## Core spring 4.2, 4.3 certification

# Topics covered:

What is dependency injection and what are the advantages?

What is an interface and what are the advantages of making use of them in Java?

What is meant by “application-context” and how do you create one?

What is the concept of a “container” and what is its lifecycle?

Dependency injection using Java configuration

Dependency injection in XML, using constructor or setter injection

Dependency injection using annotations (@Component, @Autowired)

Component scanning, Stereotypes and Meta-Annotations

Scopes for Spring beans. What is the default?

What is an initialization method and how is it declared in a Spring bean?

What is a destroy method, how is it declared and when is it called?

What is a BeanFactoryPostProcessor and what is it used for?

What is a BeanPostProcessor and how is the difference to a

BeanFactoryPostProcessor? What do they do? When are they called?

Are beans lazily or eagerly instantiated by default? How do you alter this behavior?

What does component-scanning do?

What is the behavior of the annotation @Autowired with regards to field injection,

constructor injection and method injection?

How does the @Qualifier annotation complement the use of @Autowired?

What is the role of the @PostConstruct and @PreDestroy annotations? When will they

get called?

What is a proxy object and what are the two different types of proxies Spring can create?

What is the power of a proxy object and where are the disadvantages?

What are the limitations of these proxies (per type)?

How do you inject scalar/literal values into Spring beans?

How are you going to create a new instance of an ApplicationContext?

What is a prefix?

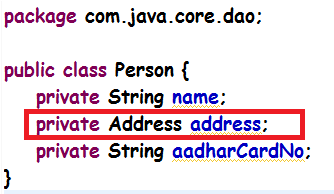
What is the lifecycle on an ApplicationContext?

What does the "@Bean annotation do?

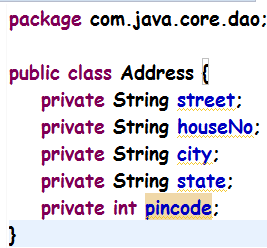
**Spring dependency Injection**

In java, everything is considered as an object and only 2 kind of relationships can exist between them. First one is a Has-a relationship and another one is a Is-A relationship. Has-a relationship signifies a dependency of one object on the member variable, whereas is-a relationship signifies inheritance.

To show a class Person which has an address field, we will demonstrate this using a has-a relationship.

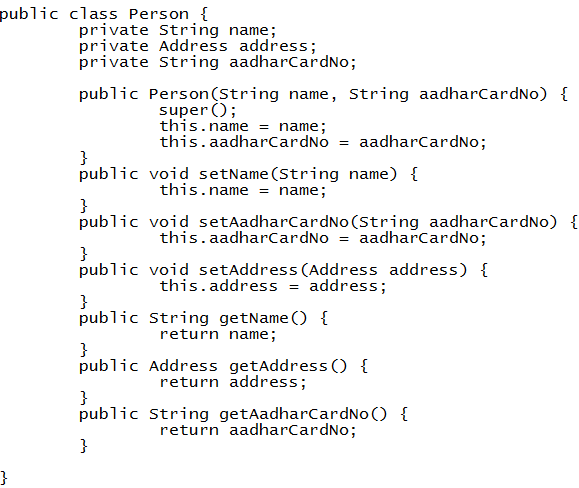


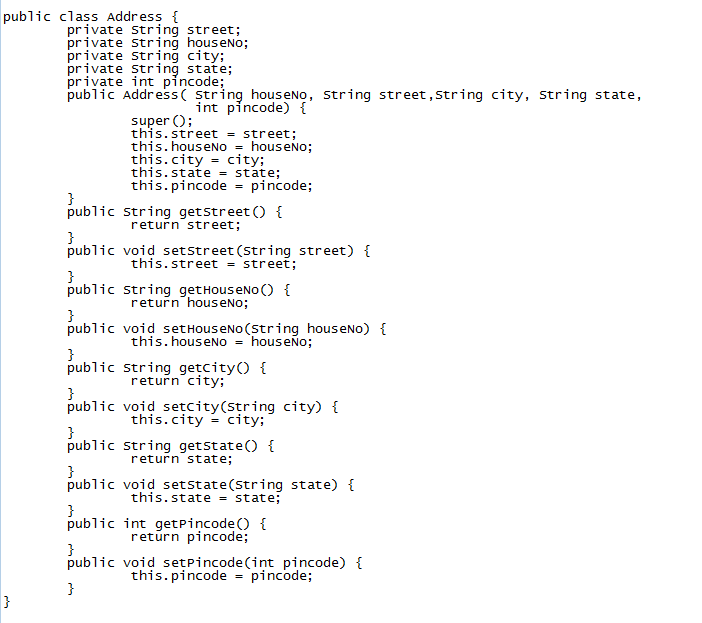
Address is also a class with member variables like street, pincode etc.

