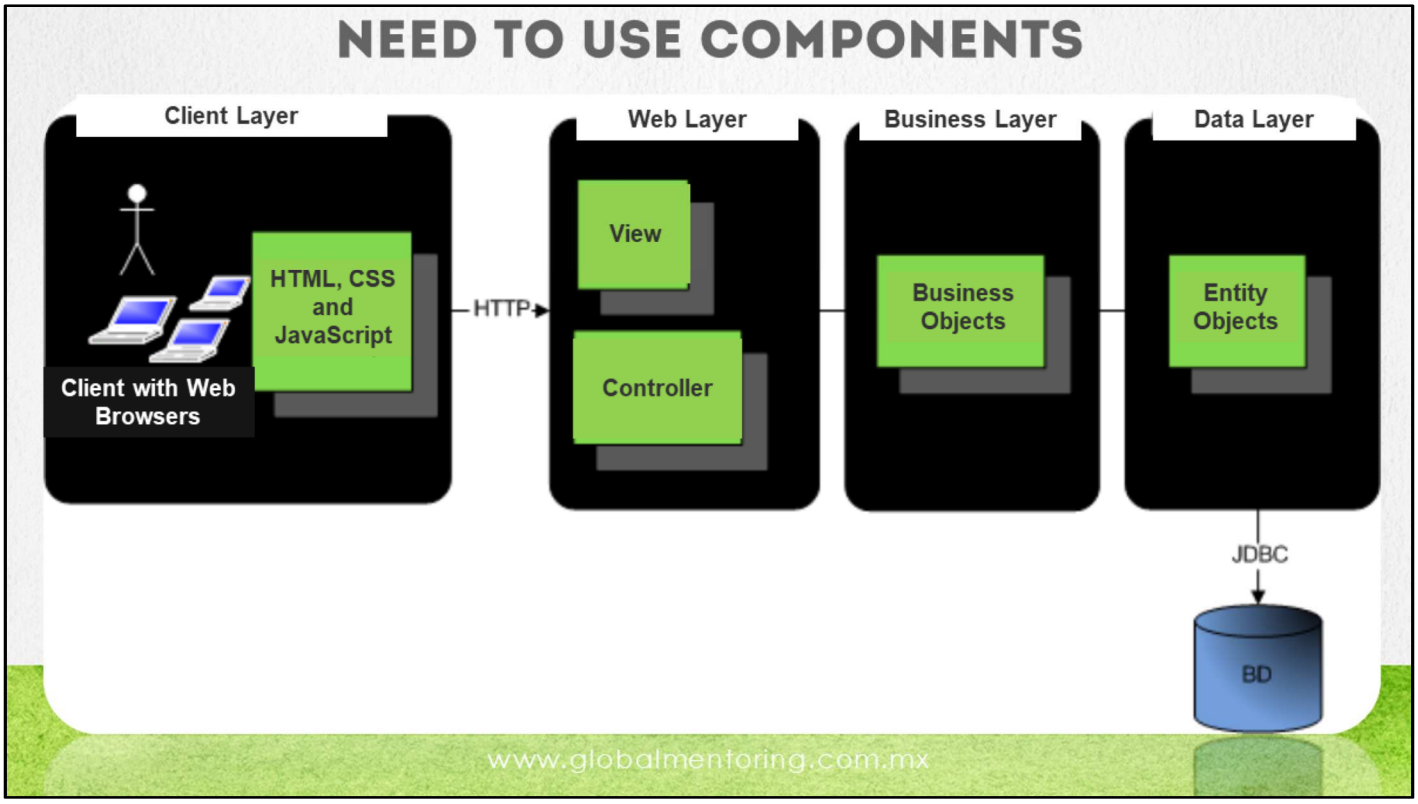
Hello, Ubaldo Acosta greets you. Welcome again. I hope you're ready to start with this lesson.

We are going to study the topic of Injection of Dependencies with Spring Framework.

