https://howtodoinjava.com/spring-core/spring-bean-life-cycle/

Intro to Inversion of Control and Dependency Injection with Spring

## **What Is Inversion of Control?**

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code

To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

* decoupling the execution of a task from its implementation
* making it easier to switch between different implementations
* greater modularity of a program
* greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

## **What Is Dependency Injection?**

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.

Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

Here's how we would create an object dependency in traditional programming:

**public** **class** **Store**

{

**private** Item item;

**public** **Store**()

{ item = **new** ItemImpl1();

}

}

In the example above, we need to instantiate an implementation of the *Item* interface within the *Store* class itself.

By using DI, we can rewrite the example without specifying the implementation of the Item that we want:

**public** **class** **Store** {

**private** Item item;

**public** **Store**(Item item)

{ **this**.item = item;

}

}

In the next sections, we'll look at how we can provide the implementation of *Item* through metadata.

Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

## **4. The Spring IoC Container**

An IoC container is a common characteristic of frameworks that implement IoC.

In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their life cycles.

The Spring framework provides several implementations of the *ApplicationContext* interface: *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext*for standalone applications, and *WebApplicationContext* for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Here's one way to manually instantiate a container:

ApplicationContext context

= **new** ClassPathXmlApplicationContext("applicationContext.xml");

To set the *item* attribute in the example above, we can use metadata. Then the container will read this metadata and use it to assemble beans at runtime.

**Dependency Injection in Spring can be done through constructors, setters or fields.**

## **5. Constructor-Based Dependency Injection**

In the case of [constructor-based dependency injection](https://www.baeldung.com/constructor-injection-in-spring), the container will invoke a constructor with arguments each representing a dependency we want to set.

Spring resolves each argument primarily by type, followed by name of the attribute, and index for disambiguation. Let's see the configuration of a bean and its dependencies using annotations:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Item **item1**() {

**return** **new** ItemImpl1();

}

@Bean

**public** Store **store**() {

**return** **new** Store(item1());

}

}

The *@Configuration* annotation indicates that the class is a source of bean definitions. We can also add it to multiple configuration classe

We use the *@Bean* annotation on a method to define a bean. If we don't specify a custom name, then the bean name will default to the method name.

For a bean with the default *singleton* scope, Spring first checks if a cached instance of the bean already exists, and only creates a new one if it doesn't. If we're using the *prototype* scope, the container returns a new bean instance for each method call.

Another way to create the configuration of the beans is through XML configuration:

<**bean** id="item1" class="org.baeldung.store.ItemImpl1" />

<**bean** id="store" class="org.baeldung.store.Store">

<**constructor-arg** type="ItemImpl1" index="0" name="item" ref="item1" />

</**bean**>

We use the *@Bean* annotation on a method to define a bean. If we don't specify a custom name, then the bean name will default to the method name.

For a bean with the default *singleton* scope, Spring first checks if a cached instance of the bean already exists, and only creates a new one if it doesn't. If we're using the *prototype* scope, the container returns a new bean instance for each method call.

Another way to create the configuration of the beans is through XML configuration:

<**bean** id="item1" class="org.baeldung.store.ItemImpl1" />

<**bean** id="store" class="org.baeldung.store.Store">

<**constructor-arg** type="ItemImpl1" index="0" name="item" ref="item1" />

</**bean**>

We can combine constructor-based and setter-based types of injection for the same bean. The Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

## **7. Field-Based Dependency Injection**

In case of Field-Based DI, we can inject the dependencies by marking them with an @Autowired annotation:

**public** **class** **Store** {

@Autowired

**private** Item item;

}

While constructing the Store object, if there's no constructor or setter method to inject the Item bean, the container will use reflection to inject Item into Store.

We can also achieve this using [XML configuration](https://www.baeldung.com/spring-xml-injection).

This approach might look simpler and cleaner, but we don't recommend using it because it has a few drawbacks such as:

* This method uses reflection to inject the dependencies, which is costlier than constructor-based or setter-based injection.
* It's really easy to keep adding multiple dependencies using this approach. If we were using constructor injection, having multiple arguments would make us think that the class does more than one thing, which can violate the Single Responsibility Principle.

More information on the @Autowired annotation can be found in the [Wiring In Spring](https://www.baeldung.com/spring-annotations-resource-inject-autowire) article.