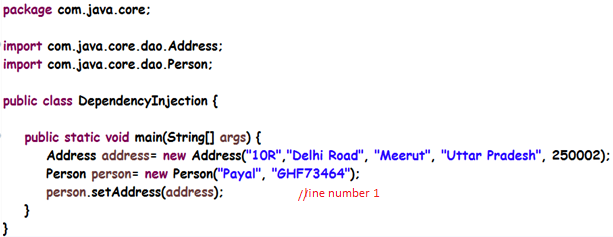


Now to set the value of an address field in a person class, it can be done by either making a constructor in a Person class with Address as a parameter which is done preferably if Address is a mandatory field for a person class i.e. a Person object should not exist if no address has been assigned to it. Else should be done through a setter.

Considering the scenario when a Person object can exist without an address been allocated to it, we’ll use the setter to set the address object in the person class. And the name of the Person and aadharCardNo is being considered as a mandatory field in this example without which a person should not exist. Hence we’ll use a parameterized constructor for it. Similarly, an address class is considered to have street, houseNo, city, state and pincode: all as mandatory fields hence using a parameterized constructor again for it. Address can be set for the Person object using a setter then.

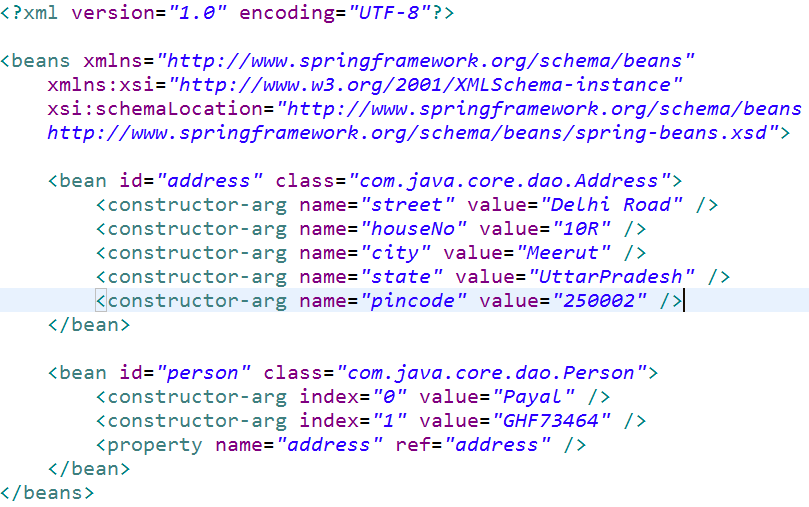






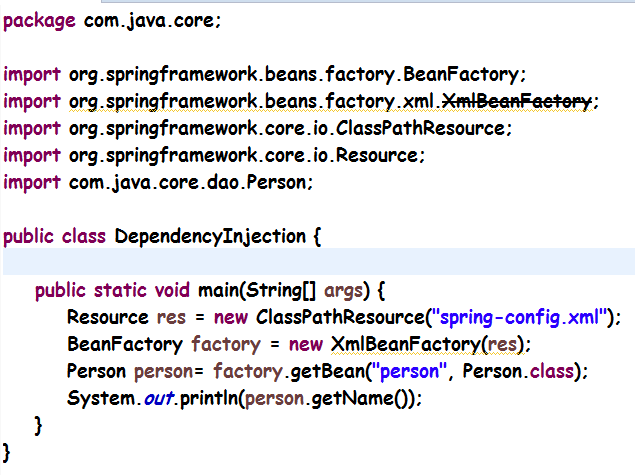
Hence we are using a setter to set the value of the address in a person object. This setter needs an object of an Address class. If the address object changes, the changes have to be done in the code at line number 1 where we are setting this object as a field in the person. This makes the code tightly coupled as it introduces the dependency of one object on the other. To reduce this dependency, spring has introduced the feature of dependency injection wherein the objects are created in the xml and dependency is injected at the runtime using either the setter or the constructor. Hence the code becomes loosely coupled.

