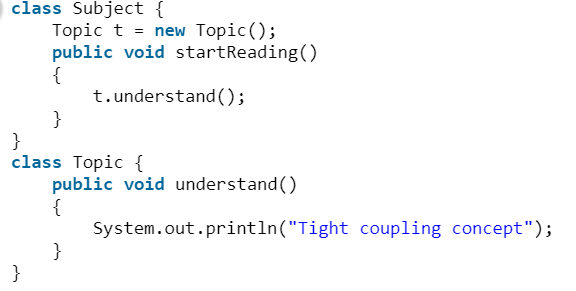
Spring

What is tight coupling and loose coupling?

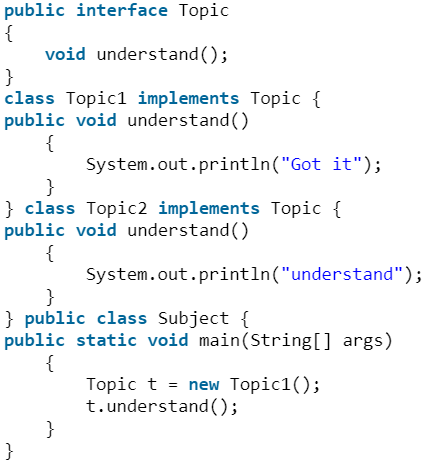
Tight coupling means classes and objects are dependent on one another. In general, tight coupling is usually not good because it reduces the flexibility and re-usability of the code while Loose coupling means reducing the dependencies of a class that uses the different class directly.

Tight coupling : In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.



Explanation: In the above program the Subject class is dependents on Topic class. In the above program Subject class is tightly coupled with Topic class it means if any change in the Topic class requires Subject class to change. For example, if Topic class understand() method change to gotit() method then you have to change the startReading() method will call gotit() method instead of calling understand() method.

Loose coupling : In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled. In order to over come from the problems of tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling



Explanation : Topic is an interface and we can inject any of the implemented classes at run time and we can provide service to the end user.

Which is better tight coupling or loose coupling?

In general, Tight Coupling is bad in but most of the time, because it reduces flexibility and re-usability of code, it makes changes much more difficult, it impedes test ability etc. loose coupling is a better choice because A loosely coupled will help you when your application need to change or grow. If you design with loosely coupled architecture, only a few parts of the application should be affected when requirements change.

Inversion of Control :

Inversion of Control is a principle in software engineering by which the control of objects is transferred to a container or framework.

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

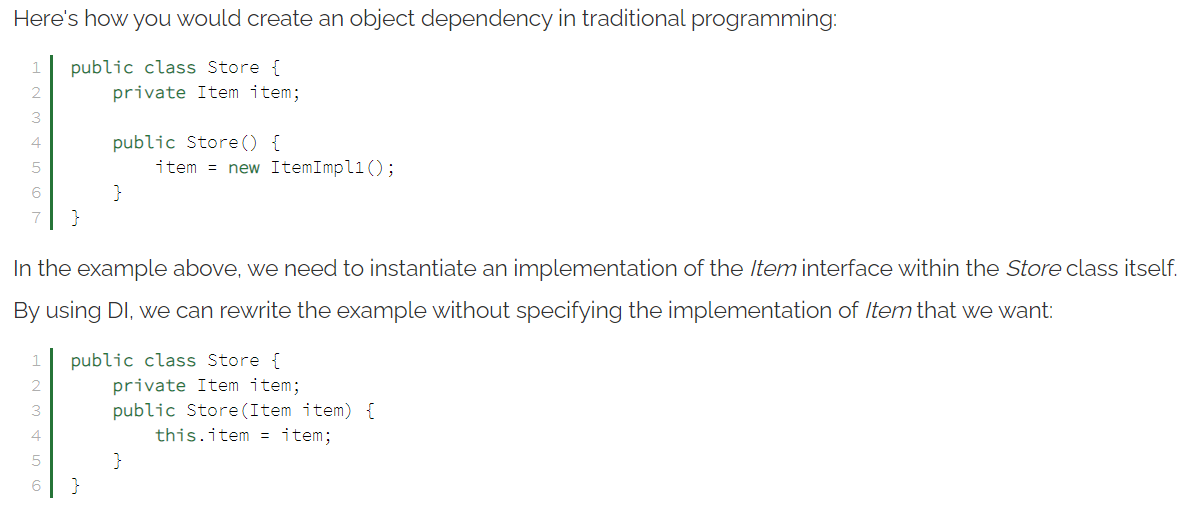
greater ease in testing a program by isolating a component or mocking its dependencies and allowing components to communicate through contracts

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

Dependency Injection:

Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.

The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.



Spring IoC Container:

The Spring container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their lifecycle.

BeanFactory and ApplicationContext are implementation of IOC container

ApplicationContext => BeanFactory + AOP feature + Internationalization + WebApplicationContext for web applications

We should use ApplicationContext except in one scenario in which the memory is concerned (like in IOT device where memory is less)

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

Beans : Spring beans are Java objects that are managed by the Spring container.

The Spring container is responsible for instantiating, configuring, and assembling the Spring beans.

Autowired : the process where spring identify the matches for the dependency and populate them

IOC : Inversion of Control is a principle in software engineering by which the control of objects is transferred to a container or framework.

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

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

@Configuration annotation indicates that the class is a source of bean definitions. Also, we can add it to multiple configuration classes.

@Bean annotation is used on a method to define a bean. If we don't specify a custom name, the bean name will default to the method name.

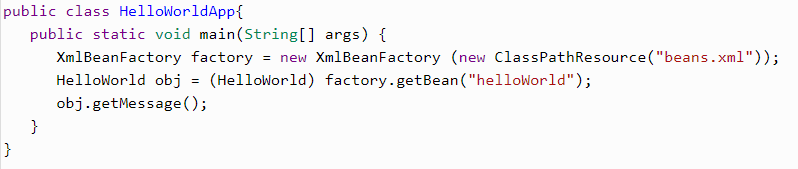
BeanFactory and ApplicationContext:

BeanFactory:

we will use Spring's dependency injection functionality using this BeanFactory interface

the implementations use lazy loading, which means that beans are only instantiating when we directly calling them through the getBean() method.