## **8. Autowiring Dependencies**

[Wiring](https://www.baeldung.com/spring-annotations-resource-inject-autowire) allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been defined.

There are four modes of autowiring a bean using an XML configuration:

* **no:** the default value – this means no autowiring is used for the bean and we have to explicitly name the dependencies.
* **byName:** autowiring is done based on the name of the property, therefore Spring will look for a bean with the same name as the property that needs to be set.
* **byType:** similar to the byName autowiring, only based on the type of the property. This means Spring will look for a bean with the same type of the property to set. If there's more than one bean of that type, the framework throws an exception.
* **constructor:** autowiring is done based on constructor arguments, meaning Spring will look for beans with the same type as the constructor arguments.

For example, let's autowire the item1 bean defined above by type into the store bean:

@Bean(autowire = Autowire.BY\_TYPE)

**public** **class** **Store** {

**private** Item item;

**public** **setItem**(Item item){

**this**.item = item;

}

}

We can also inject beans using the @Autowired annotation for autowiring by type:

**public** **class** **Store** {

@Autowired

**private** Item item;

}

If there's more than one bean of the same type, we can use the @Qualifier annotation to reference a bean by name:

**public** **class** **Store** {

@Autowired

@Qualifier("item1")

**private** Item item;

}

Now let's autowire beans by type through XML configuration:

<**bean** id="store" class="org.baeldung.store.Store" autowire="byType"> </**bean**>

Next, let's inject a bean named item into the item property of store bean by name through XML:

<**bean** id="item" class="org.baeldung.store.ItemImpl1" />

<**bean** id="store" class="org.baeldung.store.Store" autowire="byName">

</**bean**>

We can also override the autowiring by defining dependencies explicitly through constructor arguments or setters.

## **9. Lazy Initialized Beans**

By default, the container creates and configures all singleton beans during initialization. To avoid this, we can use the lazy-init attribute with value true on the bean configuration:

<**bean** id="item1" class="org.baeldung.store.ItemImpl1" lazy-init="true" />

Consequently, the item1 bean will only be initialized when it's first requested, and not at startup. The advantage of this is faster initialization time, but the trade-off is that we won't discover any configuration errors until after the bean is requested, which could be several hours or even days after the application has already been running.

# What is Spring bean?

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.

The management of a Spring bean includes:

* creating an object
* providing dependencies (e.g. other beans, configuration properties)
* intercepting object method calls to provide additional framework features
* destroying an object

## **How to define Spring bean?**

By providing **bean definitions**.

**Bean definitions tell Spring which classes the framework should use as beans**. But that’s not all. Bean definitions are like recipes. They also **describe the properties of a bean**. We’re going to discuss properties later in the article. But before we go into that, let’s focus on the bean definitions.

There are **three different ways** in which you can define a Spring bean:

* annotating your class with the stereotype *@Component* annotation (or its derivatives)
* writing a bean factory method annotated with the *@Bean* annotation in a custom Java configuration class
* declaring a bean definition in an XML configuration file

### **Spring bean as @Component**

If you own the source code, you’ll usually use the *@Component* annotation directly on a class.

|  |  |
| --- | --- |
| 1  2  3  4 | @Component  class MySpringBeanClass {      //...  } |

At runtime, Spring finds all classes annotated with *@Component* or its derivatives and uses them as bean definitions. The process of finding annotated classes is called **component scanning**.

You probably wonder what are derivatives of *@Component*.

@Conpontent derivatives are Spring stereotype annotations which themselves are annotated with @Component. This fact allows us to use them in place of *@Component*.

The list of *@Component* derivatives includes:

* @Service
* @Repository
* @Controller

### **Using @Bean for factory methods**

For classes you don’t own, you have to create factory methods with the @*Bean* annotation in a custom bean configuration class. If you don’t want to make a class dependent on Spring, you can also use this option for classes you own.

Here is a sample configuration class with a single bean factory method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Configuration  class MyConfigurationClass {       @Bean     public NotMyClass notMyClass() {         return new NotMyClass();     }    } |

Spring uses factory methods to create actual objects at runtime. As you can see, the method simply returns a new instance of a class. If such class belongs to some external library and you can’t annotate it with *@Component*, creating a factory method with *@Bean* is the only option.

