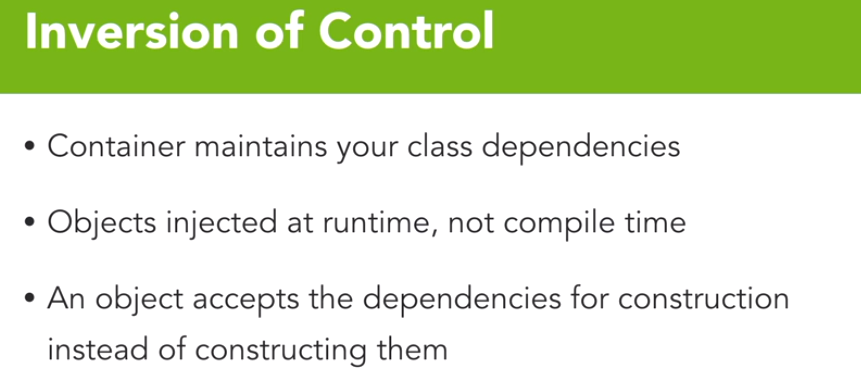
Spring IOC container manages all dependencies at run time for us. Once they're configured we never have to construct them again.

We no longer have to spend time in code managing these dependencies, we just simply let the container bring it in. And that is a huge performance improvement from the way that we traditionally wrote code, where we would construct our dependencies, manage its state throughout its life cycle and make sure that we clean it up appropriately when we're done.

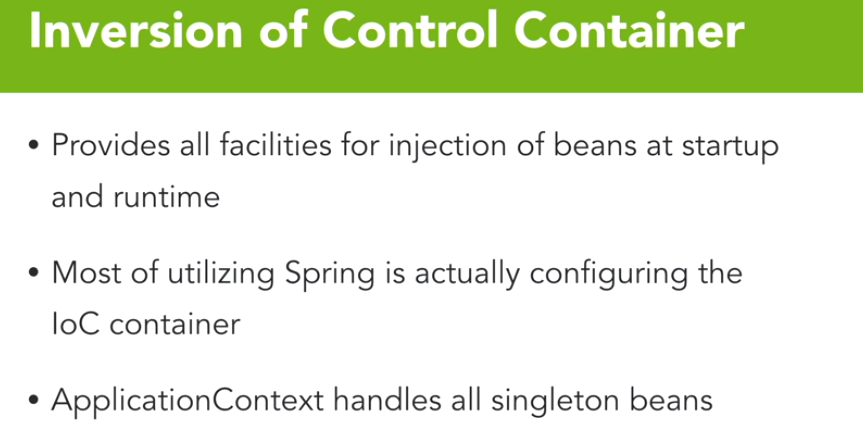


Inversion of control is essentially a way of dealing with dependency injection. Through the inversion of control container, Spring manages all our class dependencies that we rely on. And in doing so, those objects are injected at runtime, as opposed to being created at compile time. This reduces the amount of code that we need to manage within a given class file.

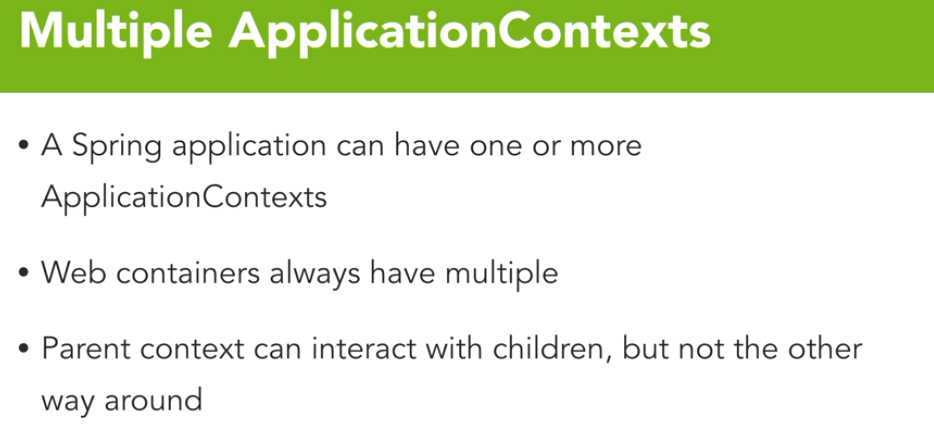
Benefits of dependency injection or inversion of control, as I just briefly alluded to, is the reduction of noise in your code. When dealing with dependency injection, because we do not copy and paste that construction code over and over, our code can focus on the business logic and not all this construction noise. It also reduces object coupling. Since an object that we are creating doesn't need to know how to create all its dependencies, that coupling is dramatically reduced.