Are you ready? OK let's go!



An application in Java is usually a medium-sized application and becomes very large. Therefore, the interaction between components is quite broad.

In the figure we observe a 3-layer architecture: Web Layer or Presentation, Business Layer and Data Layer.

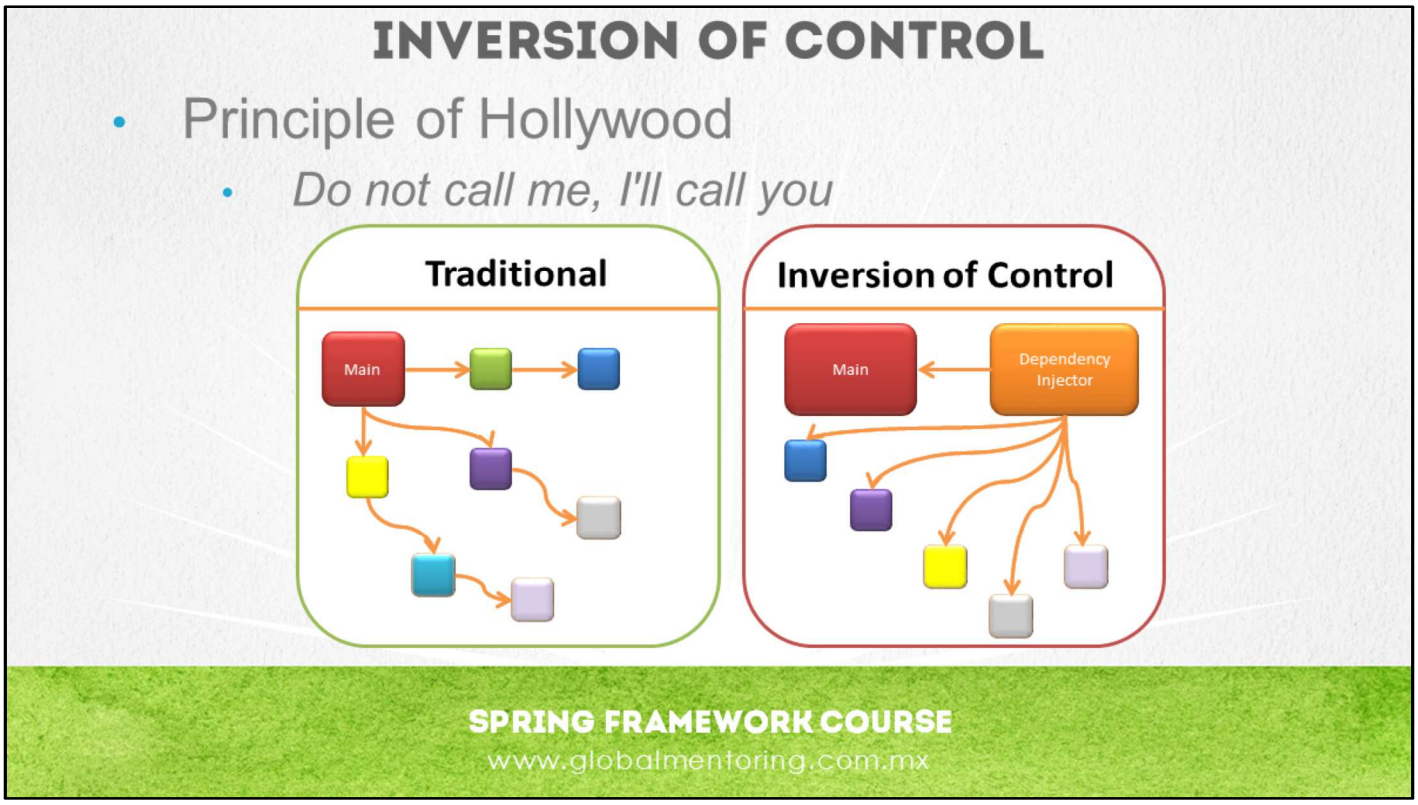
Each of the layers is responsible for a specific task. This division of responsibilities brings as benefits a low coupling and high cohesion.