The *@Configuration* annotation also comes from Spring. Actually, it’s a derivative of *@Component* but with a special purpose. The annotation marks the class as a container of @Bean definitions. You can write multiple factory methods inside a single configuration class.

## **Spring bean properties**

At this point, you already know how to mark a class as a Spring bean. The next step is to learn how to customize the features of a bean.

No matter which bean definition method you choose, they all allow describing the same set of bean properties. Properties give details about **how** Spring should create objects.

The list of Spring bean properties includes:

* class
* name
* dependencies
* scope
* initialization mode
* initialization callback
* destruction callback

To simplify bean definitions, Spring provides default values for almost all of their properties. However, it’s important to know how to customize defaults. Let’s investigate Spring bean properties one by one.

### **Bean class**

When you create a bean definition, you connect it with a single **concrete class** in your application. This class itself is the main property of a bean.

The class property is the default identifier of a bean when Spring looks for dependencies. Does that mean you can’t have multiple bean definitions for a single class? No. It’s possible. But in that case, to avoid ambiguity, you should use another bean identifier: a name.

### **Bean name**

Spring bean name is a custom string that Spring uses to **identify beans**. Unlike the bean class, names must be **unique** across the whole application. You can’t define two beans with the same name, even if their types are different.

Fortunately, you don’t have to set name for each bean you create. Spring generates names at runtime for its internal use. Unless you want to identify beans by names, you can safely go with the default setup.

The main situation in which you need to use bean names is having **several beans for the same class** defined with the *@Bean* annotation. In this case, **names allow you to identify a particular instance** which you want to use as a dependency of another bean.

#### How to name a Spring bean?

Use the *name* attribute of the *@Bean* annotation. Here’s an example with two beans with the same type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Configuration  class MyConfigurationClass {       @Bean(name = "myBeanClass")     MyBeanClass myBeanClass() {         return new MyBeanClass();     }       @Bean(name = "anotherMyBeanClass")     MyBeanClass anotherMyBeanClass() {         return new MyBeanClass();     }    } |

In practice, you don’t define bean names very often. Having multiple beans for a single class is rather a rare situation. Yet, it’s good to know about the possibility to name a bean.

### **Bean dependencies**

Objects used as beans may use other beans to execute their jobs. When Spring creates an object which defines some dependencies, the framework needs to create these dependencies first. These dependencies can also have their own dependencies.

In Object-oriented application, we usually work with a **huge graph of related objects**. Fortunately, we don’t have to think about how to construct this graph. We don’t have to think about the order in which objects should be created. Spring does all of that for us.

The only thing that Spring expects from you is the list of dependencies for a particular bean.

#### How to define bean dependencies?

Bean dependency definition is a complex topic which deserves a separate article. Consider the following paragraphs as an introduction to the topic.

When you have a class marked with *@Component* and there is only one constructor, Spring uses the list of constructor parameters as a list of mandatory dependencies. By default, the framework uses constructor parameter types to provide appropriate objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Component  class BeanWithDependency {       private final MyBeanClass beanClass;       BeanWithDependency(MyBeanClass beanClass) {         this.beanClass = beanClass;     }    } |

In the past, we used the *@Autowired* annotation on constructors. However, since Spring 4.3 it’s not mandatory if there is only one constructor. Yet, **if a bean class defines multiple constructors**, you should mark one with *@Autowired*. This way Spring knows which constructor contains the list of bean dependencies.

By the same token, a bean factory method can define its dependencies. Spring calls the method with appropriate objects.

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean  BeanWithDependency beanWithOptionalDependency(MyBeanClass beanClass) {     return new BeanWithDependency(beanClass);  } |

### **Bean scope**

The scope of a Spring bean defines **how many instances** of a particular class the framework creates at runtime. The scope also describes **on what condition** a new object is created.

Spring offers several scopes for your beans. The core of the framework comes with two:

* singleton – a single instance
* prototype – multiple instances

Moreover, Spring comes with additional bean scopes dedicated to web applications:

* request
* session
* global session
* application

**The default scope for all beans is the singleton.** When a bean has the singleton scope, Spring creates only one instance and shares it across the whole application. The singleton scope is a perfect choice for **stateless objects**. Nowadays, the vast majority of beans in our application are stateless singletons.

On the other hand, if your object contains a state, you should consider other scopes. To select the right one, you should ask yourself how long the framework should keep that state in memory. But that’s for another post.

