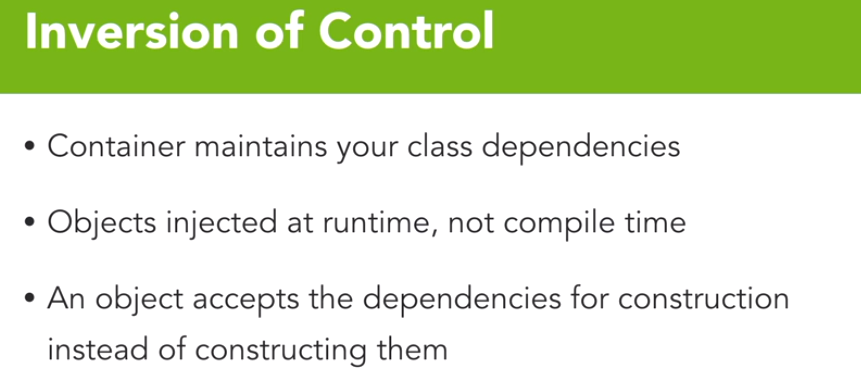
Spring IOC container manages all dependencies at run time for us. Once they're configured we never have to construct them again.

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.

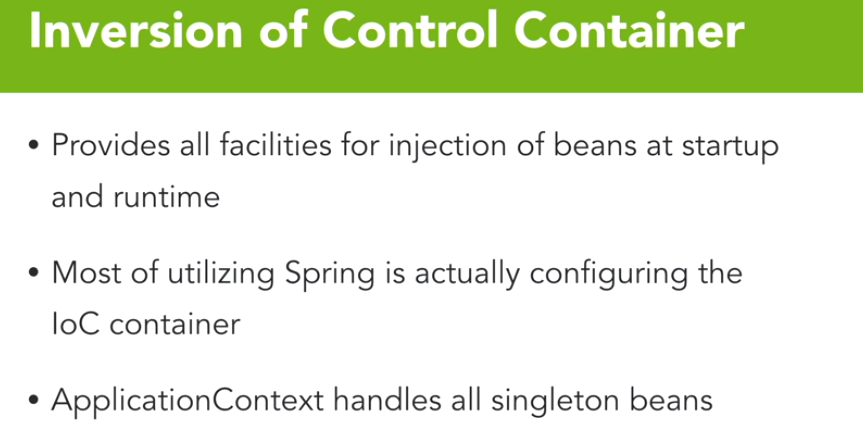


Inversion of control is essentially a way of dealing with dependency injection. Through the inversion of control container, Spring manages all our class dependencies that we rely on. And in doing so, those objects are injected at runtime, as opposed to being created at compile time. This reduces the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

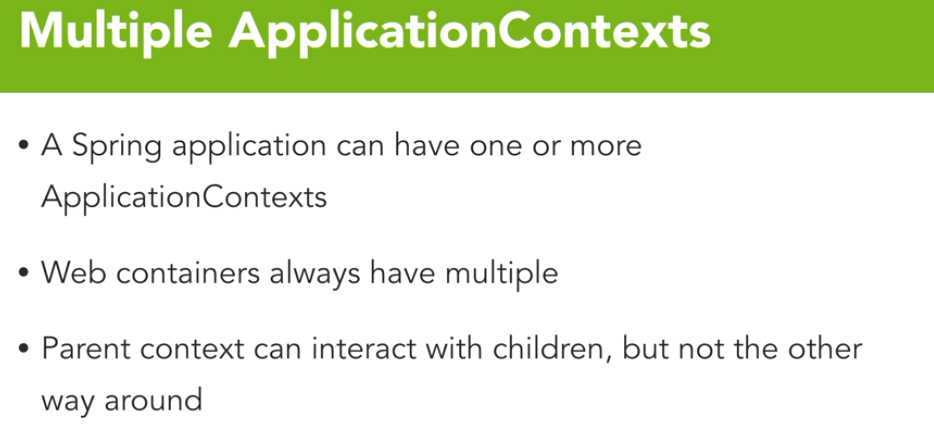
Benefits of dependency injection or inversion of control, as I just briefly alluded to, is the reduction of noise in your code. When dealing with dependency injection, because we do not copy and paste that construction code over and over, our code can focus on the business logic and not all this construction 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.