Imagine a user's login process. Everything starts with the user's request, to which the result of a page is sent, such as login.jsp.

Once the user sends their data for validation, it is the Controller who begins the actual work of the backend or Java server. The controller is the orchestrator of the requests and responsible for selecting the view to be displayed to the user.

In turn, the controller is responsible for retrieving information from the system, known as Model or Domain Classes. This manages to communicate with the Business layer. This layer in turn applies the business rules necessary to send the response back. If you need information from the database, it is when the service layer communicates with the data layer and thus the return to the controller begins. Finally, the response to the client is generated and we can conclude the process.

As we can see the interaction between Java classes and the dependency between them is quite broad. This type of architecture is one of the many reasons why Spring is one of the most used frameworks, since it simplifies the development of Enterprise Java Architectures.



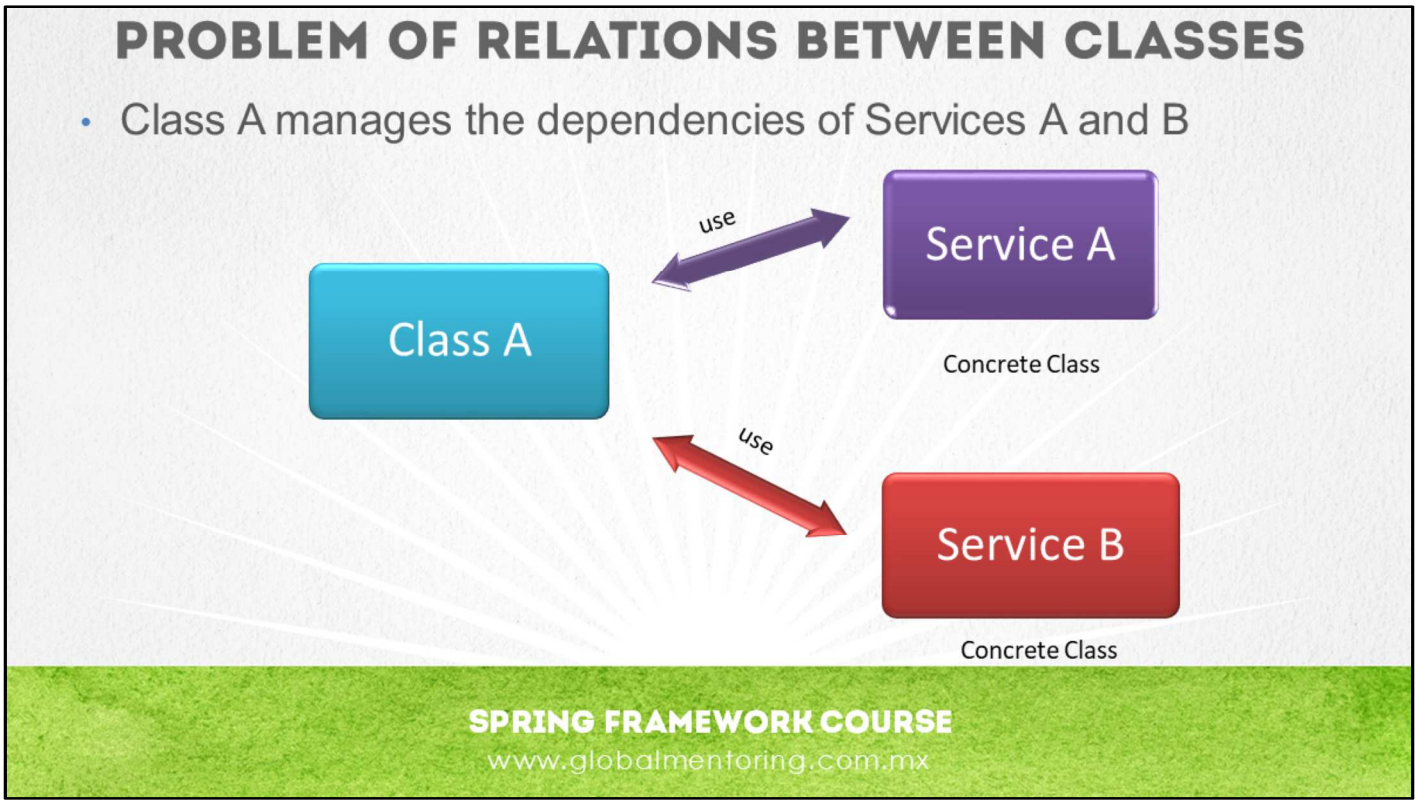
As we can see in the figure, in traditional programming a class (Main) is responsible for instantiating the objects with which it has dependency, and in turn each object is again responsible for instantiating its dependencies and completing them. This scheme leads to configurations that are not very flexible and difficult to maintain.