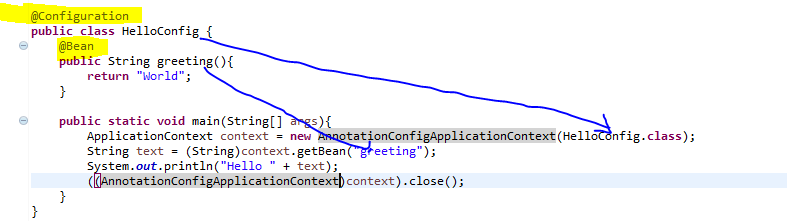


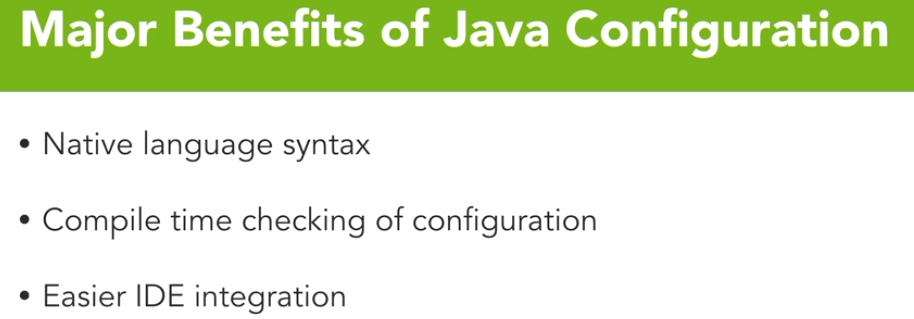
The ApplicationContext is the heart of an application leveraging the Spring Framework. ApplicationContext is a read-only wrapper of that BeanFactory, and all our runtime interactions with the BeanFactory or any beans contained in that factory is through this ApplicationContext. The ApplicationContext provides the metadata for all beans created and provides a mechanism for creating beans in the correct order. The ApplicationContext is the Inversion of Control Container and all our Dependency Injections occur here.

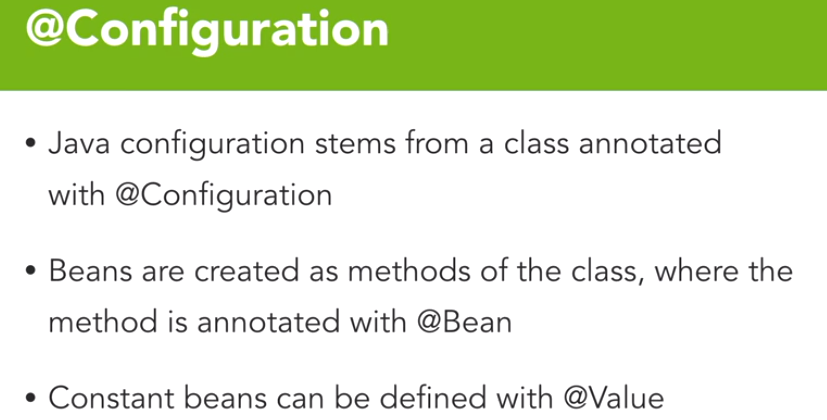


The ApplicationContext manages all our singleton beans in the application itself. It also serves all prototype or session scoped beans.









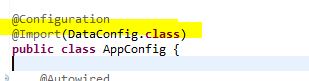
**@Configuration annotation**

When using auto configuration, this annotation allows the class to be component scanned.

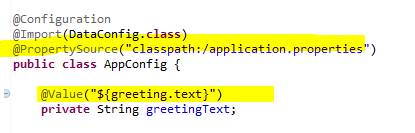
To define a Spring Bean for injection by the application context, we can create a method and annotate it with the **@Bean** annotation. The name of the bean will be the method name that we choose in our class file.

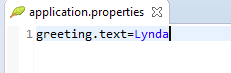
To leverage properties or static values we can create attributes of the class and annotate them with the **@Value** annotation.

To leverage a Spring configuration file, we can use import:

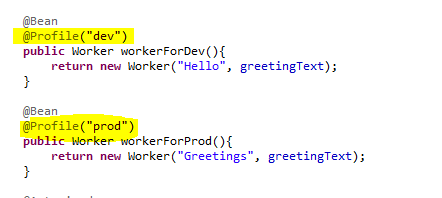


Spring provides a very simple way of managing these properties through the environment construct. In Spring, an environment is populated by default with all the environment variables passed to the application at startup. In addition to environment variables, however, you can load properties from the file system, class path, or even remote servers

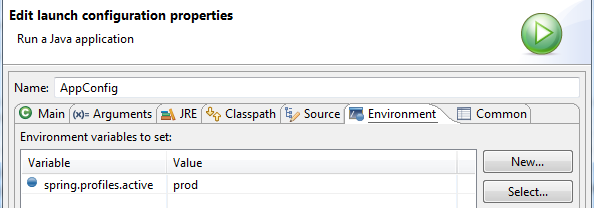




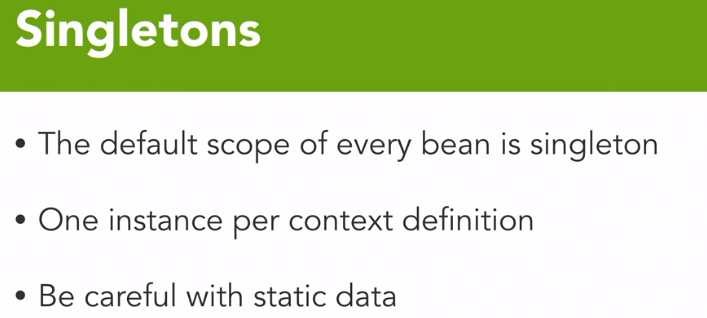
**Spring Profiles:**



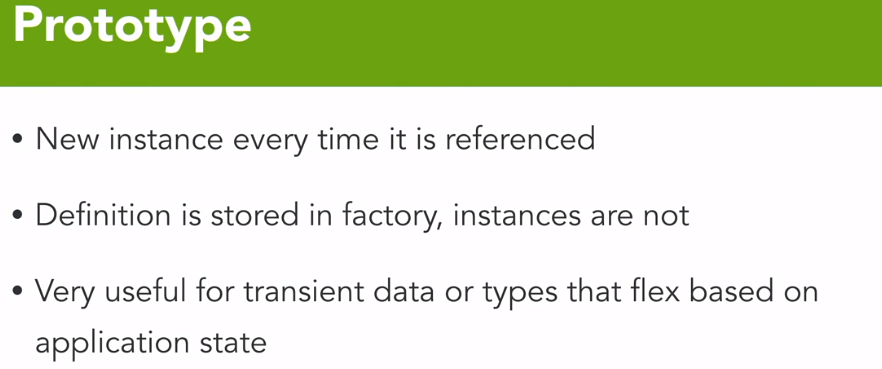
Based on the profile set in the environment variable, corresponding profile will be executed.