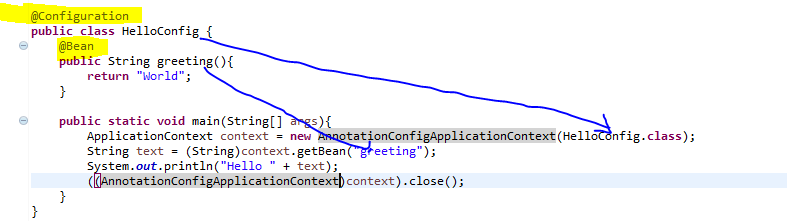


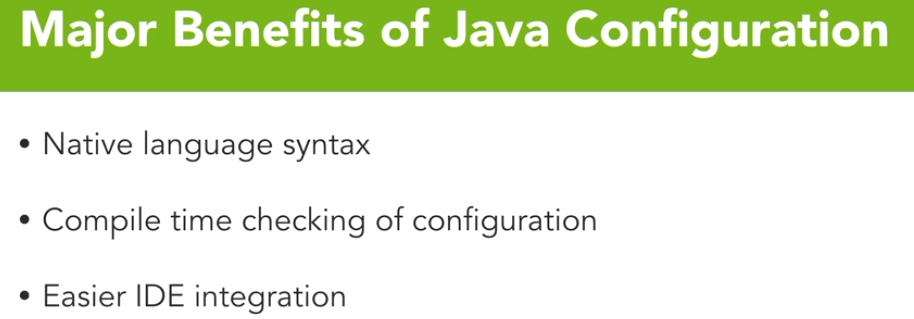
The ApplicationContext is the heart of an application leveraging the Spring Framework. ApplicationContext is a read-only wrapper of that BeanFactory, and all our runtime interactions with the BeanFactory or any beans contained in that factory is through this ApplicationContext. The ApplicationContext 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.

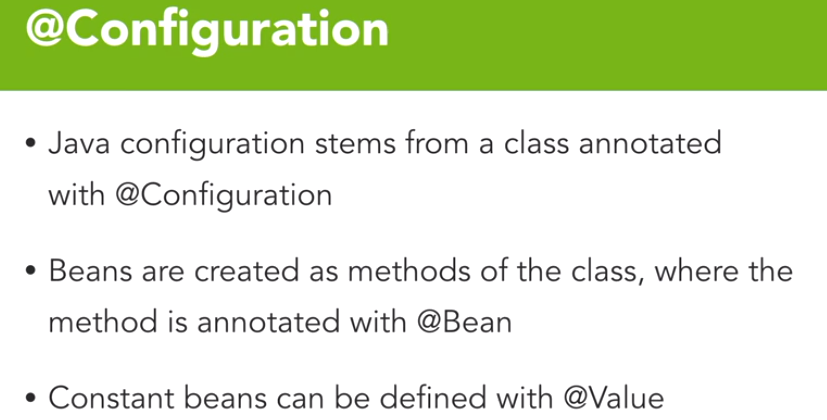


The ApplicationContext manages all our singleton beans in the application itself. It also serves all prototype or session scoped beans.









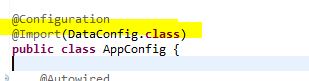
**@Configuration annotation**

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

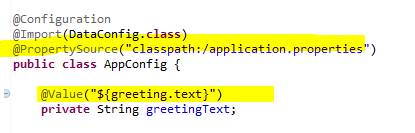
To define a Spring Bean for injection by the application context, we can create a method and annotate it with the **@Bean** annotation. The name of the bean will be the method name that we choose in our class file.

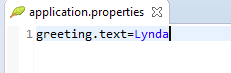
To leverage properties or static values we can create attributes of the class and annotate them with the **@Value** annotation.

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

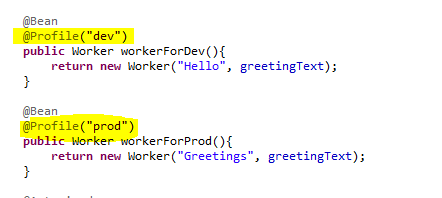


Spring provides a very simple way of managing these properties through the environment construct. In Spring, an environment is populated by default with all the environment variables passed to the application at startup. In addition to environment variables, however, you can load properties from the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