In the Inversion of Control scheme (IoC) we can see that the Main class uses an Object Factory (Container) and this in turn delivers the already configured objects, with everything and their dependencies. The container manages to complete these dependencies through the Hollywood Principle: Do not call me, I'll call you.

One of the greatest benefits of the control investment is the separation of modules and the way in which these components are connected. This allows to minimize the dependence between them.

In the figure the Dependency Injector (container) is responsible for reviewing each of the dependencies and completing tasks that were previously the responsibility of the Main class.

Dependency injection (DI) is a form of Control Inversion. The task of the container is to inject the dependencies to each object according to whether it has been configured in the Spring configuration file or through Java annotations.



The problem posed in the figure is that Class A has dependencies with 2 services. Each Service in turn must be instantiated and administered by Class A in order to be used.

The above leads to the following problems:

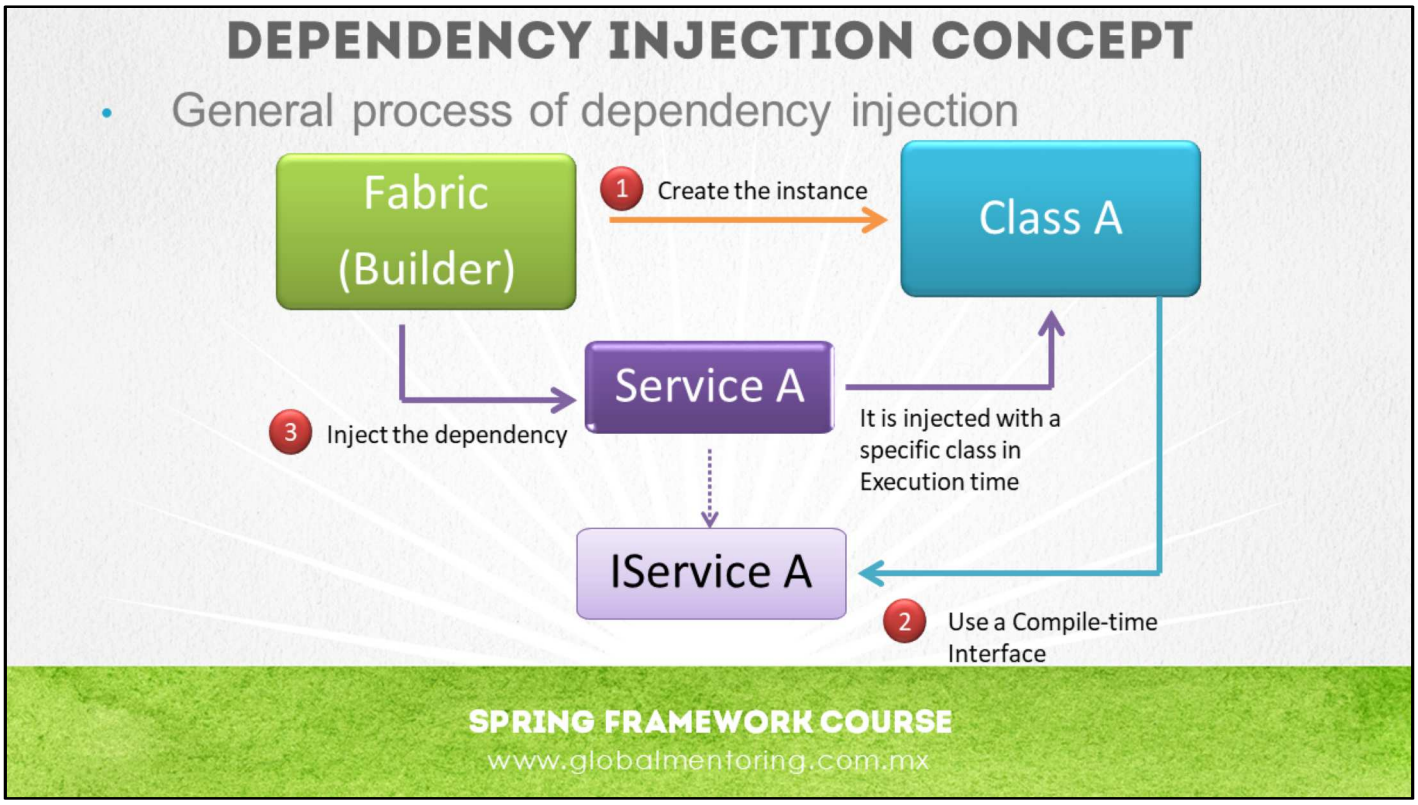
To replace or update the dependencies, the Class A code must be changed.

