**IoC** can be decomposed into two subtypes: dependency injection and dependency lookup

 In dependency pull , dependencies are pulled from a registry as required

that is, via the JNDI API to look up an EJB component

CONSTRUCTOR DEPENDENCY INJECTION

*Constructor dependency injection* occurs when a component’s dependencies are provided to it in its constructor (or constructors). The component declares a constructor or a set of constructors, taking as arguments its dependencies, and the IoC container passes the dependencies to the component when instantiation occurs.

using constructor injection is that an object cannot be created without its dependencies; thus, they are mandatory. This is eager loading.

SETTER DEPENDENCY INJECTION

In *setter* *dependency injection*, the IoC container injects a component’s dependencies via JavaBean-style setter methods. A component’s setters expose the dependencies the IoC container can manage.

An obvious consequence of using setter injection is that an object can be created without its dependencies, and they can be provided later by calling the setter. This is lazy loading.

*Constructor injection* is particularly useful when you absolutely must have an instance of the dependency class before your component is used.

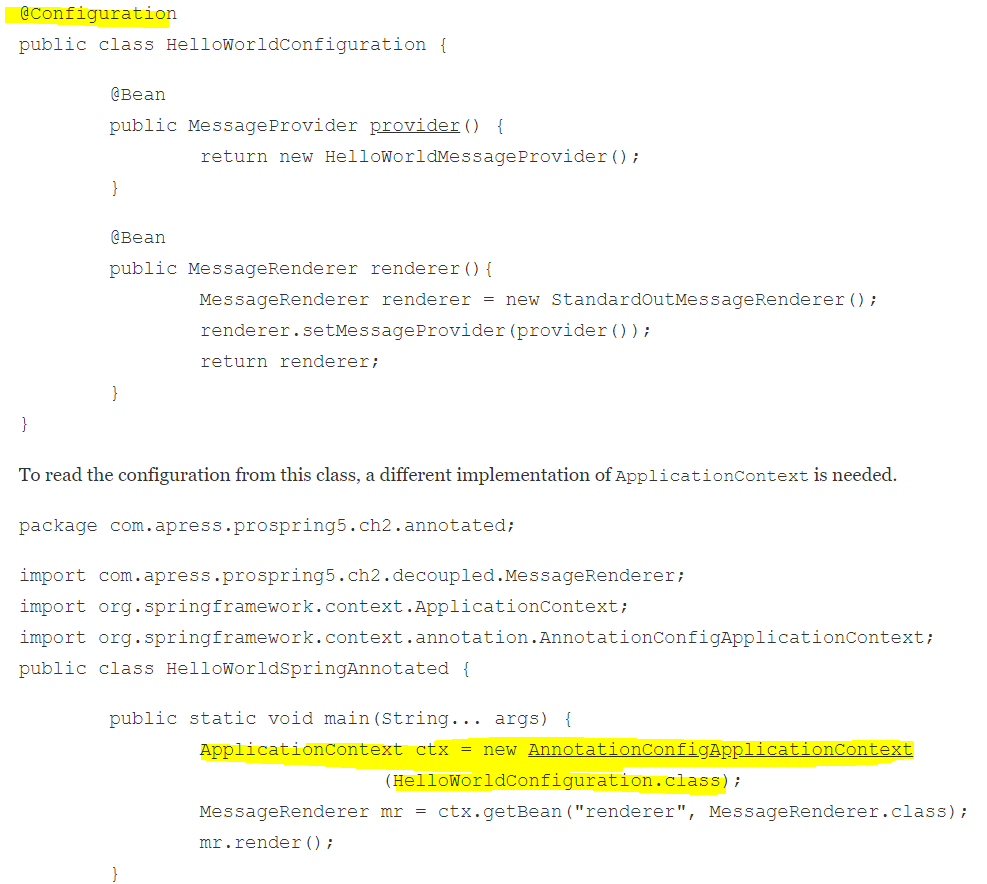
Constructor injection also helps achieve the use of immutable objects.

Setter-based injection allows dependencies to be swapped out without creating new objects and also lets your class choose appropriate defaults without the need to explicitly inject an object. Constructor injection is a good choice when you want to ensure that dependencies are being passed to a component and when designing for immutable objects.

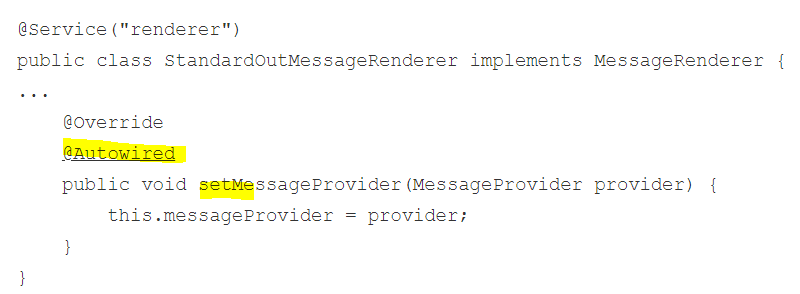
### BEANS AND BEANFACTORY

The core of Spring’s dependency injection container is the BeanFactory interface. BeanFactory is responsible for managing components, including their dependencies as well as their life cycles

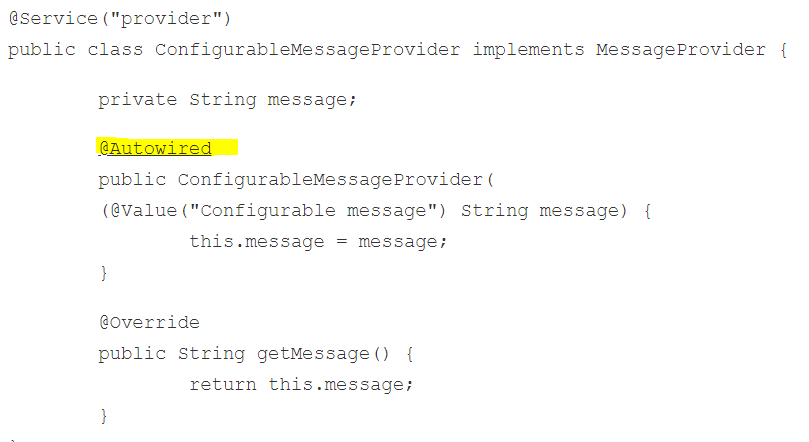
Reading configuration



Setter Injection :



Constructor Injection:



@Value, to define the value to be injected into the constructor. This is the way in Spring we inject values into a bean

 this is why using field injection is usually avoided.

* Although it is easy to add dependencies this way, we must be careful not to violate the single responsibility principle. Having more dependencies means more responsibilities for a class, which might lead to a difficulty of separating concerns at refactoring time. The situation when a class becomes bloated is easier to see when dependencies are set using constructors or setters but is quite well hidden when using field injection.
* The responsibility of injecting dependencies is passed to the container in Spring, but the class should clearly communicate the type of dependencies needed using a public interface, through methods or constructors. Using field injections, it can become unclear what type of dependency is really needed and if the dependency is mandatory or not.
* Field injection introduces a dependency of the Spring container, as the @Autowired annotation is a Spring component; thus, the bean is no longer a POJO and cannot be instantiated independently.
* Field injection cannot be used for final fields. This type of fields can only be initialized using constructor injection.
* Field injection introduces difficulties when writing tests as the dependencies have to be injected manually.