Short Circuit

Spring

Spring Documentation
Up until Container Extension Points

Summary by Emiel Bos

1 Introduction

The Spring universe actually consists of many open-source projects. The first and most important of these is the Spring Framework; all the others are built on top of it.

2 Spring Framework

Spring Framework is, overly simplified, a dependency injection container, with a couple of convenience layers on top. *Dependency injection* is a programming technique in which an object or function receives other objects or functions that it requires from some external source/container/framework, as opposed to creating them internally (with Java's `new` keyword). The dependencies are passed, or "injected", through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. Suppose you have a class (e.g. a data access object (DAO) with methods that queries a database) that requires/depends on some object (e.g. a database connection); its *dependency*, also sometimes called its *collaborator*. You could have this class construct the object with `new`, but if another class depends on the same object, you either need to copy the `new` code or have some getter that returns the same object. This would quickly get messy with multiple classes and dependencies. A natural solution is to use a singleton enum class that contains the dependency, and then all classes that need it can get it from there, which also saves on memory and construction time footprint. However, the classes still need to actively know where to get the dependency from, and the more dependencies, the more enums you need. *Inversion of Control* (IoC) solves this with *dependency injection*. The class has no control over how, where, and when they get their dependencies; rather, if you want to use the class, you have to give it its dependency. Of course, you would still need to manually construct these dependencies and construct instances of the classes in some main part of your program. This is where Spring Framework comes in. It knows which classes have which dependencies and then constructs both the class instances and their dependencies for you. DI/IoC results in cleaner code and more effective decoupling.

2.1 IoC container

The `org.springframework.context.ApplicationContext` interface represents the Spring IoC container. The IoC container manages one or more beans. *Beans* are simply Java objects that are instantiated, assembled, and otherwise managed by a Spring IoC container. The container is not limited to managing actual JavaBeans (classes with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container), but most Spring users prefer those. `ApplicationContext` is a subinterface of `org.springframework.beans.factory.BeanFactory`; the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality, namely easier integration with Spring's AOP features, message resource handling (for use in internationalization), event publication, and application-layer specific contexts such as the `WebApplicationContext` for use in web applications. The `ApplicationContext` IoC container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata.

2.2 Beans

Beans are represented in the container as `BeanDefinition` objects, of which its metadata contains (among others) a package-qualified name of its implementation class, behavioral configuration elements (e.g. scope, lifecycle callbacks, etc.), references to other beans that are its dependencies, other configuration settings to set in the newly created object (e.g. the size limit of the pool, the number of connections to use in a bean that manages a connection pool, etc.). A specified class in the bean metadata definition is just an initial class reference, but there are multiple ways in which the runtime type may differ. The recommended way to find out is `BeanFactory.getType()`, which returns the type of object that a `BeanFactory.getBean()` call is going to return for the same bean name.

Bean definitions can define a scope for the bean with the `scope` attribute in the `<bean/>` element:

- A `"singleton"` scope indicates that all dependent classes share the same bean instance. Such beans are created when the container is created (but they can be forced to be *lazily initialized*, such that they're only initialized when first needed). All other types of bean are created only once they're requested. The Spring IoC container creates exactly one instance and stores it in a cache of such singleton beans, from which it is returned for all its subsequent requests and references. Generally used for stateless beans. It is the default scope, and is by far the most occurring type of scope. Most Spring applications consist almost entirely of singleton beans, with the occasional other bean scope sprinkled in.
- A `"prototype"` scope indicates that each dependent class gets its own distinct bean instance; a new bean instance is created every time it is requested or references by other beans. Generally used for stateful beans. In some respects, the IoC container's role in regard to a prototype-scoped bean is a replacement for the Java `new` operator. All lifecycle management past that point must be handled by the client; Spring does not manage the complete lifecycle of a prototype bean. The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, but doesn't store a record of that prototype instance.

Scopes only valid in the context of a web-aware Spring `ApplicationContext` (else an `IllegalStateException` complaining about an unknown bean scope is thrown) [These require configuration: `https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-other-web-configuration`]:

- A `"session"` scope indicates that a new bean instance is created for each and every HTTP session.
- A `"request"` scope indicates that a new bean instance is created for each and every HTTP request.
- A `"session"` scope indicates that a new bean instance is created once for the entire web application. This is somewhat similar to a Spring singleton bean, but it is a singleton per `ServletContext`, not per Spring `ApplicationContext` (for which there may be several in any given web application), and it is actually exposed and therefore visible as a `ServletContext` attribute.
- A `"websocket"` scope is associated with the lifecycle of a WebSocket session and applies to STOMP over WebSocket applications.