The concrete implementations of Services A and B must be available at compile time.

Classes are difficult to test (Unit Test) because they have direct dependencies. Which implies that services can not be replaced by stub-type classes or mocks.

Class A will usually contain repeated code to create, locate and manage services A and B.

Let's review below how we can solve this type of problems using Spring and the use of Interfaces.



In the figure we can see a general configuration for the injection of dependencies. There are two new concepts, the first is the concept of the Factory and the second is the management of interfaces for services.

The Factory or Builder (1) allows us to manage the dependencies that are needed to fully instantiate our classes, in this case Class A needs a dependency on Service A, which is provided by the factory (3).

As can be seen, Class A does not have a direct relationship with the concrete class (Service A) but with an IService A (2) interface.

The above allows us to more efficiently achieve the following points:

- Decouple dependencies, allowing change or update at any time the implementation of services.
- Write classes that depend on other classes and whose implementation is not known at compile time.
- To be able to test the classes in an isolated way, without using the dependencies.
- Eliminate the responsibility of the classes to create and locate the dependencies.

Spring does not have a single container of beans, but we can select among several, which are classified into 2 broad categories:

- Bean Factories, defined by the interfaces org.springframework.beans.factory.BeanFactory
- Application Contexts, defined by the interface: org.springframework.context.ApplicationContext

The Bean Factories usually turn out to be very low level for most applications, therefore we will work and study containers of the Application Contexts type.

To instantiate the factory of the beans, it is necessary to first load the descriptor file, commonly called applicationContext.xml, into an object of type Resource, which is an interface.

Subsequently, we must instantiate the bean factory object by providing the obtained Resource type.

Our application interacts directly or indirectly with the BeanFactory object in order to obtain Spring beans including their dependencies.

The BeanFactory object when it is created, reads (parses) the Spring xml descriptor file, and it is at this moment when the dependencies of the Beans are reviewed and injected.

Once the content of the xml descriptor has been read and loaded into memory, it is possible to obtain the Spring Beans via the BeanFactory or ApplicationContext object.

