


Bean lifecycle

Theory

Practice

 100% completed, 5 problems solved

▼

Theory

⌚ 16 minutes reading

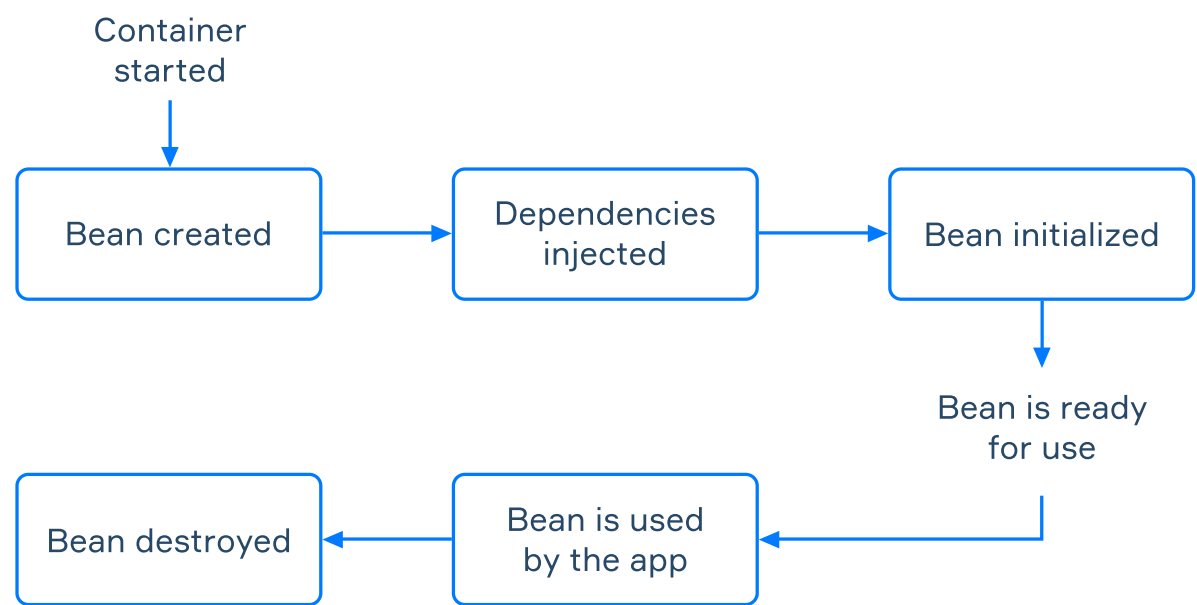
Start practicing

One of the main responsibilities of the Spring Container is managing objects known as **beans**, which form the backbone of any Spring application. Managing these objects is a complex process consisting of multiple steps, and it uses a lot of internal concepts of the framework.

In this topic, we will dive into the lifecycle of beans and learn about some additional functionalities provided by the framework to customize the lifecycle. For simplicity, we will focus only on the most essential parts of the lifecycle for now because learning all of them at once might be quite overwhelming for a single topic.

§1. High-level overview of bean lifecycle

As you know, when a Spring application is launched, the Spring Container gets started. The container is mainly responsible for managing the **lifecycle** of beans from their creation to destruction. The following picture gives us a high-level overview of it.




As you can see, once the container is started, the lifecycle of a bean begins. The container creates a new bean object, then injects its dependencies and performs some additional initialization that can be customized by a programmer. After that, the bean is ready to be used by the application. The lifecycle of a bean ends with its destruction when the container is shut down.


The presented lifecycle is the same for both `@Bean`-annotated methods and `@Component`-annotated classes. However, there is a significant difference between singletons and beans annotated with `Scope("prototype")`. Spring doesn't destroy prototypes and doesn't allow us to customize them.

Considering that any Spring application contains multiple beans and the beans have dependencies on each other, the container creates them in the right order according to their dependencies.

Note that separate parts of this lifecycle can be customized according to the specific needs of a particular application. Bean initialization and destruction are especially worth considering because programmers often customize them. Other


1 required topic



 [Scopes of beans](#)

▼

2 dependent topics

 [Logging in Spring Boot](#)

▼

[Application layers](#)

▼

parts of the lifecycle can be adjusted as well, however, it is required less often in practice.

§2. Customizing bean initialization and destruction

To get a bean into the **ready state**, some **initialization** after dependency injection might be required. The Spring container does it automatically, but it also allows us to customize the initialization according to the needs of our application. For example, we can load some resources, read a file, connect to a database, and so on. At the same time, when a bean is no longer required in the application, some custom cleanup may be required before destroying the bean, such as closing some connections, cleaning files, and so on.

There are several ways to add these customizations to your code:

- using special annotations (`@PostConstruct` , `@PreDestroy` , `@Bean`);
- implementing some interfaces (`InitializingBean` , `DisposableBean`);
- using an XML bean definition file, which is an outdated way mostly used for legacy applications (we will skip it here).