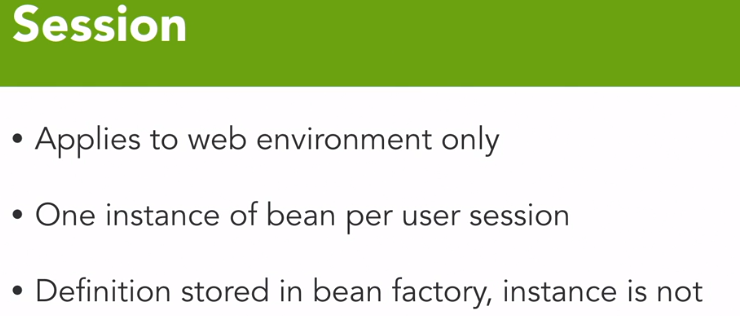
**Spring Expression Language or SpEL**



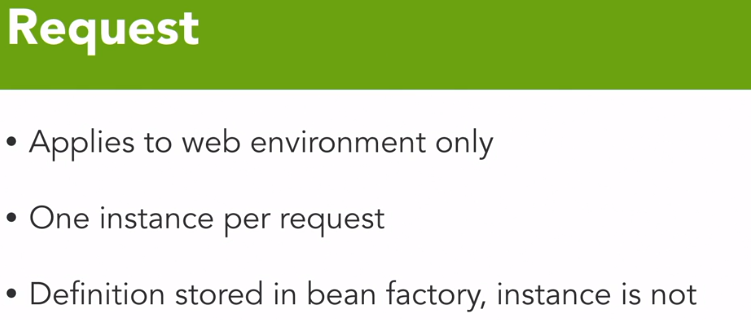
The most common bean scope and the one that you get by default, is the singleton bean. And just like its name implies, you get once instance of this bean per application context and as such, you need to be very careful with singleton beans and the storing of static data within them. However, because most beans in spring are singletons, it allows you to have a lot of flexibility in not creating a lot of instances but having a lot of behavior that is replicated across your application.



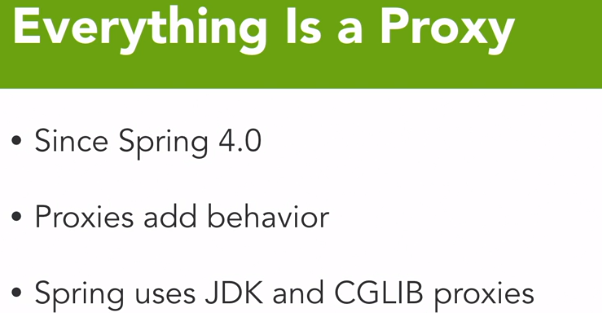
prototype bean is unique in that you get a new instance of it every time it's referenced. Its definition is stored in the bean factory, but the instances are not, and unlike a singleton bean that closes and cleans up its beans through garbage collection as the application shuts down, a prototype bean is available for garbage collection the minute the instance itself goes out of scope.



session scoped bean, as its name implies, allows you to have one instance of that bean, per user session. So, if you choose to, this allows you to store session specific data in a bean, knowing that it's going to go out of scope when the session itself goes out of scope.

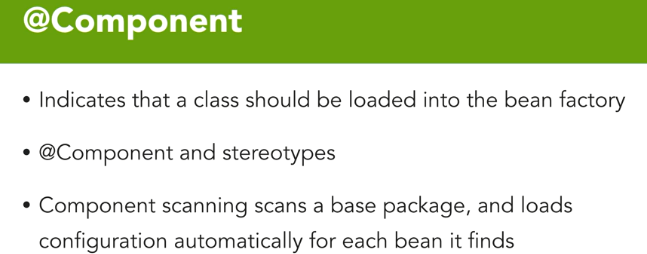


once instance of a request scope bean per request and that's a little bit more of a stateless model that we would deal with in a web environment and once again, just like prototype and session scope beans, the definition of these is stored in the bean factory, but the instance itself is not and is available for garbage collection, once it goes out of scope.

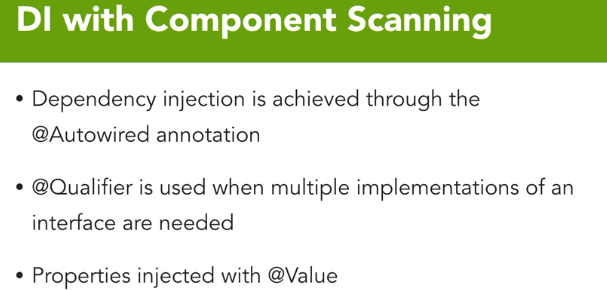


Proxies are aspects. These aspects add various behaviors to your class. Many of the common proxies include transaction management and caching. Custom aspects can be written with which we will discuss later.