The most used API that implements the BeanFactory is the XmlBeanFactory.



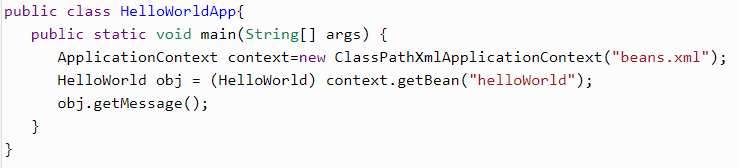
ApplicationContext:

It implements the BeanFactory interface. Hence, the ApplicationContext includes all functionality of the BeanFactory and much more! Its main function is to support the creation of big business applications.

WebApplicationContext for web application

Spring AOP

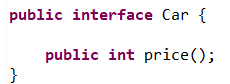
I18n capability

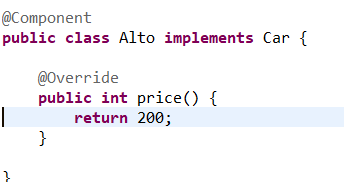


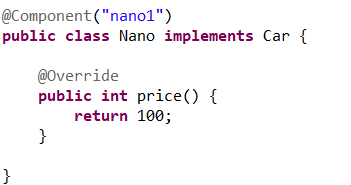
NOTE:

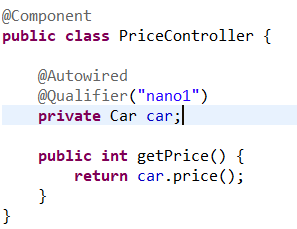
The ApplicationContext includes all the functionality of the BeanFactory. It is generally recommended to use the former. There are some limited situations, such as in mobile applications, where memory consumption might be critical. In those scenarios, it would be justifiable to use the more lightweight BeanFactory. However, in most enterprise applications, the ApplicationContext is what you will want to use.

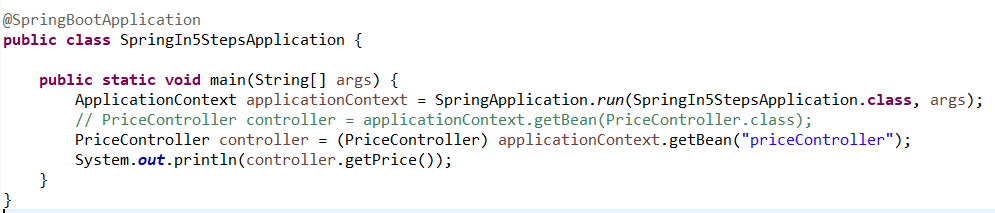
BeanPostProcessor:





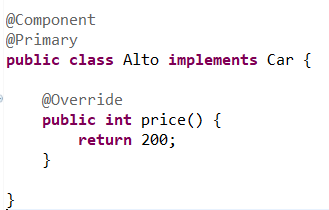




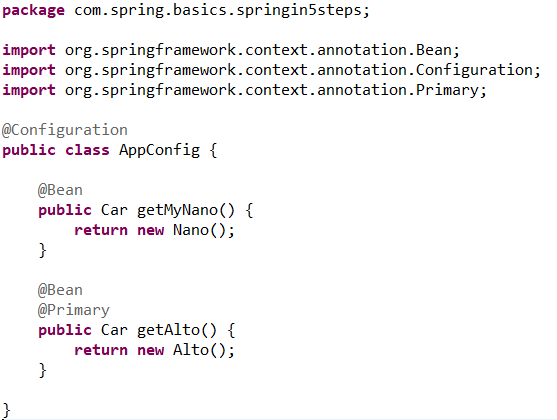


If we are creating bean of class using @Component annotation, then by default name of that bean is camel convention of the class name (i.e. nano for Nano class)

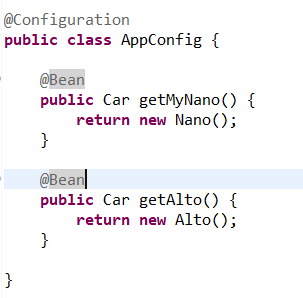
* if we have multiple bean of same type then either we can use @Qualifier or @Primary
* Use @Primary on top of class if we are creating Bean using @Component

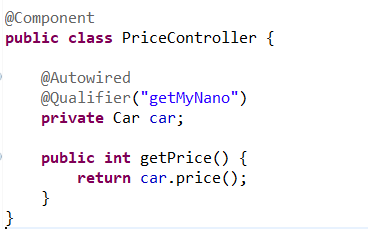


* Bean creation using @Configuration and @Bean
* In this case we should add @Component on top of class, we will create a class and annotate with @Configuration then create method with any name and annotate with @Bean



* In case of multiple bean of same class, name of the bean is same name of method, like name of Nano bean is getMyNano and name of Alto bean is getAlto.





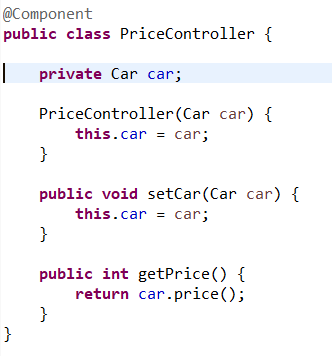
* If we have both @Qualifier and @Primary, then priority goes to @Qualifier

Add below to application.properties file to see what spring is doing in background

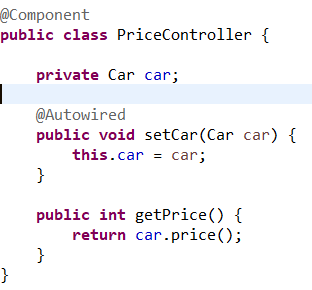
logging.level.org.springframework = debug

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

1. Constructor injection:

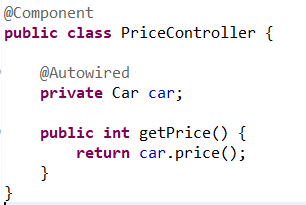


2. Setter injection:



Spring documentation recommends using constructor-based injection for mandatory dependencies, and setter-based injection for optional ones.