#### How to set the scope of a Spring bean?

No matter if you annotate a class directly with *@Component* or create a factory method with *@Bean*, the method is the same. Use the *@Scope* annotation and its string attribute to select a scope.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component  @Scope("prototype")  class MyPrototypeClass {      //...  } |
| 1  2  3  4  5 | @Bean  @Scope("prototype")  MyPrototypeClass myPrototypeClass() {     return new MyPrototypeClass();  } |

What is more, for web scopes Spring comes with additional alias annotations. You can use these annotations as a replacement for *@Scope*:

* *@RequestScope*
* @SessionScope
* @ApplicationScope

### **Bean initialization mode**

When your application starts, Spring eagerly creates all singleton beans at the startup. This default behavior allows us to quickly detect errors in bean definitions. On the other hand, **eager bean initialization makes the startup of your application slow**.

Fortunately, you can delay the creation of beans to the moment when they are actually needed. You can do this using the *@Lazy* annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Component  @Lazy  class MyLazyClass {      //...  } |

Because the annotation is commonly misunderstood, I wrote a separate [article dedicated the *@Lazy* annotation](http://dolszewski.com/spring/spring-lazy-annotation-use-cases/).

### **Bean initialization callback**

You may want to run some object initialization logic once Spring creates a new instance based on the bean definition.

If this logic doesn’t depend on the framework, you can run it in the constructor of the object. However, to make sure the logic is run after Spring initializes the object (e.g. after optional dependency injection), you should use the initialization callback.

#### How to set bean initialization callback?

If you define a bean using the *@Component* annotation, you have two options:

* Make the bean class implement *InitializingBean*. The interfaces will force you to implement the initialization method.
* Write a custom initialization method and mark it with [the javax @PostContruct annotation](https://docs.oracle.com/javaee/7/api/javax/annotation/PostConstruct.html).

In both cases, Spring will run initialization callbacks for you.

What about beans defined with factory methods?

You set the initialization callback using *@Bean* annotation and its attribute called *initMethod*. The attribute expects a string with the name of an initialization method.

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean(initMethod = "someInitMethodName")  MySpringBeanClass meBeanClass() {     return new MySpringBeanClass();  } |

Interestingly, the method used as an **initialization callback can have private access control**. Spring uses the reflection mechanism to call the method.

### **Bean destruction callback**

Similarly to the initialization callback, you can define a method which Spring should call when a bean object is destroyed. Predestroy callbacks are much less used but it’s good to be aware they exist.

#### How to set bean destruction callback?

Again, if you have access to the source code of a bean class, you can go with one of two options:

* Implement the *DisposableBean* interface. Spring uses its only method a destruction callback.
* Write a custom method and annotate it with [@PreDestroy](https://docs.oracle.com/javaee/7/api/javax/annotation/PreDestroy.html) from the Javax API.

For factory method, you use the *@Bean* annotation and its *destroyMethod* attribute.

|  |  |
| --- | --- |
| 1  2  3  4 | @Bean(name = "myBeanClass", destroyMethod = "cleanUpMethod")  MySpringBeanClass meBeanClass() {     return new MySpringBeanClass();  } |

## **How does Spring create objects from bean definitions?**

Spring beans don’t live in a vacuum.

When you start a Spring application, the framework firstly creates a special object called ApplicationContext. The ApplicationContext, also known as the Inversion of Control (IoC)  container, is **the heart of the framework**.

The ApplicationContext is a container in which your bean objects exist.

## 1. What is Life Cycle of a Bean?

definition. The framework may also be required to perform some pre and post-initialization steps to get the bean into a usable state.

After that, when the bean is no longer required, it will be removed from the IoC container. Like the initialization phase, the Spring framework may need to perform pre-and post-destruction steps to free the other system resources.

The [Spring bean factory](https://howtodoinjava.com/spring-core/how-to-create-beans-using-spring-factorybean/) is responsible for managing the life cycle callbacks of the beans which are created in the spring containers.

### 1.1. Life Cycle Callback Methods

Spring bean factory controls the creation and destruction of beans. To execute some custom code, the bean factory provides the callback methods, which can be categorized broadly into two groups:

1. **Post-initialization** callback methods
2. **Pre-destruction** callback methods



Spring Bean Life Cycle