[Dit gaat nog over AOP en scopes: `https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html#beans-factory-scopes-other-injection`]

You can even define your own scopes.[1] To integrate your custom scopes into the Spring container, you need to implement the `org.springframework.beans.factory.config.Scope` interface, which has four methods to get objects from the scope (`Object get(String name, ObjectFactory<?> objectFactory)`), remove them from the scope (`Object remove(String name)`), and let them be destroyed by registering a callback that the scope should invoke when it is destroyed or when the specified object in the scope is destroyed (`void registerDestructionCallback(String name, Runnable destructionCallback)`). The `String getConversationId()` method obtains the conversation identifier for the underlying scope (e.g. the session identifier for a session-scoped implementation). After you write a custom `Scope` implementation, you need to make the Spring container aware of your new scopes: `void registerScope(String scopeName, Scope scope)`, a method declared on the `ConfigurableBeanFactory interface`, which is available through the `BeanFactory` property on most of the concrete `ApplicationContext` implementations that ship with Spring. You can then use the supplied `scopeName` as a scope.

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 assigned. The IoC container can detect circular dependencies (basically chicken-and-egg scenarios), in which case it will throw a `BeanCurrentlyInCreationException`.

Passing dependencies as constructor arguments and passing dependencies as arguments to a factory method are very

[1] Or even redefine existing scopes, although this is considered bad practice and you cannot override the built-in singleton and prototype scopes.

similar, and are called *constructor-based* DI. *Setter-based* DI is accomplished by the container calling setter methods on your beans after instantiating your bean. It is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies (that can be assigned reasonable default values). Values are either references to other managed beans (collaborators) or as primitive values defined inline.

Spring offers a wide range of `...Aware` interfaces that let beans indicate to the container that they require a certain infrastructure dependency, even though this is not recommended, as it ties your code to the Spring API and does not follow the Inversion of Control style. When an `ApplicationContext` creates an object instance that implements such an `...Aware` interface, the instance is provided with the corresponding dependency. As a general rule, the name indicates the dependency type. For example, `ApplicationContextAware` declares `setApplicationContext(ApplicationContext applicationContext)`, which `ApplicationContext` uses to give the bean a reference to itself during its creation. One use would be the programmatic retrieval of other beans. Beans can also cast the reference to a known subclass of this interface (such as `ConfigurableApplicationContext`, which exposes additional functionality). Implementing the `BeanNameAware` interface provides the bean with its own name. There are many more `...Aware` interfaces.

2.3 Configuration

Configuration metadata can be represented in XML, Java annotations, or Java code, and the container is totally decoupled from this format. XML was historically the main way of supplying configurations, but Java-based configurations are the most common nowadays.

2.3.1 XML-based configuration

XML-based configuration metadata configures beans as `<bean/>` elements inside a top-level `<beans/>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

   <import resource="external.xml"/> <!-- Load bean definitions from another file -->

   <!-- id is a string that identifies the individual bean definition -->
   <!-- class is the fully qualified class name and defines the type of the bean -->
  <bean id="myClass" class="org.mypackage.MyClass">
      <property name="myDependency" ref="myDependency"/> # ref refers another bean name
    <!-- additional dependencies and configuration for this bean go here -->
  </bean>

  <bean id="myDependency" class="MyDependency">
    <!-- dependencies and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

You then supply this XML file location as a `String` to the constructor of an `ApplicationContext` implementation, such as `ClassPathXmlApplicationContext` if the file is in the Java classpath, or `FileSystemXmlApplicationContext` if the file is somewhere else in your filesystem. Bean definitions can span multiple XML files, e.g. when each individual XML configuration file represents a logical layer or module in your architecture. You can supply multiple `String`s to the `ApplicationContext` constructor, or use the `<import/>` element to import definitions into one XML.