Note : If we are not using constructor or setter injection then by default spring do injection using setter injection.



There are many key differences between constructor injection and setter injection:

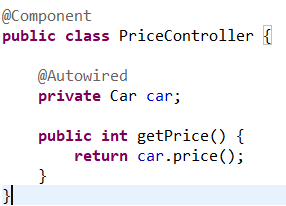
Partial dependency: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.

Overriding: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.

Changes: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So setter injection is flexible than constructor injection.

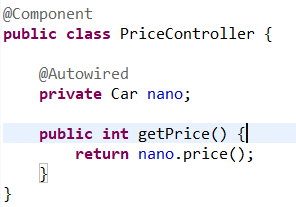
Autowiring by type:

Spring will search for the bean by its type. As per below example spring will search for bean of Type Car



Autowiring by name:

Spring will search for the bean by its name. As per below example spring will search for bean of same name nano.

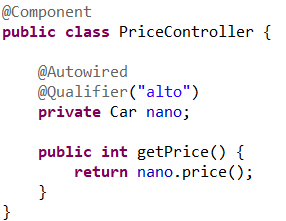


Suppose we are having two bean of Car type then, in that case spring search for bean name nano.

Autowiring priority:

* By type
* Qualifier
* Primary
* By name

As per below example, Qualifier will get prefernce over by name



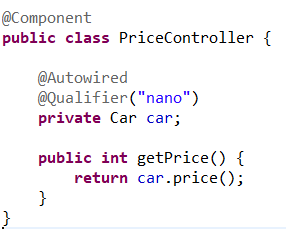
Qualifier:

The @Qualifier annotation is used to resolve the autowiring conflict, when there are multiple beans of same type.

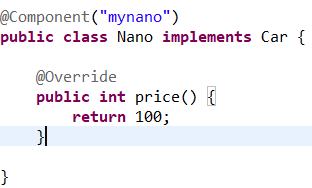
The @Qualifier annotation can be used on any class annotated with @Component or on method annotated with @Bean. This annotation can also be applied on constructor arguments or method parameters.

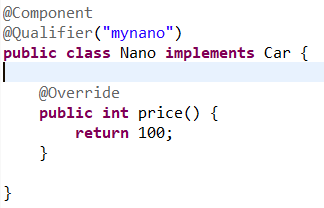
By using the @Qualifier annotation, we can eliminate the issue of which bean needs to be injected.

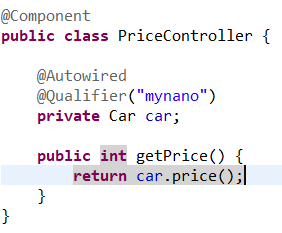
Ex-> we are having two bean of type Car which is nano and alto, so using @Qualifier we can specify which bean need to inject by passing the name bean



we can give name to bean like below,

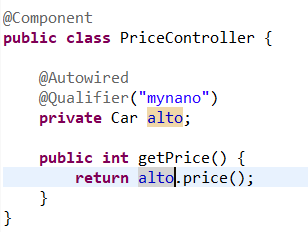






Note:

As per below example we have two bean of type Car name mynano and alto, and if we use @Qualifier and injection by name (private Car alto), then @Qualifier will get preference.



Scope of Bean:

The latest version of Spring framework defines 6 types of scopes:

singleton

prototype

request

session

application

websocket

The last four scopes mentioned request, session, application and websocket are only available in a web- application.

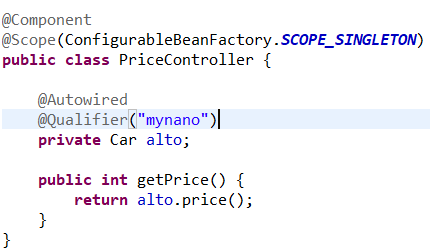
1. Singleton:

Defining a bean with singleton scope means the container creates a single instance of that bean, and all requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. This scope is the default value if no other scope is specified.

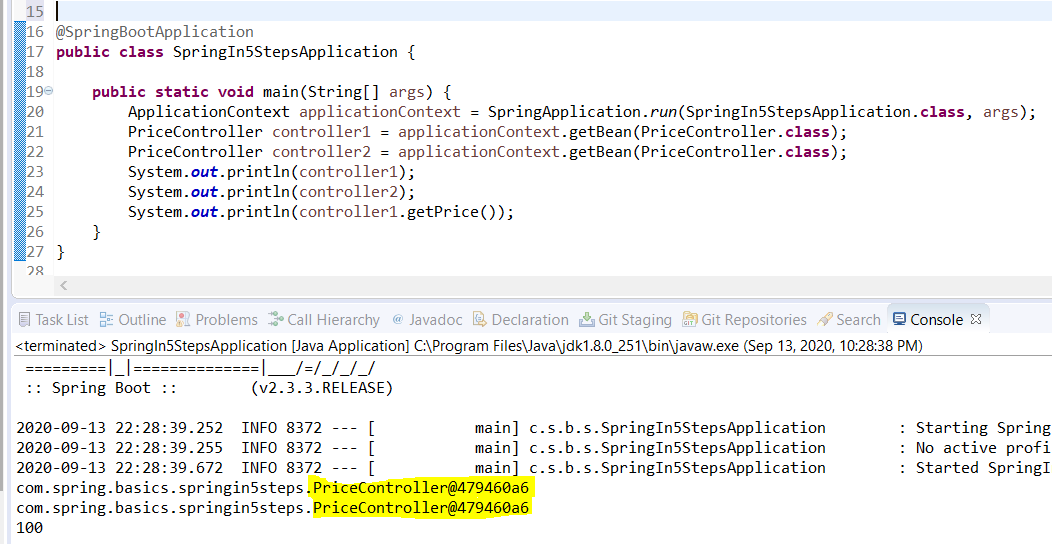
@Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON)