There are several ways to obtain the Spring Application Context container. The method to select depends on the type of class or integration that creates the instance of the Spring factory.

Here are some examples of each way to obtain the Spring factory:

- `ClassPathXmlApplicationContext:`
  `ApplicationContextcontext=new ClassPathXmlApplicationContext("spring.xml");`

- `FileSystemXmlApplicationContext:` Load the Spring file from the file system of our operating system.
  `ApplicationContext context = new FileSystemXmlApplicationContext("c:/spring.xml");`

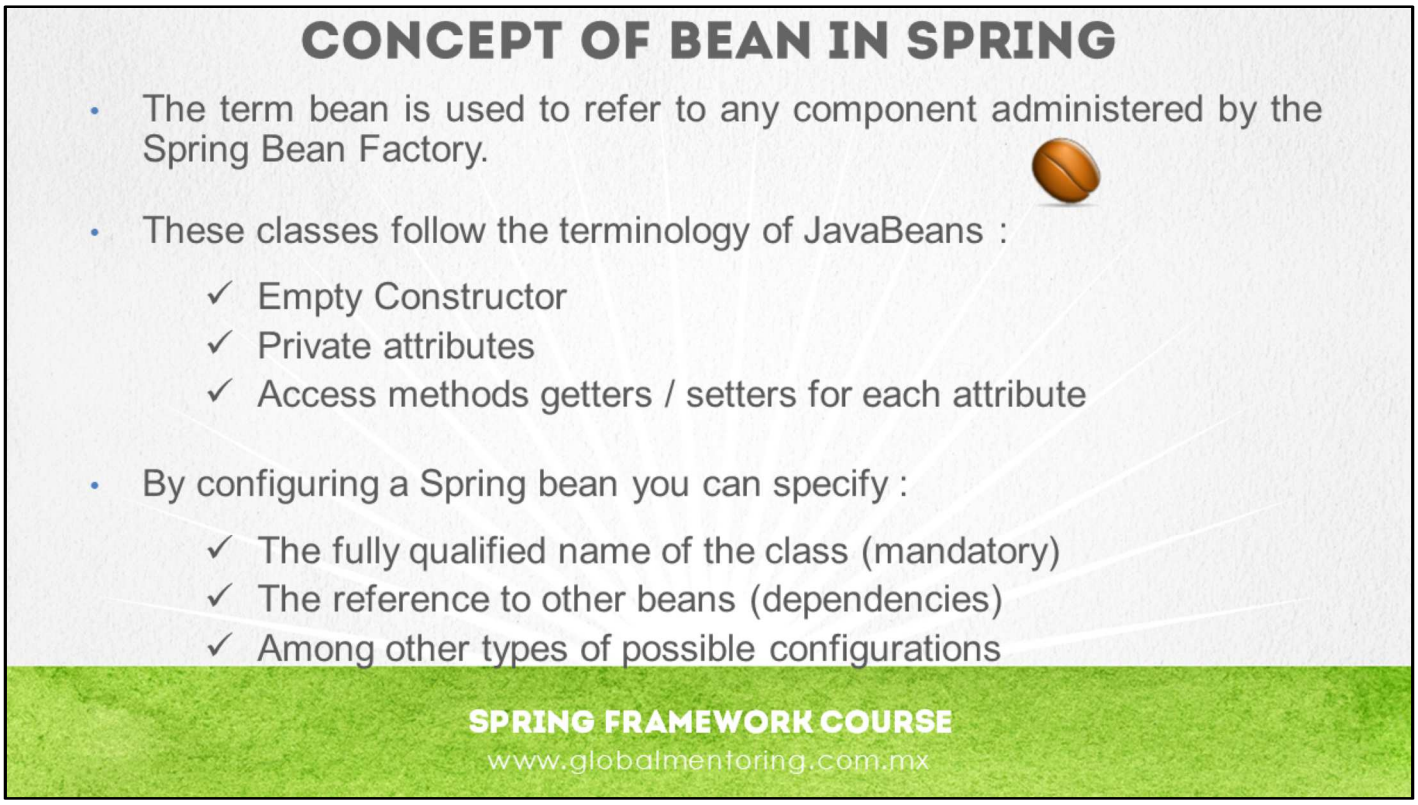
- `XmlWebApplicationContext: Inside the web.xml file we have:`

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The name of the factory file can be whatever we define.