The way we define the beans in spring xml configuration file is:



Here, constructor-arg is used to set the arguments for the constructor and property is used to set the property value using the setter. So first the constructor of the class is called passing in the values for the constructor as the arguments and after the object is created, the properties are set using the setter for that property.

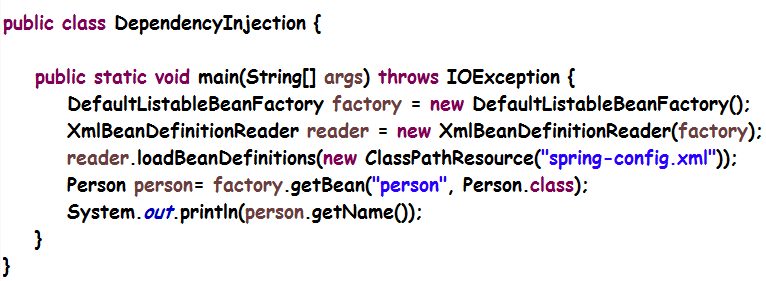
In the application, the objects can then be used fetching the appropriate bean object from the xml.



BeanFactory is an interface which is also said to be a core container for spring where the objects are created by the spring reading the xml file and lifecycle is managed by the spring container.

XmlBeanFactory is the implementation class which reads an xml file and the root tag should be beans tag for the xml. This implementation is now deprecated.

Alternatively, can use DefaultListableBeanFactory along with XmlBeanDefinitionReader to start the container. This allows for reading from multiple XML resources and is highly configurable in its actual XML parsing behavior.

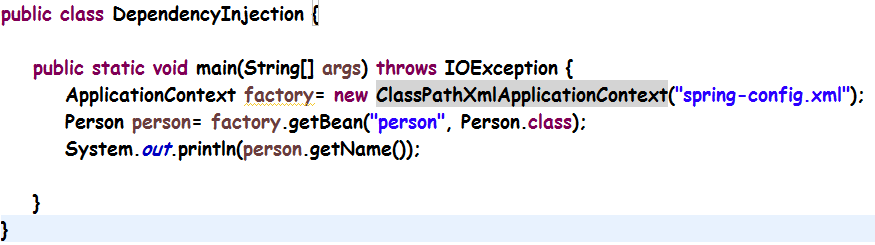


Before spring came up, many design patterns were studied to create objects like the Singleton pattern, builder pattern, decorator, factory, abstract factory pattern, proxy pattern, service locator pattern and even reflection.

The dependency is not managed by the client but by an external resource i.e. the container while it is creating the objects, hence termed as **Inversion of Control**.

There is also an advanced container which is also a core container initialized by creating an instance of **ApplicationContext**. It adds easier integration with Spring AOP , message resource handling features( used for internationalization) and application layer specific contexts like WebApplicationContext for use in web applications.

. In standalone applications it is common to create an instance of [ClassPathXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.1.RELEASE/javadoc-api/org/springframework/context/support/ClassPathXmlApplicationContext.html) or [FileSystemXmlApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.1.RELEASE/javadoc-api/org/springframework/context/support/FileSystemXmlApplicationContext.html)



Generally, we don’t inject pojo objects using spring.

//TODO modify the above example for service, repository instead

@Component: stereotype annotation to make the object creation by itself using component scan.