Or,

@Scope("singleton")

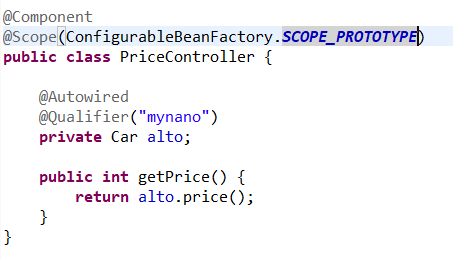


Hashcode is same

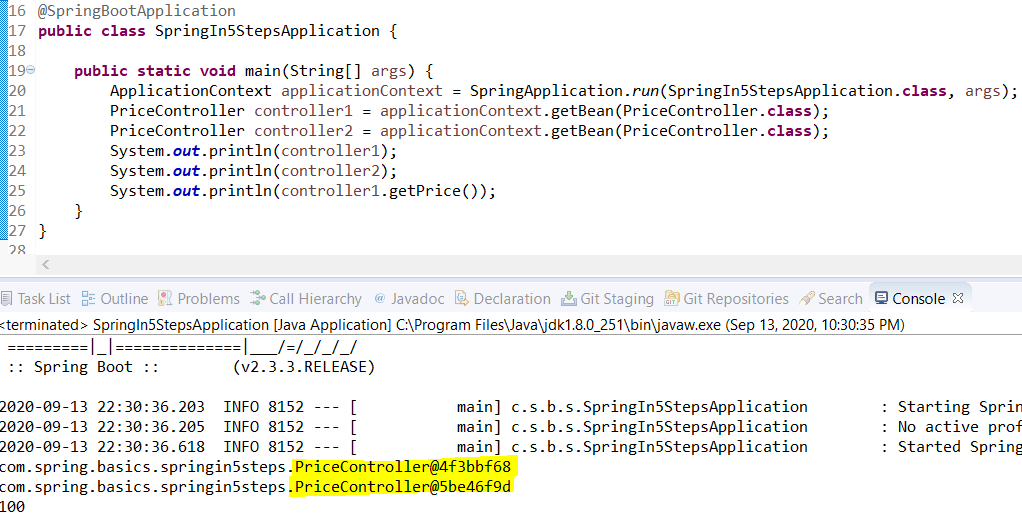


2. Prototype:

A bean with prototype scope will return a different instance every time it is requested from the container

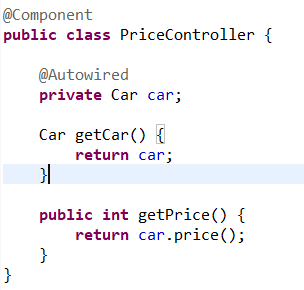


Hashcode is different

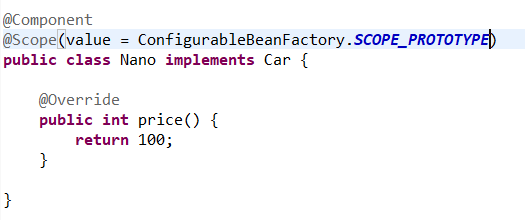


* Scope scenario:

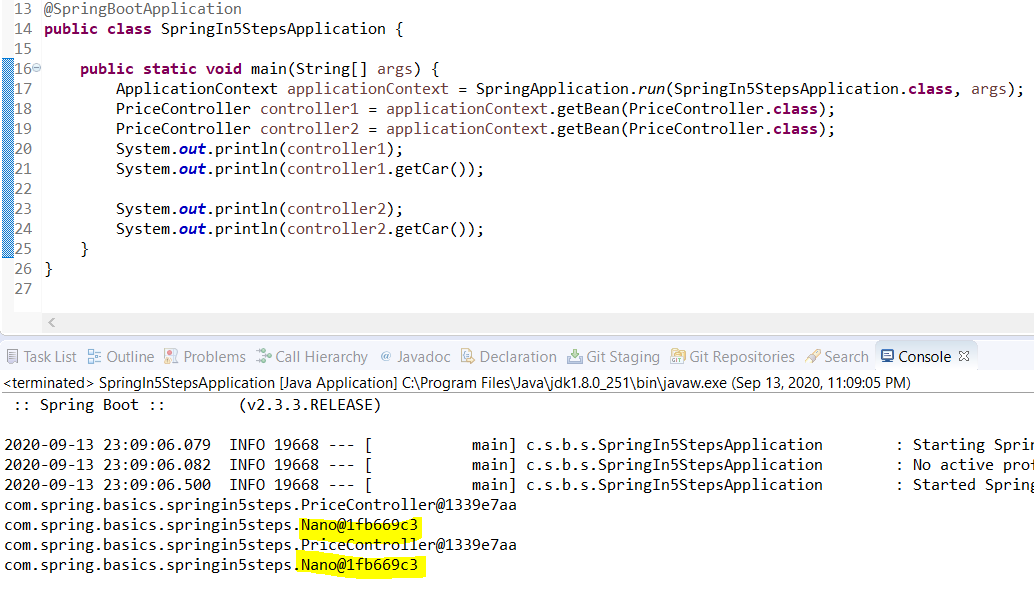
PriceController class is having singleton scope