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

A bean has an identifier specified with the `id` attribute that must be unique within its container. One or more comma-separated identifiers called aliases can be specified in the optional `name` attribute. If you do not supply a `id` or `name`, the container generates a unique name for that bean, but you mush if you want to refer to that bean by name (e.g. with the `ref` attribute or a Service Locator style lookup). You can also specify an alias outside a bean definition, with `<alias name="fromName"alias="toName"/>`. You can use the `depends-on` attribute to specify one or more (comma-separated) identifiers of beans that should be initialized before the one at hand, useful e.g. when static initializer in a class needs to be triggered, such as for database driver registration. `depends-on` beans are also destroyed first, so this mechanism also controls shutdown order. Use `lazy-init="true"` to make the bean lazy initialized (initialized when requested instead of during creation of the container). You can make all beans lazy initialized by default with the `default-lazy-init="true"` attribute in the overarching `beans` element.

When defining a bean that is created with a `static` factory method, use the `class` attribute to specify the class that contains the factory method and the `factory-method` to specify the name of the factory method itself. It could also be that another class contains an instance (non-`static`) factory method for producing the class at hand (e.g. a `Bakery` class that has a factory method for producing `Bread`). In that case, leave the `class` attribute empty, specify the name of the bean in the current (or parent or ancestor) container that contains the instance method in the `factory-bean` attribute, and set the name of the factory method itself with the `factory-method` attribute, e.g.:

```
<!-- the factory bean, which contains a method called bakeBread() -->
<bean id="bakery" class="examples.Bakery"> </bean>

<!-- the bean to be created via the factory bean -->
<bean id="bread" factory-bean="bakery" factory-method="bakeBread"/>
```

If you have the following constructor-based Java class:

```
public class MyClass {
  private MyDependency myDependency;
  private int myValue;

  public MyClass (MyDependency myDependency, int myValue) {
    this.myDependency = myDependency;
    this.myValue = myValue;
  }
}
```

The XML could look like this:

```
<bean id="myClass" class="examples.MyClass">
  <constructor-arg ref="myDependency"/>
  <constructor-arg type="int" value="1"/>
</bean>

<bean id="myDependency" class="examples.MyDependency"/>
```

You have to specify the type of primitive type dependencies. Spring's conversion service converts these values from a `String` to the actual type of the property or argument. Constructor arguments are matched by argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, the order of the arguments in a bean definition is the order in which they are supplied to the appropriate constructor. Else, you can either use the `index` attribute of the `constructor-arg` element to explicitly specify the index of constructor arguments (e.g. `index="0"`), or you can also use the `name` attribute to identify the parameter name.

If you have the following constructor-based Java class with a `static` factory method that returns instances:

```
public class MyClass {
  private ExampleBean(...) { }

  public static MyClass createInstance(MyDependency myDependecy, int myValue) {
    return new MyClass(...);
  }
}
```

The only change would be to add the `factory-method="createInstance"` attribute to the `bean` element. The type returned by the factory method does not have to be of the same type as its containing class. The configuration is the samme for instance (non-static) factory methods, aside from the use of the `factory-bean` attribute instead of `class`.

If you have the following setter-based Java class:

```
public class MyClass {
  private MyDependency myDependency;
  private int myValue;

  public void setMyDependency(MyDependency myDependency) {
    this.myDependency = myDependency;
  }

  public void setMyValue(int myValue) {
    this.myValue = myValue;
  }
}
```

The XML could look like this:

```
<bean id="myClass" class="examples.MyClass">
  <property name="myDependency" ref="myDependency"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="myDependency" class="examples.MyDependency"/>
```

i.e. the only big change is the use of the `property` element rather than `constructor-arg`. When referencing another bean, rather than

```
<constructor-arg/property name="myDependency" value="myDependency"/>
```

its preferable to use

```
<property name="myDependency">
   <idref bean="myDependency"/>
</property>
```

because using the `idref` tag lets the container validate at deployment time that the referenced, named bean actually exists. In the first form, typos are only discovered when the client bean is actually instantiated (which may be long after the container is deployed if the bean has prototype scope).