No matter which way you choose, you need to have separate methods that conduct some custom bean initialization and destruction. These are the rules for such methods:

- the methods can have any names and access modifiers;
- the methods must not have any arguments, otherwise, an exception will be thrown.

To demonstrate different customization ways, we will use a simple example with a library of technical books that comprises an in-memory list of strings. It should be enough for now to give you the main idea.

§3. Using annotations for customization

The simplest way to customize the initialization and destruction processes is to add the `@PostConstruct` and `@PreDestroy` annotations to the methods of a container-managed class.

Here is an example with the `init` and `destroy` methods annotated with `@PostConstruct` and `@PreDestroy`. If we run an application containing this component, Spring will call the annotated methods only once.

▼ Java

```
1  @Component
2  class TechLibrary {
3      private final List<String> bookTitles =
4          Collections.synchronizedList(new ArrayList<>());
5
6      @PostConstruct
7      public void init() {
8          bookTitles.add("Clean Code");
9          bookTitles.add("The Art of Computer Programming");
10
11          bookTitles.add("Introduction to Algorithms");
12
13          System.out.println("The library has been initialized: " +
bookTitles);
14      }
15
16      @PreDestroy
17
18      public void destroy() {
```

```
1
6         bookTitles.clear();
1
7         System.out.println("The library has been cleaned: " +
bookTitles);
1
8     }
1
9 }
```

▼ Kotlin

```
1  @Component
2  class TechLibrary {
3      private val bookTitles: MutableList<String> =
Collections.synchronizedList(ArrayList())
4
5      @PostConstruct
6      fun init() {
7          bookTitles.add("Clean Code")
8          bookTitles.add("The Art of Computer Programming")
9          bookTitles.add("Introduction to Algorithms")
1
0          println("The library has been initialized: $bookTitles")
1
1      }
1
2
1
3      @PreDestroy
1
4      fun destroy() {
1
5          bookTitles.clear()
1
6          println("The library has been cleaned: $bookTitles")
1
7      }
1
8  }
```

Here is the output produced by the `TechLibrary` class:

```
The library has been initialized: [Clean Code, The Art of Computer
Programming, Introduction to Algorithms]

2022-04-22 12:08:06.515  INFO Started HsSpringApplication in 0.382
seconds (JVM running for 5.698)

The library has been cleaned: []

Process finished with exit code 0
```

If you work with a `@Bean`-annotated method instead of a component, you can achieve the same result. What you need to do is specify the names of the `init` and `destroy` methods as the values of `initMethod` and `destroyMethod` properties of the `@Bean` annotation.

The following code is equivalent to the one above, but this new version uses `@Bean` instead of `@Component`.

▼ Java

```
1  @Configuration
2  class Config {
3
4      @Bean(initMethod = "init", destroyMethod = "destroy")
5      public TechLibrary library() {
```

```
6         return new TechLibrary();
7     }
8 }
9
1
0 class TechLibrary {
1
1     private final List<String> bookTitles =
1
2         Collections.synchronizedList(new ArrayList<>());
1
3
1
4     public void init() {
1
5         bookTitles.add("Clean Code");
1
6         bookTitles.add("The Art of Computer Programming");
1
7         bookTitles.add("Introduction to Algorithms");
1
8         System.out.println("The library has been initialized: " +
bookTitles);
1
9     }
2
0
2
1     public void destroy() {
2
2         bookTitles.clear();
2
3         System.out.println("The library has been cleaned: " +
bookTitles);
2
4     }
2
5 }
```

▼ Kotlin

```
1 @Configuration
2 class Config {
3
4     @Bean(initMethod = "init", destroyMethod = "destroy")
5     fun library(): TechLibrary {
6         return TechLibrary()
7     }
8 }
9
1
0 class TechLibrary {
1
1     private val bookTitles: MutableList<String> =
Collections.synchronizedList(ArrayList())
1
2
1
3     fun init() {
1
4         bookTitles.add("Clean Code")
1
5         bookTitles.add("The Art of Computer Programming")
1
6         bookTitles.add("Introduction to Algorithms")
1
7         println("The library has been initialized: $bookTitles")
1
8     }
1
9 }
```

```
2
0      fun destroy() {
2
1          bookTitles.clear()
2
2          println("The library has been cleaned: $bookTitles")
2
3      }
2
4  }
```

If you run this code, the result will be exactly the same as before.

As an alternative, we can still add the `@PostConstruct` and `@PreDestroy` annotations to the `init` and `destroy` methods instead of specifying the attributes of the `@Bean` annotation, even if we use a `@Bean`-annotated method.

§4. Using interfaces for customization

Another way to customize the initialization and destruction processes is to implement the `InitializingBean` and `DisposableBean` interfaces, and then override their `afterPropertiesSet` and `destroy` methods correspondingly.