  A meta annotation is an annotation that can be applied to another annotation

Core Spring provides several stereotype annotations out of the box, including but not limited to: @Component, @Service, @Repository, @Controller, @RestController, and @Configuration. @Repository, @Service, etc. are specializations of @Component. A ***composed annotation*** is an annotation that is *meta-annotated* with one or more annotations with the intent of combining the behavior associated with those meta-annotations into a single custom annotation. For example, an annotation named @TransactionalService that is meta-annotated with Spring's @Transactional and @Service annotations is a composed annotation that combines the semantics of @Transactional and @Service. @TransactionalService is technically also a custom *stereotype annotation*.

There are five annotation types in the java.lang.annotation package called **meta-annotations**. These annotation types are used to annotate other annotation types.

## Documented[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit&section=1" \o "Edit section: Documented)]

If a member is annotated with a type itself marked as @Documented, then that member will be documented as annotating that type.

|  |  |
| --- | --- |
| Computer code | **Code listing 1.1: Use of @Documented**  @interface Secret { }  @Documented  @interface NotSecret { }  @Secret  @NotSecret  public class Example {  } |

In the documentation for the Example class, such as the JavaDoc, Example will be shown as annotated with @NotSecret, but not @Secret.

|  |  |
| --- | --- |
| Clipboard | **To do:** Add the render of a Javadoc. |

## Inherited[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit&section=2" \o "Edit section: Inherited)]

Exactly as the name sounds, an @Inherited annotation type is inherited by subclasses of an annotated type.

|  |  |
| --- | --- |
| Computer code | **Code listing 1.2: Use of @Inherited**  @Inherited  @interface ForEveryone { }  @interface JustForMe { }  @ForEveryone  @JustForMe  class Superclass { }  class Subclass extends Superclass { } |

In this example, Superclass has been explicitly annotated with both @ForEveryone and @JustForMe. Subclass hasn't been explicitly marked with either one; however, it inherits @ForEveryone because the latter is annotated with @Inherited. @JustForMe isn't annotated, so it isn't inherited by Subclass.

## Repeatable[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit&section=3" \o "Edit section: Repeatable)]

|  |  |
| --- | --- |
| https://upload.wikimedia.org/wikipedia/commons/thumb/9/91/Book_important2.svg/40px-Book_important2.svg.png | **This section is a stub.** You can help Wikibooks by [expanding it](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit). |

A @Repeatable annotation type is repeatable - i.e. can be specified multiple times on the same class. This meta-annotation was added in Java 8.

## Retention[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit&section=4" \o "Edit section: Retention)]

Different annotation types have different purposes. Some are intended for use with the compiler; others are meant to be reflected dynamically at runtime. There's no reason for a compiler annotation to be available at runtime, so the @Retention meta-annotation specifies how long an annotation type should be retained. The value attribute is one of the java.lang.annotation.RetentionPolicy enum constants. The possible values, in order from shortest to longest retention, are as follows:

RetentionPolicy.SOURCE

The annotation will not be included in the class file. This is useful for annotations which are intended for the compiler only.

RetentionPolicy.CLASS

The annotation will be included in the class file, but cannot be read reflectively.

RetentionPolicy.RUNTIME

The annotation can be reflected at runtime.

If no @Retention policy is specified, it defaults to RetentionPolicy.CLASS.

## Target[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Meta-Annotations&action=edit&section=5" \o "Edit section: Target)]

The @Target meta-annotation determines what may be marked by the annotation. The value attribute is one or more of the java.lang.annotation.ElementType enum constants. Those constants are ElementType.ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, and TYPE.

If @Target is not specified, the annotation may be used on any program element.

Annotations can be viewed as a source of defining meta-data for a piece of code in Java. The annotation @CodeDescription used in the following sections does not come as a part of the Java API.

## Annotation Type Declaration[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Custom_Annotations&action=edit&section=1" \o "Edit section: Annotation Type Declaration)]

Before you can use an annotation with classes, theirs members and statements or expressions, you need to define an ***annotation type***. Following is the syntax on how to define a type for the mentioned annotation.

|  |  |
| --- | --- |
| Computer code | **Code listing 1.1: Annotation type declaration**  @interface CodeDescription  {  String author();  String version();  } |

That's it! Our first ever annotation has been defined. Now, we can use it with any of our classes. An annotation definition if you look closely resembles the definition of a normal interface, except that the interface keyword is preceded by the **@** character. Some refer to this syntactical declaration as the ***annotation type declaration*** due to the fact that @ is 'AT' or 'Annotation Type' for that very instance.