Regardless of which method we have selected to start the Spring factory, once we have the instantiated object, we can request any of the beans from the factory, which already includes its dependencies, simply using the id of the previously declared bean.



The term bean is used to refer to any component administered by the Spring factory.

These classes follow the JavaBean terminologys:

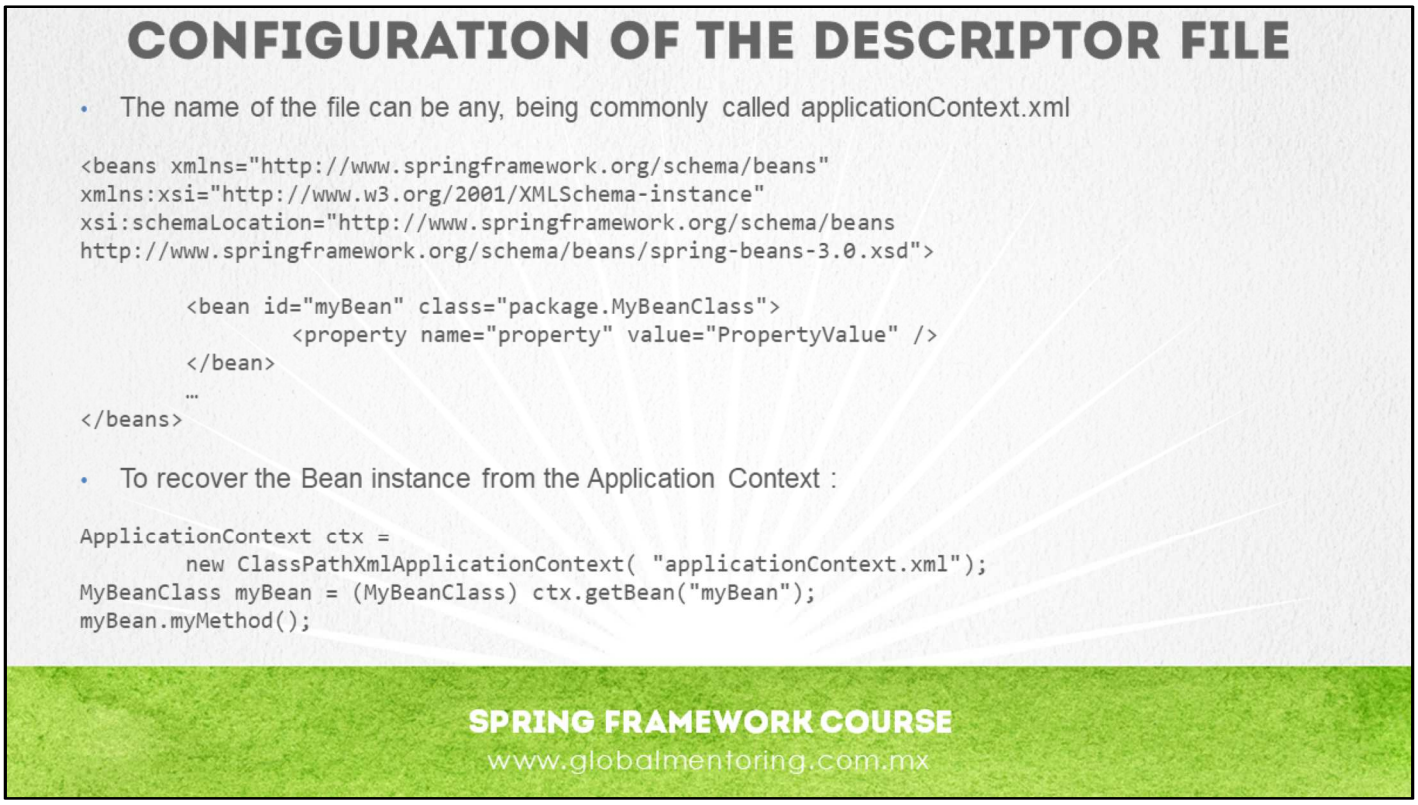
- Class with empty constructor or without arguments
- Private attributes
- For each private attribute, its get and set method is generated (getters and setters)