Here is the modified version of the previous example:

▼ Java

```
1  @Component
2  class TechLibrary implements InitializingBean, DisposableBean {
3      private final List<String> bookTitles =
4          Collections.synchronizedList(new ArrayList<>());
5
6      @Override
7      public void afterPropertiesSet() throws Exception {
8          bookTitles.add("Clean Code");
9          bookTitles.add("The Art of Computer Programming");
1
10         bookTitles.add("Introduction to Algorithms");
1
11         System.out.println("The library has been initialized: " +
bookTitles);
1
2      }
1
3
1
4      @Override
1
5      public void destroy() {
1
6          bookTitles.clear();
1
7          System.out.println("The library has been cleaned: " +
bookTitles);
1
8      }
1
9  }
```

▼ Kotlin

```
1  @Component
2  class TechLibrary : InitializingBean, DisposableBean {
3      private val bookTitles: MutableList<String> =
Collections.synchronizedList(ArrayList())
4
5      override fun afterPropertiesSet() {
6          bookTitles.add("Clean Code")
```

Table of contents:

- [1 Bean lifecycle](#)
- [§1. High-level overview of bean lifecycle](#)
- [§2. Customizing bean initialization and destruction](#)
- [§3. Using annotations for customization](#)

```
7         bookTitles.add("The Art of Computer Programming")
8         bookTitles.add("Introduction to Algorithms")
9         println("The library has been initialized: $bookTitles")
1
0     }
1
1
1
2     override fun destroy() {
1
3         bookTitles.clear()
1
4         println("The library has been cleaned: $bookTitles")
1
5     }
1
6 }
```

[§3.6. Customization](#)

[§4. Using interfaces for customization](#)

[§5. Post-processors for beans](#)

[§6. Conclusion](#)

[Discussion](#)

After running this code, the result will be exactly the same as it was with the annotations.

§5. Post-processors for beans

Now that we've considered a few ways of initializing beans to make them ready for use, let's look at another way that is even more straightforward. You can initialize beans using the `BeanPostProcessor` interface that allows for custom modification of bean instances. Using the post-processor, it is possible to run any custom operation before or right after a bean is initialized and even return a modified bean.

To start using post-processors, your class should:

- 1. Implement the `BeanPostProcessor` interface;
- 2. Override the `postProcessBeforeInitialization` and/or `postProcessAfterInitialization` methods.

Here is an example of a post-processor that only prints the name of the processed bean:

▼ Java

```
1 @Component
2 class PostProcessor implements BeanPostProcessor {
3
4     @Override
5     public Object postProcessBeforeInitialization(
6         Object bean, String beanName) throws BeansException {
7
8         System.out.println("Before initialization: " + beanName);
9
10        return BeanPostProcessor.super
11
12            .postProcessBeforeInitialization(bean, beanName);
13    }
14
15    @Override
16    public Object postProcessAfterInitialization(
17
18        Object bean, String beanName) throws BeansException {
19
20
21
22        System.out.println("After initialization: " + beanName);
```

```
1
9
2
0         return BeanPostProcessor.super
2
1             .postProcessAfterInitialization(bean, beanName);
2
2     }
2
3 }
```

▼ Kotlin

```
1  @Component
2  class PostProcessor : BeanPostProcessor {
3
4      override fun postProcessBeforeInitialization(bean: Any,
beanName: String): Any? {
5          println("Before initialization: $beanName")
6          return super.postProcessBeforeInitialization(bean, beanName)
7      }
8
9      override fun postProcessAfterInitialization(bean: Any, beanName:
String): Any? {
1         println("After initialization: $beanName")
1
1         return super.postProcessAfterInitialization(bean, beanName)
1
2     }
1
3 }
```

If you run this code, you will see a long list of beans in your application at different stages of their lifecycle.

It is important that `BeanPostProcessor` is executed for each bean defined in the Spring context, including the beans created by the framework. At the same time, it is possible to keep some of the beans from being modified by writing a particular condition.

Post-processors is a much more advanced concept and for now, it is enough to get only the basic idea of it. Unlike `@PreDestroy` , `@PostConstruct` and other approaches for custom initialization, post-processors are used for processing multiple beans the same way. The processors aren't usually tied to business logic and provide some infrastructure code that modifies or wraps the beans. If you want to learn more, you can take a look at [this article](#) (Java) on Spring bean lifecycle.

§6. Conclusion

In this topic, you've learned about the main stages of bean lifecycle managed by the Spring Container. You have also studied several ways to customize the initialization and destruction parts of the lifecycle using the `@PostConstruct` and `@PreDestroy` annotations, as well as the `InitializingBean` and `DisposableBean` interfaces. And finally, we've introduced you to the concept of a post-processor that allows for custom modification of new bean instances.

 Report a typo

72 users liked this piece of theory. 3 didn't like it. What about you?



Start practicing

[Comments \(2\)](#)

[Useful links \(0\)](#)

[Show discussion](#)



- Tracks
- Pricing
- For organizations

- About
- Contribute
- Careers

Become beta tester

Be the first to see what's new

Terms Support

Made with by Hyperskill and JetBrains