Since Spring 4.0, every class you load into the bean factory gets at least one proxy. Now as I mentioned, these proxies add behavior to your classes during runtime. Spring leverages two primary mechanisms for proxies: the JDK-based proxies which leverage an interface model, and the CGLIB-based proxies which leverage a subclass model.

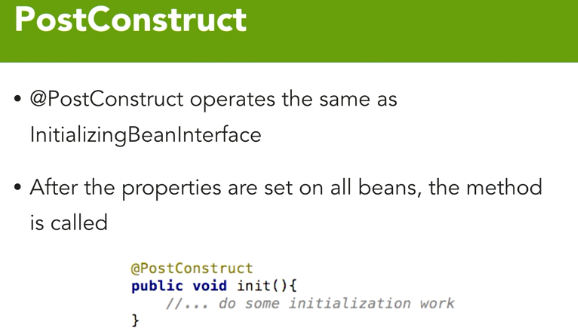


Annotation-based configuration, or component scanning, is another way of configuring your application context. Indicating beans that need to be managed by the bean factory is as simple as an annotation on the class of @component or one of its stereotypes. Typical stereotypes include **@controller**, **@service** and **@repository**. Some of these stereotypes add additional behavior to your bean through the proxies that come along with them, such as the @repository.

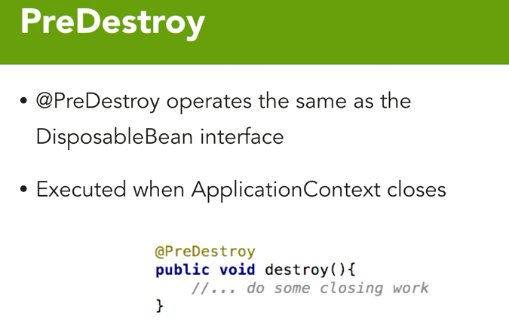


Dependency injection with component scanning is achieved through auto-wiring. Now, because you could only have a single implementation when using simple auto-wiring, Spring allows you to use the @qualifier annotation to specify which bean to inject in which case. To complement this, you can provide a bean name on the @component annotation.

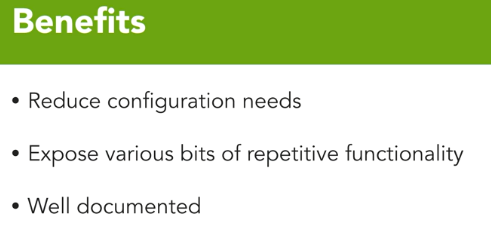
With component scanning, common configuration can be achieved by simply annotating the class, instead of defining a bean for each class. The framework will do all the work for us.

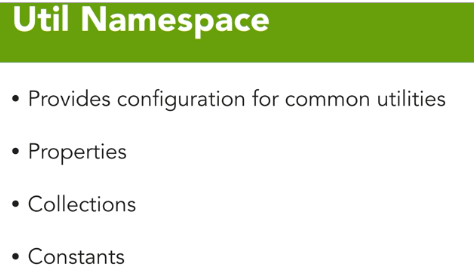


The PostConstruct annotation allows you to specify a single method on a class that is executed after construction

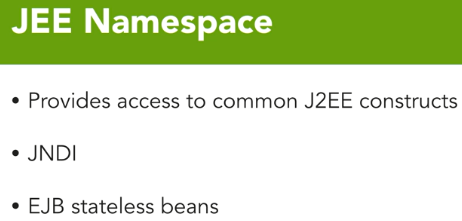


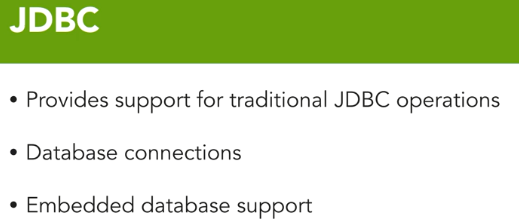
**XML namespaces:**





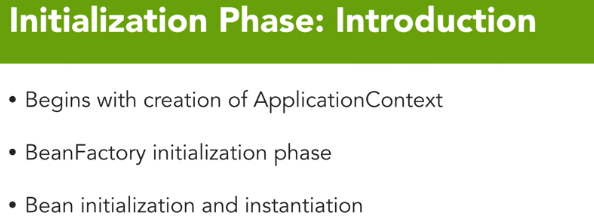
The Util Namespace provides configuration for common utilities. These include things like properties files, collections like lists and maps and sets, as well as constants. And all too often, you'll need to configure those one times and this Util Namespace makes this much easier.





**Spring Life Cycle:**

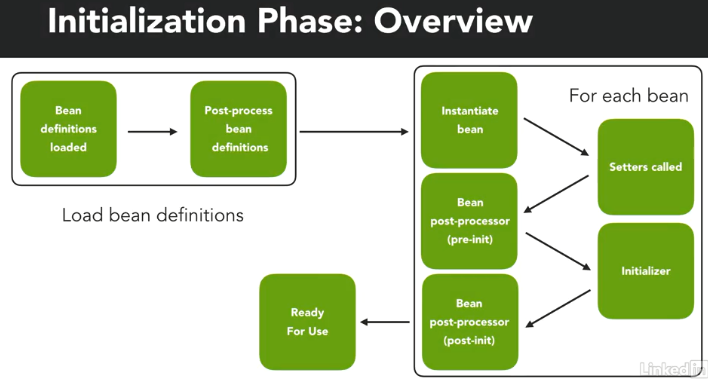
1. Initialization phase
2. Use phase
3. Destruction Phase



The initialization phase of a Spring Bean application's lifecycle begins with the creation of the ApplicationContext.