## Annotation Element Declarations[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Custom_Annotations&action=edit&section=2" \o "Edit section: Annotation Element Declarations)]

What look like methods in the body of the annotation definition are called ***annotation element declarations***. These are the named entities that we used with the annotation body in the example in the [previous section](https://en.wikibooks.org/wiki/Java_Programming/Annotations/Introduction). However, for the sake of clarity, code below also represents the calling of the following annotation:

|  |  |
| --- | --- |
| Computer code | **Code listing 1.2: Calling of annotation**  public class MyMethod  {  @CodeDescription  (  author = "Unknown",  version = "1.0.0.1"  )  public void doSomething()  {  ...  }  } |

|  |
| --- |
| **Note:** Instead of using the declaration with the class, the annotation is used with the method doSomething(). This might not demonstrate the power of annotations yet, but more will be explored in later chapters. |

## Using a default value[[edit](https://en.wikibooks.org/w/index.php?title=Java_Programming/Annotations/Custom_Annotations&action=edit&section=3" \o "Edit section: Using a default value)]

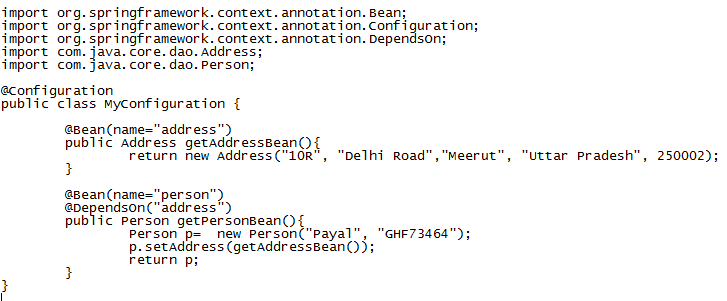
Now, for instance, you want the annotation to know that if no value for the version element is present, then it should use a ***default value***. Declaring a default value would be done the following way.

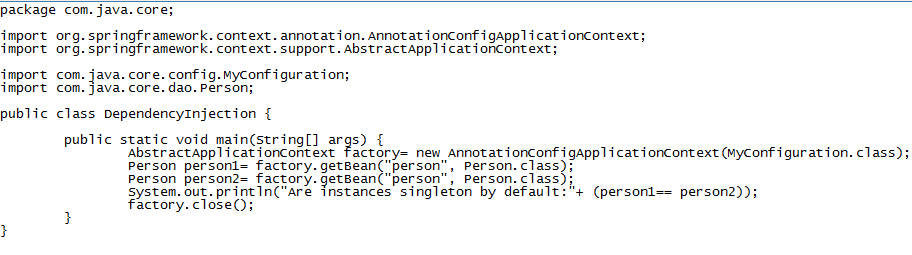
|  |  |
| --- | --- |
| Computer code | **Code listing 1.3: Using default values.**  @interface CodeDescription  {  String author();  String version() default "1.0.0.1";  } |

So, now if you use the same code again, you can ignore the version element because you know that the value is to be provided by default.

|  |  |
| --- | --- |
| Computer code | **Code listing 1.4: Pre-defined value.**  public class MyMethod  {  @CodeDescription(author = "Sysop")  public void doSomething()  {  ...  }  } |

**Java Configuration for spring**:





Typically you define service layer objects, data access objects (DAOs), presentation objects such as Struts Action instances, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring’s integration with AspectJ to configure objects that have been created outside the control of an IoC container

Using applicationContext can load more than 1 resource/ file

ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");

**Use of import to import more than one configuration file in Spring**

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the ApplicationContext’s BeanFactory via the method getBeanFactory() which returns the BeanFactory implementation DefaultListableBeanFactory. DefaultListableBeanFactory supports this registration through the methods registerSingleton(..) and registerBeanDefinition(..).