Besides

```
<property name="myDependency" ref="myDependency"/>
```

you can do

```
<property name="myDependency">
   <ref bean="myDependency"/>
</property>
```

where the `ref` attribute in the first example and the `bean` attribute example in the second example refer to the `id` of one of the `name`s of another bean. Rather than the `bean` attribute, you can use the `parent` attribute to refer to a bean that is in a parent container of the current container. You should use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that has the same name as the parent bean.
Instead of a `ref` element, you can also inline an entire bean definition for an inner bean, e.g.

```
<bean id="outerClass" class="examples.OuterClass">
  <property name="dependency">
    <bean class="examples.InnerClass">
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

The inner bean does not need and/or use an identifier and/or scope flag. It is not possible to access inner beans independently or to inject them into collaborating beans other than into the enclosing bean. Inner beans typically simply share their containing bean's scope.[2]

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements set the properties and arguments of the Java `Collection` types `List`, `Set`, `Map`, and `Properties`, respectively:

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <property name="adminEmails"> <!-- results in a setAdminEmails(java.util.Properties) call -->
    <props>
```

[2]As a corner case, it is possible to receive destruction callbacks from a custom scope, e.g. for a request-scoped inner bean contained within a singleton bean. The creation of the inner bean instance is tied to its containing bean, but destruction callbacks let it participate in the request scope's lifecycle.

```
        <prop key="administrator">administrator@example.org</prop>
        <!-- etc. -->
      </props>
    </property>
    <property name="someList"> <!-- results in a setSomeList(java.util.List) call -->
      <list>
        <value>a list element followed by a reference</value>
        <ref bean="myDataSource" />
      </list>
    </property>
    <property name="someMap"> <!-- results in a setSomeMap(java.util.Map) call -->
      <map>
        <entry key="entry" value="some string"/>
        <entry key="a ref" value-ref="myDataSource"/>
          <!-- etc. -->
      </map>
    </property>
    <property name="someSet"> <!-- results in a setSomeSet(java.util.Set) call -->
      <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
      </set>
    </property>
  </bean>
```

The value of a map key or value, or a set value, can also be any of `bean`, `ref`, `idref`, `list`, `set`, `map`, `props`, `value`, `null`. The (keys and) values specified are type converted to the appropriate generic type specified in the corresponding `Collection` declaration, which Spring knows about by reflection. `<null/>` elements are treated as Java `null` values.

Rather than specifying dependencies (i.e. properties or constructor arguments) manually, you can let Spring *autowire* those, meaning it resolves those relationships by inspecting the contents of the `ApplicationContext`. You specify autowiring per bean, with the `autowire` attribute of the `<bean/>` element, which can be any of:

- `no`; default.
- `byName`; autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.
- `byType`; lets a property be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown. If there are no matching beans, nothing happens and the property is not set.
- `constructor`; analogous to `byType` but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

You can exclude a specific bean from being autowired by type with `autowire-candidate="false"` in the `<bean/>` element, but this doesn't work if it is matched by name. Excluded beans can still themselves use autowiring. You can also limit autowire candidates based on pattern-matching against bean names with the `default-autowire-candidates` attribute in the top-level `<beans/>` element, e.g. to limit autowire candidate status to any bean whose name ends with `Repository`, provide a value of `*Repository`. You can provide multiple comma-separated patterns. Autowiring has downsides: you cannot autowire simple properties such as primitives, `String`s, and `Class`es (and arrays of such simple properties) (so you need to set those manually; explicit dependencies in `property` and `constructor-arg` settings always override autowiring). Autowiring is less exact than explicit wiring, although Spring is careful to avoid guessing in case of ambiguity. The relationships between your Spring-managed objects are no longer documented explicitly; to some extent, explicit wiring documents the structure of a system. In short, Autowiring works best when it is used consistently across a project, and it can be especially useful during development without negating the option of switching to explicit wiring when the code base becomes more stable.

A bean can specify a parent with the `parent` attribute (or pragmatically with the `ChildBeanDefinition` class), in which case it becomes a child bean and inherits configuration data from a parent definition: scope, constructor argument values, property values, and method overrides from the parent. The child definition can override some values or add others as needed. The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, and lazy init. A child bean definition uses the bean class from the parent definition if none is specified, but can also override it, in which case the child bean class must naturally be compatible with the parent. A parent can be marked as abstract with the `abstract` attribute, in which case it cannot be instantiated and can only serve as a parent definition for child definitions.