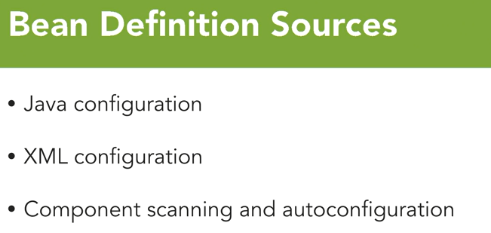
When we’re running a web application the ApplicationContext is started as part of the Servlet initialization phase itself. The Servlet calls in and starts the ApplicationContext and indeed the ServletContext is what has the handle to the ApplicationContext. The first part of the initialization phase, after the ApplicationContext starts is the BeanFactory initialization phase.

ApplicationContext, again, is the wrapper for the BeanFactory, but the BeanFactory itself must be brought up correctly. And that is done during the BeanFactory initialization phase of the Spring Bean lifecycle. After the BeanFactory is primed and ready to run we'll get into individual bean initialization and instantiation operations.



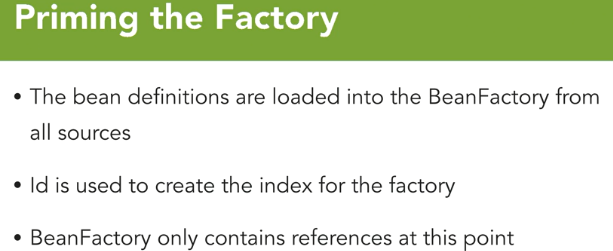
The bean definitions themselves are loaded into the BeanFactory through their metadata and then there's a process by which we do post-processing on the bean definitions themselves, called BeanFactory post-processing. Once the BeanFactory itself is initialized and ready to be used, then we iterate through all the beans in that factory and go through a process that occurs on each individual bean. Each bean is instantiated, set with appropriate setters, bean post-processors are then executed that happen before the initializer

**Bean Definitions Loaded:**



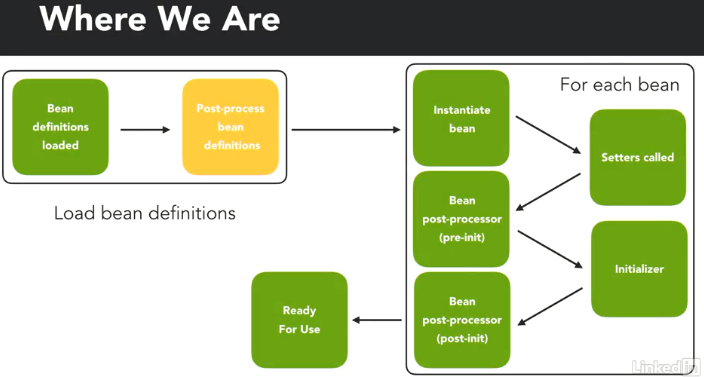
The first is through Java config which we're spent quite a bit of time looking at how to configure the application context and what we're doing here providing the metadata for the bean factory.

The second is the XML configuration which we also have looked at. And the final piece is that component scanning and auto config that can occur.



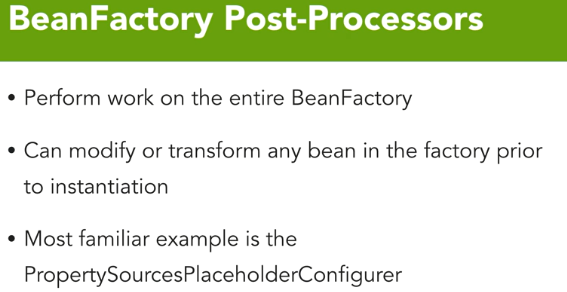
All the beans have been loaded into the bean factory from all sources.

They can be Java config and XML and then we can throw in a little auto config if we want to. And Id is used to create the index of each bean in the bean factory. Now an important thing to note here is that the bean factory at this point only contains references, no classes have been created no beans have been defined and no proxies have been wrapped around them. We only have references to beans in our bean factory at this point of the life cycle.

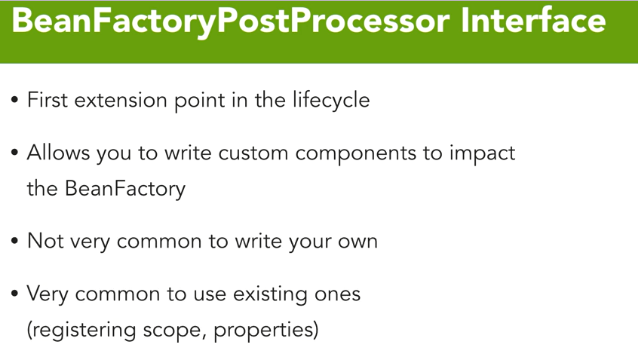


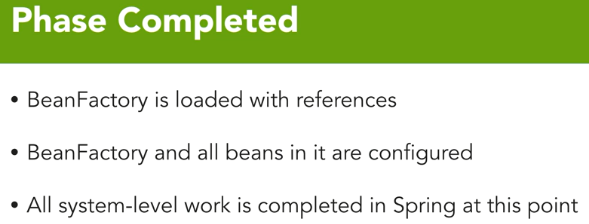
After all the beans have been loaded into the BeanFactory, Spring takes a moment to process all those beans.