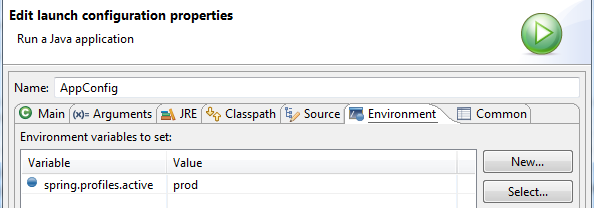




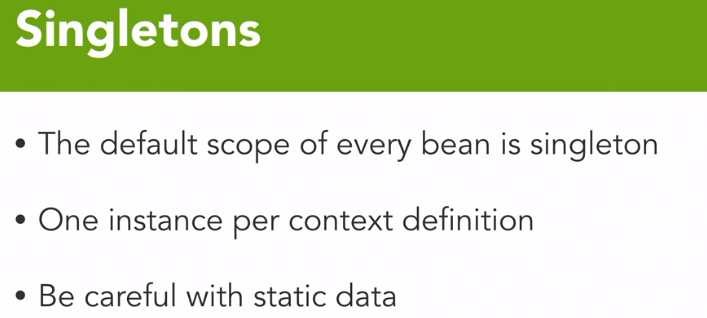
**Spring Profiles:**



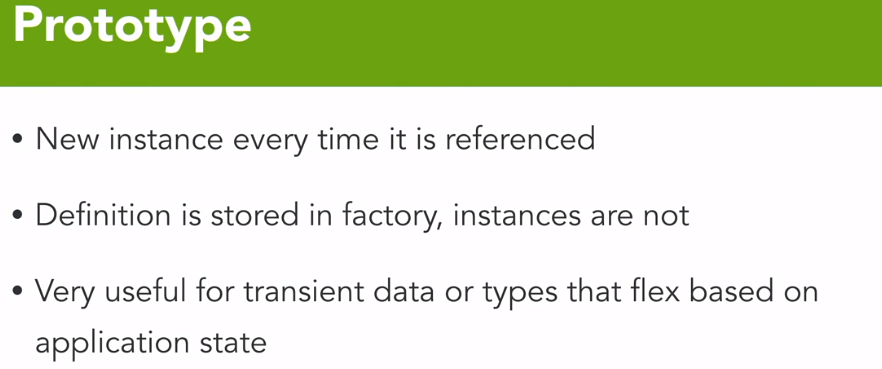
Based on the profile set in the environment variable, corresponding profile will be executed.



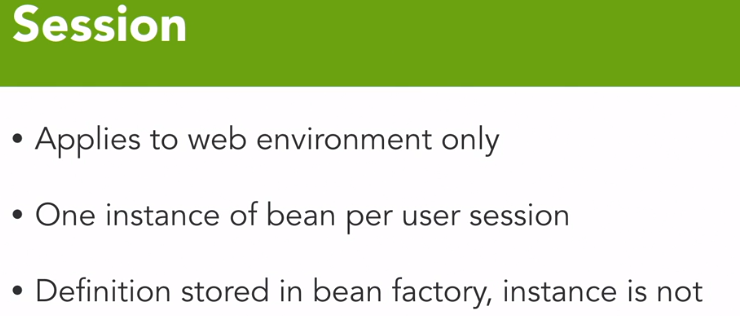
**Spring Expression Language or SpEL**



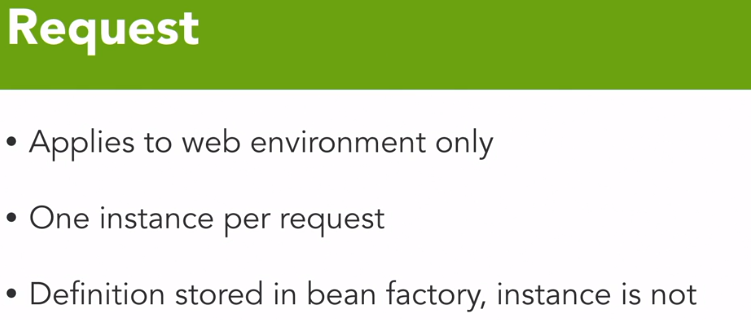
The most common bean scope and the one that you get by default, is the singleton bean. And just like its name implies, you get once instance of this bean per application context and as such, you need to be very careful with singleton beans and the storing of static data within them. However, because 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.



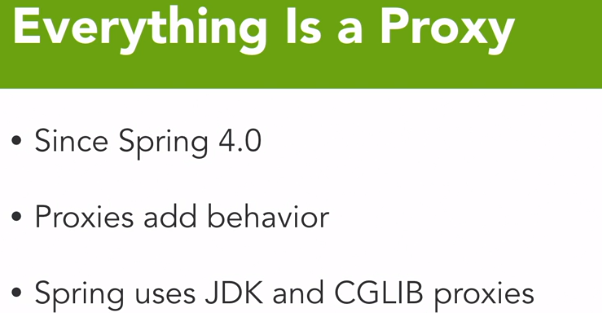
prototype bean is unique in that you get a new instance of it every time it's referenced. Its definition is stored in the bean factory, but the instances are not, and unlike a singleton bean that closes and cleans up its beans through garbage collection as the application shuts down, a prototype bean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