Inner class names

**Inner class names**

If you want to configure a bean definition for a static nested class, you have to use the *binary* name of the nested class.

For example, if you have a class called Foo in the com.example package, and this Foo class has a static nested class called Bar, the value of the 'class' attribute on a bean definition would be…​

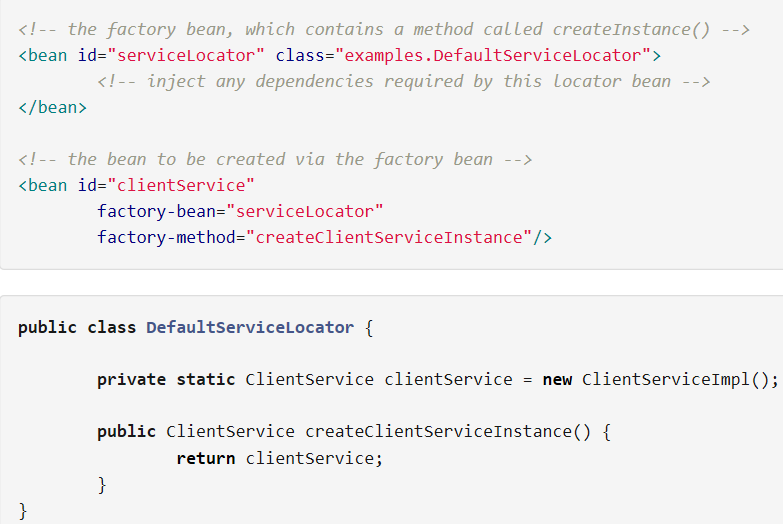
com.example.Foo$Bar

Notice the use of the $ character in the name to separate the nested class name from the outer class name.

**Factory method**

****

**Instance factory method:**

****

**@Required**

Note that use of the [@Required](https://docs.spring.io/spring/docs/5.0.1.RELEASE/spring-framework-reference/core.html#beans-required-annotation) annotation on a setter method can be used to make the property a required dependency.

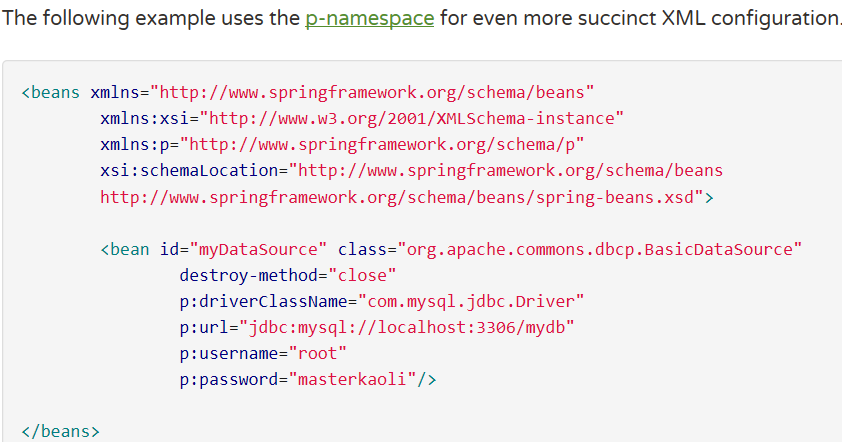
Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in [Bean scopes](https://docs.spring.io/spring/docs/5.0.1.RELEASE/spring-framework-reference/core.html#beans-factory-scopes). Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigneds

**Circular Dependency**

Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a **BeanCurrentlyInCreationException**

The value we specify to the property in spring is considered as String . But Spring’s [**conversion service**](https://docs.spring.io/spring/docs/5.0.1.RELEASE/spring-framework-reference/core.html#core-convert-ConversionService-API) is used to convert these values from a String to the actual type of the property or argument.