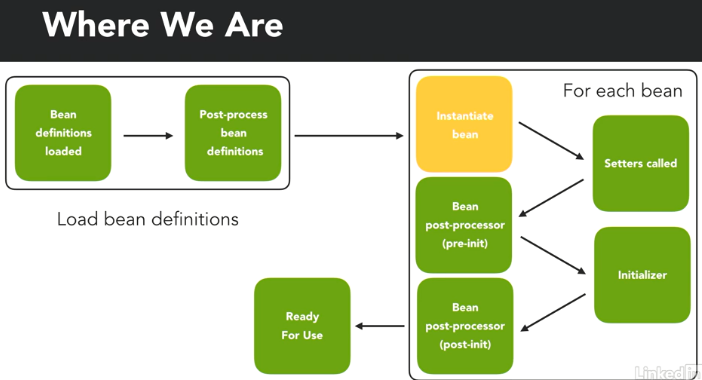
BeanFactory Post-Processors perform their work on the entire BeanFactory. They can modify or transform any bean in the factory, prior to its initialization or instantiation.



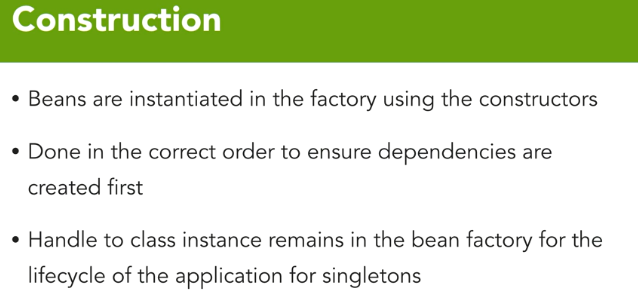
The PropertySourcesPlaceholderConfigurer takes property files, parces them, and injects the property values into the bean before it's ever instantiated, and that is the typical behavior of what a BeanFactory Post-Processor will do.





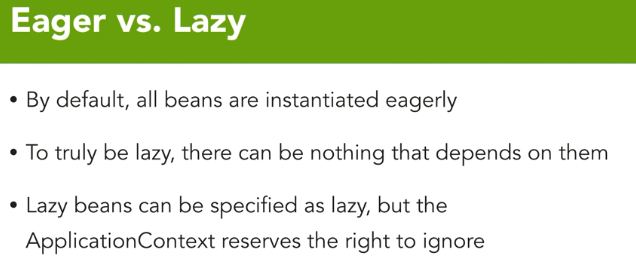


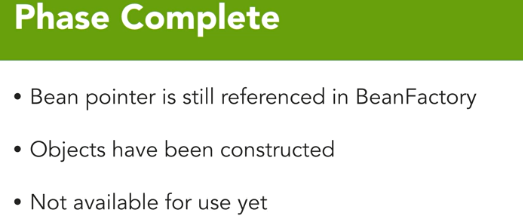
In the beginning, beans themselves are instantiated in the factory using their constructors. Now they are done in the correct order to ensure the dependencies are created first so that we're not injecting null values into beans that are bean created through this instantiation phase.

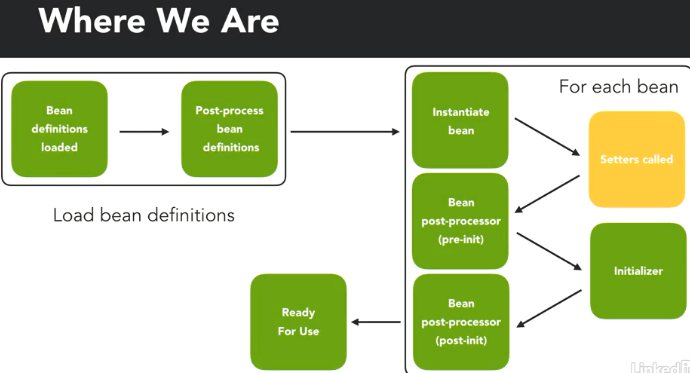


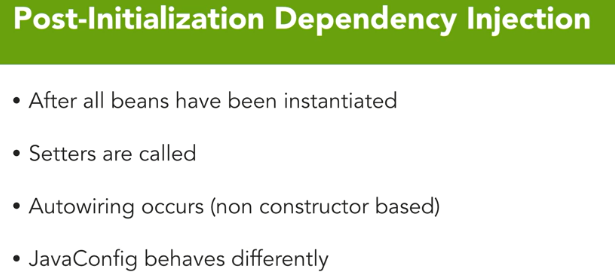
There are two ways that beans themselves can be instantiated, eager or lazy. By default, all beans are instantiated eagerly, meaning that as the bean factory itself is processed, every bean within that factory is instantiated.

For a bean to be lazily instantiated, it not only needs to be annotated as lazy, but it also must have nothing else in the bean factory that uses them as a dependency. If a bean is a dependency of another bean, it cannot be lazily instantiated