By configuring a Spring bean you can specify:

- The fully qualified name of the class (mandatory)
- The reference to other beans (dependencies)
- Among other types of possible configurations

When defining a bean in the Spring descriptor, we can specify the following attributes:

- Name of the class (class)
- Name of the Bean (name)
- Scope of the Class (scope)
- Arguments of the constructor (constructor arguments)
- Properties (properties)
- Auto-injection mode (autowiring)
- Delayed start mode (lazy-initialization mode)
- Initialization method of the bean (initialization method)
- Method of destruction of the bean (destruction method)



Spring, like many Java frameworks, uses an xml file to configure many of the features of the framework. In the most recent versions, many of the features described in this file are being replaced by the use of annotations, which simplify the configuration process and use the xml file to a lesser extent.

The name of the Spring descriptor file can be any, being commonly called applicationContext.xml. Normally the Spring configuration file is placed in the root of the classpath for easy access.

Spring allows to configure the descriptors supporting the previous versions, the only restriction would be that the new features of the latest versions would not work in our project.

Each bean is defined within a <bean> tag within the <beans> tag. The id attribute is used to assign the bean name and must be unique. The class attribute specifies the type of the bean (bean class) from which the instance is to be generated.

It is possible to use the name attribute to specify alternative bean names (separated by a comma). However, the name that takes priority is the attribute id.

To retrieve the Bean instance from the Application Context, we can use the following code:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext( "applicationContext.xml");
MyBeanClass myBean = (MyBeanClass) ctx.getBean("myBean");
myBean.myMethod();
```

NAMESPACES IN SPRING

Namespace	Purpose
aop	Provides elements to declare aspects in Spring.
beans	Enables the declaration of beans and how they should be linked.
context	It contains elements to configure the Spring application context, including self detection of beans and auto-linking (autowired).
jee	Provides integration with the Java EE API, such as JNDI and EJB.
jms	Provides the configuration of Java classes of type message-driven.
lang	It talks about the declaration of beans that are implemented with languages like Groovy, JRuby or BeanShell.
mvc	Enables the Spring MVC features including annotations.
oxm	Supports the configuration for the object-to-xml mapping in Spring.
tx	Provides the configuration for transactional management in a declarative way.
util	Provides a set of utilities, for example declaration of beans as a collection.

An XML namespace allows you to define uniquely identifiable elements within an XML configuration file. This same technique is applied by Spring to use several of the features supported by the Framework. We can roughly understand a namespace as an import of a Java class, which allows us to add certain features to our Java application.

The namespaces specify the characteristics that will be used in the project, for example: Security, Transactional Management, Bean Autowired, etc.

Spring Framework by default contains 10 namespaces, as we can see in the table. Each of them allows to enable features of the Framework as described.

In addition to the namespaces that come by default in Spring, several of the Spring portfolio projects, such as Spring Security, Spring Web Flow, and spring Dynamic Modules provide their own namespaces.

As we advance in the course, we will put into practice several of these namespaces, and thus be able to review the characteristics that they offer us.


ONLINE COURSE
SPRING FRAMEWORK
By: Eng. Ubaldo Acosta
Global Mentoring
SPRING FRAMEWORK COURSE
www.globalmentoring.com.mx