**P-namespace**

****

**Properties Instance**

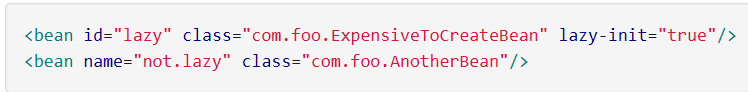
****

**Null and Empty values**

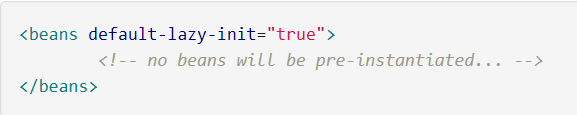
****

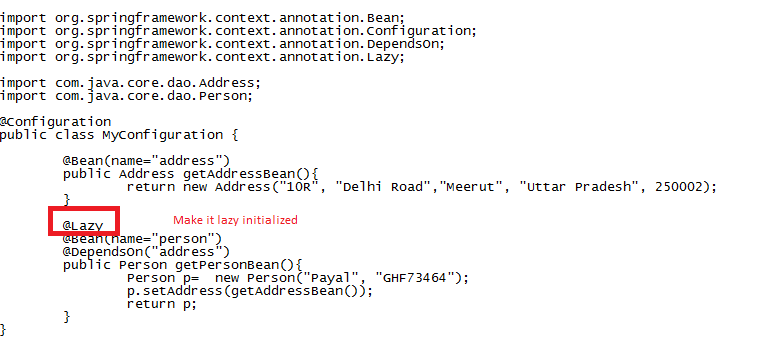
**Lazy Initialized beans:**

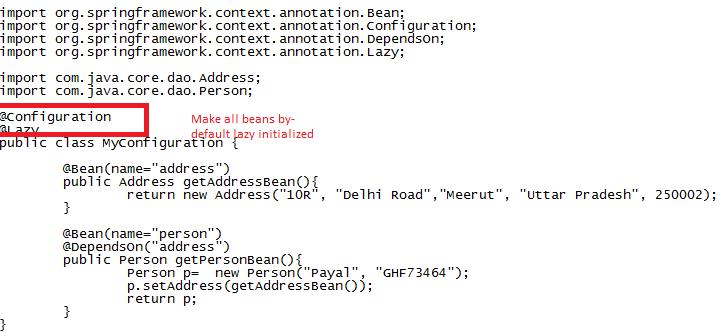
By default, ApplicationContext implementations eagerly create and configure all [singleton](https://docs.spring.io/spring/docs/5.0.1.RELEASE/spring-framework-reference/core.html#beans-factory-scopes-singleton) beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup

****

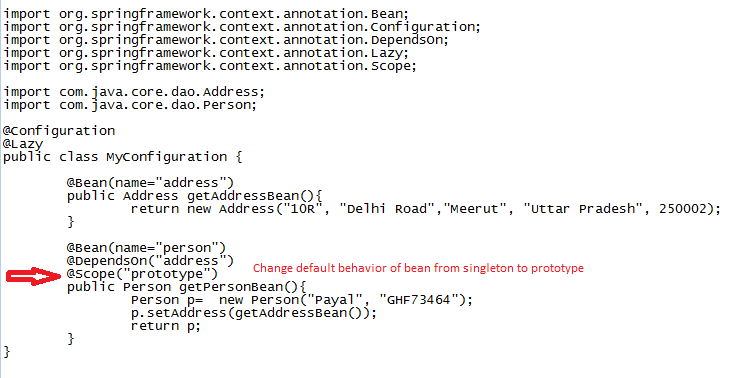
**To make all the beans by-default lazy initialized:**

****

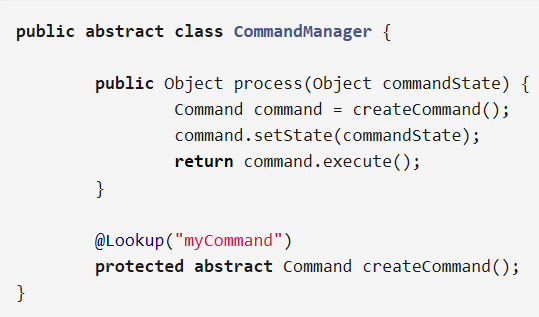
****

****

**Change bean from singleton to prototype:**

****

**@Lookup**

****

**Arbitrary method replacement**