2.3.2 Annotation-based configuration

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {

   @Bean
   @Scope("singleton")
   public MyDependency myDependency() {
      return new MyDependency();
   }

   @Bean
   public MyClass myClass() {
      return new MyClass(myDependency());
   }

}
```

The methods in the configuration are like factory methods. You can control the scope of the objects created from a particular bean definition with the `@Scope` annotation.

2.3.3 Java-based configuration

Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class.

2.4 `ApplicationContext`

`ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. It can read bean definitions from multiple, diverse configuration sources.[3]

```java
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MyApplication {
   public static void main(String[] args) {
      ApplicationContext ctx = new
          AnnotationConfigApplicationContext(MyApplicationContextConfiguration.class); // Construct
          context by passing a configuration
      MyClass myClass = ctx.getBean(MyClass.class); // Returns a fully configured MyClass, i.e. one
          with its MyDependency dependency set
      MyDependency dependency = ctx.getBean(MyDependency.class); // Returns the same MyDependency
          that it sets inside the MyClass
   }
}
```

However, ideally, your application code should have no calls to `getBean()` or any other methods for retrieving beans that `ApplicationContext` provides and thus have no dependency on Spring APIs at all.

2.5 Method injection

When a singleton bean needs to collaborate with another singleton bean or a non-singleton bean needs to collaborate with another non-singleton bean, you can define the dependency as a property of the other. A problem arises when the bean lifecycles are different, e.g. a singleton bean needing a prototype bean. In that case, a new prototype bean is instantiated and then dependency-injected into the singleton bean at instantiation time (of the container and singleton bean). This prototype instance is the sole instance that is ever supplied to the singleton-scoped bean. The container cannot provide the singleton bean with a new instance of the prototype bean later on (e.g. for each call of a certain method), because that injection occurs only once. A suboptimal solution is to forego some inversion of control by making the singleton bean aware of the container by implementing the `ApplicationContextAware` interface and by using `getBean("PrototypeBean")` call

[3]Even though typical applications work solely with beans defined through regular bean definition metadata, `ApplicationContext` implementations can register objects created outside the container (by users), which is done by accessing the `ApplicationContext`'s `BeanFactory` through `getBeanFactory()` that returns the `DefaultListableBeanFactory` implementation. `DefaultListableBeanFactory` supports this registration through `registerSingleton()` and `registerBeanDefinition()`. The registration of new beans at runtime (concurrently with live access to the factory) is not officially supported and may lead to concurrent access exceptions, inconsistent state in the bean container, or both.

to the container to ask for (a typically new) bean instance:

```
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class SingletonBean implements ApplicationContextAware {

  private ApplicationContext applicationContext;

  protected PrototypeBean createPrototypeBean() {
    return this.applicationContext.getBean("prototypeBean", PrototypeBean.class);
  }

  public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
    this.applicationContext = applicationContext;
  }

   // ...other methods that grab a new PrototypeBean instance each call...
}
```

However, the business code is aware of and coupled to the Spring Framework, which you don't want. Method injection, a somewhat advanced feature of the Spring IoC container, lets you handle this use case cleanly. You instead declare the method to be injected with the following signature:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

like so (notice no more Spring imports and/or Spring API calls):

```
public abstract class SingletonBean {

  protected abstract Command createPrototypeBean();

   // ...other methods that grab a new PrototypeBean instance each call...
}
```

The Spring Framework then dynamically generates a subclass that overrides the method (by using bytecode generation from the CGLIB library) to return the lookup result for another named bean in the container when requested. Naturally, for this to work, the class to-be-subclassed and the method to-be-overridden both cannot be `final`. In the `<bean/>` element, instead of a `<constructor-arg/>` or `<property/>` element, we can declare a `<lookup-method/>`:

```
<lookup-method name="createPrototypeBean" bean="prototypeBean"/>
```

Alternatively, within the annotation-based component model, you can declare a lookup method through the `@Lookup` annotation, e.g. `@Lookup("prototypeBean")` or simply `@Lookup` (which relies on the target bean getting resolved against the declared return type of the lookup method).[4]

2.6 Lifecycle callbacks

Spring allows beans to define (`void`, no-argument) callback methods: initialization methods are called after the container has set all necessary properties on the bean[5] to let a bean perform initialization work, and destruct methods are called when the container that contains it is destroyed. There are three ways for controlling bean lifecycle behavior. If multiple of these are used with a different method name, then each configured method is run in the following order (if the same method name is configured, that method is only run once):

- The `@PostConstruct` and `@PreDestroy` annotations on the callbacks. Generally considered best practice
- Implementing `InitializingBean` and/or `DisposableBean` interfaces and their `afterPropertiesSet()` and/or `destroy()` callback methods. Not recommended, as your beans are coupled to Spring-specific interfaces.
- Custom callback methods specified in the configuration. With Java configuration, use the `initMethod` and/or `destroyMethod` attribute of `@Bean`. With XML, use the `init-method` and/or `destroy-method` attributes of a `<bean/>`

[4]Note that you should typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring's component scanning rules where abstract classes get ignored by default. This limitation does not apply to explicitly registered or explicitly imported bean classes.

[5]But before any AOP interceptors and so forth are applied to the bean.

– or the `default-init-method` and/or `default-destroy-method` of `<beans/>` – to specify them. You may assign `(inferred)` to `destroy-method`, which instructs Spring to automatically detect a public `close()` or `shutdown()` method on the bean class. This is the default behavior for `@Bean` methods in Java configuration classes and automatically matches `java.lang.AutoCloseable` or `java.io.Closeable` implementations.

Internally, the Spring Framework uses `BeanPostProcessor` implementations – and you can implement your own – to process any callback interfaces it can find and call the appropriate methods.
Initialization methods in general are executed within the container's singleton creation lock. The bean instance is only considered as fully initialized and ready to be published to others after returning from the `@PostConstruct` method. For a scenario where expensive post-initialization activity is to be triggered, e.g. asynchronous database preparation steps, your bean should either implement `SmartInitializingSingleton.afterSingletonsInstantiated()` or rely on the context refresh event: implementing `ApplicationListener<ContextRefreshedEvent>` or declaring its annotation equivalent `@EventListener(ContextRefreshedEvent.class)`. Those variants come after all regular singleton initialization and therefore outside of any singleton creation lock.

In addition to the initialization and destruction callbacks, beans may also implement the `Lifecycle` interface, which defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process), as driven by the container's own lifecycle: `start()` and `stop()`. These are called when the `ApplicationContext` itself receives start and stop signals. On regular shutdown, all `Lifecycle` beans first receive a stop notification before the general destruction callbacks are being propagated. However, on hot refresh during a context's lifetime or on stopped refresh attempts, only destroy methods are called. For fine-grained control over auto-startup and for graceful stopping of a specific bean (including startup and stop phases), consider implementing the extended `SmartLifecycle` interface instead.
If a "depends-on" relationship exists between any two beans, the dependendee starts after its dependency and it stops before it. However, the direct dependencies may be unknown, and you may only know that objects of a certain type should start prior to objects of another type. In those cases, the `SmartLifecycle` interface defines the `getPhase()` method as defined on its super-interface, `Phased`. Objects with the lowest phase start first and stop last, and objects with larger phases likely depend on other processes to be running. The defaullt phase is `0` for beans implementing the regular `Lifecycle`. `SmartLifecycle`' `stop(Runnable callback)` accepts a callback on which the `run()` method has to be called by any implementing code after that implementation's shutdown process is complete. This enables asynchronous shutdown; `DefaultLifecycleProcessor` waits up to its timeout value (default is 30 seconds) per phase for the group of beans within that phase to call this callback. You can change the default timeout by implementing `LifecycleProcessor` in a bean called `lifecycleProcessor` and setting `<property name="timeoutPerShutdownPhase"value="10"/>`.
`LifecycleProcessor` is an extension of the `Lifecycle` interface, adding two callback declarations for reacting to the context being closed and reacting to the context being refreshed. `onClose()` drives the shutdown process as if stop() had been called explicitly, but it happens when the context is closing. `onRefresh()` enables another feature of `SmartLifecycle` beans: when the context is refreshed (after all objects have been instantiated and initialized), that callback is invoked. At that point, the default lifecycle processor checks the boolean returned by each `SmartLifecycle` object's `isAutoStartup()` method. If `true`, that object is started at that point rather than waiting for an explicit invocation of the context's or its own `start()` method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The phase value and any "depends-on" relationships determine the startup order as described earlier.

Spring's web-based ApplicationContext implementations already have code in place to gracefully shut down the Spring IoC container and calling relevant destroy methods on singleton beans when the relevant web application is shut down, but in a non-web application environment you have to register a shutdown hook with the JVM by calling `registerShutdownHook()` on your `ConfigurableApplicationContext`.