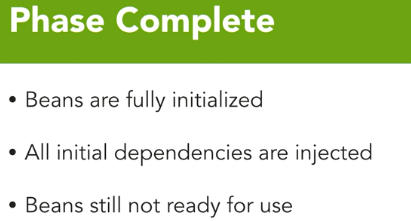


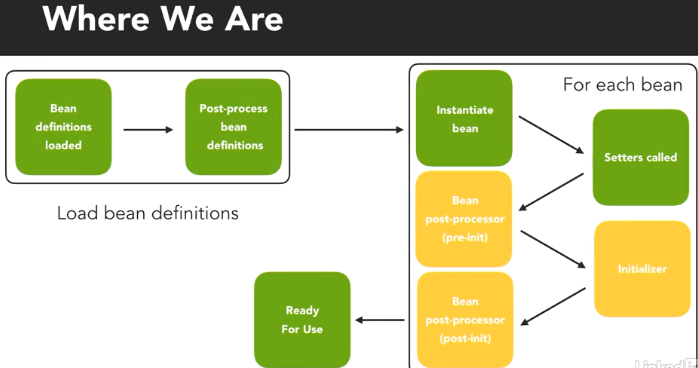


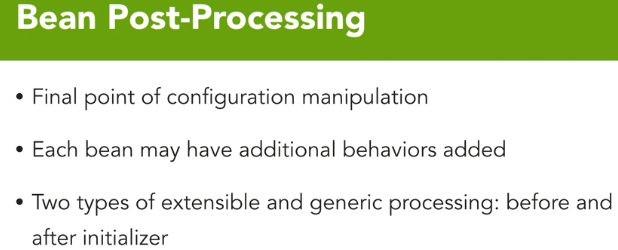




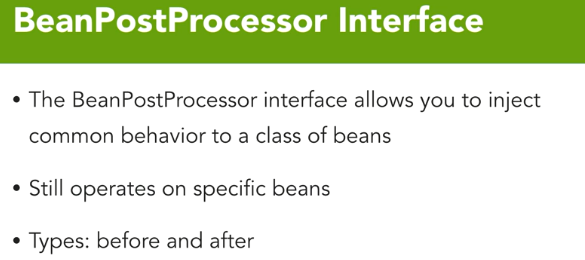
Once all these beans have been created, all those methods with set property will be called if they are appropriately constructed through property injection or auto wiring. However, field-based injection also occurs here. All auto wiring regardless of whether it's a setter or a field occurs here, except for constructor-based auto wiring.

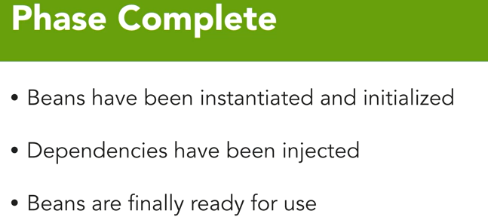


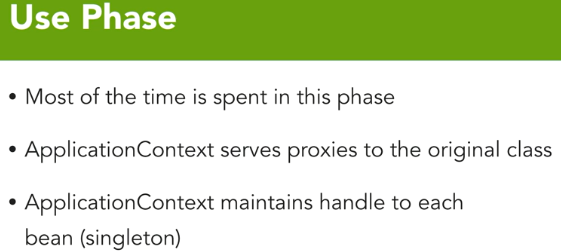


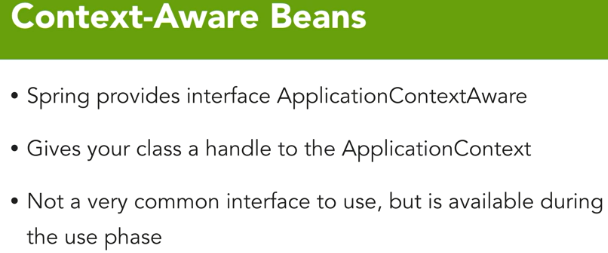


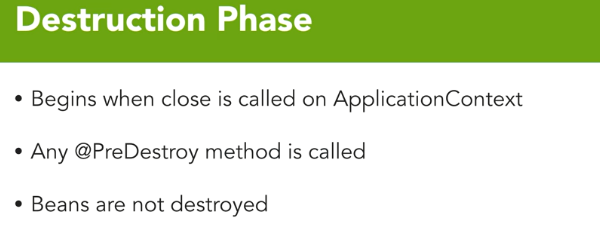
Bean post-processing is the final point of configuration manipulation. Each bean may have additional behaviors added either by the framework itself or by you the developer through extending specific interfaces. There are two types of extensible and generic processing that occurs before and after the initializer itself through this bean post-processing process.



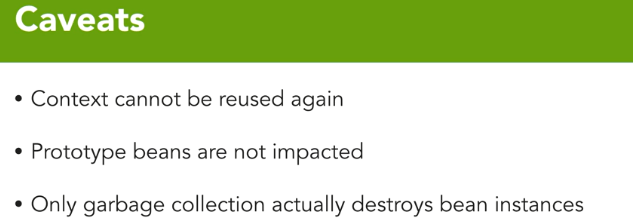








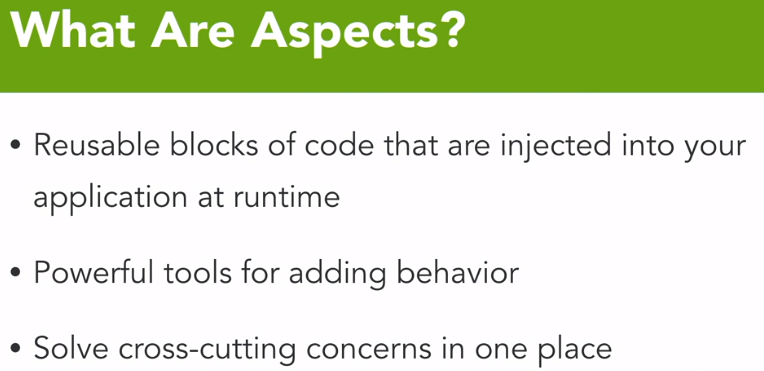
In a Java world, the only thing that can destroy a class itself is the garbage collector so calling close on the application context makes every bean contained within it go out of scope and allow it to be garbage collected during the normal processing.