Nano class’s scope is Prototype

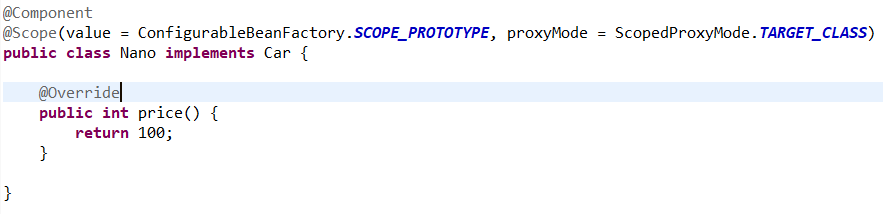


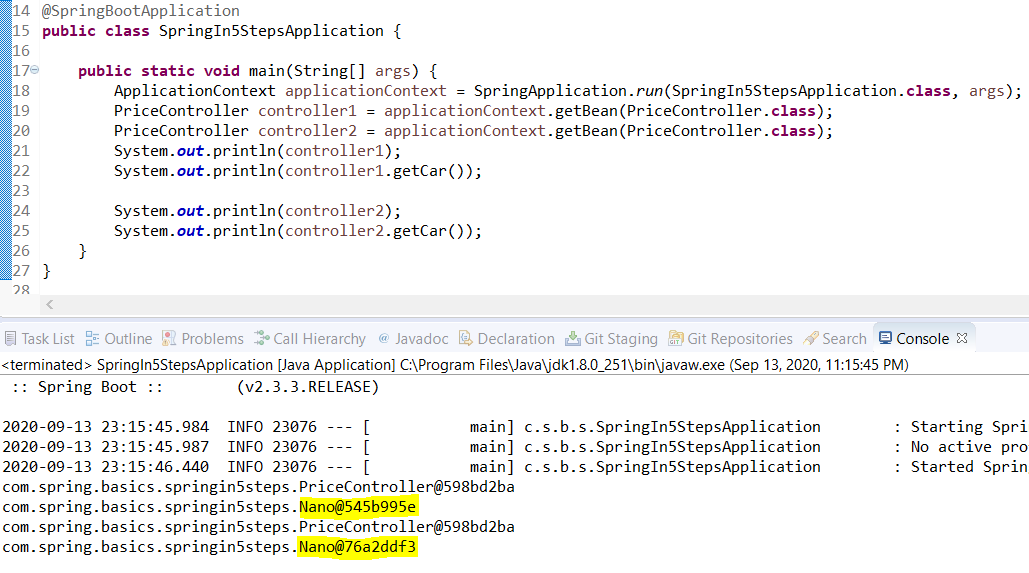
But we are getting same hashcode for 2 different PriceController’s Car class, but Car class is prototype



Sol-> to resolve this we should use proxyMode

Spring will create a proxy to be injected as a dependency, and instantiate the target bean when it is needed





GOF singleton vs Spring singleton:

As per Gang of Four, singleton means one instance of bean per JVM

But as per Spring singleton one instance per ApplicationContext, i.e. if we have 2 ApplicationContext then we will have 2 instance in JVM

* Component scan:

define the packages that have to be scanned.

If your other package hierarchies are below your main app with the @SpringBootApplication annotation, you’re covered by the implicit Component Scan.

If there are beans/components in other packages that are not sub-packages of the main package, you should manually add them as @ComponentScan

In this case, you would need to add the new package into Component Scan.You have two options:

Define @ComponentScan(“com.in28minutes.springboot”)

This would scan the entire parent tree of com.in28minutes.springboot.

Or define two specific Component Scans by using an array.

@ComponentScan({“com.in28minutes.springboot.basics.springbootin10steps”,”com.in28minutes.springboot.somethingelse”})

Aware interfaces:

These are used to access the Spring Framework infrastructure. The aware interfaces are largely used within the framework and rarely used by Spring programmer

Though Spring Aware interfaces, you can access the Spring context, or Spring Bean life cycle events.

Spring aware interfaces:

following three aware interfaces.

BeanNameAware: ThesetBeanName() callback of this interface supplies the name of the bean.

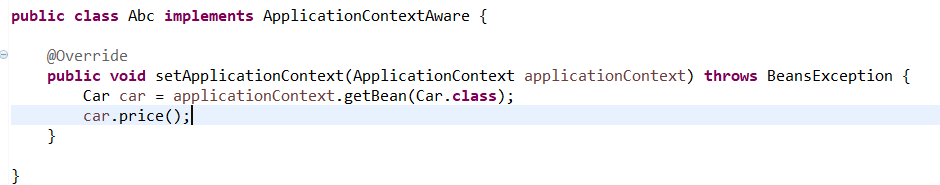
BeanFactoryAware: Provides setBeanFactory(), a callback that supplies the owning factory to the bean instance.

ApplicationContextAware: ThesetApplicationContext() method of this interface provides the ApplicationContext object of this bean.

ApplicationContextAware Interface:

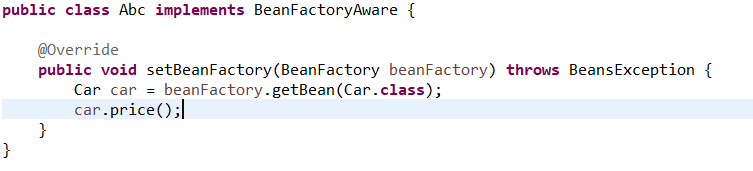
Spring provides an ApplicationContextAware interface that allows beans access to the ApplicationContext.

if my bean needs to look up some other beans



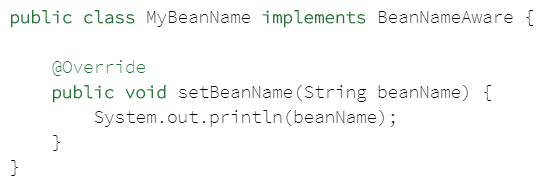
BeanFactoryAware Interface:

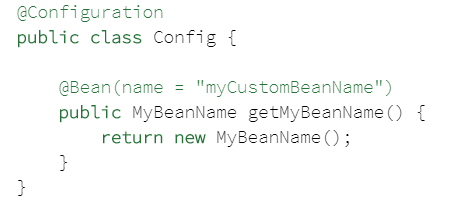
BeanFactoryAware is used to inject the BeanFactory object. This way we get access to the BeanFactory which created the object.

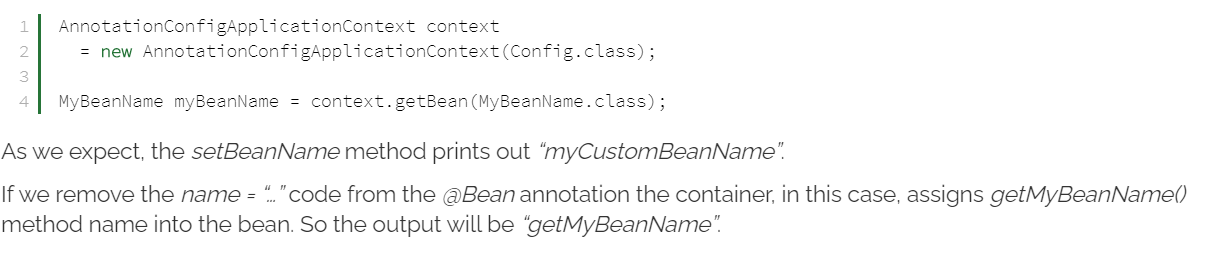


BeanNameAware:

The BeanNameAware interface is implemented by beans that need access to its name defined in the Spring container. This interface provides the setBeanName() method.





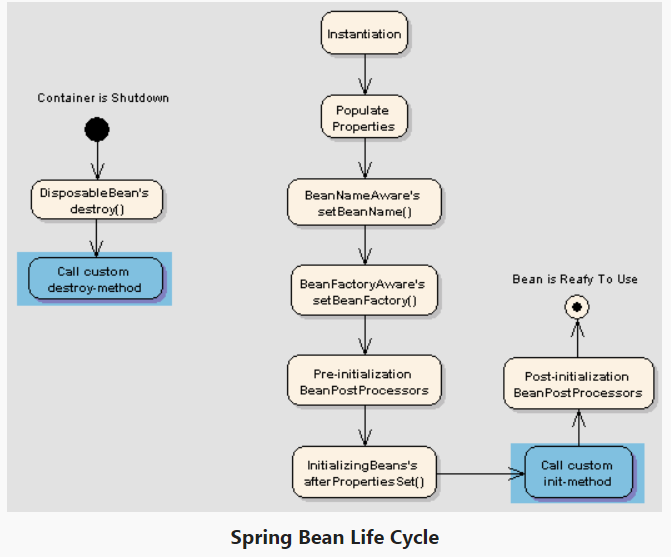


Note : we should avoid using any of the Aware interfaces, unless we need them. Implementing these interfaces will couple the code to the Spring framework.

Spring Bean Lifecycle:

When container starts – a Spring bean needs to be instantiated. It may also be required to perform some post-initialization.

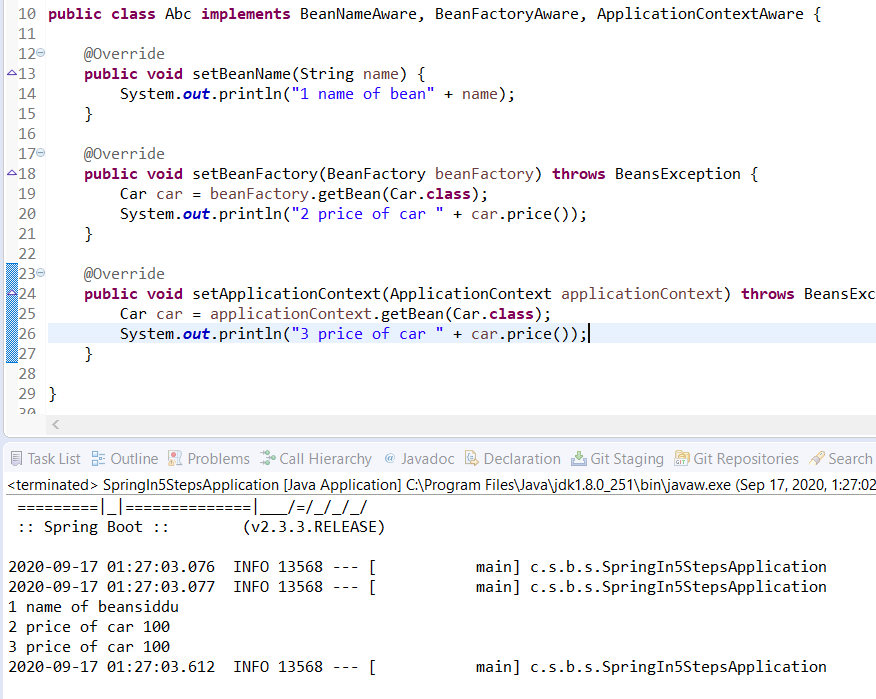
After that, when the bean is no longer required, it will be removed from the IoC container.

Spring bean factory is responsible for managing the life cycle of beans created through spring container.

ways for controlling life cycle events of a bean:

* \*Aware interfaces for specific behavior

Spring offers a range of \*Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean.

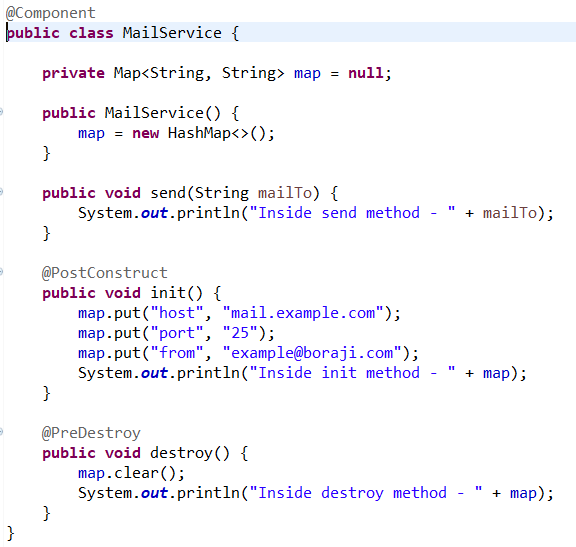


* @PostConstruct and @PreDestroy annotations

we can manage lifecycle of a bean by using method-level annotations @PostConstruct and @PreDestroy.