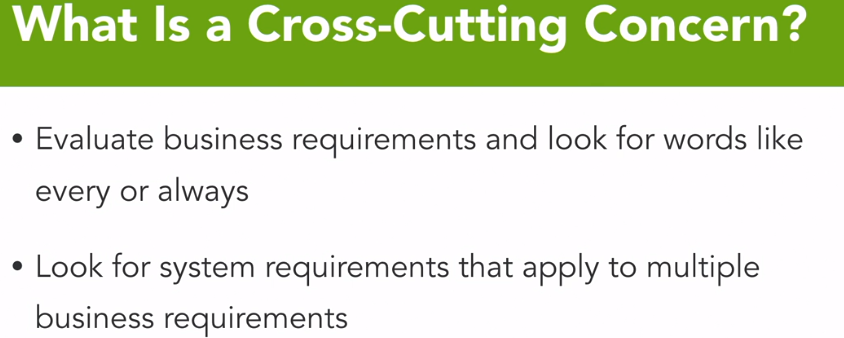
Application context itself cannot be reused again.

When close is called, the application context goes out of scope and all handles to it are released. Prototype beans are not impacted by the destruction phase either. Because a prototype bean no longer is handled by the application context once it's constructed, they go out of scope immediately when the application class itself no longer needs them. And as I previously mentioned, garbage collection is the only thing that can destroy an instance of a bean.

Nothing else that we've talked about in the destruction phase will make a bean itself be destroyed.

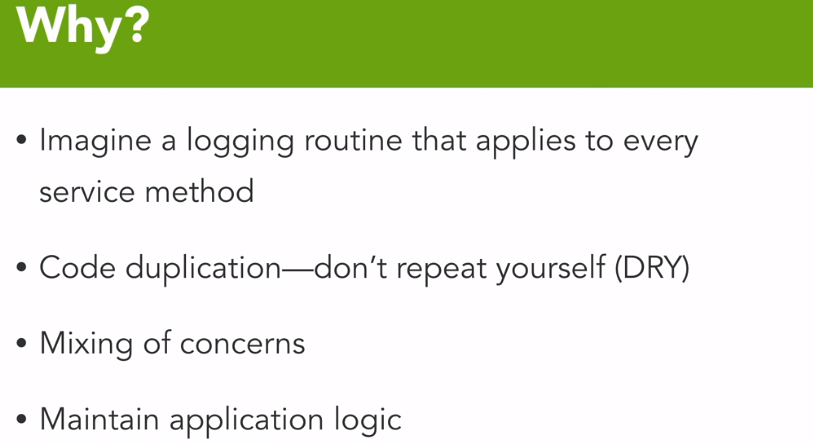




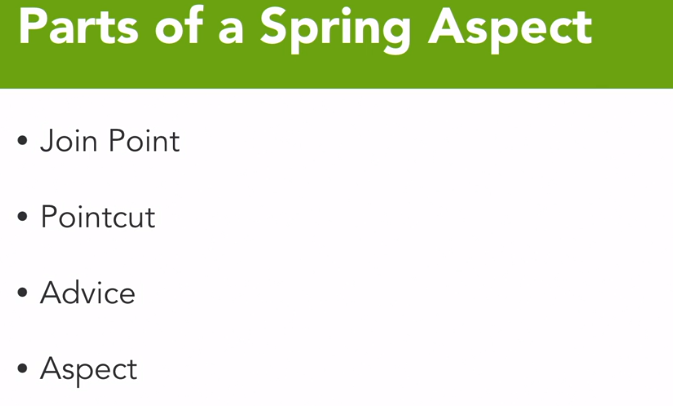


If we have a system level requirement that says any time a user logs in, log all the actions that they perform, or things like every time we execute a database method, we want specific logging written out to our system logs.

These are cross-cutting concerns that apply to system requirements and these are great places to solve that concern with an Aspect



By leveraging Aspects, we can maintain our application logic in very concise format, which improves readability as well as maintainability of our code. Now Spring Aspecting leverages AspectJ underneath for all its aspecting needs. This is done through byte code modification, so called run time interweaving our Aspects at run time.



Spring Aspects by their nature are dynamic proxy-based Aspects, which is why every being that gets injected into an application through the inversion of control container has a proxy around it.

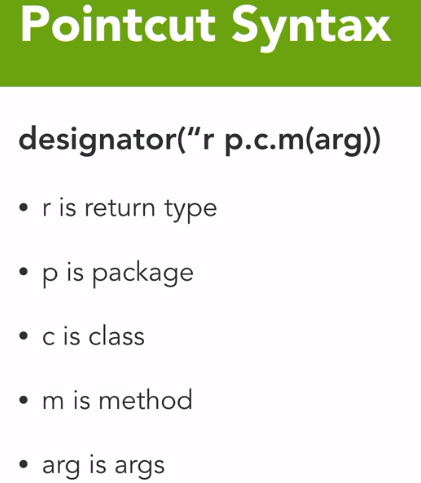
A **join point** is a point in code in the program where execution of an Aspect is targeted towards.

So, this is your method, or your line of code or your annotation that the Aspect is going to target.

The **pointcut** is the expression that identifies that join point through some sort of regular expression matching.

The **advice** is the code that you execute at a join point that was selected by a pointcut. So, the advice is your cross-cutting concern routine that we are applying to a join point in our application.

An **Aspect** is a module that contains all your pointcuts, as well as all our advice that is then injected at the run time of your application.



The pointcut is the expression that is used to identify the selection criteria of the join-point. There's a designator and then we need to specify the package class and method. Each of these values can be replaced with wildcards to select groups of join-points.