The @PostConstruct annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization.

The @PreDestroy annotation is used on methods as a callback notification to signal that the instance is in the process of being removed by the container.





Note:

As you can see, the init() and destroy() methods of MailService bean are called only once, when the scope of bean is singleton (default scope).

In case of prototype scope, the destroy method of the MailService bean will not work.

CDI:

CDI (Contexts and Dependency Injection) is standard dependency injection framework included in Java EE 6 and higher.

So spring implements CDI

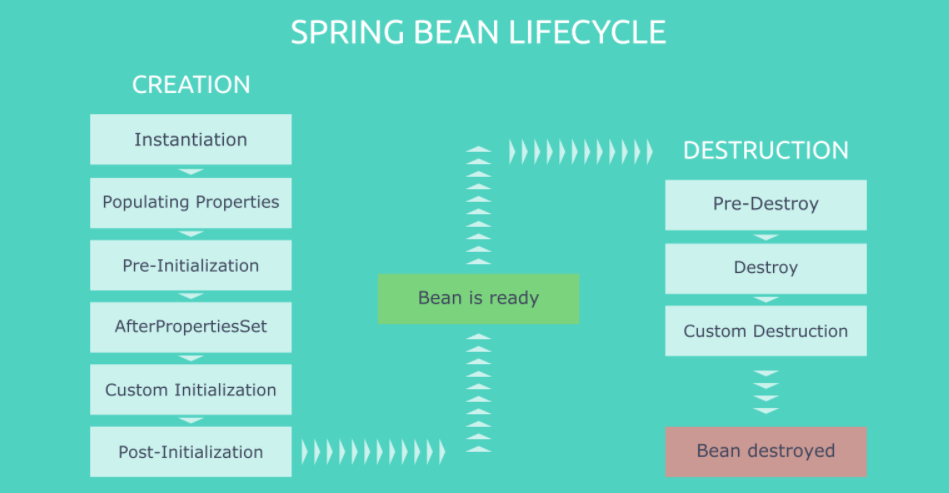
Spring supports most annotation:

@Inject => @Autowired

@Named => @Component

@Scope => @Scope

* The Spring Bean Lifecycle:



Bean Creation Phases

**Instantiation**: This is where everything starts for a bean. Spring instantiates bean objects just like we would manually create a Java object instance.

**Populating Properties**: After instantiating objects, Spring scans the beans that implement Aware interfaces and starts setting relevant properties.

**Pre-Initialization**: Spring’s BeanPostProcessors get into action in this phase. The postProcessBeforeInitialization() methods do their job. Also, @PostConstruct annotated methods run right after them.

**AfterPropertiesSet**: Spring executes the afterPropertiesSet() methods of the beans which implement InitializingBean.

**Custom Initialization**: Spring triggers the initialization methods that we defined in the initMethod attribute of our @Beanannotations.

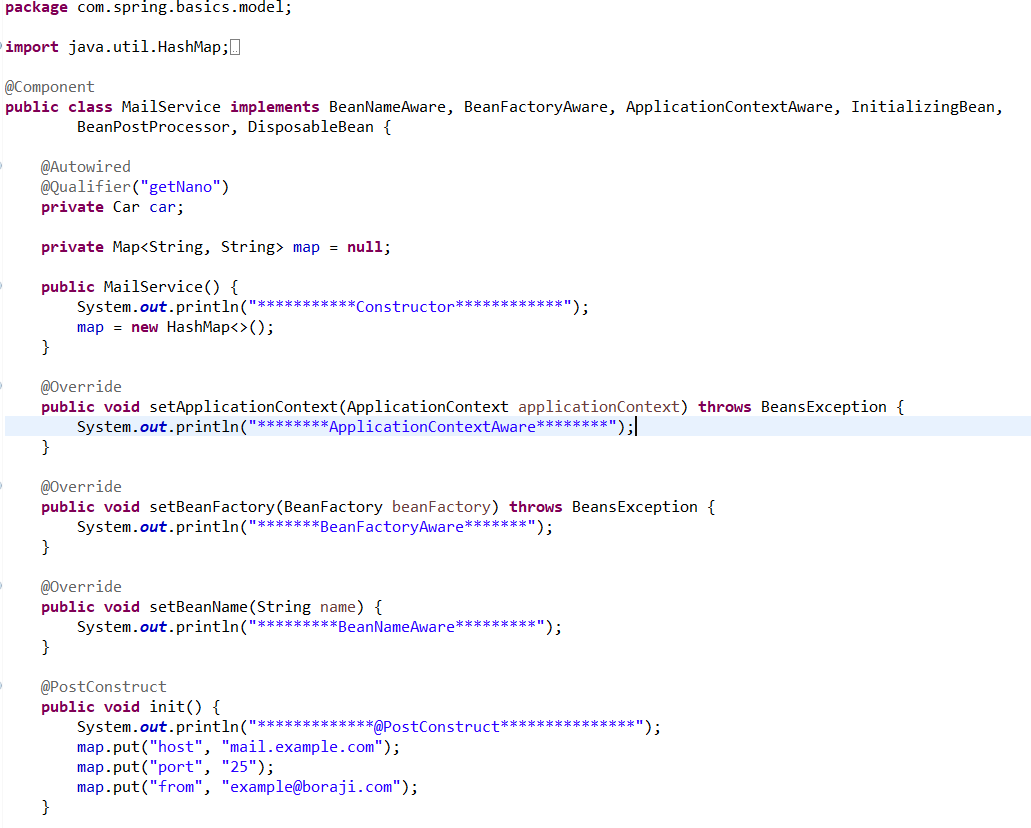
**Post-Initialization**: Spring’s BeanPostProcessors are in action for the second time. This phase triggers the postProcessAfterInitialization() methods.

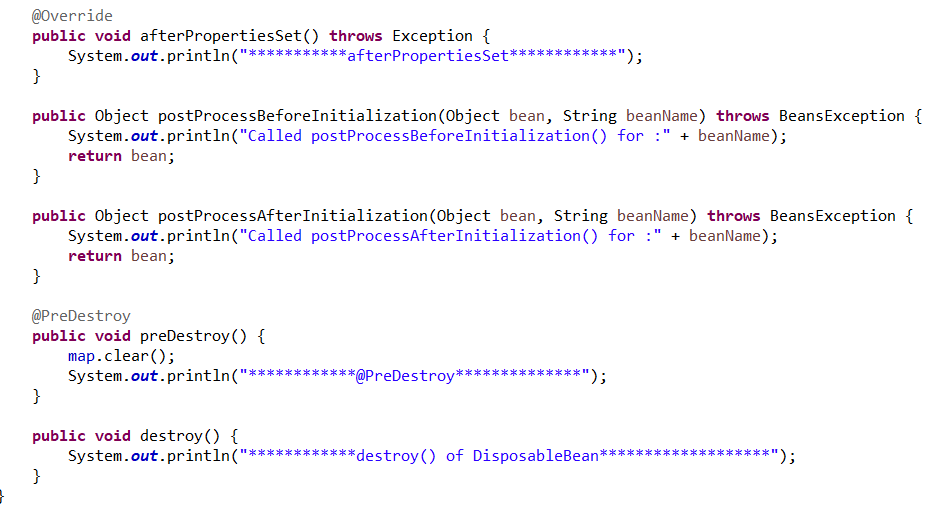
Bean Destruction Phases

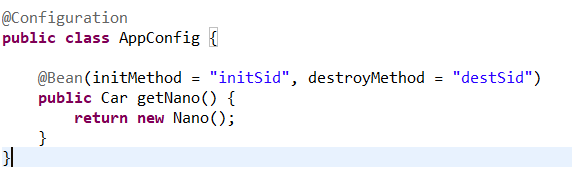
**Pre-Destroy**: Spring triggers@PreDestroy annotated methods in this phase.

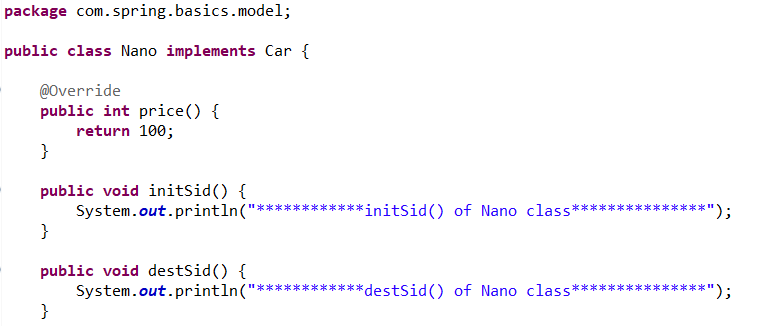
Destroy: Spring executes the destroy() methods of DisposableBean implementations.

**Custom Destruction**: We can define custom destruction hooks with the destroyMethod attribute in the @Bean annotation and Spring runs them in the last phase.

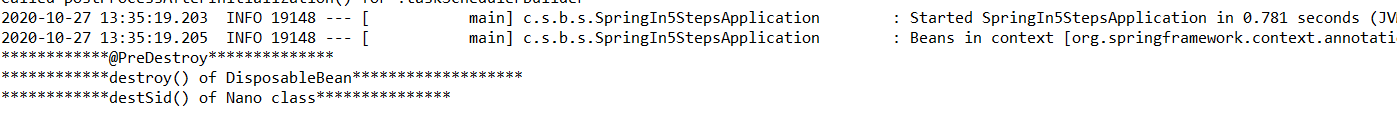






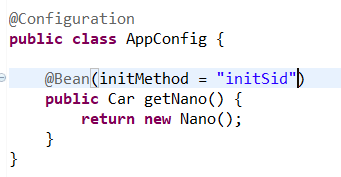


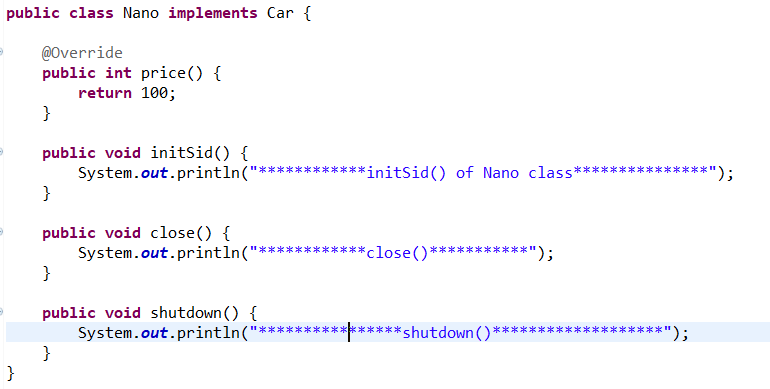




If we have public close() or shutdown() method in bean then this method get automatically get called without keeping destroyMethod attribute in @Bean

Note -> In that case it priority is close() if not then shutdown()





* Stereotype annotation:

1. @ Component -> generic component

2. @Controller -> at controller level

3. @Service -> at business layer

4. @Repository -> at DAO layer

* Register a Properties File via Java Annotations:

